Computer Programming: C

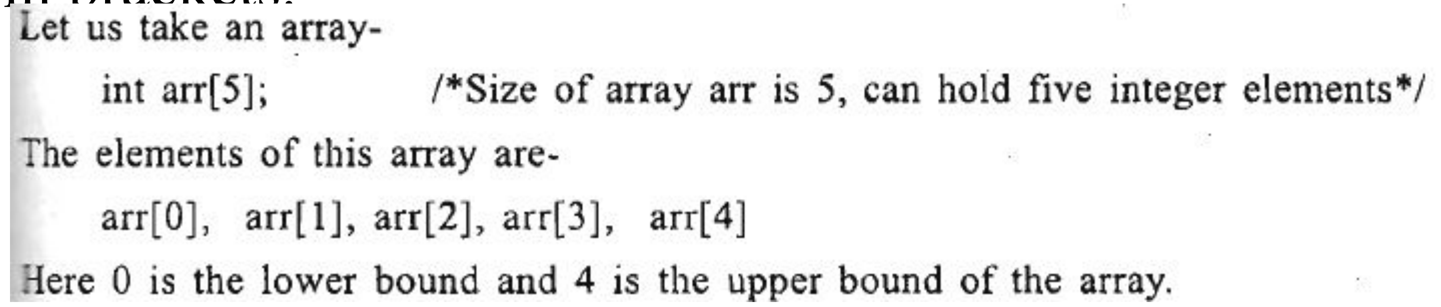
By: Ritu Devi

Array

- An *array* is a collection of similar type of data items and each data item is called an element of the array.
- The elements of array share the same variable name but each element has a different index number known as ***subscript***.
- Before using an array its type and dimension must be declared.
- Arrays can be *single dimensional* or *multidimensional*. The number of subscripts determines the dimension of array.

One Dimensional Array

- One-dimensional arrays are known as vectors and it has one subscript.
- **Declaration of an array :** Syntax is
  - ***Data_type array_name[size];***
  - Size is number of element that can be stored in array. It may be a positive integer constant or constant integer expression or symbolic constant.
- **Accessing 1-D Array Elements:** The elements of an array can be accessed by specifying the array name followed by subscript in brackets.


Let us take an array-

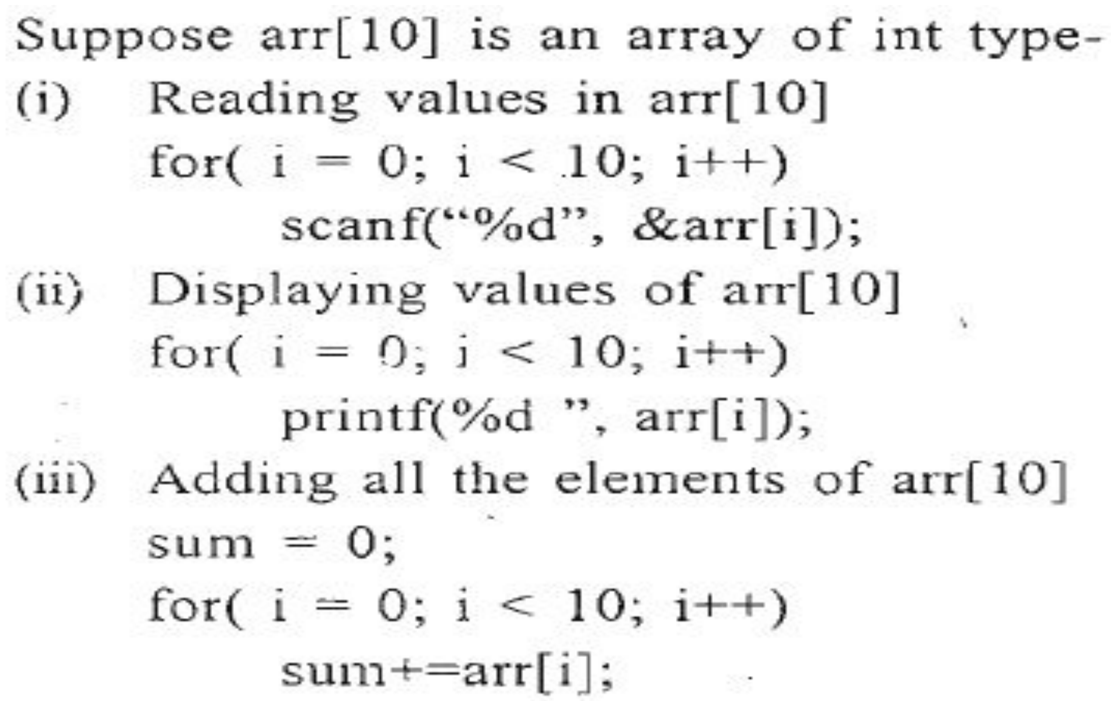
```
    int arr[5];        /*Size of array arr is 5, can hold five integer elements*/
```

The elements of this array are-

```
    arr[0],  arr[1], arr[2], arr[3],  arr[4]
```

Here 0 is the lower bound and 4 is the upper bound of the array.

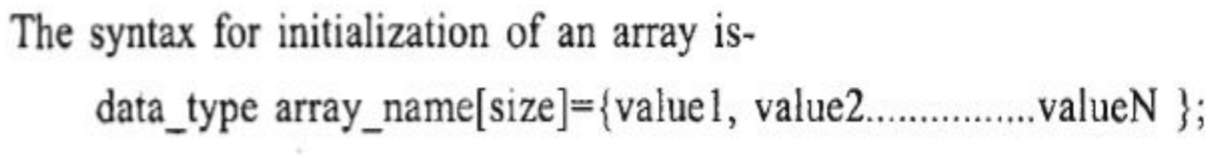
- The subscript can be any expression that yields an integer value. It can be any integer constant, integer variable, integer expression or return value(int) from a function call.
- **Processing 1-D Arrays:**



Suppose arr[10] is an array of int type-

```
(i)    Reading values in arr[10]
       for( i = 0; i < 10; i++)
               scanf(“%d”, &arr[i]);
(ii)   Displaying values of arr[10]
       for( i = 0; i < 10; i++)
               printf(%d ”, arr[i]);
(iii)  Adding all the elements of arr[10]
       sum = 0;
       for( i = 0; i < 10; i++)
               sum+=arr[i];
```

