CE143: COMPUTER CONCEPTS & PROGRAMMING

UNIT-11
Pointers

N. A. Shaikh

nishatshaikh.it@charusat.ac.in

Topics to be covered

- Introduction to pointer, Declaration & Initialization
- Access value using pointer , Indirection (*) operator
- Chain of Pointers
- Pointers in expressions , scale factor
- 1D-array and pointer
- Pointer with strings
- Array of pointers
- Pointer as arguments in function , Call by address
- Functions returning pointers
- Pointers and structures

Introduction to pointer

- Pointer is a **derived data type** in C.
- Different from other normal variables which can store values, pointers are special variables that can **hold the address of a variable** as their values.
- Since they store memory address of a variable, the pointers are very commonly said to “**point to variables**”.
- Since these memory addresses are the location in the computer memory where program instruction and data are stored
- pointers can be used to access and manipulate data stored in the memory

What is pointer?

A **pointer** is a variable that stores the address of another variable

Advantages:

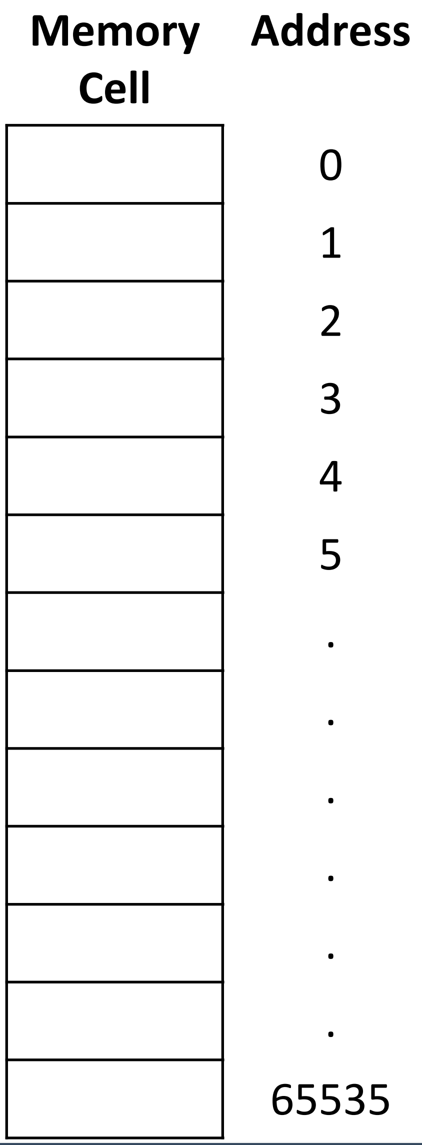
- More efficient in handling arrays and data tables.
- It can be used to **return multiple values** from a function via function argument.
- Use of pointer arrays to character strings results in **saving of data storage space** in memory.
- Pointer allow C to **support dynamic memory management**

Advantages of pointer

- Provide an efficient tool for **manipulating dynamic data structures** (structures, linked lists, queues, stacks and trees)
- It **reduce length and complexity** of programs
- **Increase the execution speed** and thus **reduce the program execution time.**

Understanding Pointers

- The computer's Memory is a sequential collection of storage cells as shown in the figure.
- Each cell known as **byte** has a number called address associated with it.
- The addresses are numbered consecutively starting from **zero**.
- The last address depends on the memory size.
- A computer system having **64 K** memory will have its last address as **65,535**

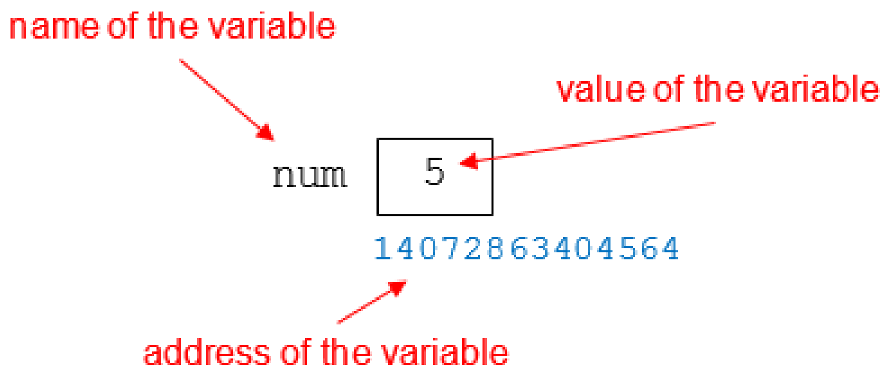


Understanding Pointer

- Whenever we declare a variable, the system allocates an appropriate location(address) to hold the value of the variable

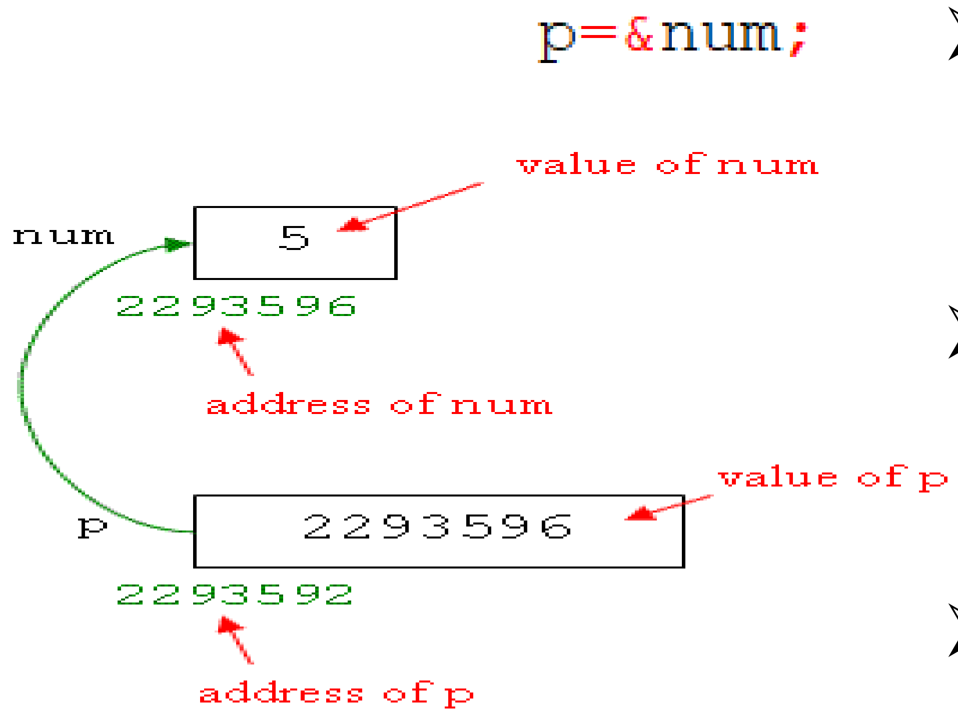
Example:

```
int num=5;
```



Understanding Pointer

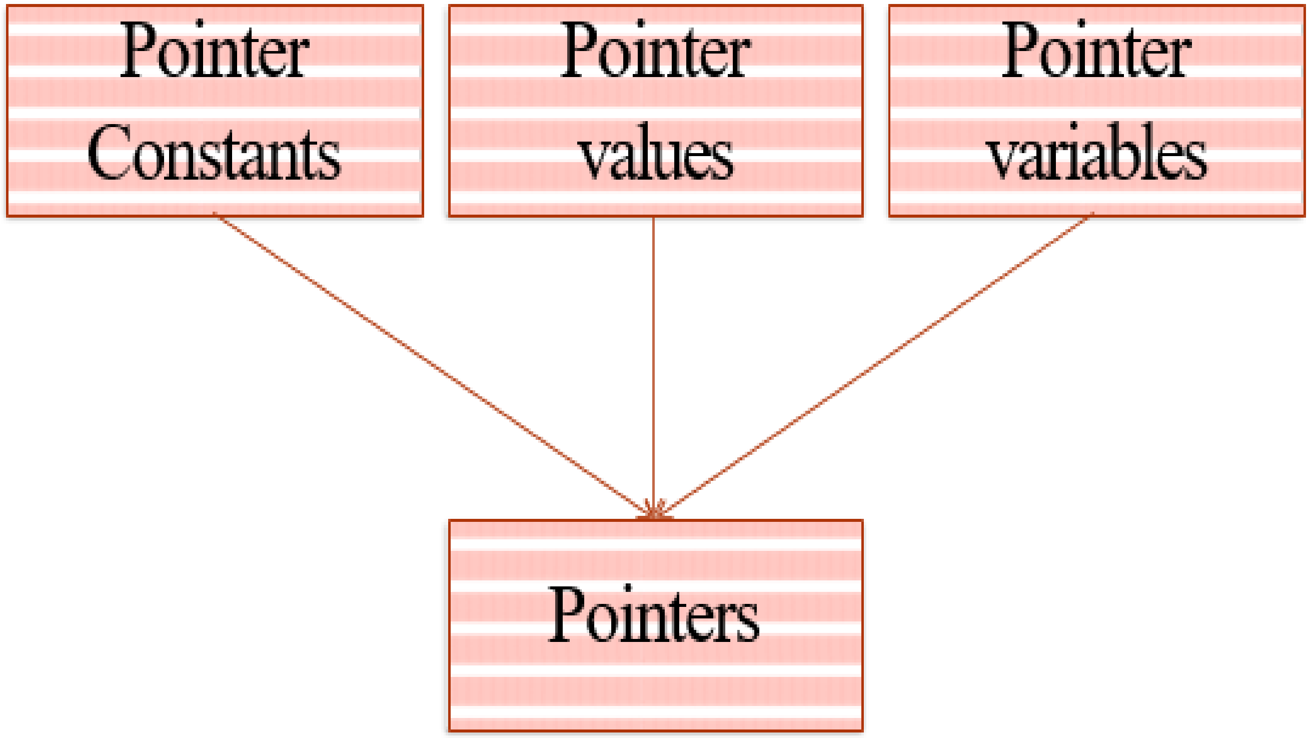
- Since pointer is a variable, its value is also stored in the memory in another location
- Suppose, we assign address of num to a variable p



- Now we can access the **value of num using p**
- Therefore, we say that the **variable p points to variable num**
- Thus p gets the name '**pointer**'

Underlying Concepts of Pointers

- Pointers are built on the three underlying concepts:



Understanding Pointer

Pointer Constants: Memory addresses within a computer are referred to as **pointer constants**. We cannot change them; we can only use them to store data values. They are like house numbers.

Pointer Values: We cannot save the value of a memory address directly. We can only obtain the value through the variables stored there using the address operator (&).The value obtained is known as **pointer value**. Pointer values may change from one run of a Program to another.

Pointer Variables: Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a **pointer variable**.

Accessing the address of a variable

- To determine the address of variable, operator & will be used.
- The **& operator** is known as ‘**address of**’.
- The operator & immediately preceding a variable **returns the address of a variable** associated with it.

Example:

```
p=&num;
```

Accessing the address of a variable

- The & operator can be used only with a simple variable or an array element.

Simple variable Example:

```
p=&num;
```

would assign the address of num to the pointer variable p.

Array element example:

- If x is an array, then the expression such as &x[0] and &x[i+ 3] are valid and represents the address of 0th and (i+3)th elements of x.

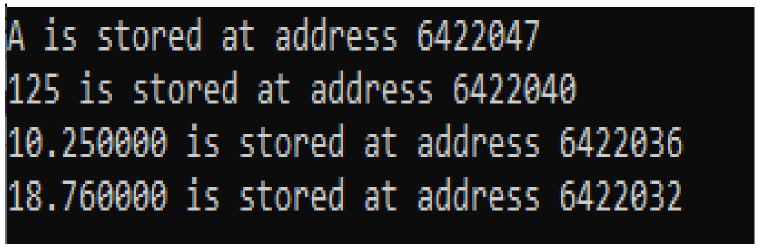
Accessing the address of a variable

The following are illegal use of address operator:

1. &125 (pointing at constants)
2. int x [10];   &x (Pointing at array names)
3. &(x+y)   (Pointing at expressions).

Accessing the address of a variable

```c
void main()
{
    char a='A';
    int x=125;
    float p=10.25, q= 18.76;

    printf("%c is stored at address %u\n",a,&a);
    printf("%d is stored at address %u\n",x,&x);
    printf("%f is stored at address %u\n",p,&p);
    printf("%f is stored at address %u\n",q,&q);
}
```



```
A is stored at address 6422047
125 is stored at address 6422040
10.250000 is stored at address 6422036
18.760000 is stored at address 6422032
```

Declaring Pointer Variables

They must be declared as pointers before we use them

Syntax:

```
data_type *poinetr_name;
```

This tells the compiler three things about the variable pointer_name.

1. The asterisk (*) tells that the variable pointer_name is a pointer variable.
2. pointer_name needs a memory location.
3. pointer_name points to a variable of type data_type.

Declaring Pointer Variables

Exapmle:

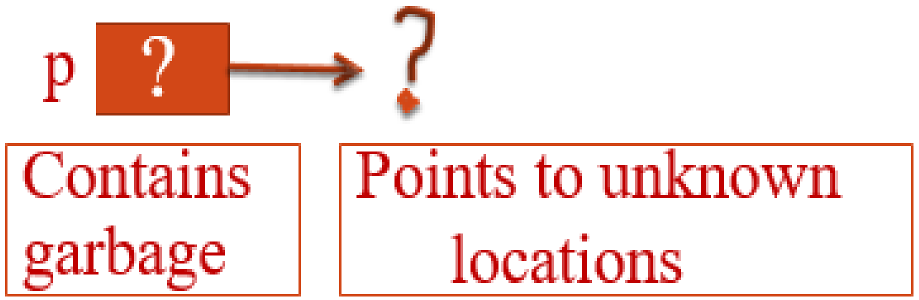
```
float *p;
```

- Declares the variable p as a pointer variable that points to an float data type.
- **The type float refers to the data type of the variable being pointed to by p and not the type of the value of the pointer.**
- The declaration causes the compiler to allocate memory locations for the pointer variable p.

Declaring Pointer Variables

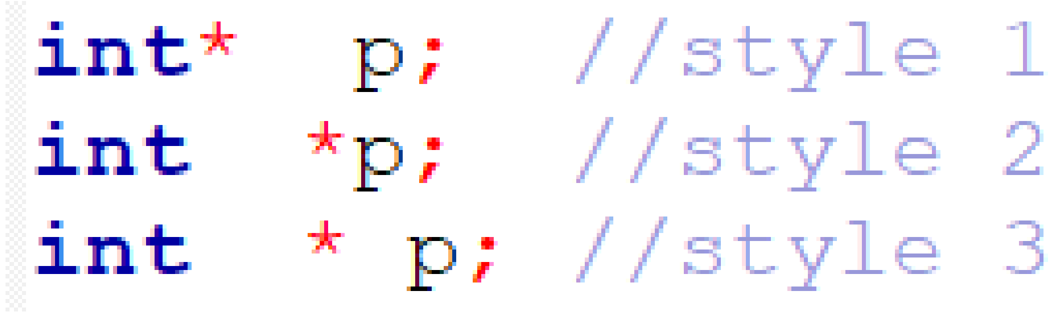
- Since the memory allocation have not been assigned any values, these locations may contain some unknown values in them and therefore they **point to unknown locations** as shown

```
int *p;
```



Pointer Declaration Style

- Pointer variable are declared similarly as normal variables except for the addition of the **unary * operator.**
- This symbol can appear anywhere between the type name and the pointer variable name.



```
int*   p;  //style 1
int   *p;  //style 2
int   * p; //style 3
```

Pointer Declaration Style

However style 2 is becoming popular due to following reasons:

1. This style is convenient to have multiple declaration in the same statement

```
int *p, x, *q;
```

2. This style matches with the format used for accessing the target values

```
int x,*p;
x=10;
p=&x;
*p=20; //assigning 20 to x
```

Initialization of Pointer Variables

- The process of assigning the address of a variable to a pointer variable is known as **initialization**.
- All **uninitialized pointer** will have some **unknown values**
- They may not be valid addresses or they may point to some values that are wrong.
- Since the compilers do not detect these errors, the programs with uninitialized pointers will produce **erroneous results.**
- It is therefore important to **initialize pointer variables before they are used**

Initialization of Pointer Variables

- Once a pointer variable has been declared we can use the assignment operator to initialize the variable.

