Computer Programming & Problem Solving

CS100

Mrs Sanga G. Chaki
Department of Computer Science and Engineering
National Institute of Technology, Goa
January, 2023

Array Basics

Topics

1. **Array Declaration**
2. **Accessing Elements of an Array**
3. **Entering Data into an Array**
4. **Reading Data from an Array**
5. **Array Initialization**
6. **Array Elements in Memory**
7. **Copy one array into another**
8. **Array bounds checking**
9. **Things you cannot do**

Arrays

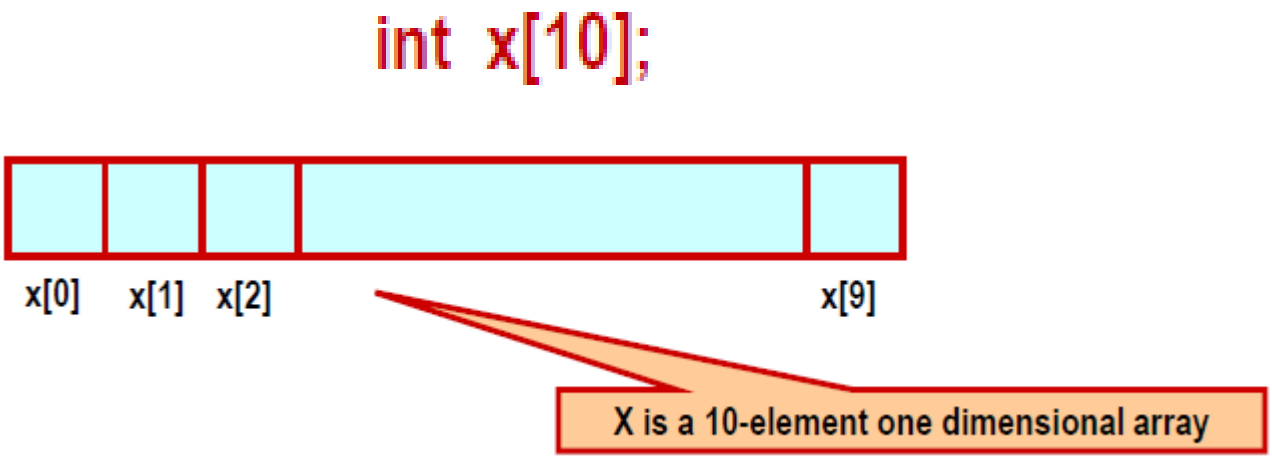
1. **English: An ordered series of a particular type of thing**
2. **C: Array is a data structure which can represent a collection of data items having the same data type (float/int/char/…)**

Why Arrays? – Examples

1. **Many applications require multiple data items that have common characteristics**
2. **Example: Finding the minimum of a set of *n* numbers.**
   - **if-else works fine if *n* value is low.**
   - **But what happens if *n* = 100? Or even more?**
   - **Do we use 100 different variables? No.**
   - **We use arrays - one variable capable of storing or holding all the hundred values.**

Using Arrays

1. **In mathematics, we often express such groups of data items in indexed form:** $x_1, x_2, x_3, \ldots, x_n$
2. **All the data items constituting the group share the same name.**
3. **Individual elements are accessed by specifying the index**



Declaring Arrays

1. **Like variables, the arrays used in a program must be declared before they are used**

General syntax:

```
type   array-name [size];
```

- type specifies the type of element that will be contained in the array (int, float, char, etc.)
- size is an integer constant which indicates the maximum number of elements that can be stored inside the array

How are Arrays stored in Memory?

1. **Starting from a given memory location, the successive array elements are allocated space in consecutive memory locations**

- x: starting address of the array in memory
- k: number of bytes allocated per array element

A[i] is allocated memory location at address x + (i*k)

12	34	66	-45	23	346	77	90
65508	65510	65512	65514	65516	65518	65520	65522

Example

```c
#include <stdio.h>
int main() {
    int a[10];
    int i;
    for(i=0;i<10;i++){ //inserting values in array
        a[i] = i;
    }
    for(i=0;i<10;i++){ //printing values in array
        printf("\nValue in a[%d] = %d",i,a[i]) ;
    }
    printf("\nValue in a[%d] = %d",10,a[10]) ;
    return 0;
}
```

Example

Output

```
/tmp/SQkURl93n3.o
Value in a[0] = 0
Value in a[1] = 1
Value in a[2] = 2
Value in a[3] = 3
Value in a[4] = 4
Value in a[5] = 5
Value in a[6] = 6
Value in a[7] = 7
Value in a[8] = 8
Value in a[9] = 9
Value in a[10] = -1834157824
```

Example

1. In the above example you can use scanf() to insert user-input values in your array.

Some more Array Declarations

```
int x[10];
char line[80];
float points[150];
char name[35];
```

1. **If we are not sure of the exact size of the array, we can define an array of a large enough size.**

Accessing Array Elements

1. **A particular element of the array can be accessed by specifying two things:**
   a) **Name of the array**
   b) **Index (relative position) of the element in the array**
2. **In C, the index of an array starts from 0, not 1**

- An array is defined as `int x[10];`
- The first element of the array x can be accessed as x[0], fourth element as x[3], tenth element as x[9], etc.