- **Initialization of 1-D Array:**



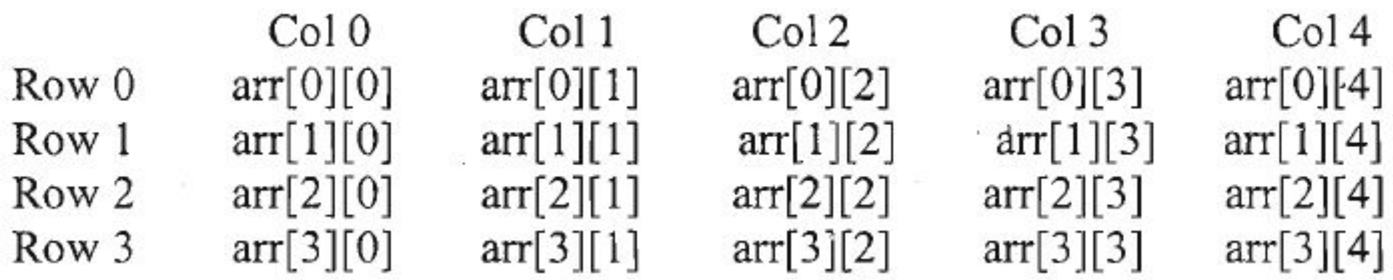
The syntax for initialization of an array is-

```
data_type array_name[size]={value1, value2................valueN };
```

- For example
  - int marks[5] = {50, 85, 70, 65, 95};
  - int marks[] = { 99, 78, 50,45, 67, 89};
  - int marks[5] = { 99, 78};

Two Dimensional Array

- Two-dimensional arrays are known as matrix and it has two subscript.
- **Declaration :** Syntax is
  - ***data_type array_name[rowsize][columnsize];***
  - The total number of elements in the array are rowsize * columnsize.
- For Example: int arr[4][5];



	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]	arr[0][4]
Row 1	arr[1][0]	arr[1][1]	arr[1][2]	arr[1][3]	arr[1][4]
Row 2	arr[2][0]	arr[2][1]	arr[2][2]	arr[2][3]	arr[2][4]
Row 3	arr[3][0]	arr[3][1]	arr[3][2]	arr[3][3]	arr[3][4]

Processing 2-D Arrays:

For processing 2-D arrays, we use two nested for loops. The outer for loop corresponds to the row and the inner for loop corresponds to the column.

```
int arr[4][5];
```

(i) Reading values in arr

```
for( i = 0;  i < 4; i++ )
    for( j = 0; j< 5; j++ )
            scanf("%d', &arr[i][j]);
```

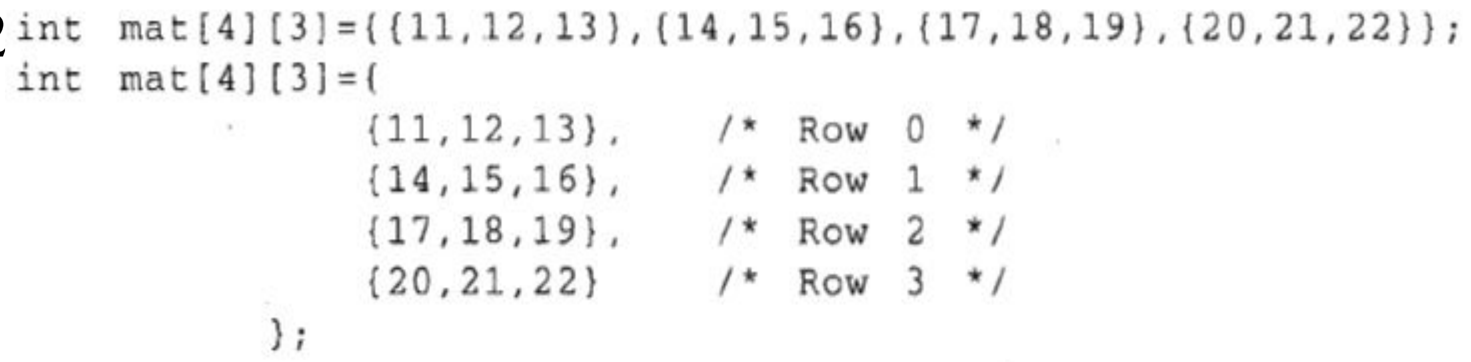
(ii) Displaying values of arr

```
for( i = 0;  i < 4; i++ )
    for( j = 0; j< 5; j++ )
            printf( "%d  ", arr[i][j]);
```

- **Initialization of 2-D Arrays:**

  int mat[4][3] = { 11, 12, 13, 14, 15, 16, p, 18, 19, 20, 21,

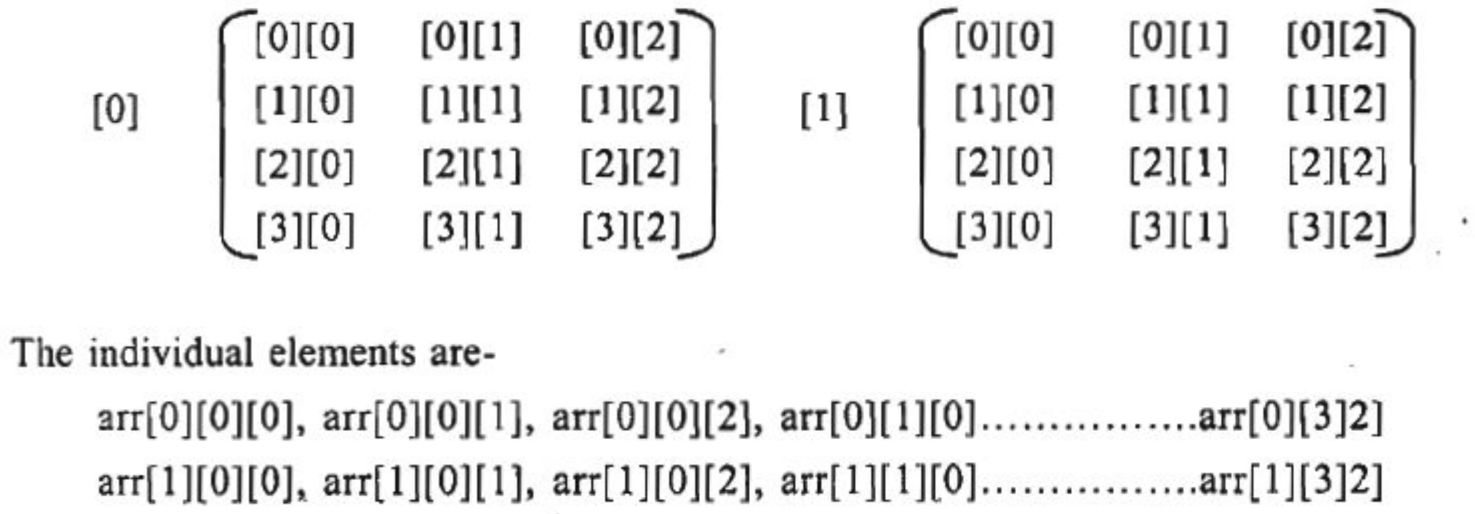
2



```
int  mat[4][3]={{11,12,13},{14,15,16},{17,18,19},{20,21,22}};
int  mat[4][3]={
                {11,12,13},     /*  Row  0  */
                {14,15,16},     /*  Row  1  */
                {17,18,19},     /*  Row  2  */
                {20,21,22}      /*  Row  3  */
            };
```