Example:

```
int qty;
int *P; //declaration
P=&qty; //initialization
```

- We can also combine the initialization with the declaration

```
int *p=&qty;   //Valid
```

- The only requirement here is that the variable qty must be declared before the initialization.

Initialization of Pointer Variables

- We must ensure that the pointer variables always point to the corresponding type of data.

Example:

```
float a, b ;
int x=10, *p;
p = &a;   //Wrong
p = &x;
```

- Will result in **erroneous output** because we are trying to assign the address of a float variable to an integer pointer.
- When we declare a pointer to be an int type , the system assumes that any address that the pointer will hold will point to an integer variable.
- **Compiler will not detect such errors and so care should be taken to avoid wrong pointer assignments.**

Initialization of Pointer Variables

- If is also possible to combine the declaration of data variable, the declaration of pointer variable, & the initialization of the pointer variable in one step.

Example:

```
int x,  *p=&x;   //Valid

int *p=&x,  x ;  //Invalid
```

- We can also define a pointer variable with an initial value **of NULL or 0(zero).**

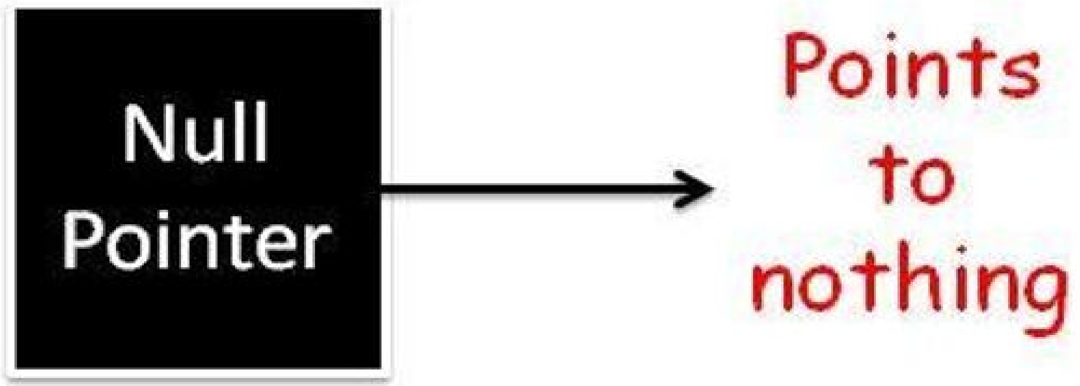
```
int *p = NULL;
int *p = 0;
```

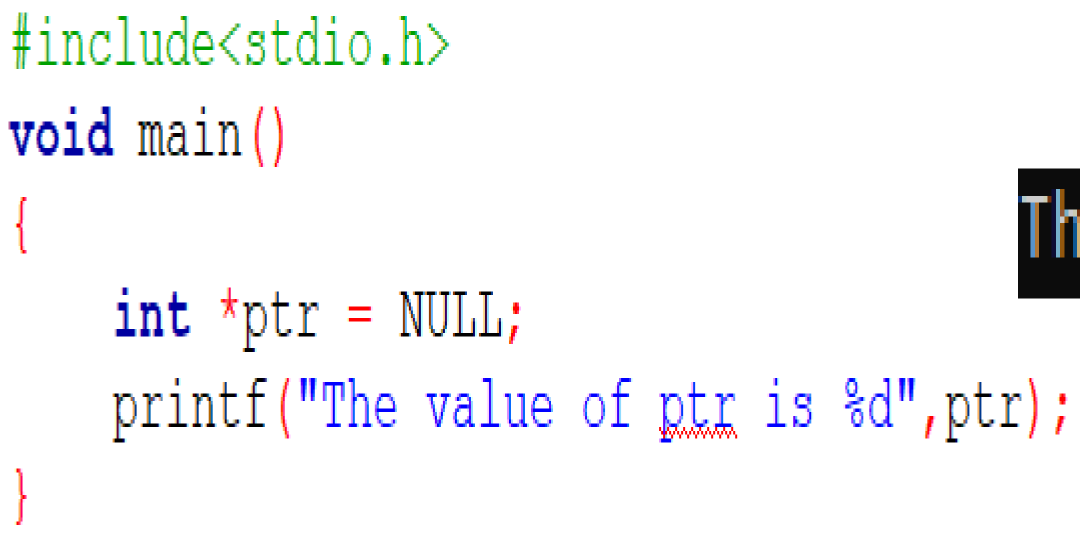
- With an exception of NULL and 0 , no other constant value can be assigned to a pointer variable.

```
int *p = 5360 //Invalid
```

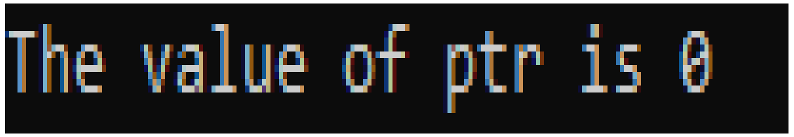
NULL POINTER

- **NULL pointers are pointers that contain as address the location 0**





```
#include<stdio.h>
void main()
{
    int *ptr = NULL;
    printf("The value of ptr is %d",ptr);
}
```

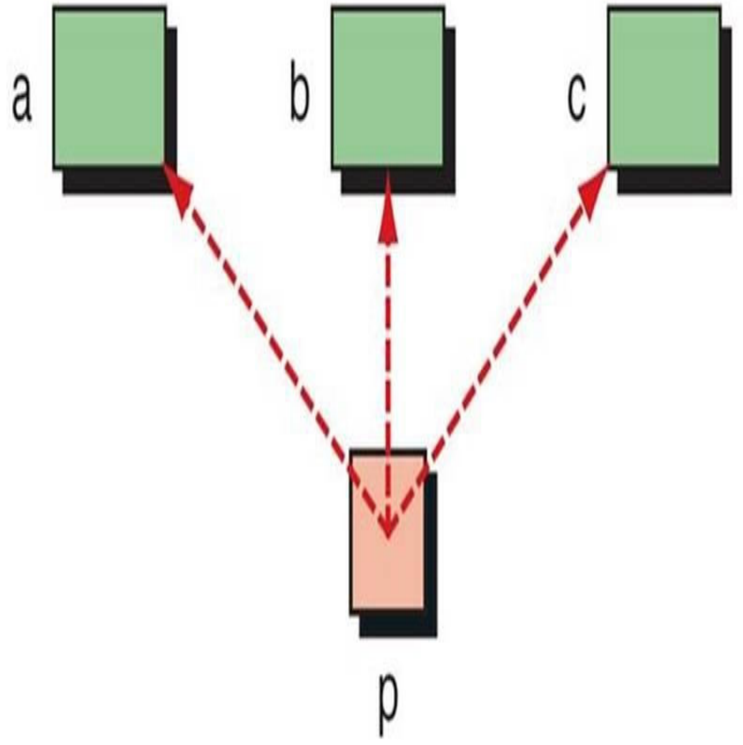


```
The value of ptr is 0
```

Pointer Flexibility: Pointers are Flexible

- We can make the same pointer to point to different data variables in different statement.

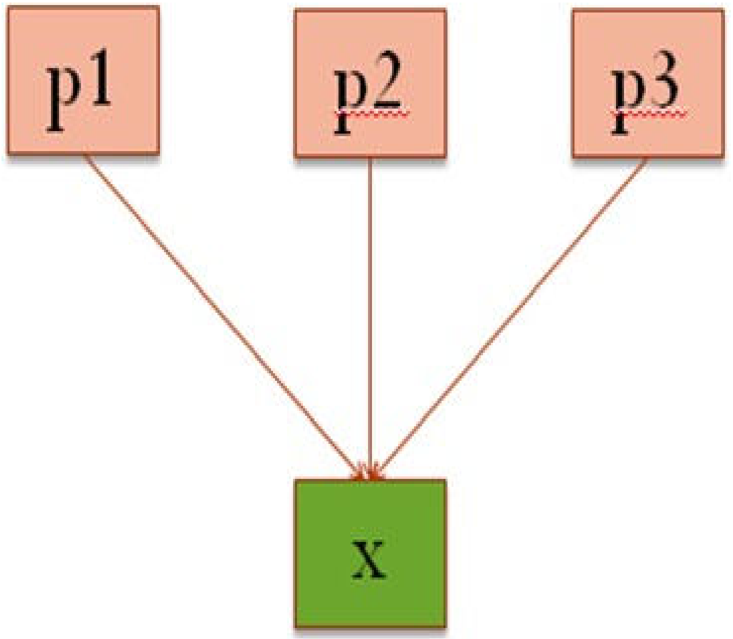
```
int a,b,c,*p;
...............
p = &a;
...............
p = &b;
..................
p = &c;
..................
```



Pointer Flexibility: Pointers are Flexible

- We can also use different pointers to point to the same data variable .

```
int x;
int *p1,*p2,*p3;
p1 = &x;
……………
p2 = &x;
……………
p3 = &x;
……………
```



Accessing a variable through its Pointer

- Once a pointer has been assigned the address of a variable, we can **access** the value of the variable using an operator ***** (asterisk), usually known as the **indirection operator**.
- Another name for the indirection operator is the **dereferencing operator**.

Example:

```
int qty,*p,n;
qty=179;
p=&qty;
n=*p;
```

NOTE: When the * operator is placed before a pointer variable in an expression(right hand side of equal sign), the pointer returns the value of the variable of which the pointer value is the address.

Accessing a variable through its Pointer

- In this case * p returns the value of the variable qty, because p is the address of qty.
- The * can be remembered as "**value at address**".
- The value of n would be 179.

Two statements

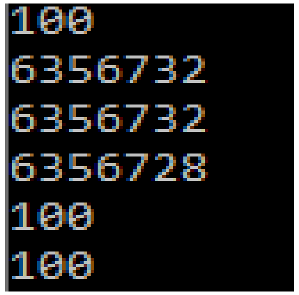
```
p = & qty;
n=*p;
```

are equal to

```
n=*&qty; (or) n=qty;
```

Accessing a variable through its Pointer

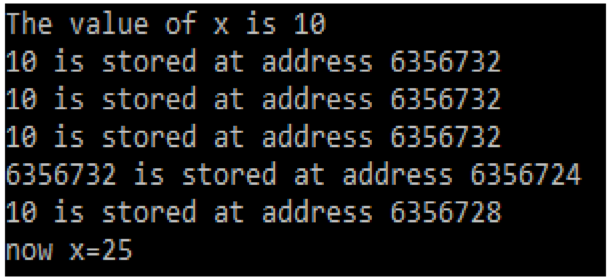
```c
void main()
{
     int i=100;
     int *p;
     p=&i;

     printf("\n%d",i);
     printf("\n%u",p);

     printf("\n%u",&i);
     printf("\n%u",&p);

     printf("\n%d",*p);
     printf("\n%d",*(&i));
}
```



```
100
6356732
6356732
6356728
100
100
```

Accessing a variable through its Pointer

```c
void main()
{
	int x,y;
	int *ptr;
	x=10;
	ptr=&x;
	y=*ptr;

	printf("The value of x is %d\n",x);
	printf("%d is stored at address %u\n", x,&x);
	printf("%d is stored at address %u\n", *&x,&x);
	printf("%d is stored at address %u\n",*ptr, ptr);
	printf("%d is stored at address %u\n", ptr, &ptr);
	printf("%d is stored at address %u\n", y, &y);

	*ptr=25;

	printf("now x=%d\n",x);
}
```



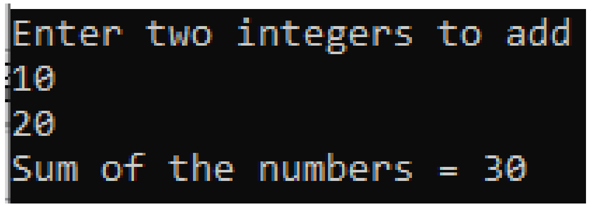
```
The value of x is 10
10 is stored at address 6356732
10 is stored at address 6356732
10 is stored at address 6356732
6356732 is stored at address 6356724
10 is stored at address 6356728
now x=25
```

NOTE:

```
x = *(&x) = *ptr = y
&x = &*ptr
```

Example

```c
//WAP that adds two integers using pointers
#include <stdio.h>
void main()
{
   int first, second, *p, *q, sum;

   printf("Enter two integers to add\n");
   scanf("%d%d", &first, &second);

   p = &first;
   q = &second;

   sum = *p + *q;

   printf("Sum of the numbers = %d\n", sum);
}
```

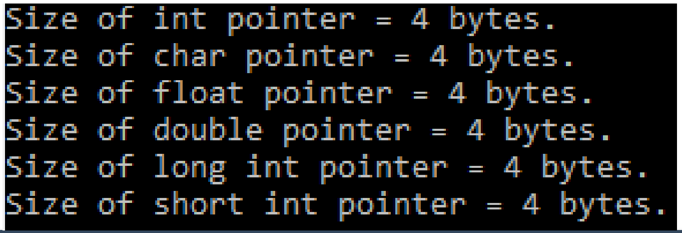


```
Enter two integers to add
10
20
Sum of the numbers = 30
```

Size of Pointers

- Any pointer (int, char, double, etc) size will be 2/4 byte depending on processor.
- **sizeof()** operator can be used to evaluate size of a variable/pointer in C.

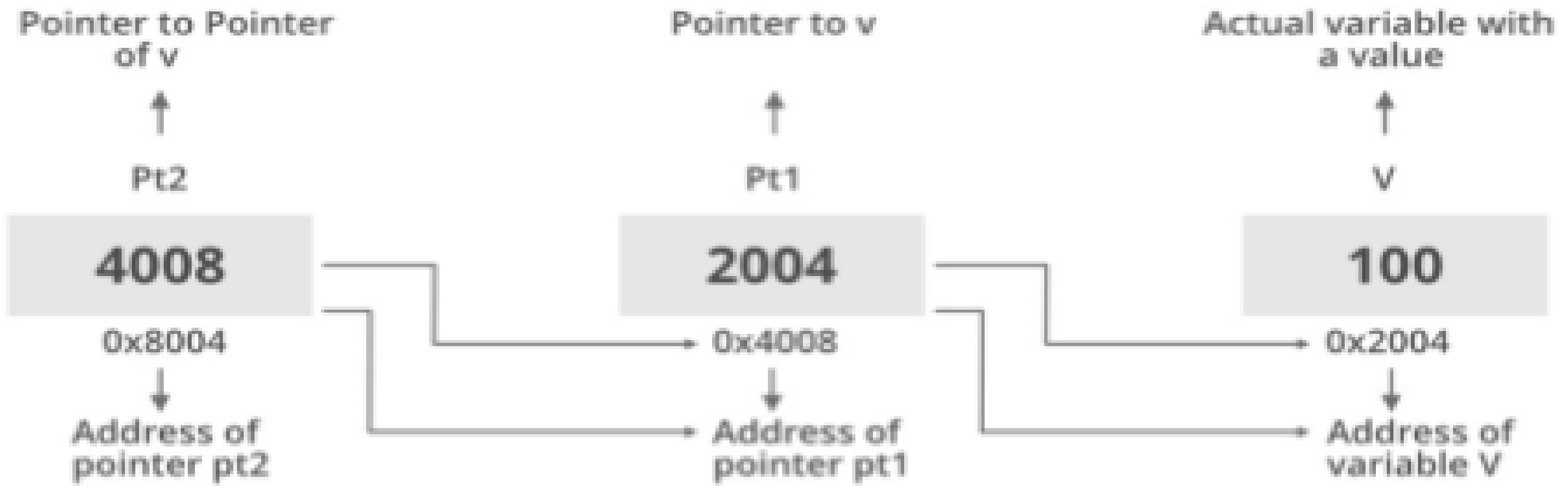
```c
#include<stdio.h>

int main()
{
    printf("Size of int pointer = %d bytes.\n", sizeof(int*));
    printf("Size of char pointer = %d bytes.\n", sizeof(char*));
    printf("Size of float pointer = %d bytes.\n", sizeof(float*));
    printf("Size of double pointer = %d bytes.\n", sizeof(double*));
    printf("Size of long int pointer = %d bytes.\n", sizeof(long*));
    printf("Size of short int pointer = %d bytes.\n", sizeof(short*));

    return 0;
}
```



```
Size of int pointer = 4 bytes.
Size of char pointer = 4 bytes.
Size of float pointer = 4 bytes.
Size of double pointer = 4 bytes.
Size of long int pointer = 4 bytes.
Size of short int pointer = 4 bytes.
```

Chain of Pointers / Pointer to Pointer

- It is possible to make a pointer to point to another pointer, thus creating a **chain of pointers**.