Initializing Arrays

- **General form:**

```
type array_name[size] = {list of values};
```

- **Examples:**

```
int  marks[5] = {72, 83, 65, 80, 76};
char name[4] = {'A', 'm', 'i', 't'};
```

The size may be omitted. In such cases the compiler automatically allocates enough space for all initialized elements.

```
int  flag[] = {1, 1, 1, 0};
char name[] = {'A', 'm', 'i', 't'};
```

Copy the elements of one array to anoth

Copy individual elements

```
for ( j = 0; j < 25; j++ )
    a[ j ] = b[ j ];
```

A Warning!

In C, while accessing array elements, array bounds are not checked

Example:

```
int   marks[5];
:
:
marks[8] = 75;
```

- The above assignment would not necessarily cause an error during compilation
- Rather, it may result in unpredictable program results, which are very hard to debug

Things you cannot do

- use = to assign one array variable to another

```
a = b;   /* a and b are arrays */
```

- use == to directly compare array variables

```
if  (a = = b)  ……..
```

- directly scanf or printf arrays

```
printf (“……”, a);
```

Example: What is happening here?

```
main( )
{
      int  avg, sum = 0 ;
      int  i ;
      int  marks[30] ;  /* array declaration */

      for ( i = 0 ; i <= 29 ; i++ )
      {
            printf ( "\nEnter marks " ) ;
            scanf ( "%d", &marks[i] ) ;  /* store data in array */
      }

      for ( i = 0 ; i <= 29 ; i++ )
            sum = sum + marks[i] ;  /* read data from an array*/

      avg = sum / 30 ;
      printf ( "\nAverage marks = %d", avg ) ;
}
```

SEARCHING and SORTING

Topics

1. Searching
    a) Linear search
    b) Binary search
2. Sorting
    a) Bubble sort

Searching

1. Check if a given element occurs in the array.
2. Two ways:
    a) If the array elements are unsorted - Linear search
    b) If the array elements are sorted - Binary search

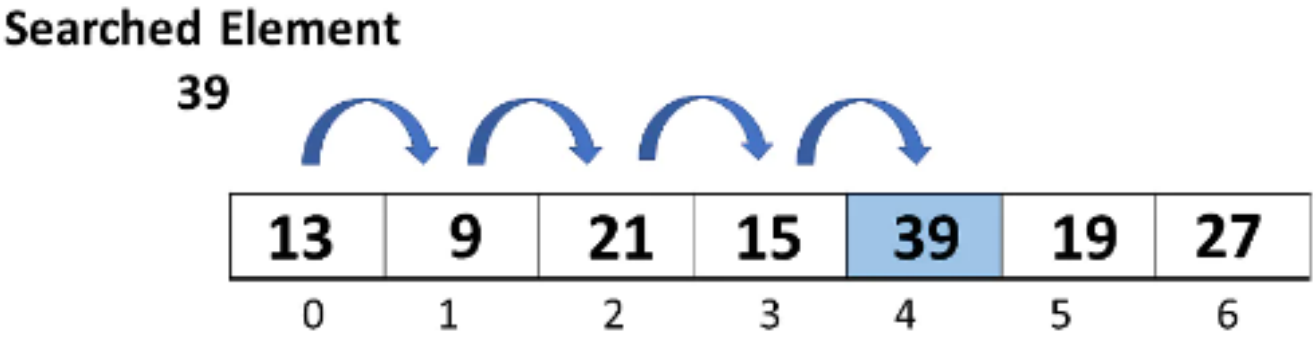
Linear Search

1. **Basic idea:**

   a) **Start at the beginning of the array.**

   b) **Inspect elements one by one to see if it matches the key (the element being searched).**

Linear Search - Example



Linear Search - Example

```c
#include <stdio.h>

int main(){
    int arr[] = {2, 3, 4, 5, 7, 1, 10};
    int x = 4;
    int N = sizeof(arr) / sizeof(arr[0]);
    int i;
    int flag = 0;
    for (i = 0; i < N; i++){
        if (arr[i] == x){
            printf("\nElement is present at index %d", i);
            flag += 1;
//          break;
        }
        else
            continue;
    }
    if(!flag)
        printf("Element is not present in array");
    return 0;
}
```

3 new terms of Algorithmic Performance

1. In computer science, best, worst, and average cases of a given algorithm express measures of usage - at least (best case), at most (worst case) and on average (average case), respectively.
2. In case of searching and sorting algorithms, it measures number of comparisons made in each case.
3. Best case: case where the minimum number of steps on input data of n elements are needed.
4. Worst case : case where maximum number of steps on input data of size n are needed.
5. Average case : case where an average number of steps on input data of n elements re needed.