Arrays With More Than Two Dimensions

- We can think of a 3-d array as an array of 2-D arrays. For example:
  - int arr[2][4][3] ;



$$[0]\begin{pmatrix}[0][0] & [0][1] & [0][2] \\ [1][0] & [1][1] & [1][2] \\ [2][0] & [2][1] & [2][2] \\ [3][0] & [3][1] & [3][2]\end{pmatrix} \qquad [1]\begin{pmatrix}[0][0] & [0][1] & [0][2] \\ [1][0] & [1][1] & [1][2] \\ [2][0] & [2][1] & [2][2] \\ [3][0] & [3][1] & [3][2]\end{pmatrix}$$

The individual elements are-

arr[0][0][0], arr[0][0][1], arr[0][0][2], arr[0][1][0]...............arr[0][3]2]

arr[1][0][0], arr[1][0][1], arr[1][0][2], arr[1][1][0]...............arr[1][3]2]

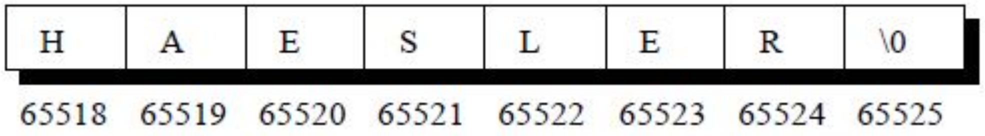
- Total number of elements in the above array are: 2*4*3=24.

- In C there is no check on bounds of the array. So it is the responsibility of programmer to provide array bounds checking wherever needed.

- Usually, an array of characters is called a '**string**', whereas an array of ints or floats is called simply an array.

String

- String in C is an array of characters that ends with a null character (‘\0’).
- This null character is an escape sequence with ASCII value 0.
- Strings are generally used to store and manipulate data in text form like words or sentences.
- A string constant is a one-dimensional array of characters .
- For example,
  - char name[ ] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' } ;
  - Memory representation



- **Declaration of strings**:
  - char str_name[size];
- **Initializing a String**:

  These are the two forms of initialization of a string variable
  - char str[10] = {'I', 'n', 'd', 'i', 'a', '\0' };
  - char str[10] = "India"; /*Here the null character is automatically placed at the end by compiler*/

Input and output of strings

```
#include<stdio.h>
main( )
{
    char  str[10]=“Anpara”;
    printf(“String  is  :  %s\n”,str);
    printf(“Enter  new  value  for  string  :  ” );
    scanf(“%s”,str);
    printf(“String  is  :  %s\n”,str);
}
```

Output:

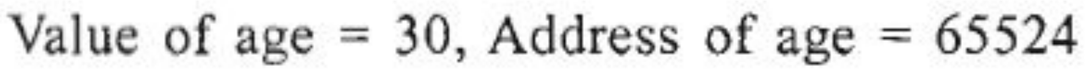
```
String is : Anpara
Enter new value for string : Bareilly
String is : Bareilly
```

gets( ) and puts( ) for the input and output of strings

```
// C program to read strings

#include<stdio.h>

    int main()
    {
      char str[50]; // declaring string

      gets(str); // reading string

      puts(tr); // print string

      return 0;
    }
```

Pointers

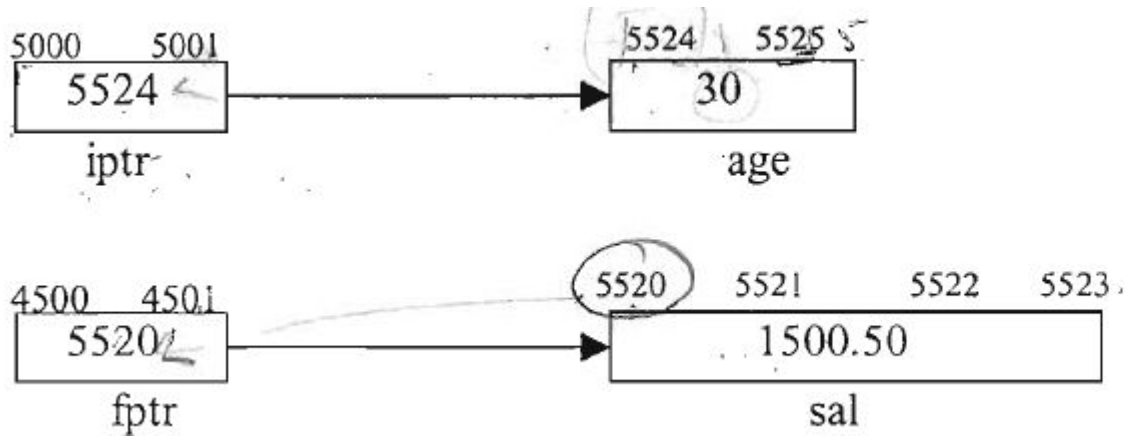
- Suppose we have declared a variable :
  - int age;
    - The compiler reserves 2 consecutive bytes from memory for this variable and associates the name age with it. The address of first byte from the two allocated bytes is known as the address of variable age.
- If we assign some value:
  - age=30;
    - Now this value will be stored in these 2 bytes (in binary form).
- C provides an address operator '&', which returns the address of a variable when placed before it.
  - printf ("Value of age = %d, Address of age,= %u\n",age,&age);

- Output is: 