- Here, the pointer variable ptr2 contains the address of the pointer variable Ptr1, which points to the location that contains the desired value.
- This is known as **multiple indirections**.

Chain of Pointers / Pointer to Pointer

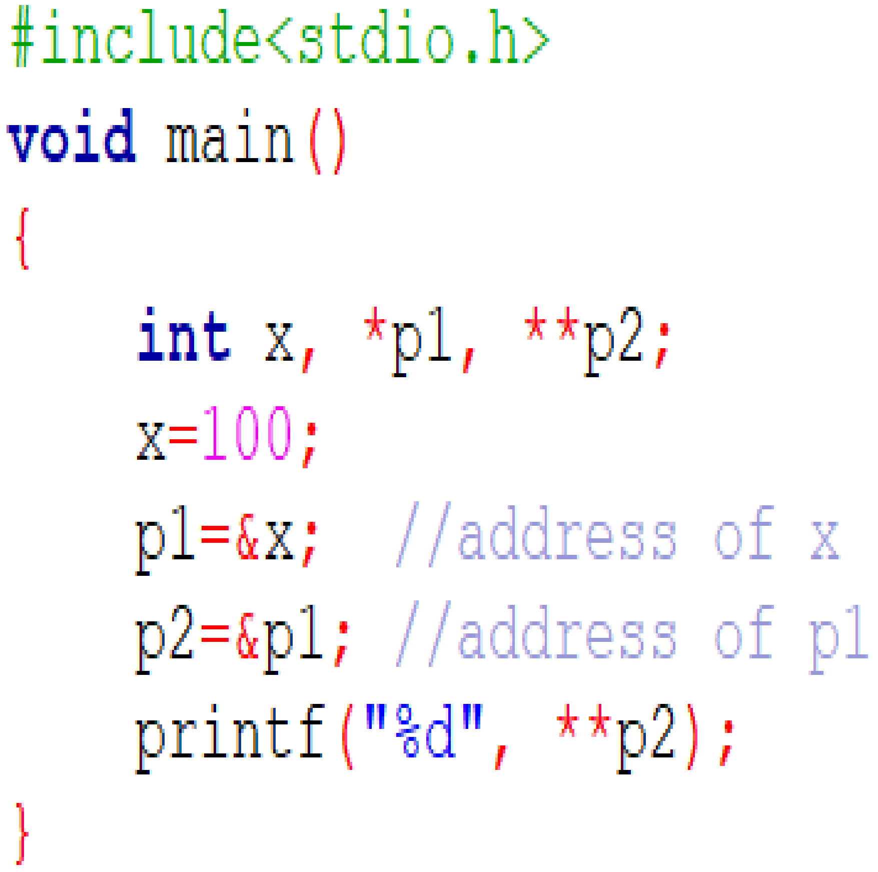
- A variable that is a pointer to a pointer must be declared using **additional indirection operator** symbols in front of the name.

For example,

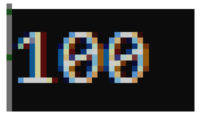
```
int **ptr2;
```

- This declaration tells the compiler that p2 is a pointer to a pointer of int type.
- The Pointer p2 is not an pointer to an integer ,But, a pointer to an integer pointer.
- We can access the target value indirectly pointed to by pointer to a pointer by applying indirection operator **twice**.

Chain of Pointers / Pointer to Pointer



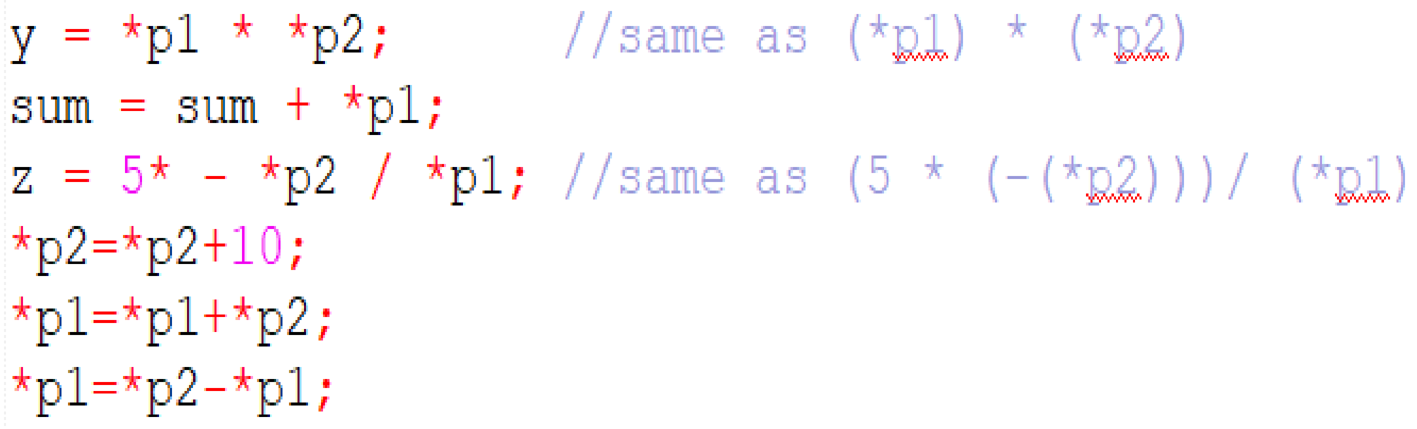
```
#include<stdio.h>
void main()
{
    int x, *p1, **p2;
    x=100;
    p1=&x;  //address of x
    p2=&p1; //address of p1
    printf("%d", **p2);
}
```



```
100
```

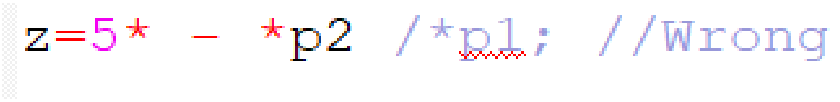
Pointer Expressions

Like other variables, pointer variables can be used in expressions. If p1 and p2 are properly declared and initialized pointers, then the following statements are valid:



```
y = *p1 * *p2;        //same as (*p1) * (*p2)
sum = sum + *p1;
z = 5* - *p2 / *p1; //same as (5 * (-(*p2)))/ (*p1)
*p2=*p2+10;
*p1=*p1+*p2;
*p1=*p2-*p1;
```

NOTE: in the third statement there is a **blank space between '/' and ***



```
z=5* - *p2 /*p1; //Wrong
```

because the symbol /*is considered as beginning of the comment and therefore the statement fails.

Pointer Expressions

- C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another.

```
p1+4;
p2-3;
p1-p2;
```

- If p1 and p2 are both pointers to the same array, then p2-p1 gives the number of elements between p1 and p2.
- We may also use short-hand operators with the pointers

```
p1++;
sum += *p2;
```

- Pointers can also be compared using the relational operators

```
p1>p2;
p1==p2;
p1!=p2;
```

Pointer Expressions

- We may not use pointers in division or multiplication.

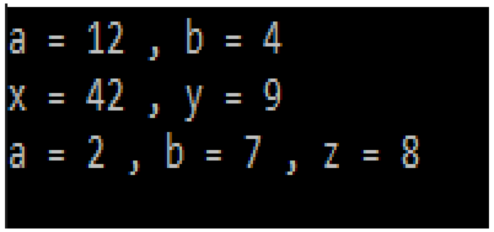
```
p1/p2;   //Not Allowed
p1*p2;   //Not Allowed
p1/3;    //Not Allowed
```

- Similarly two pointers cannot be added.

```
p1+p2;   //illegal
```

Pointer Expressions

```c
#include<stdio.h>
void main()
{ int a = 12 ,b = 4 ,*p1 = &a , *p2 = &b,  x , y , z ;

    x = *p1 * *p2 - 6;
    y = 4* - *p2/ *p1 + 10;

    printf("a = %d , b = %d \n x = %d , y = %d\n", a , b, x , y);

    *p2 = *p2 +3;
    *p1 = *p2 - 5;
     z = *p1 * *p2 - 6;

     printf("a = %d , b = %d , z = %d\n", a , b, z);
}
```



```
a = 12 , b = 4
x = 42 , y = 9
a = 2 , b = 7 , z = 8
```

Pointer increments and scale factor

- We have seen that the pointers can be incremented like

```
p1=p2+2;
p1=p1+1;
p1++;
```

Will cause the pointer to point to the next value of its type

- For example, if p1 is an integer pointer with an initial value, say 2800, then after the operation p1=p1+1, the value of p1 will be 2802, and not 2801.
- That is, when we increment a pointer, its value is increased by the '**length**' of the data type that it points to.
- This length called **scale factor.**

Rules of Pointer Operations

1. Pointer Variable Can be Assigned the **address of another Variable**
2. Pointer Variable Can be Assigned the **value of another Pointer Variable**
3. Pointer Variable Can be **initialized with zero or NULL value**
4. Pointer variable Can be **Pre-fixed or Post-fixed with Increment or Decrement Operators.**
5. Integer value can be **added** or **Subtracted** from Pointer variable
6. When two pointers point to the same array , one pointer variable can be **subtracted** from another.
7. When two pointers point to the objects of same data types, then they can be **compared** using the Relational Operators.
8. A pointer variable cannot be **multiplied** or **divided** by an integer constant.
9. Two pointer variables **cannot be added**
10. A Value cannot be assigned to an arbitrary address.

    For example: &x=10 is illegal.

Pointer and Arrays

- Pointer to Array
- Array of Pointers

Pointer to 1D Array

- When an array is declared, the compiler allocates a **base address** and **sufficient amount of storage** to contain all the elements of the array in contiguous memory locations.
- The base address is the address of the first element(index 0) in the array.
- The compiler also defines the **array name as a constant pointer** to the first element.
- Suppose we declare an array as date as follows:

```
int date[5] = {23,56,78,87,90};
```

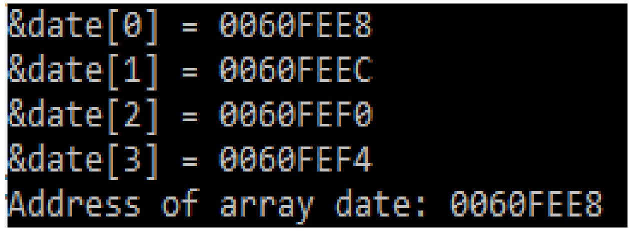
Elements	date[0]	date[1]	date[2]	date[3]	date[4]
Values	23	56	78	87	90
Address	2000	2004	2008	2012	2016

Pointer to 1D Array

- The name date is defined as a constant pointer pointing to the first element, date[0] and therefore the value of date is 2000, the location where date[0] is stored.

date = &date[0] = 2000

```c
//Address of &date[0] and date are same
#include<stdio.h>
int main()
{
    int date[5],i;

    for(i=0;i<4;++i)
    {
        printf("&date[%d] = %p\n", i , &date[i]);
    }

    printf("Address of array date: %p",date);

    return 0;
}
```



```
&date[0] = 0060FEE8
&date[1] = 0060FEEC
&date[2] = 0060FEF0
&date[3] = 0060FEF4
Address of array date: 0060FEE8
```

Pointer to 1D Array

- If we declare p as an integer pointer , then we can make the pointer p to point to the array date by

```
p = date;        is equivalent        p= &date[0];
```

- Now, we can access every value of date using p++ to move from one element to another.

```
date   = p   = &date[0] = 2000       *(date)   = *(p)   = date[0] = 23
date+1 = p+1 = &date[1] = 2004       *(date+1) = *(p+1) = date[1] = 56
date+2 = p+2 = &date[2] = 2008       *(date+2) = *(p+2) = date[2] = 78
date+3 = p+3 = &date[3] = 2012       *(date+3) = *(p+3) = date[3] = 87
date+4 = p+4 = &date[4] = 2016       *(date+4) = *(p+4) = date[4] = 90

date+i = p+i = &date[i]              *(date+i) = *(p+i) = date[i]
```

Pointer to 1D Array

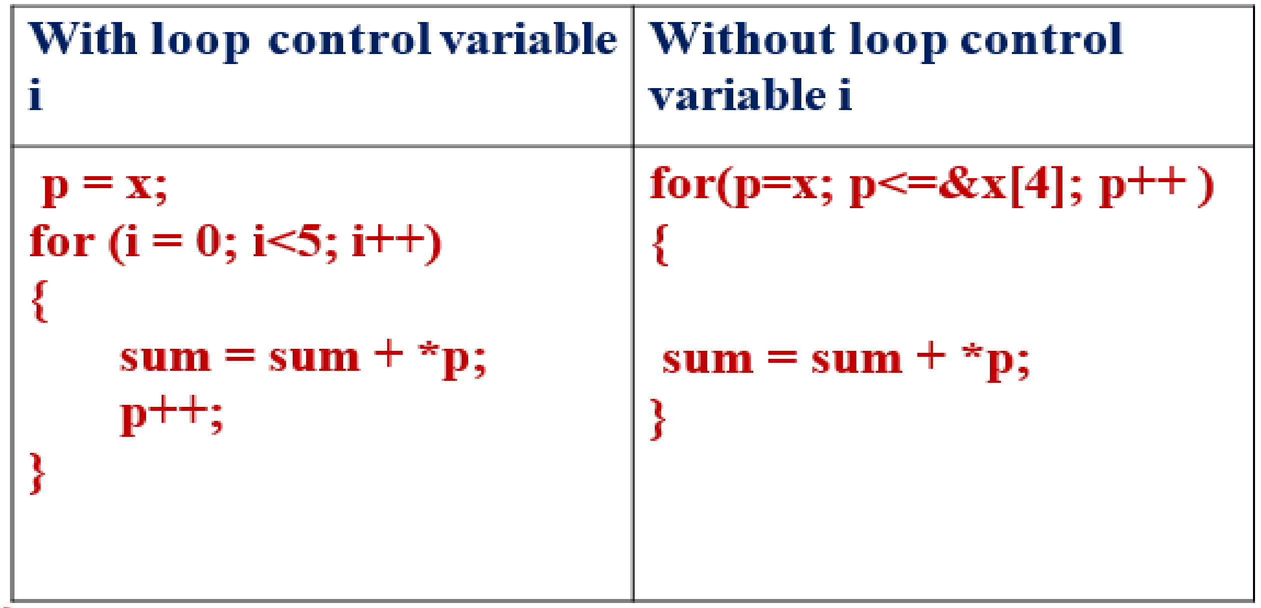
- The address of an element is calculated using its **index** and the **scale factor** of the data type.

Address of date[3] = base address + (3 * scale factor of int)
= 2000+ 3*4
= 2012

- When handling arrays, instead of using **array indexing**, we can **use pointers** to access array elements.
- Note that *(p+3) gives the value of date[3] .
- The pointer accessing method is much **faster** than indexing.

Pointer to 1D Array

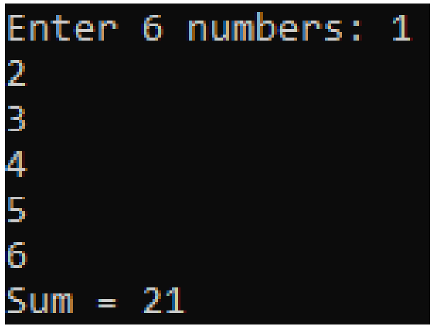
- We can avoid loop control variable i by writing the following code.