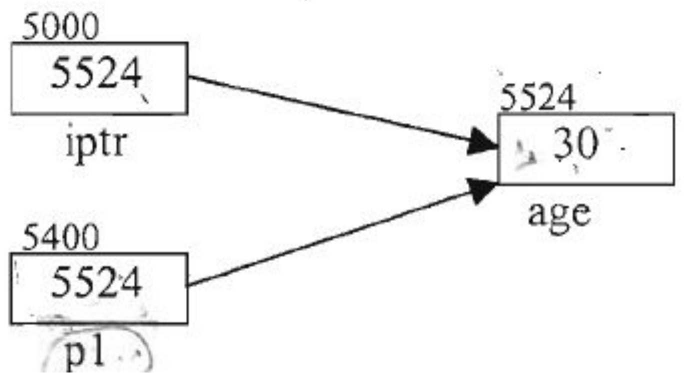
```
Value of age = 30, Address of age = 65524
```

- Addresses are just whole numbers. These addresses may be different each time you run your program, it depends on which part of memory is allocated by operating system for this program.
- The address operator cannot be used with a constant or an expression.
- It is used in ***scanf()*** function to know where to write the input value

- A pointer is a variable that stores memory address.
- The general syntax of declaration is
  - ***Data_type *ptr_name;***
  - Some Examples are:
    - int *iptr; /* iptr is a pointer that should point to variables of type int */
    - float *fptr;
    - char *cptr, chI, ch2;
- Type of variable iptr is 'pointer to int' or (int *).
- Pointers may be assigned the addres of a variable
    - iptr = &age;
    - Fptr=&sal;



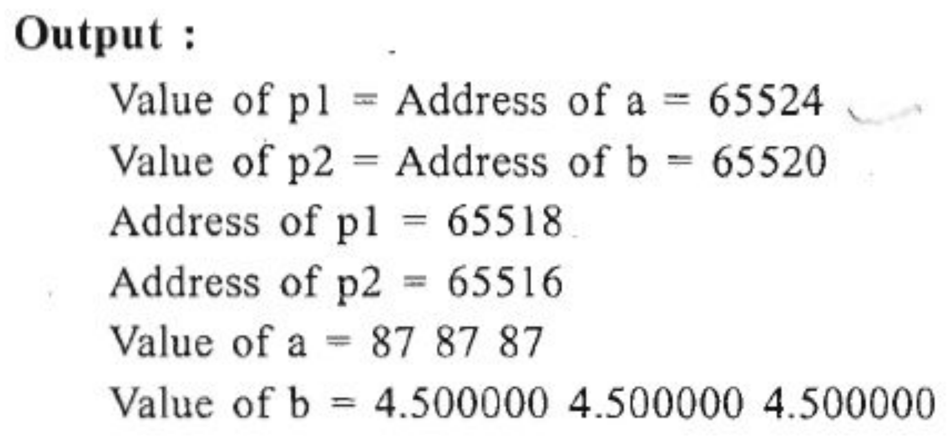
- We can assign the value of one pointer variable to other but their base type should be same.
  - int *p1;
  - p1=iptr;



- We can assign a pointer as null
  - iptr=NULL; //A symbolic constant NULL is defined in stdio.h
- We can also access a variable indirectly using pointer it is also called as indirection operator(*).
- if we place ’*’ before iptr then we can access the variable whose address is stored in age.
  - printf(“%d %f”, *iptr, *fptr); is equivalent to printf(“%d %f”, age, sal);
  - scanf(“%d %f”, iptr, fptr); is equivalent to printf(“%d %f”, &age, &sal);

Dereferencing of pointers

```
#include<stdio.h>
main()
{
    int  a=87;
    float  b=4.5;
    int  *p1=&a;
    float  *p2=&b;
    printf("Value of p1 = Address of a = %u\n",p1);
    printf("Value of p2 = Address of b = %u\n",p2);
    printf("Address of p1 = %u\n",&p1);
    printf("Address of p2 = %u\n",&p2);
    printf("Value of a = %d %d %d \n",a,*p1,*(&a));
    printf("Value of b = %f %f %f \n",b,*p2,*(&b));
}
```

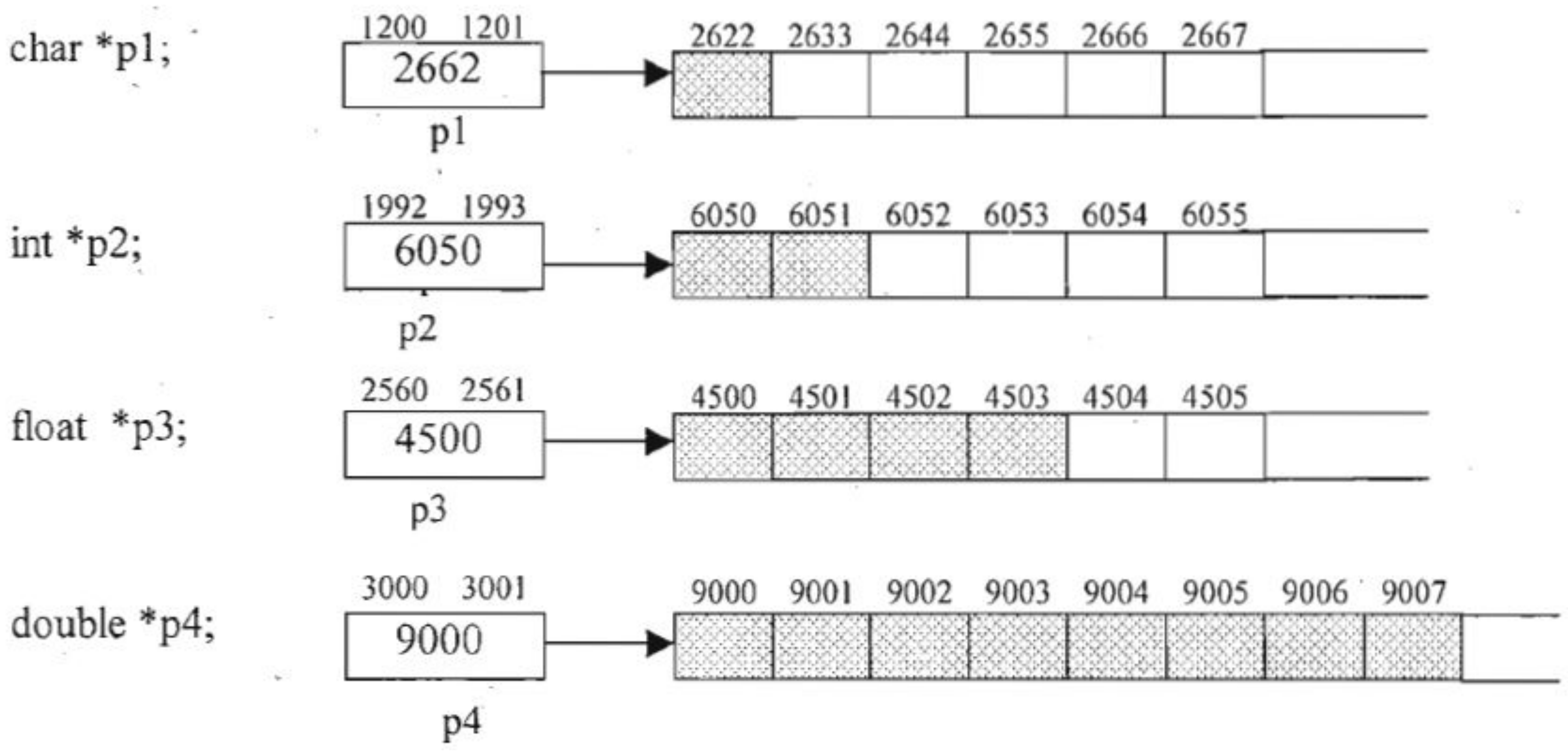


Output :

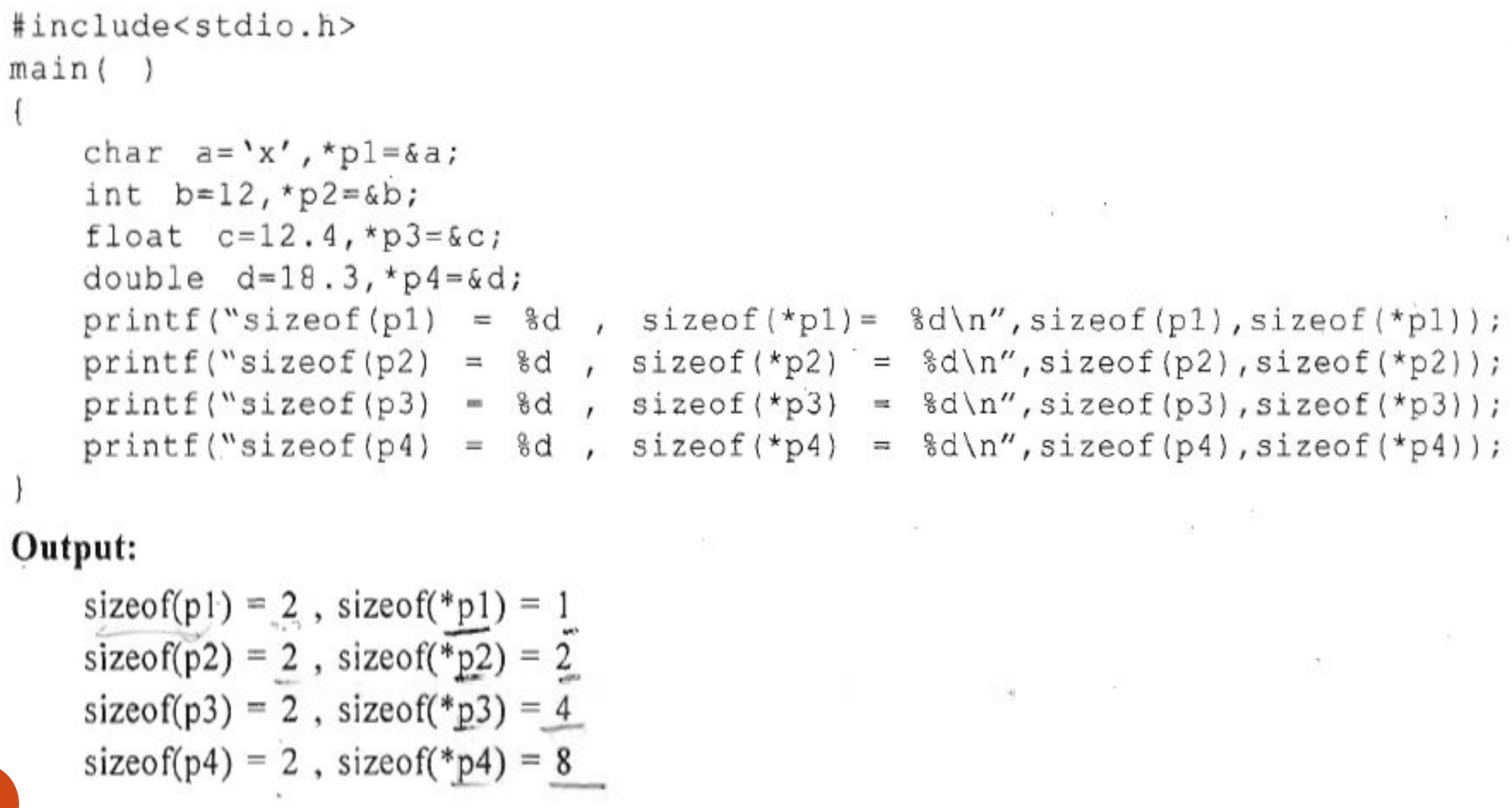
```
    Value of p1 = Address of a = 65524
    Value of p2 = Address of b = 65520
    Address of p1 = 65518
    Address of p2 = 65516
    Value of a = 87 87 87
    Value of b = 4.500000 4.500000 4.500000
```

- The value of the pointer only tells the address of starting byte.
- So the compiler will look at the base type of the pointer and will retrieve the information depending on that base type.

- The size of pointer variable is same for all type of pointers but the memory that will be accessed while dereferencing is different.



- Program to print the size of pointer variable and size of value dereferenced by that pointer.



```
#include<stdio.h>
main( )
{
    char  a='x',*p1=&a;
    int  b=12,*p2=&b;
    float  c=12.4,*p3=&c;
    double  d=18.3,*p4=&d;
    printf("sizeof(p1)  =  %d  ,  sizeof(*p1)=  %d\n",sizeof(p1),sizeof(*p1));
    printf("sizeof(p2)  =  %d  ,  sizeof(*p2)  =  %d\n",sizeof(p2),sizeof(*p2));
    printf("sizeof(p3)  =  %d  ,  sizeof(*p3)  =  %d\n",sizeof(p3),sizeof(*p3));
    printf("sizeof(p4)  =  %d  ,  sizeof(*p4)  =  %d\n",sizeof(p4),sizeof(*p4));
}
```

Output:

```
    sizeof(p1) = 2 , sizeof(*p1) = 1
    sizeof(p2) = 2 , sizeof(*p2) = 2
    sizeof(p3) = 2 , sizeof(*p3) = 4
    sizeof(p4) = 2 , sizeof(*p4) = 8
```