With loop control variable i	Without loop control variable i
p = x;	for(p=x; p<=&x[4]; p++)
for (i = 0; i<5; i++)	{
{	
sum = sum + *p;	sum = sum + *p;
p++;	}
}	

Example: Pointer to 1D Array

```c
//WAP using pointer to compute the sum of array
#include <stdio.h>
void main()
{
  int i, x[6], sum = 0,*p;
  p=x;  //initializing with base add of x

  printf("Enter 6 numbers: ");
  for(i = 0; i < 6; ++i)
  {
      // Equivalent to scanf("%d", &x[i]);
      // Equivalent to scanf("%d", x+i);
      scanf("%d", p+i);
  }

  for(i = 0; i < 6; ++i)
  {
      // Equivalent to sum = sum+ x[i]
      // Equivalent to sum = sum+ *(x+i)
      sum = sum+ *(p+i);
  }
  printf("Sum = %d", sum);
}
```



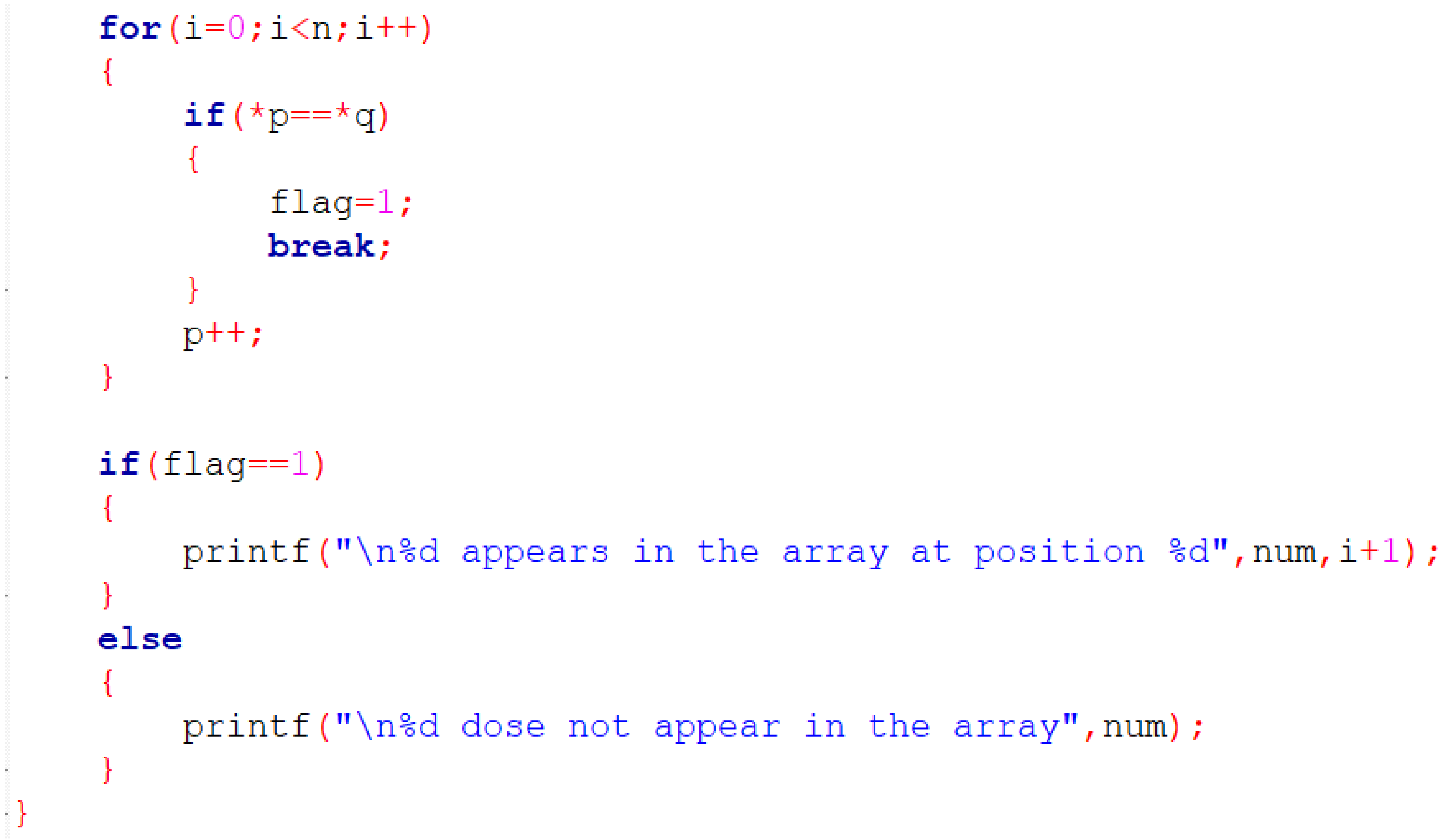
```
Enter 6 numbers: 1
2
3
4
5
6
Sum = 21
```

Example: Pointer to 1D Array

Write a program to enter N integer elements in array using pointer. Enter a number and search whether it exists in array or not. Assume no duplicates exist in array.

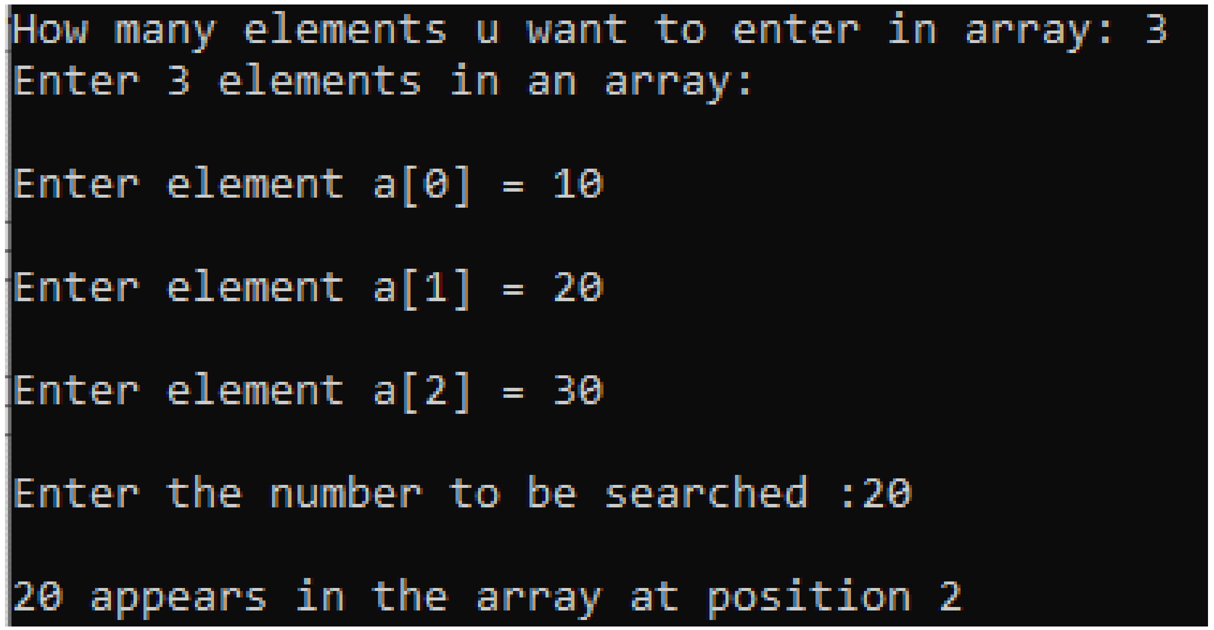
```
void main()
{
    int a[20],i,num,n,flag=0;
    int *p,*q;

    printf("How many elements u want to enter in array: ");
    scanf("%d",&n);

    printf("Enter %d elements in an array: \n",n);
    p=a;
    for(i=0;i<n;i++,p++)
    {
        printf("\nEnter element a[%d] = ",i);
        scanf("%d",p);
    }

    p=a;
    printf("\nEnter the number to be searched :");
    scanf("%d",&num);
    q=&num;
```

Example: Pointer to 1D Array



```
  for(i=0;i<n;i++)
  {
    if(*p==*q)
    {
      flag=1;
      break;
    }
    p++;
  }

  if(flag==1)
  {
    printf("\n%d appears in the array at position %d",num,i+1);
  }
  else
  {
    printf("\n%d dose not appear in the array",num);
  }
}
```

Example: Pointer to 1D Array



```
How many elements u want to enter in array: 3
Enter 3 elements in an array:

Enter element a[0] = 10

Enter element a[1] = 20

Enter element a[2] = 30

Enter the number to be searched :20

20 appears in the array at position 2
```

Practical 11.1

Write a program to read the marks of 10 students for the subject CE143 Computer concepts and Programming and computes the number of students in categories FAIL, PASS, FIRST CLASS and DISTINCTION using Pointers and Arrays.

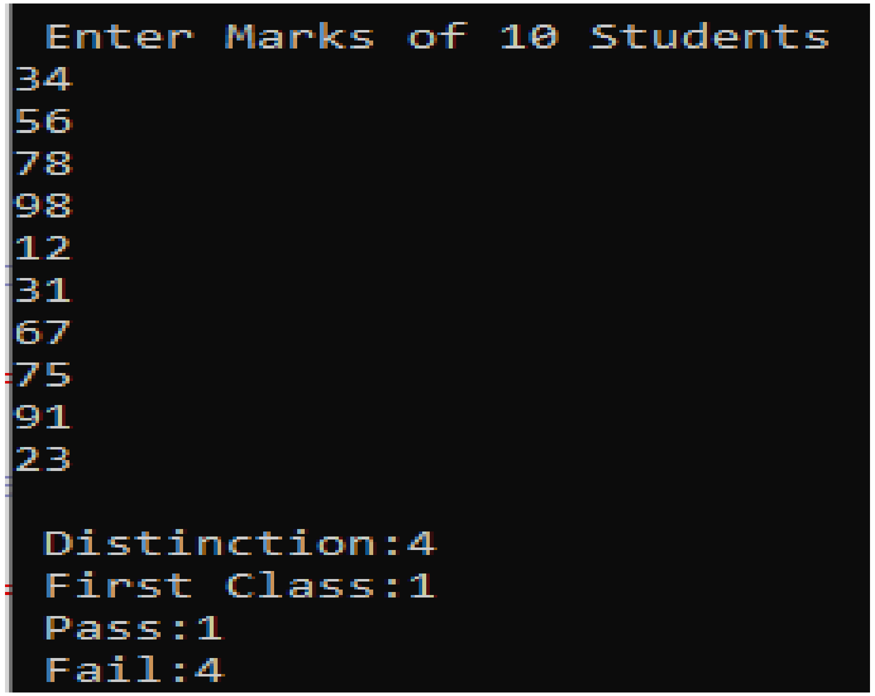
Marks	Categories
70 or Above	DISTINCTION
69 to 60	FIRST CLASS
59 to 40	PASS
Below 40	FAIL

For example, if following marks of 10 students are entered:
34 56 78 98 12 31 67 75 91 23
Then the output should be : **DISTINCTION 4 FIRST CLASS 1 PASS 1 FAIL 4**

Practical 11.1

```c
void main()
{
    int s_mark[10],i,*p,c1=0,c2=0,c3=0,c4=0;
    p=s_mark;

    printf("\n Enter Marks of 10 Students ");
    for(i=0;i<10;i++)
    {
        scanf("%d",(p+i));
    }

    for(i=0;i<10;i++)
    {
        if(*p>=70)
        {
            c1++; //distinction
        }
        else if(*p>=60 && *p<=69)
        {
            c2++;  //First Class
        }
        else if(*p>=40 && *p<=59)
        {
            c3++;  //Pass
        }
        else
        {
            c4++;  //Fail
        }
        p++;
    }
    printf("\n Distinction:%d ",c1);
    printf("\n First Class:%d ",c2);
    printf("\n Pass:%d ",c3);
    printf("\n Fail:%d ",c4);
}
```

Practical 11.1



```
 Enter Marks of 10 Students
34
56
78
98
12
31
67
75
91
23

 Distinction:4
 First Class:1
 Pass:1
 Fail:4
```

Let's Practice

1. WAP using pointer to read an array of integers and print its element in reverse order
2. Write a program using pointer to read in array of integers and sort an array of n elements into ascending order using pointer.

Pointer to 2D Array

- Pointers can be used to manipulate 2D array as well
- We know that in a 1D array x, following expression represents the element x[i]

```
*(x+i)   or   *(p+i)
```

- Similarly, an element in a 2D array can be represented by the pointer expression as follows:

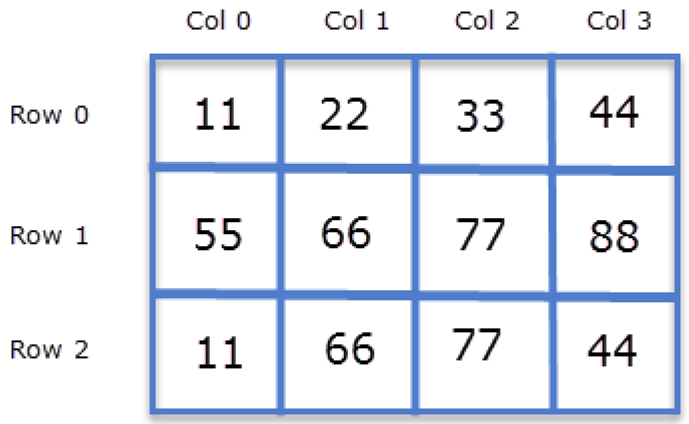
```
*(*(a+i)+j)   *(*(p+i)+j)
```

Pointer to 2D Array

- Suppose we declare an array as follows:

```
int arr[3][4] = {
                    {11,22,33,44},
                    {55,66,77,88},
                    {11,66,77,44}
                };
```

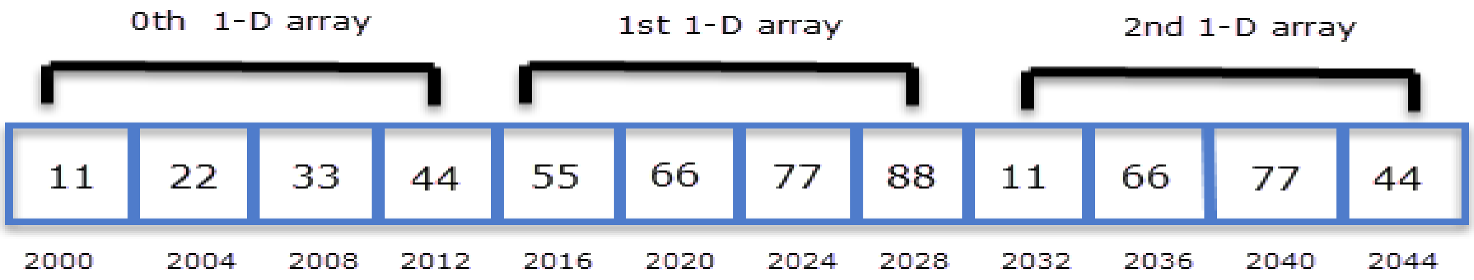
Logical Representation of memory:



	Col 0	Col 1	Col 2	Col 3
Row 0	11	22	33	44
Row 1	55	66	77	88
Row 2	11	66	77	44

Pointer to 2D Array

Actual Representation of memory:

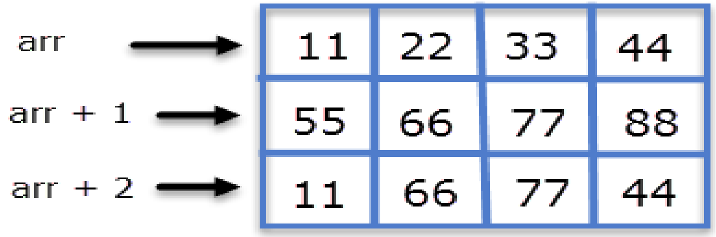


- ➢ we have already discussed that the name of a 1-D array is a constant pointer to the 0th element.
- ➢ In the case, of a 2-D array, **0th element is a 1-D array.**

`arr` points to 0th 1-D array.

`(arr + 1)` points to 1st 1-D array.

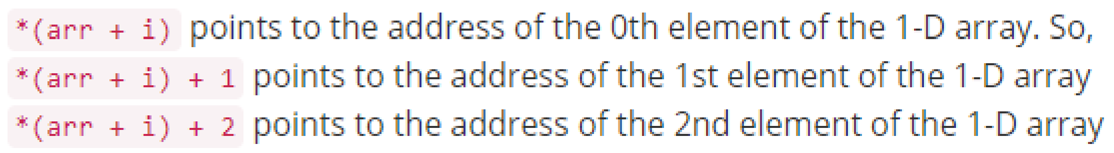
`(arr + 2)` points to 2nd 1-D array.



In general we can write

`(arr + i)` points to ith 1-D array.

Pointer to 2D Array



`*(arr + i)` points to the address of the 0th element of the 1-D array. So,
`*(arr + i) + 1` points to the address of the 1st element of the 1-D array
`*(arr + i) + 2` points to the address of the 2nd element of the 1-D array

Hence we can conclude that:

***(arr + i) + j** points to the base address of jth element in the ith row.

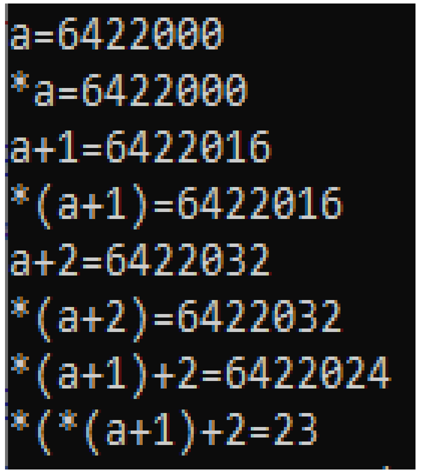
On dereferencing ***(arr + i) + j** we will get the value of jth element of ith row as shown below:

***(*(arr + i) + j)**

Pointer to 2D Array

a	→	**pointer to first row**
a+I	→	pointer to ith row
*(a+i)	→	pointer to first element in the ith row
*(a+i)+j	→	pointer to jth element in ith row
((a+i)+j)	→	Value stored in the cell(i,j) (ith row and jth column)

```
void main()
{
     int a[3][4]={11,12,13,14,21,22,23,24,31,32,33,34};

     printf("\na=%u",a);
     printf("\n*a=%u",*a);

     printf("\na+1=%u",a+1);
     printf("\n*(a+1)=%d",*(a+1));

     printf("\na+2=%u",a+2);
     printf("\n*(a+2)=%d",*(a+2));

     printf("\n*(a+1)+2=%d",*(a+1)+2);

     printf("\n*(*(a+1)+2)=%d",*(*(a+1)+2));
}
```

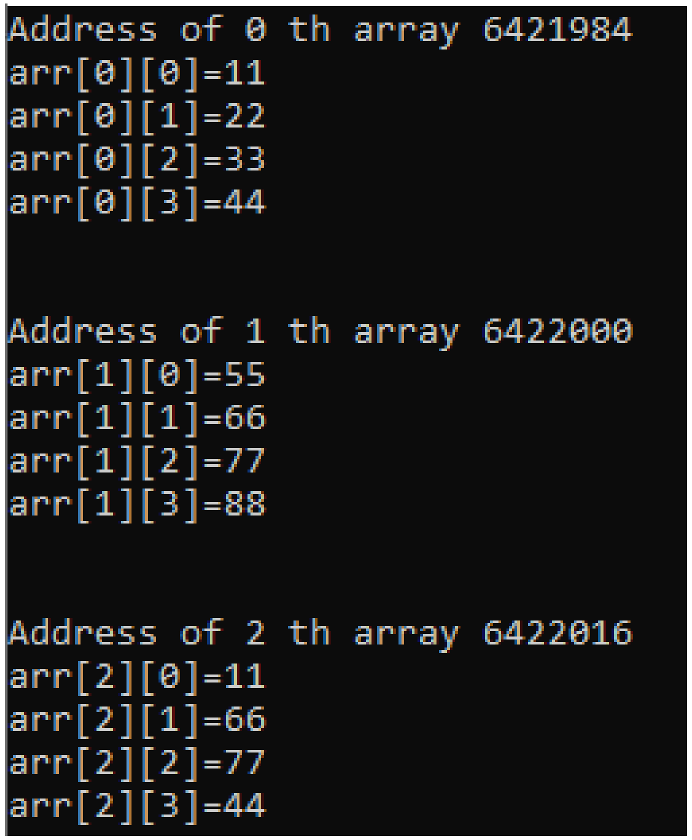


```
a=6422000
*a=6422000
a+1=6422016
*(a+1)=6422016
a+2=6422032
*(a+2)=6422032
*(a+1)+2=6422024
*(*(a+1)+2=23
```

Example: Pointer to 2D Array

```c
void main()
{
    int arr[3][4] = {
                        {11,22,33,44},
                        {55,66,77,88},
                        {11,66,77,44}
                    };
    int i, j;

    for(i = 0; i < 3; i++)
    {
        printf("Address of %d th array %u \n",i , *(arr + i));
        for(j = 0; j < 4; j++)
        {
            printf("arr[%d][%d]=%d\n", i, j, *( *(arr + i) + j) );
        }
        printf("\n\n");
    }
}
```

Example: Pointer to 2D Array



```
Address of 0 th array 6421984
arr[0][0]=11
arr[0][1]=22
arr[0][2]=33
arr[0][3]=44


Address of 1 th array 6422000
arr[1][0]=55
arr[1][1]=66
arr[1][2]=77
arr[1][3]=88


Address of 2 th array 6422016
arr[2][0]=11
arr[2][1]=66
arr[2][2]=77
arr[2][3]=44
```

Let's Practice

Write a program to find transpose of matrix using pointers and arrays.

Array of Pointers

Just like we can declare an array of int, float or char etc, we can also declare an **array of pointers**,

Syntax:

```
datatype *array_name[size];
```

Example:

```
int *arr[5];
```

Here arr is an array of 5 integer pointers. It means that this array can hold the address of 5 integer variables.

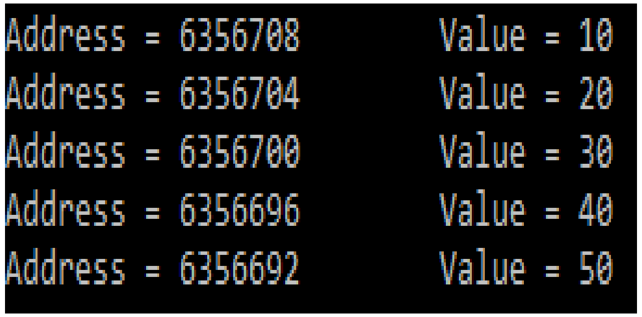
NOTE:

arr[index] will have address

*arr[index] will print the value.

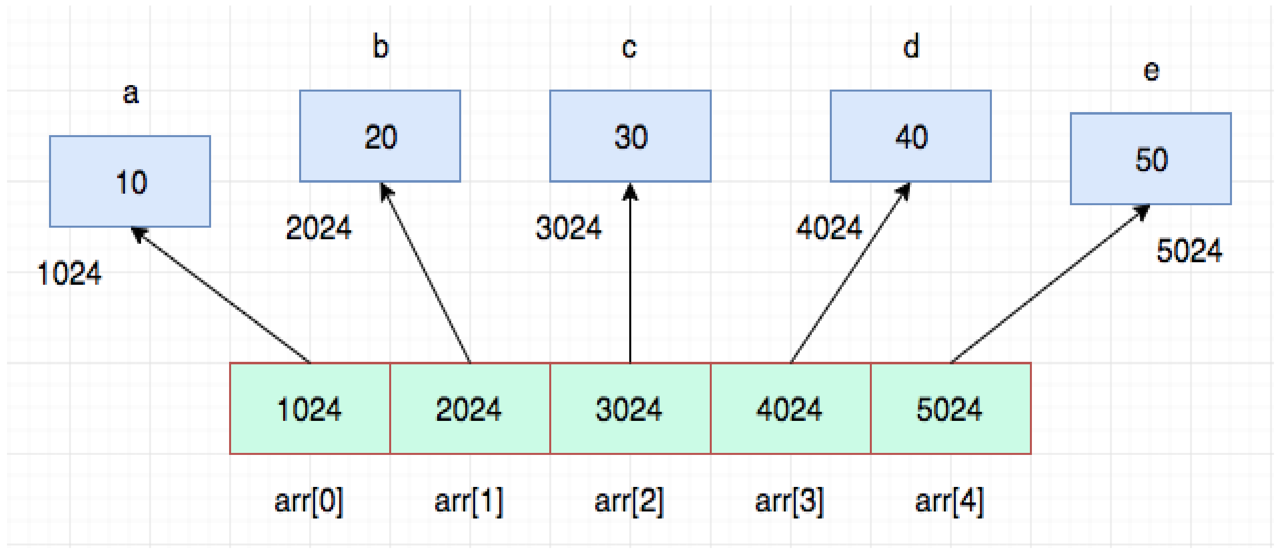
Array of Pointers

```c
void main()
{
    int *arr[5];
    int a = 10, b = 20, c = 30, d = 40, e = 50, i;

    arr[0] = &a;
    arr[1] = &b;
    arr[2] = &c;
    arr[3] = &d;
    arr[4] = &e;

    for(i = 0; i < 5; i++)
    {
        printf("Address = %u\t Value = %d\n", arr[i], *arr[i]);
    }
}
```

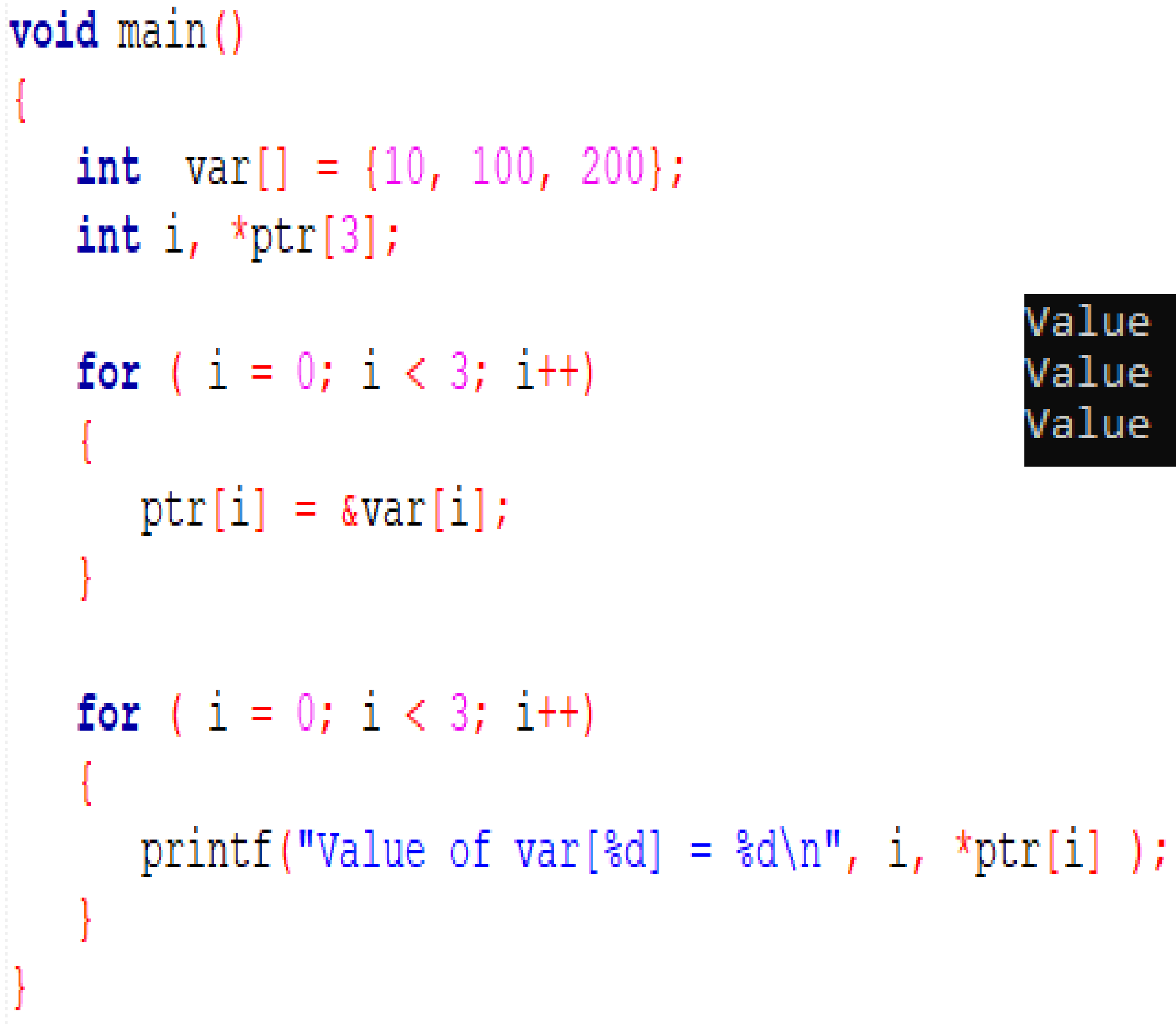


```
Address = 6356708       Value = 10
Address = 6356704       Value = 20
Address = 6356700       Value = 30
Address = 6356696       Value = 40
Address = 6356692       Value = 50
```

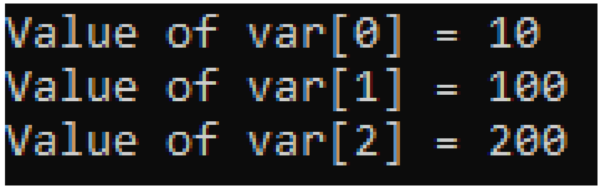
Array of Pointers



Array of Pointers



```c
void main()
{
   int  var[] = {10, 100, 200};
   int i, *ptr[3];

   for ( i = 0; i < 3; i++)
   {
      ptr[i] = &var[i];
   }

   for ( i = 0; i < 3; i++)
   {
      printf("Value of var[%d] = %d\n", i, *ptr[i] );
   }
}
```



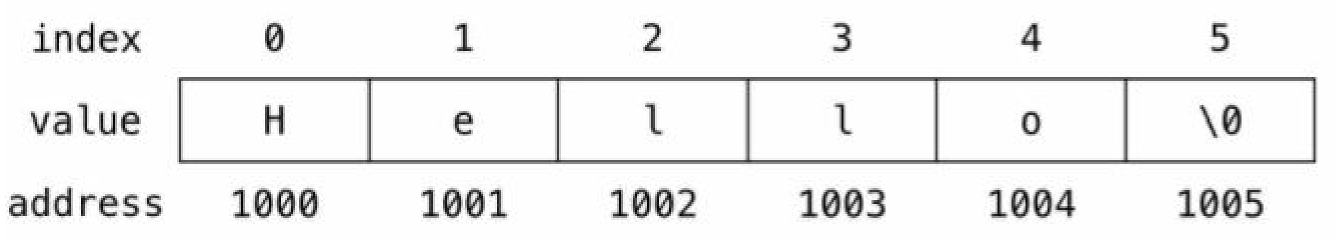
```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

Pointers and Character Strings

- Character arrays or string are declared and initialized as follows:

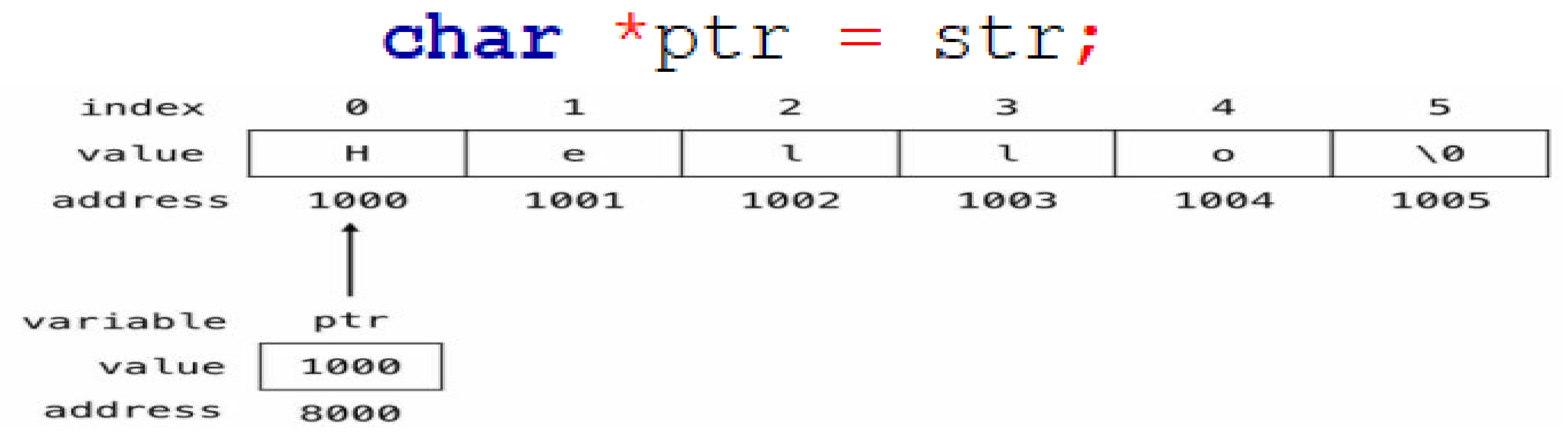
```
char str[6] = “hello”;
```

- The compiler automatically inserts the null character **‘\0’** at the end of the string.

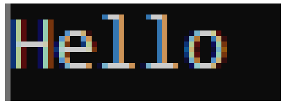


index	0	1	2	3	4	5
value	H	e	l	l	o	\0
address	1000	1001	1002	1003	1004	1005

Pointers and Character Strings