Pointer Arithmetic

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

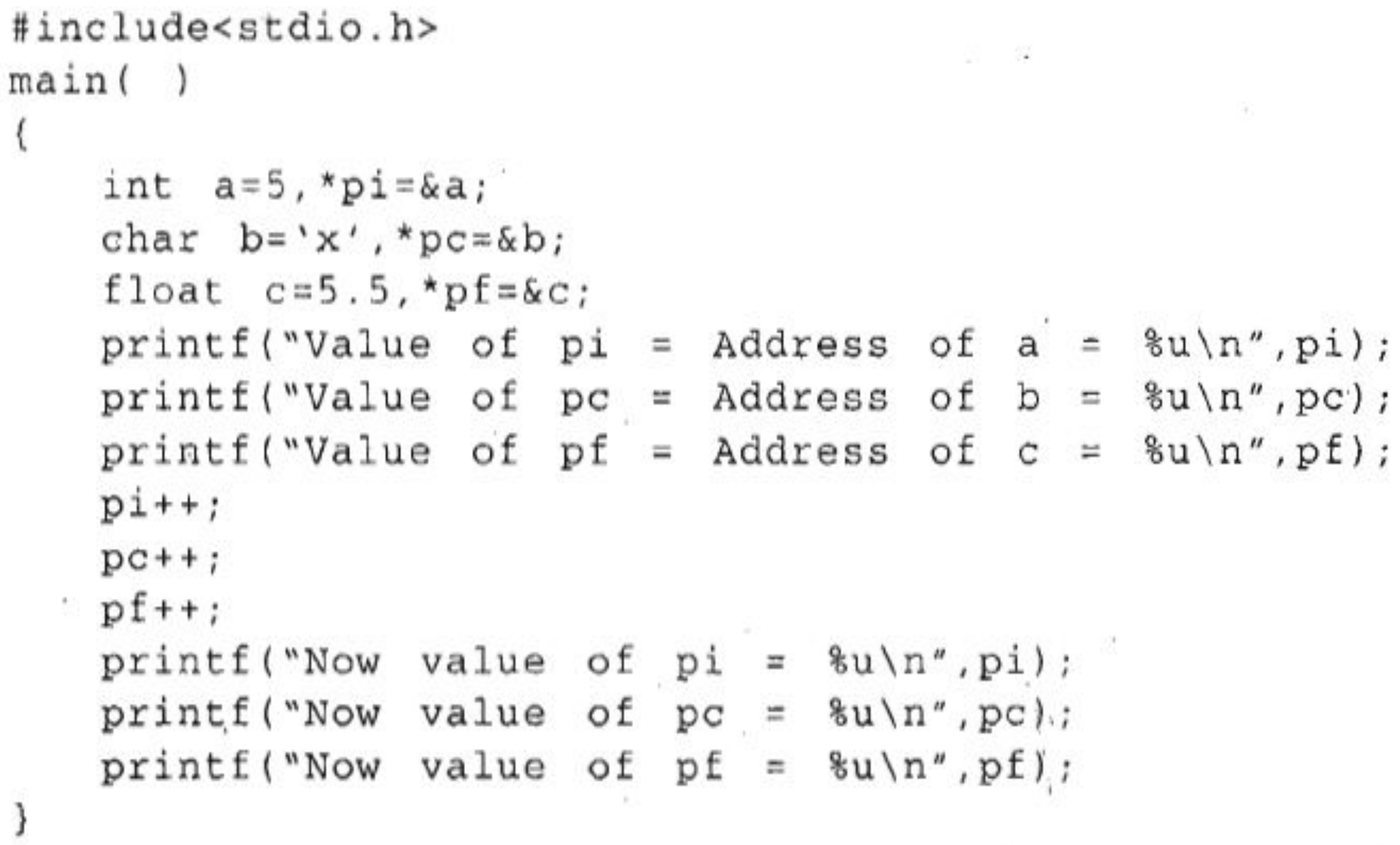
  (I) Addition of an integer to a pointer and increment operation.

  (2) Subtraction of an integer from a pointer and decrement operation

  (3) Subtraction of a pointer from another pointer of same type.

- Here all arithmetic is performed relative to the size of base type of pointer

Program to show pointer arithmetic



```
#include<stdio.h>
main( )
{
    int  a=5,*pi=&a;
    char  b=‘x’,*pc=&b;
    float  c=5.5,*pf=&c;
    printf(“Value of pi = Address of a = %u\n”,pi);
    printf(“Value of pc = Address of b = %u\n”,pc);
    printf(“Value of pf = Address of c = %u\n”,pf);
    pi++;
    pc++;
    pf++;
    printf(“Now value of pi = %u\n”,pi);
    printf(“Now value of pc = %u\n”,pc);
    printf(“Now value of pf = %u\n”,pf);
}
```

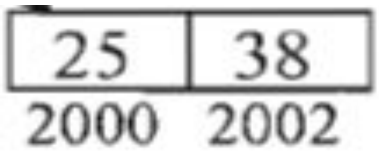
- **Output:**

```
Value of pi = Address of a = 1000
Value of pc =Address of b = 4000
Value of pf = Address of c = 8000
Now value of pi = 1002
Now value of pc = 4001
Now value of pf = 8004
```

- 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.

Precedence Of Dereferencing Operator(‘*’) And ‘++’or ‘—’ Operators

- Precedence level of * and increment/decrement operators, is same and their associativity is from right to left.
- Suppose
  - int a=25,x;
  - int *ptr=&a;
- Now we'll see how the following pointer expressions are interpreted.
  1) x =*ptr++; //is equivalent to *(ptr++);
  2) x=*++ptr; //is equivalent to *(++ptr);
  3) x=++*ptr; //is equivalent to ++(*ptr);
  4) x=(*ptr)++;



1) x=*ptr;
   ptr=ptr+1;

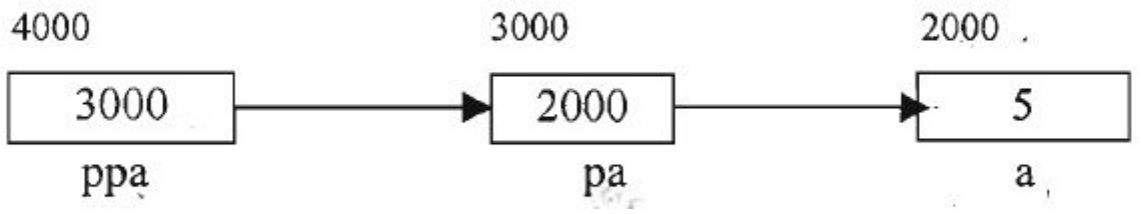
2) ptr=ptr+1;
   x=*ptr;