```
void main(void)
{
      char str[6] = "Hello";
      char *ptr = str;

      while(*ptr != '\0')
      {
            printf("%c", *ptr);
            ptr++;
      }
}
```



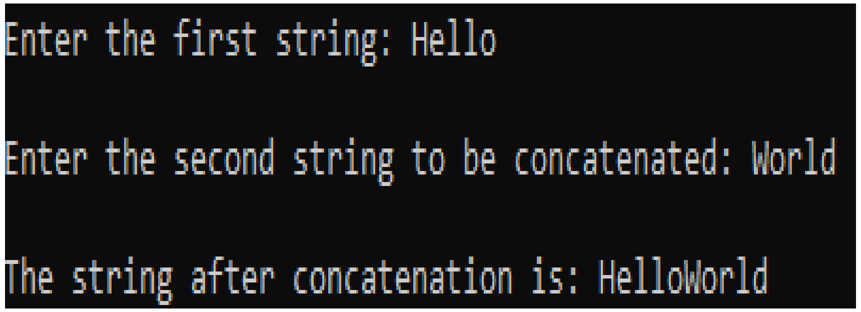
```
Hello
```

Pointers and Character Strings

Write a program to concatenate two string using pointer.
(**Pointers and Strings**)

Pointers and Character Strings

```
void main()
{
     char s1[10], s2[10];
     char *p1,*p2;
     printf("\nEnter the first string: ");
     scanf("%s",s1);
     printf("\nEnter the second string to be concatenated: ");
     scanf("%s",s2);

     p1 = s1;
     while(*p1!='\0')
     {
          p1++;
     }

     p2=s2;
     while(*p2!='\0')
     {
          *p1 = *p2;
          p1++;
          p2++;
     }
     *p1 = '\0';
     printf("\nThe string after concatenation is: %s ", s1);
}
```



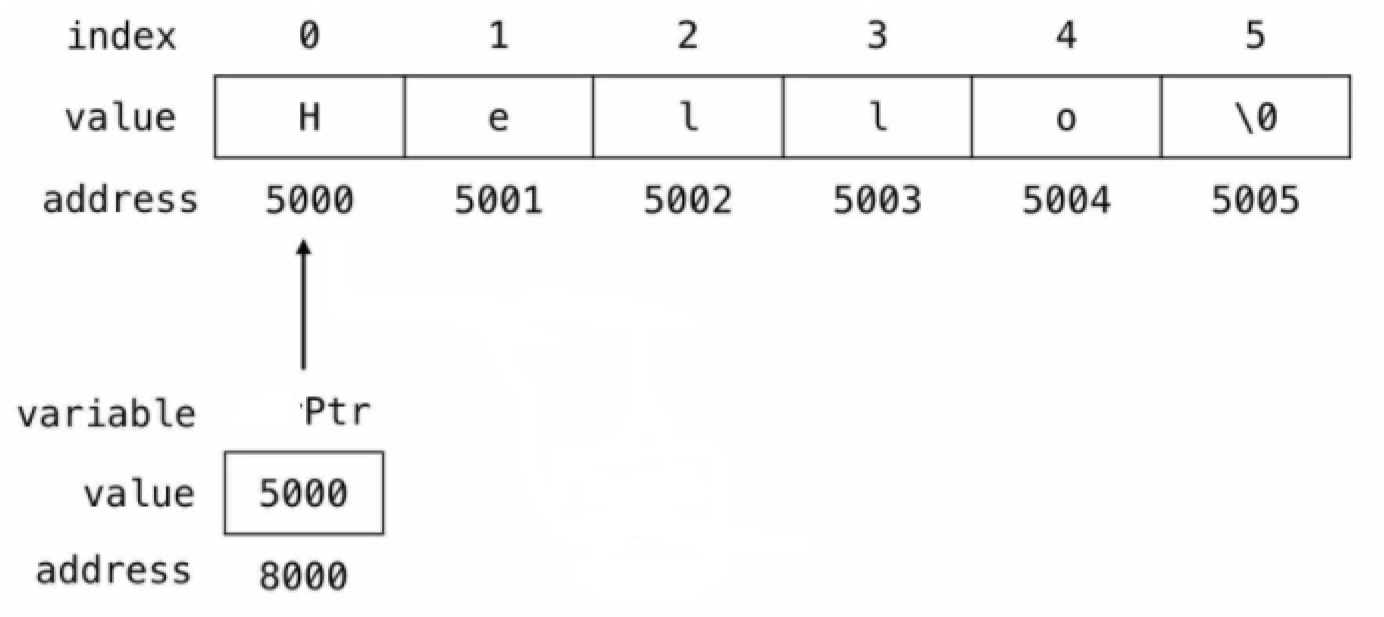
```
Enter the first string: Hello

Enter the second string to be concatenated: World

The string after concatenation is: HelloWorld
```

Pointers and Character Strings

Alternate method :

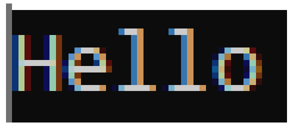
```
char *Ptr="hello";
```

This creates a string for the literal and then stores its address in the pointer variable ptr.



Pointers and Character Strings

```c
void main(void)
{
    char *Ptr = "Hello";

    while(*Ptr != '\0')
    {
        printf("%c", *Ptr);
        Ptr++;
    }
}
```



```
Hello
```

Pointers and Character Strings

We can also write

```
char * str;
str = "hello";
```

NOTE:

1. Here, str = “hello” is not a string copy because the variable str is a pointer, not a string
2. These type of assignment does not apply to character arrays

```
char str[20];
str="hello";
```

INVALID

Pointers and Character Strings

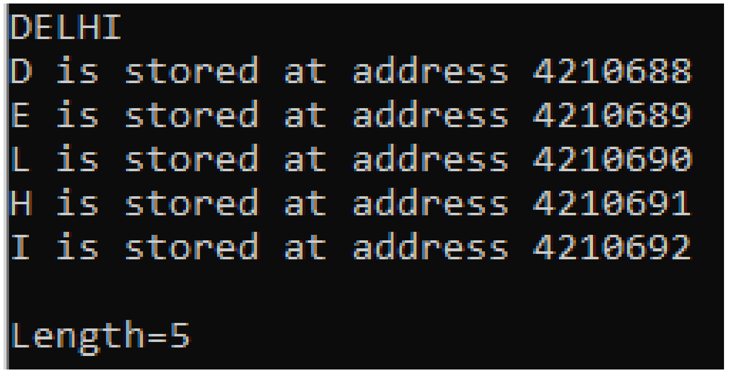
We can print the content

```
printf("%s", str);
puts(str);
```

NOTE: Although str is a pointer to the string, it is also the name of the string. Therefore We don’t need to use indirection operator * in printf

Pointers and Character Strings

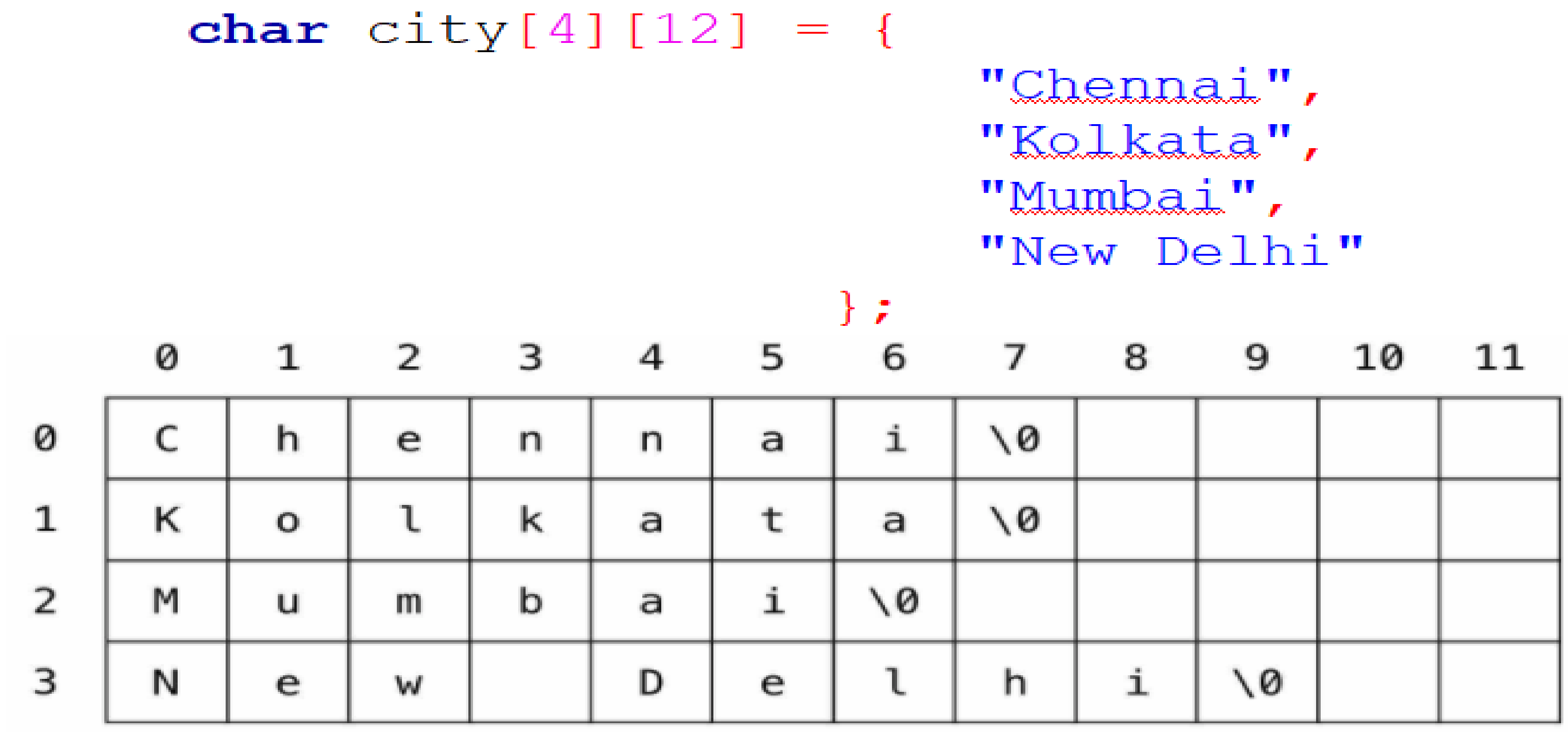
```c
void main(void)
{
    char *name;
    name="DELHI";
    int length;
    char *cptr=name;
    printf("%s\n",name);

    while(*cptr!='\0')
    {
        printf("%c is stored at address %u\n",*cptr,cptr);
        cptr++;
    }
    length=cptr-name;
    printf("\nLength=%d\n",length);
}
```



```
DELHI
D is stored at address 4210688
E is stored at address 4210689
L is stored at address 4210690
H is stored at address 4210691
I is stored at address 4210692

Length=5
```

Array of Pointers

One important use of pointers is in handling of a table of strings(2D char array).



```
char city[4][12] = {
                        "Chennai",
                        "Kolkata",
                        "Mumbai",
                        "New Delhi"
                   };
```

	0	1	2	3	4	5	6	7	8	9	10	11
0	C	h	e	n	n	a	i	\0				
1	K	o	l	k	a	t	a	\0				
2	M	u	m	b	a	i	\0					
3	N	e	w		D	e	l	h	i	\0		

Array of Pointers

Problem:

- Here, we are allocating 4x12 = 48 bytes memory to the city array and we are only using 33 bytes

- We can save those unused memory spaces by using pointers.

Array of Pointers

Solution:

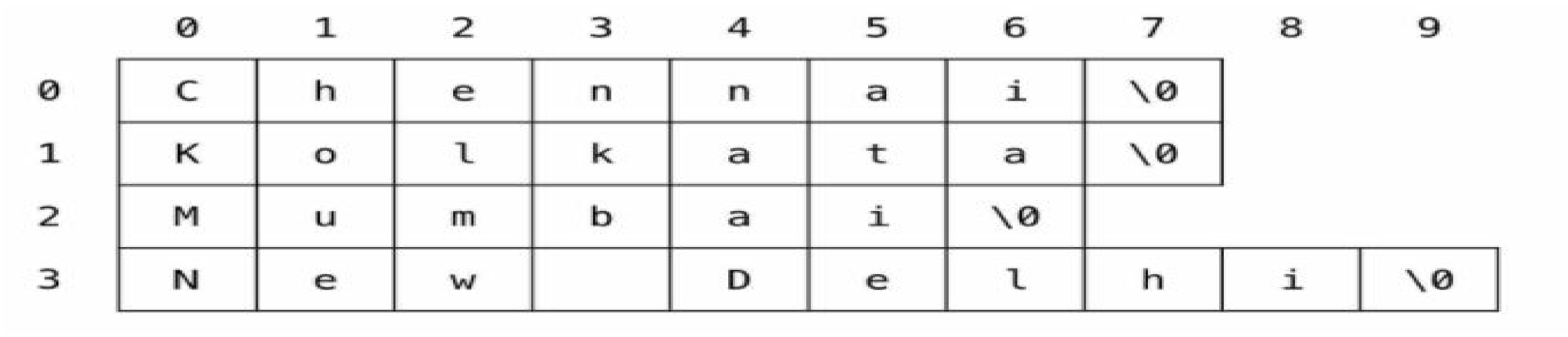
- We know that rarely the individual strings will be of equal lengths.
- Therefore, instead of making each row a fixed number of characters, we can make it a pointer to a string of varying length.

For example,

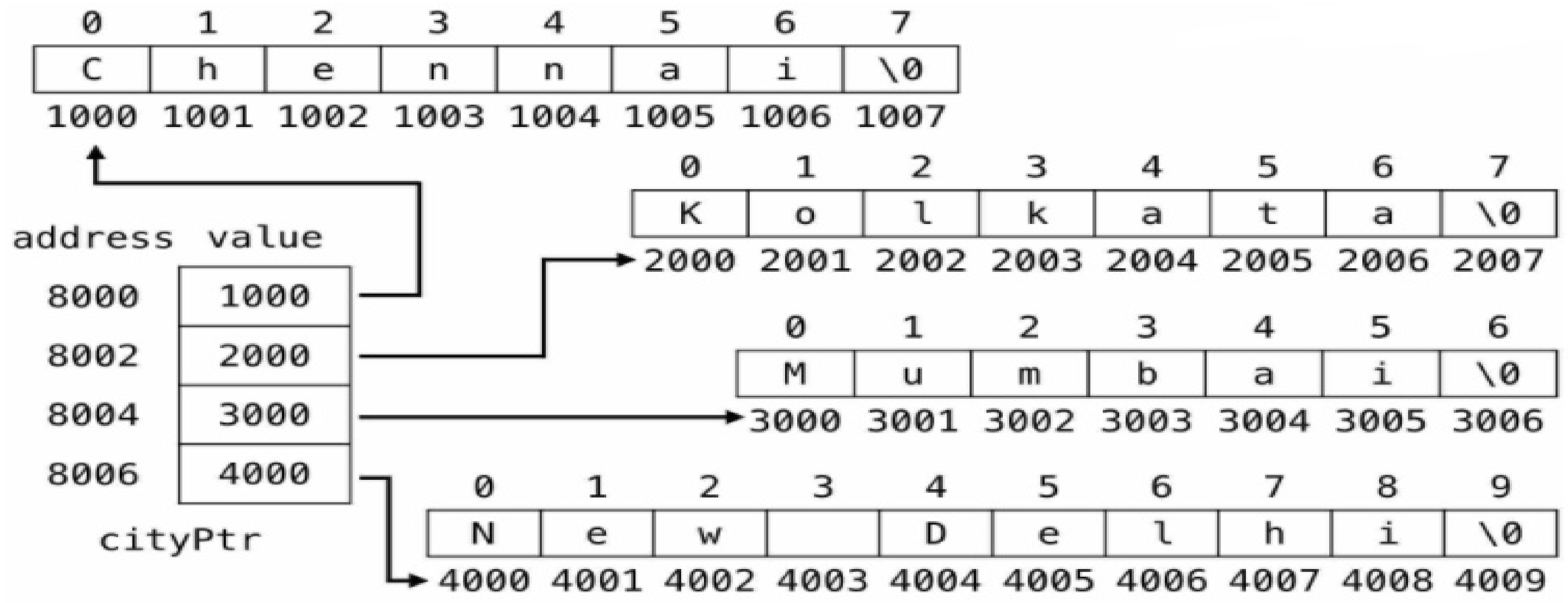
```
char *cityPtr[4] = {
                        "Chennai",
                        "Kolkata",
                        "Mumbai",
                        "New Delhi"
                   };
```

declares cityPtr to be an array of 4 pointers to characters, each pointer pointing to a particular city

Array of Pointers

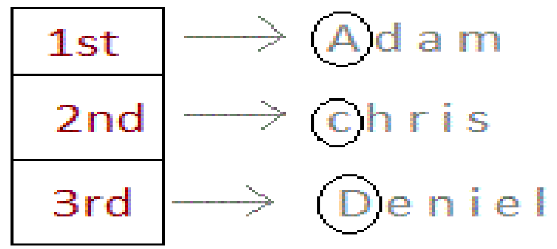


	0	1	2	3	4	5	6	7	8	9
0	C	h	e	n	n	a	i	\0		
1	K	o	l	k	a	t	a	\0		
2	M	u	m	b	a	i	\0			
3	N	e	w		D	e	l	h	i	\0



Array of Pointers

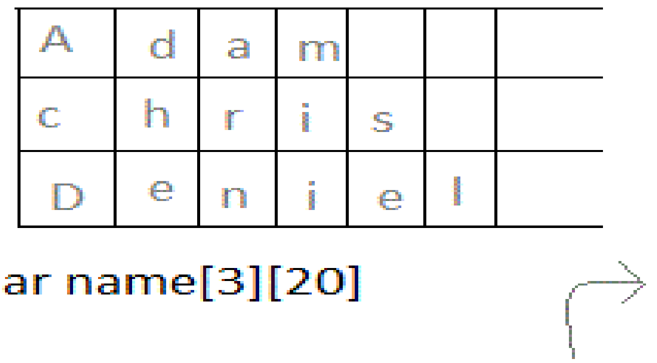
Using Pointer



char* name[3]

Only 3 locations for pointers, which will point to the first character of their respective strings.

Without Pointer



char name[3][20]

extends till 20 memory locations

Array of Pointers

NOTE:

1. The 2D character arrays with the rows of varying length(variable number of columns in each row) are called “**ragged arrays**”. Also known as “**jagged arrays**”

2. **Difference between the notation *p[3] and (*p)[3]:**

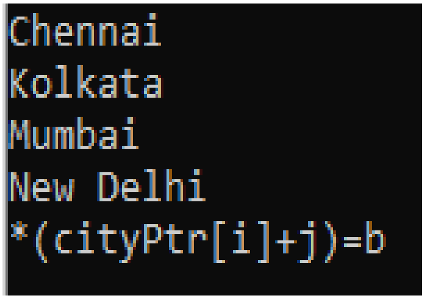
➢ Since * has a lower precedence than [ ] ,

➢ *p[3] declares p as an array of 3 pointers

➢ (*p) [3] declares p as a pointer to an array of three elements.

Array of Pointers

```
void main()
{
    char *cityPtr[4] = {
                            "Chennai",
                            "Kolkata",
                            "Mumbai",
                            "New Delhi"
                        };
    int i,j;
    for(i=0;i<=3;i++)
        printf("%s\n",cityPtr[i]);

    //To access jth character in the ith row
    i=2; j=3;
    printf("*(cityPtr[i]+j)=%c\n",*(cityPtr[i]+j));
}
```



```
Chennai
Kolkata
Mumbai
New Delhi
*(cityPtr[i]+j)=b
```

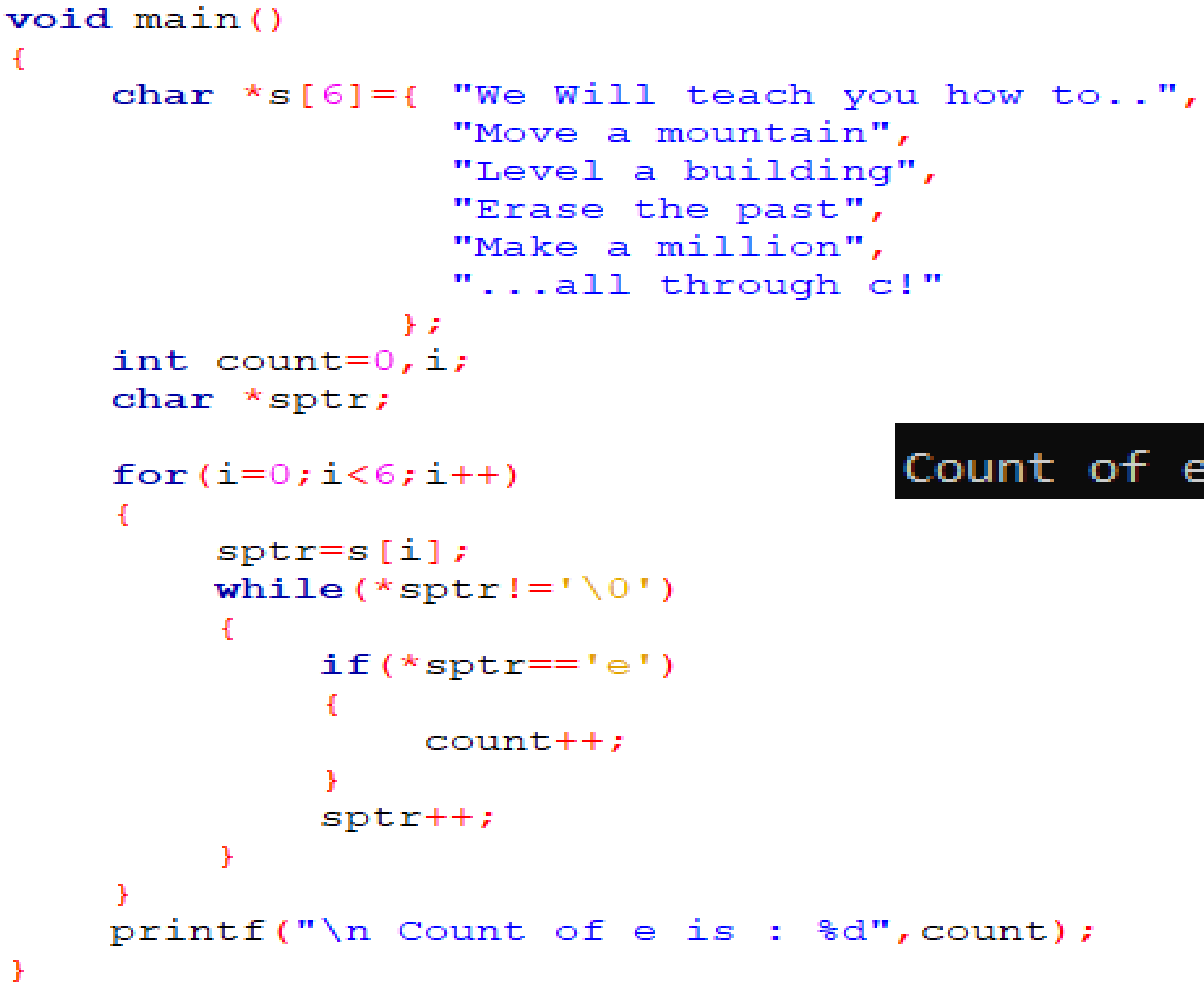
Practical 11.2

Write a program that counts the number of ‘e’ in the following array of pointers to strings.

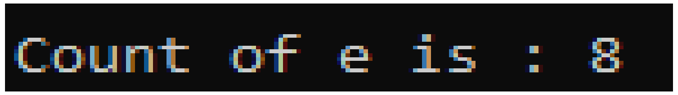
(Array of Pointers)

```
char *s[ ] =
{
        "We will teach you how to...",
        "Move a mountain",
        "Level a building",
        "erase the past",
        "Make a million",
        "...all through C!"
} ;
```

Practical 11.2



```
void main()
{
     char *s[6]={ "We Will teach you how to..",
                  "Move a mountain",
                  "Level a building",
                  "Erase the past",
                  "Make a million",
                  "...all through c!"
                };
     int count=0,i;
     char *sptr;

     for(i=0;i<6;i++)
     {
          sptr=s[i];
          while(*sptr!='\0')
          {
               if(*sptr=='e')
               {
                    count++;
               }
               sptr++;
          }
     }
     printf("\n Count of e is : %d",count);
}
```



```
Count of e is : 8
```

Pointer as function Arguments

- We can pass **address of a variable** as an argument to a function.
- When we pass addresses to a function, the parameters receiving the addresses should be **pointers**.
- The process of calling a function using pointers to pass the addresses of a variable is known as **Call by Address** or **Pass by Address** or **pass by pointers**.
- The process of passing actual value of a variables is known as **call by value**.
- The function which is called by reference **can change the value** of the variable used in the call.

Pointer as function Arguments

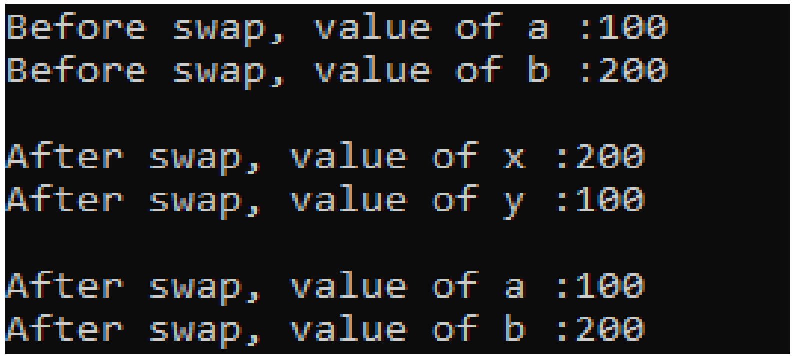
```c
void change(int);

void main()
{
    int x;
    x=20;
    change(x);  //call by value
    printf("%d\n",x);
}

void change(int p)
{
    p=p+10;
}
```

```c
void change(int *);

void main()
{
    int x;
    x=20;
    change(&x);  //call by address
    printf("%d\n",x);
}

void change(int *p)
{
    *p=*p+10;
}
```

```
20
```

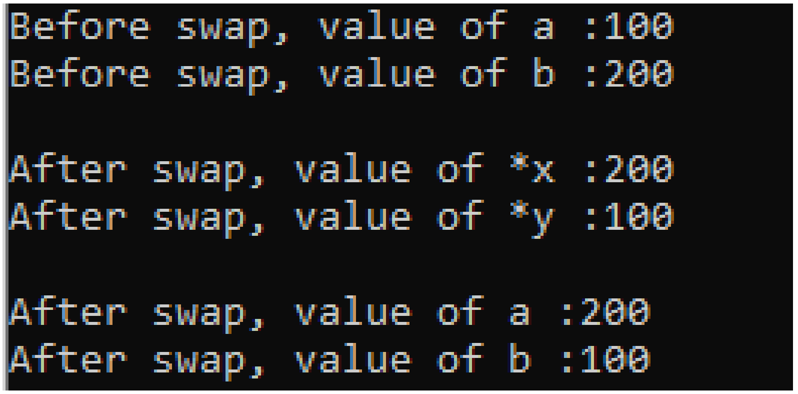
```
30
```

```c
//WAP to exchange values of two variables using call by value
void swap(int x,int y);

void main()
{
     int a = 100;
     int b = 200;

     printf("Before swap, value of a :%d\n",a );
     printf("Before swap, value of b :%d\n", b );

     swap(a,b); //call by value

     printf("\nAfter swap, value of a :%d\n",a );
     printf("After swap, value of b :%d\n", b );
}

void swap(int x, int y)
{
     int temp;
     temp =x;
     x = y;
     y = temp;

     printf("\nAfter swap, value of x :%d\n",x );
     printf("After swap, value of y :%d\n", y );
}
```



```
Before swap, value of a :100
Before swap, value of b :200

After swap, value of x :200
After swap, value of y :100

After swap, value of a :100
After swap, value of b :200
```

```
//Write a function using pointers  to exchange the values stored in two locations
void swap(int *x,int *y);

void main()
{
    int a = 100;
    int b = 200;

    printf("Before swap, value of a :%d\n",a );
    printf("Before swap, value of b :%d\n",b );

    swap(&a,&b); //call by address

    printf("\nAfter swap, value of a :%d\n",a );
    printf("After swap, value of b :%d\n", b );
}

void swap(int *x, int *y)
{
    int temp;
    temp =*x;
    *x = *y;
    *y = temp;

    printf("\nAfter swap, value of *x :%d\n",*x );
    printf("After swap, value of *y :%d\n", *y );
}
```



```
Before swap, value of a :100
Before swap, value of b :200

After swap, value of *x :200
After swap, value of *y :100

After swap, value of a :200
After swap, value of b :100
```

Pointer as function Arguments

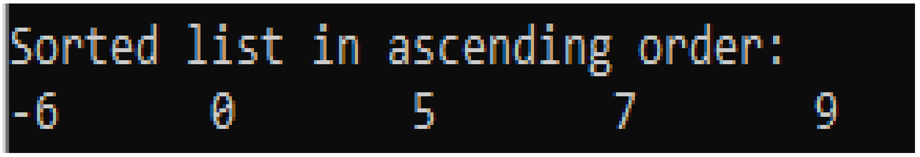
NOTE:

1. The function parameters are declared as pointers.
2. The dereferenced pointers are used in the function body.
3. When the function is called, the addresses are passed as actual arguments.

Pointer as function Arguments

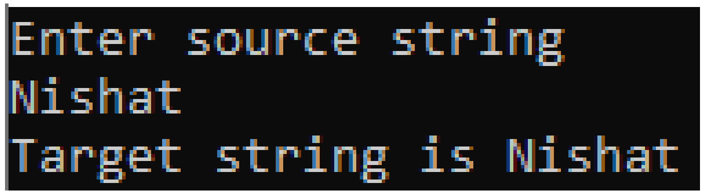
- When an array is passed to a function as an argument, only the **address of the first element** of the array is passed, but not the actual values of the array elements.
- If x is an array, when we call **sort(x)**, the address of **x[0] is passed** to the function sort.
- In function declaration/definition, function sort can be written **using pointers** instead of array indexing.