3) *ptr=*ptr+1;
   x=*ptr;

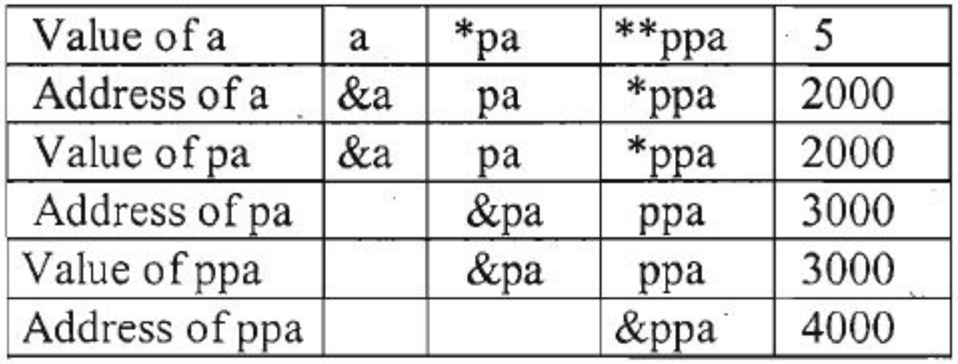
4) x=*ptr;
   *ptr=*ptr+1;

Pointer To Pointer

- We can store the address of a pointer variable in some other variable, which is known as a pointer to pointer variable.
- Similarly we can have a pointer to pointer to pointer variable and this concept can be extended to any limit, but in practice only pointer to pointer is used.
- It is generally used while passing pointer variables to functions.
- Syntax is:
  - ***data_type **pptr;***
- For example
  - int a = 5; // type of variable a is int
  - int *pa = &a; // type of variable pa is (int*) or pointer to int
  - int **ppa = &pa;// type of variable ppa is (int**) or pointer to pointer to int



- To access the value indirectly pointed to by a pointer to pointer, we can use double indirection operator. The table given below will make this concept clear.



Value of a	a	*pa	**ppa	5
Address of a	&a	pa	*ppa	2000
Value of pa	&a	pa	*ppa	2000
Address of pa		&pa	ppa	3000
Value of ppa		&pa	ppa	3000
Address of ppa			&ppa	4000

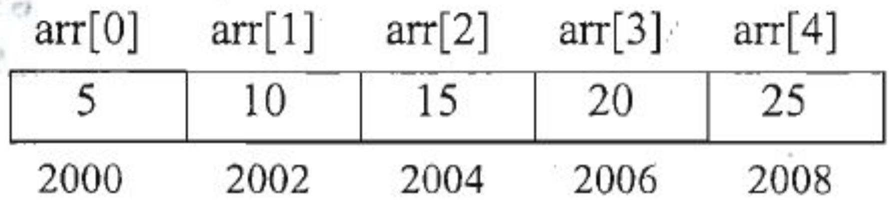
Pointers and One Dimensional Arrays

- The elements of an array are stored in contiguous memory locations.
- The address of first element of the array is also known as the base address of the array.
- In C language, pointers and arrays are closely related. We can access the array elements using pointer expressions.
- Actually the compiler also access the array elements by converting *subscript notation* to *pointer notation*.
- The name of an array is a **constant pointer** that points to the first element of the array

For example-

```
    int arr[5] = { 5 , 10 , 15 , 20, 25 }
```

Here arr[5] is an array that has 5 elements each of type int.



- The name of the array 'arr' denotes the address of 0th element of array which is 2000.
- The address of 0th element can also be given by &arr[0], so arr and &arr[0] represent the same address.
- (arr+1) will denote the address of the next element arr[1].

- The pointer expression (arr+i) denotes the same address as &arr[i].
- If we dereference arr, *arr or *(arr+i) represents ith element of array.
- So in subscript notation the address of an array element is &arr[i] and its value is arr[i], while in pointer notation the address is (arr+i) and the element is *(arr+i).
- Accessing array elements by pointer notation is faster than accessing them by subscript notation.
- Array subscripting is commutative, i.e. arr[i] is same as i[arr].

```
main ( )
{
   int arr[5]={5,lO,15,20,25};
   int i
   for(i=O;i<5;i++)
   {
     printf ("Value of arr [%d] = %d\t",i,*(arr+i));
     printf("Address of arr[%d]= %u\n", i,arr+i);
   }
}
```

Subscripting Pointer Variables

- Suppose

  ```
  int *ptr;
  ptr=arr;
  ```

- On applying pointer arithmetic and dereferencing we can see that the expression (ptr+i) denotes the address of ith element of array and the expression *(ptr+i) denotes the value of ith element of array.
- The name of array is constant pointer. It will always point to the address of 0th element of the array. We cannot assign another address to it.

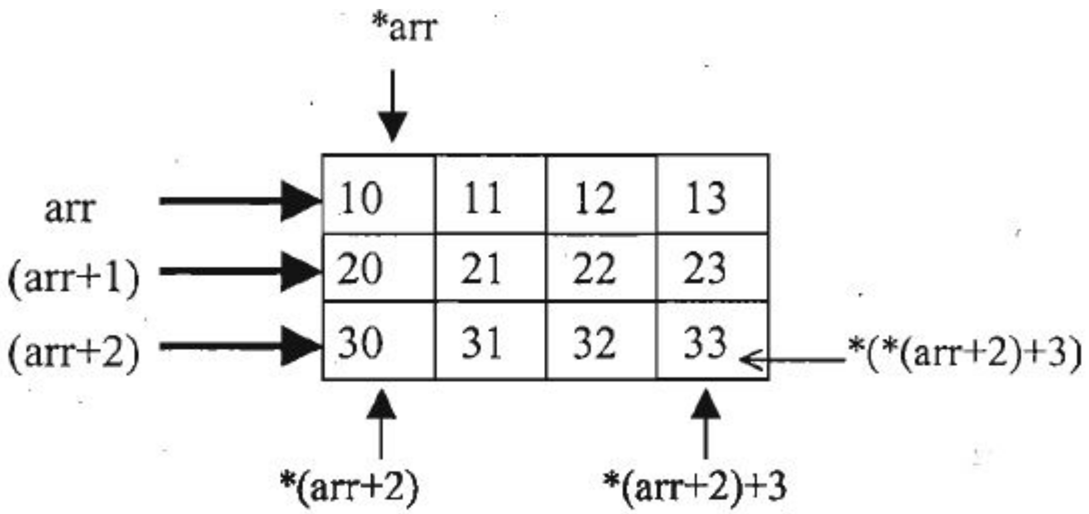
```
main( )
{
  int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
  int i, *j ;
  j = num ; /* assign address of zeroth element */
  for ( i = 0 ; i <= 5 ; i++ )
  {
    printf ( "\naddress = %u ", j ) ;
    printf ( "element = %d", *j ) ;
    j++ ; /* increment pointer to point to next location */
  }
}
```

Pointer to an Array

- We can also declare a pointer that can point to the whole array.
- int (*ptr)[10];
  - ptr is pointer that can point to an array of 10 integers.
  - the type of ptr is 'pointer to an array of 10 integers'.
  - sizeof(*ptr) = 20.