```
//WAP to sort an array of n elements using pointer
//Hint: Make a function to sort an array
void sort(int m,int *x);

void main()
{
	int c[5]={9,-6,5,0,7};

	sort(5,c);
}

void sort(int m,int *x)
{
	int i,j,swap;
	for(i=0;i<m-1;i++)
	{
		for(j=0;j<m-1-i;j++)
		{
			if(*(x+j) > *(x+j+1))
			{
				swap = *(x+j);
				*(x+j) = *(x+j+1);
				*(x+j+1) = swap;
			}
		}
	}
	printf("\nSorted list in ascending order:\n");
	for(i=0;i<m;i++)
	{
		printf("%d\t",*(x+i));
	}
}
```



```
Sorted list in ascending order:
-6      0       5       7       9
```

```c
//WAP to Copy one string into another using pointers
void copy_string(char*, char*);

void main()
{
    char source[100], target[100];
    printf("Enter source string\n");
    gets(source);

    copy_string(target, source);

    printf("Target string is %s", target);
}

void copy_string(char *target, char *source)
{
    while(*source!='\0')
    {
        *target = *source;
        source++;
        target++;
    }
    *target = '\0';
}
```



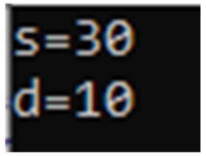
```
Enter source string
Nishat
Target string is Nishat
```

Function returning Multiple Values

- Return Statement can **Return only one value**.
- We can Return multiple values by using the argument not only to receive information but also to send back information to the calling function.
- The arguments that are used to send out information are called **output arguments** or parameters.
- Here x and y are input arguments and sum and diff are output argument.

Function returning Multiple Values

```c
#include<stdio.h>
int calc(int x,int y,int *s,int *d);

void main()
{
    int x=20,y=10,s,d;
    calc(x,y,&s,&d);
    printf("s=%d\n d=%d\n",s,d);
}

int calc(int a,int b,int *sum,int *diff)
{
    *sum=a+b;
    *diff=a-b;
}
```



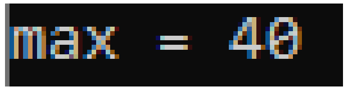
```
s=30
d=10
```

Function returning Pointers

- Function can return a **single value** by its name
- Function can return **multiple values** through pointer parameter
- Since pointers are a data type in C, function can also return a **pointer** to the calling function.

Function returning Pointers

```
#include<stdio.h>
int * larger(int*,int*);

void main()
{
      int a=40, b=20;
      int *p;

      p=larger(&a,&b);

      printf("max = %d",*p);
}

int* larger(int *x,int*y)
{
      if(*x>*y)
      {
            return x;   //address of a
      }

      else
      {
            return y;   //address of b
      }
}
```

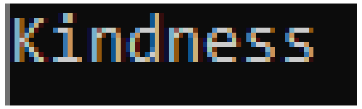


```
max = 40
```

Function returning Pointers : Practical 11.3.2

Write output for the following programs:

```c
//Functions Returning Pointers
char *copy (char*,char *);

void main()
{
      char *str;
      char source[] = "Kindness";
      char target[10];
      str=copy(target,source);
      printf("%s\n",str);
}

char *copy(char *t,char *s)
{
      char * r;
      r = t;
      while(*s!='\0')
      {
            *t=*s;
            t++;
            s++;
      }
      *t='\0';
      return(r);
}
```



```
Kindness
```

Pointers to Functions

- A function, like a variable, has a type and an address location in the memory.
- It is therefore, possible to declare a **pointer to a function**, which can then be used as an argument in another function.
- A pointer to a function is declared as follows:

```
type (*fptr) ();
```

- This tells the compiler that fptr is a pointer to a function, which returns type value.

Pointers to Functions

NOTE:

- The parenthesis around *fptr are necessary.

Remember,

```
type *gptr();
```

This would declare gptr as a function returning a pointer to type.

Pointers to Functions

We can make a function pointer(**p1**) to point to a specific function(**mul**) by simply assigning the name of the function to the pointer.

```
double mul(int,int);
double (*p1)();
p1=mul;
```

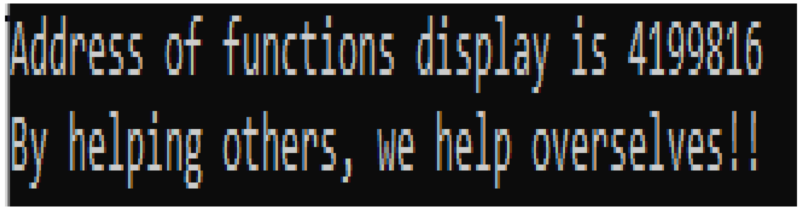
Function call:

```
(*p1)(x,y);  //equivalent to mul(x,y)
```

Pointers to Functions : Practical 11.3.1

Write output for the following programs:

```c
//Pointers to Functions
#include<stdio.h>
void display();

void main()
{
    void (*func_ptr)();
    func_ptr=display;
    printf("Address of functions display is %u\n",func_ptr);
    (*func_ptr)();  //equivalent to display()
}

void display()
{
    puts("By helping others, we help overselves!!");
}
```



```
Address of functions display is 4199816
By helping others, we help overselves!!
```

Pointers to Structures

- 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.**

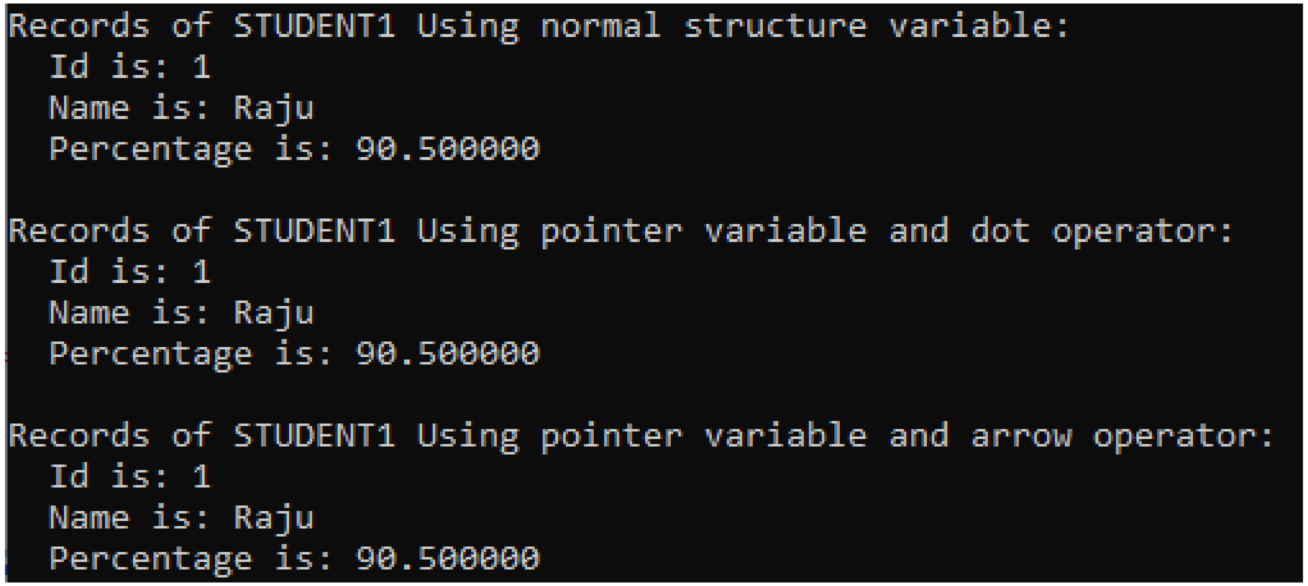
C structure can be accessed in 2 ways:

1. Using normal structure variable
   - **Dot(.) operator is used to access the data**
2. Using pointer variable
   - Using indirection(*****) operator and dot(**.**) operator
   - **Using arrow (->) operator or membership operator**

Pointers to Structures

```c
struct student
{
    int id;
    char name[30];
    float percentage;
};

void main()
{
    struct student record1 = {1, "Raju", 90.5};
    struct student *ptr;
    ptr = &record1;

    printf("Records of STUDENT1 Using normal structure variable: \n");
    printf("  Id is: %d \n", record1.id);
    printf("  Name is: %s \n", record1.name);
    printf("  Percentage is: %f \n\n", record1.percentage);

    printf("Records of STUDENT1 Using pointer variable and dot operator: \n");
    printf("  Id is: %d \n", (*ptr).id);
    printf("  Name is: %s \n", (*ptr).name);
    printf("  Percentage is: %f \n\n", (*ptr).percentage);

    printf("Records of STUDENT1 Using pointer variable and arrow operator: \n");
    printf("  Id is: %d \n", ptr->id);
    printf("  Name is: %s \n", ptr->name);
    printf("  Percentage is: %f \n\n", ptr->percentage);
}
```

Pointers to Structures



```
Records of STUDENT1 Using normal structure variable:
  Id is: 1
  Name is: Raju
  Percentage is: 90.500000

Records of STUDENT1 Using pointer variable and dot operator:
  Id is: 1
  Name is: Raju
  Percentage is: 90.500000

Records of STUDENT1 Using pointer variable and arrow operator:
  Id is: 1
  Name is: Raju
  Percentage is: 90.500000
```

Pointers and Array of Structures

- Recall that the name of an array is the address of its 0-th element
  - Also true for the names of arrays of structure variables.

```
struct inventory
{
    char name[30];
    int number;
    float price;
}product[3],*ptr;
```

Here,

- **product** is an array of two elements of type struct inventory
- The name product represents the address of its zeroth element.
- **ptr** as a pointer to data objects of the type struct inventory

Pointers and Array of Structures

➢Pointer ptr will point to product[0] as below

```
ptr=product;
```

➢Its members can be accessed as below

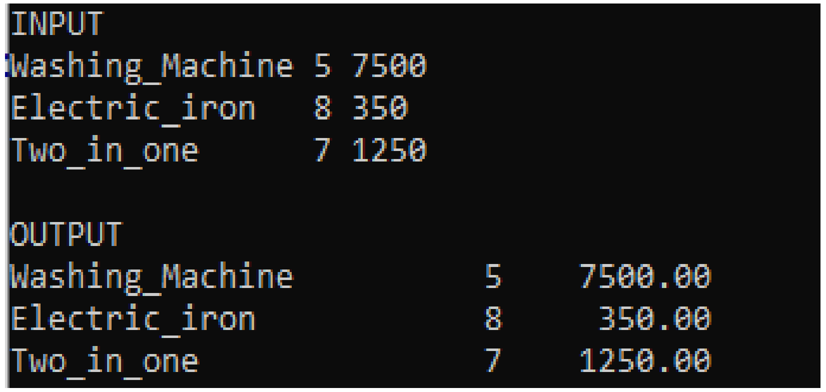
```
ptr->name
ptr->number
ptr->price
```

NOTE:

➢ When the pointer ptr is incremented by one, it is made to point to the next record, i.e., product[1].

Pointers and Array of Structures

```
struct invent
{
    char name[20];
    int number;
    float price;
};

void main()
{
    struct invent product[3], *ptr;

    printf("INPUT\n");
    for(ptr = product; ptr < product+3; ptr++)
        scanf("%s %d %f", ptr->name,&ptr->number,&ptr->price);

    printf("\nOUTPUT\n");
    for(ptr = product; ptr < product+3; ptr++)
        printf("%-20s %5d %10.2f\n", ptr->name,ptr->number,ptr->price);
}
```



```
INPUT
Washing_Machine 5 7500
Electric_iron   8 350
Two_in_one      7 1250

OUTPUT
Washing_Machine          5    7500.00
Electric_iron            8     350.00
Two_in_one               7    1250.00
```

Pointers and Array of Structures

NOTE: When using structure pointers, be careful of operator precedence

- **Member operator "." has higher precedence than "*"**
  - ptr->roll and (*ptr).roll mean the same thing
  - *ptr.roll will lead to error
- **The operator "->" enjoys the highest priority among operators**
  - ++ptr->roll increments roll
  - (++ptr)->roll increments ptr first and then links roll
  - ptr++ -> roll increments ptr after accessing roll

Let's Practice

Create a structure engine_parts having serial number starting from AA0 to FF9, year of manufacturing and quantity manufactured as data members. Write a program which takes input of 3 engine parts using pointer to structure and display information using pointer to structure. (Pointers & Structures).

Structure as function arguments Using Pointer

Passing Structure To Function can be done in below 3 ways:

1. Passing Structure Members as arguments to Function
2. Passing structure to a function **by value**
   (Passing Structure Variable as Argument to a Function)
3. Passing structure to a function **by address**
   (Passing Structure Pointers as Argument to a Function)

REFER UNIT-10

void pointers

- A void pointer is a pointer that has **no associated data type with it.**
- A void pointer can hold address of any type and can be type casted to any type.

```
int a=10;
char b='x';
void *p;
p=&a;  //void pointer holds address of int a
p=&b;  //void pointer holds address of char b
```

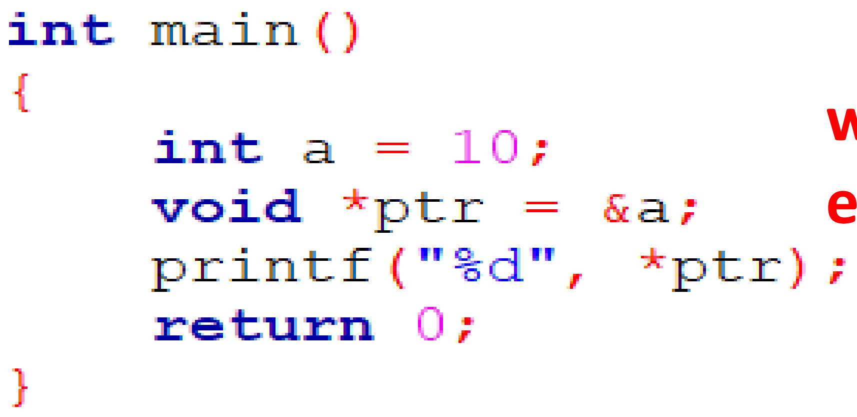
- Also known as generic pointer(pointer which can point to any type of data)

Syntax:

```
void *ptr;
```

Interesting facts about void pointers

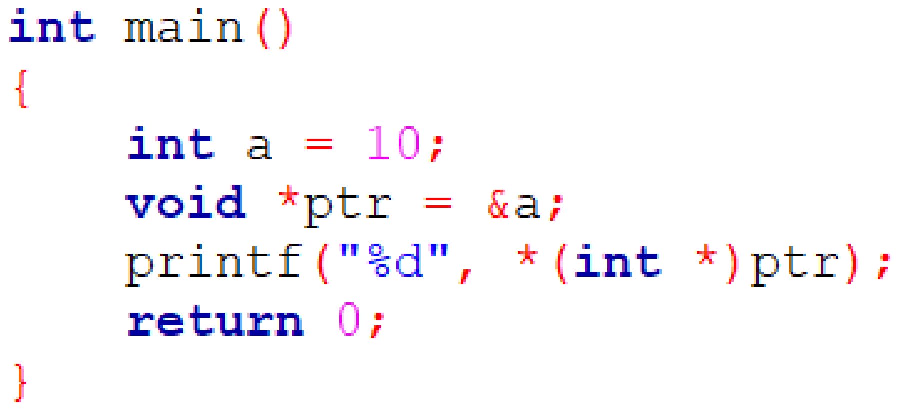
void pointers cannot be dereferenced without



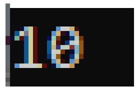
```c
int main()
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *ptr);
    return 0;
}
```

warning: dereferencing 'void *' pointer

error: invalid use of void expression



```c
int main()
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *(int *)ptr);
    return 0;
}
```



```
10
```

Troubles with Pointers

- Pointers give us tremendous **power** and **flexibility**.
- However, they could become a nightmare when they are not used correctly.
- The major problem with wrong use of pointers is that the **compiler may not detect the error** in most cases and therefore the program is likely to **produce unexpected results.**
- The output may not given us any clue regarding the use of a bad pointer.
- **Debugging** therefore becomes a difficult task.

Troubles with Pointers

Below are some pointer errors that are more commonly committed:

- Assigning values to uninitialized pointers

```
int *p , m =100;
*p=m; /*Error */
```

- Assigning values to pointer variables

```
int *p , m =100;
p=m; /*Error */
```

- Not dereferencing a pointer when required.

```
int *p , m =100;
p=&m;
printf("%d",p); /*Error */
```

Troubles with Pointers

- Assigning the address of an uninitialized variable

```
int *p , m;
p=&m; /*Error */
```

- Comparing pointers that point to different objects.

```
char name1[20] ;
char name2[20] ;
char *p1= name1;   char *p2= name2;
if(p1>p2)…… /*Error */
```

Troubles with Pointers

NOTE:

- We must be careful in declaring and assigning values to **pointers correctly** before using them.
- We must also make sure that we apply the **address operator & and referencing operator *** correctly to the pointers
- That will save us from sleepless nights.

End of Unit-11