Pointers And Two Dimensional Arrays

- A two-dimensional array can be considered as a collection of one-dimensional arrays.
- we can access any element arr[i][j] of 2-D array using the pointer expression *(*(arr+i)+ j).



Subscripting Pointer To An Array

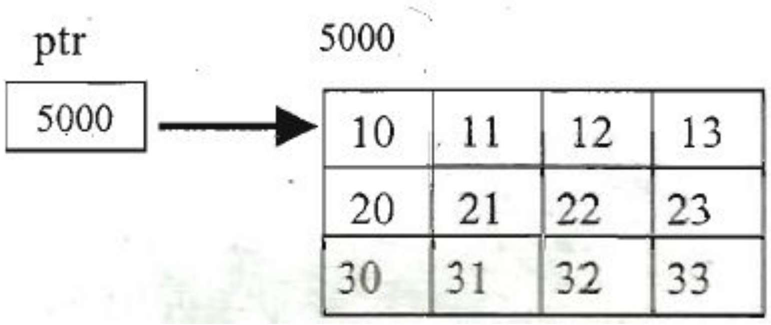
- Subscript a pointer to an array that contains the base address of a 2-D array. For example

  int arr[3][4],= { {10, 11, 12, 13}, {20, 21, 22, 23}, {30, 31, 32, 33} };

  int (*ptr)[4]; // ptr is a pointer to an array of 4 integer

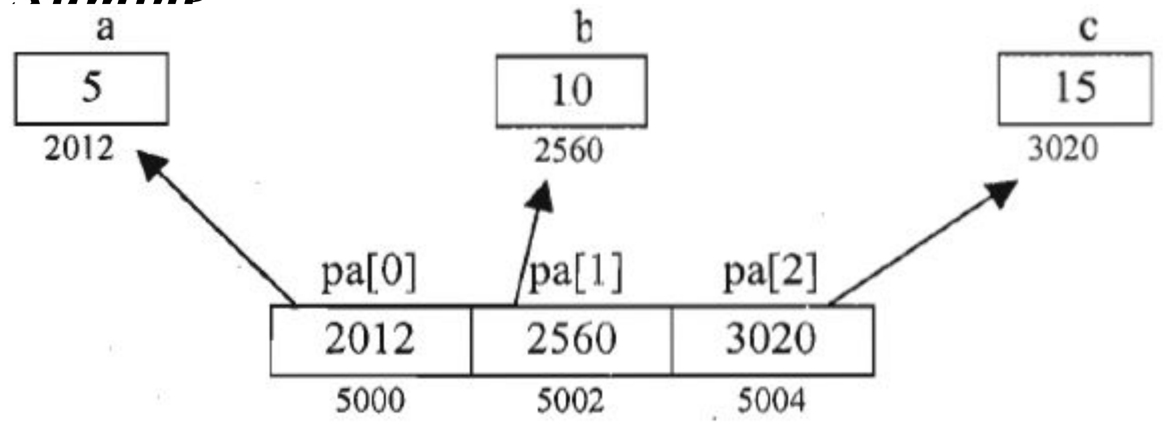
  ptr = arr;

- ptr+i will point to ith row,



Array Of Pointers

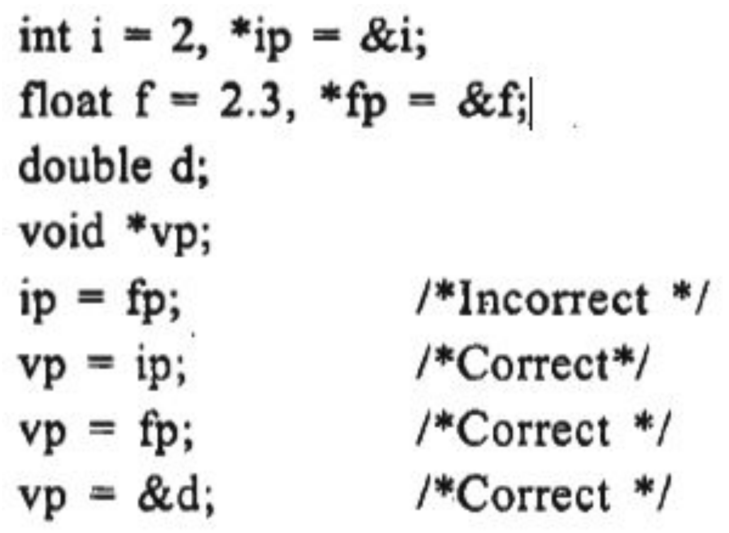
- We can declare an array that contains pointers as its elements.
- Every element of this array is a pointer variable that can hold address of any variable of appropriate type.
- Syntax is: ***datatype *arrayname[size];***
- ***Example:***



```
main ( )
{
   int *pa[3];
   int i,a=5,b=10,c=15;
   pa[0]=&a;
   pa[1]=&b;
   pa[2]=&c;
   for(i=0;i<3;i++)
   {
     printf ("pa [%dl=%u\t" , i, pa [ i 1 ) ;
     printf("*pa[%d] = %d\n",i,*pa[i]);
   }
}
```

void Pointers

- A pointer to void is a generic pointer that can point to any data type. Syntax of declaration is:
- void *vpt; //a pointer of void type.



```
int i = 2, *ip = &i;
float f = 2.3, *fp = &f;
double d;
void *vp;
ip = fp;            /*Incorrect */
vp = ip;            /*Correct*/
vp = fp;            /*Correct */
vp = &d;            /*Correct */
```

- void pointers are generally used to pass pointers to functions which have to perform same operations on different data types.