3 new terms of Algorithmic Performance

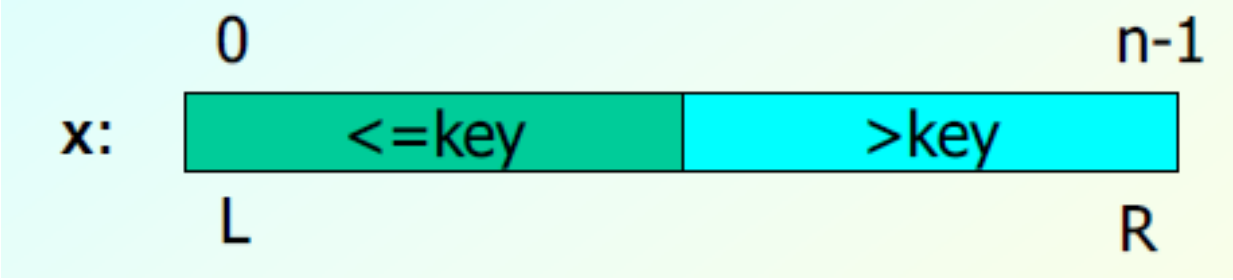
1. If there are *n* elements in the array:
    a) Best case: match found in first element (1 search operation)
    b) Worst case: no match found, or match found in the last element (n search operations)
    c) Average case: (n + 1) / 2 search operations

Binary Search

1. Basic idea: Binary search works if the array is sorted.
    a) Look for the target in the middle.
    b) If you don't find it, you can ignore half of the array, and repeat the process with the other half.
2. In every step, we reduce the number of elements to search in by half.

Binary Search

1. Strategy: Find split between values larger and smaller than key
2. Situation while searching: Initially L and R contains the indices of first and last elements.
3. Then, look at the element at index [(L+R)/2].
4. Move L or R to the middle depending on the outcome of test.



Binary Search - Example

Binary Search - Example

```c
// Binary Search in C
#include <stdio.h>
int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 8; //to be searched
  int mid, high, low, res = -1;
  low = 0;
  high = n-1;
  // Repeat until the pointers low and high meet each other
  while (low <= high) {
    mid = low + (high - low) / 2;
    if (array[mid] == x)
      res = mid;
    if (array[mid] < x)
      low = mid + 1;
    else
      high = mid - 1;
  }
  if (res == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", res);
  return 0;
}
```

Why use Binary Search?

1. Suppose that the array x has 1000 elements.
2. In Linear search – If key is a member of x, it would require 500 comparisons on the average.
3. In Binary search
   a) after 1st compare, left with 500 elements.
   b) after 2nd compare, left with 250 elements.
   c) after at most 10 steps, you are done.
   d) If there are n elements in the array, number of searches required in the worst case: $log_2 n$

ceil() and floor()

1. Standard library functions to roundoff float values to integers
2. The ceil function in C returns the nearest integer greater than the provided argument – argument is a float.
3. floor() function returns the nearest integer smaller than the argument

ceil() and floor()

```c
#include <stdio.h>
#include <math.h>

int main(){
    double num = 8.33;
    int r1, r2;
    r1 = ceil(num);
    printf("Ceiling integer of %.2f = %d", num, r1);
    r2 = floor(num);
    printf("\nFloor integer of %.2f = %d", num, r2);
    return 0;
}
```

Output

```
/tmp/gGn0GYZUCQ.o
Ceiling integer of 8.33 = 9
Floor integer of 8.33 = 8
```

Sorting

1. Basic Problem: Given an array

$$x[0], x[1], \ldots, x[size-1]$$

reorder entries so that

$$x[0] <= x[1] <= \ldots <= x[size-1]$$

So that, the array is in non-decreasing or non-increasing order.

2. Example: If original list: 10, 30, 20, 80, 70, 10, 60, 40, 70

a) Sorted in non-decreasing order: 10, 10, 20, 30, 40, 60, 70, 70, 80

b) Sorted in non-increasing order: 80, 70, 70, 60, 40, 30, 20, 10, 10

Bubble Sort

1. The sorting process proceeds in several passes.
2. In every pass, we go on comparing neighboring pairs, and swap them if out of order.
3. If we are sorting in ascending order, in every pass, the largest of the elements under consideration will bubble to the top (i.e., the right).
4. Number of comparisons: n(n-1)/2, if there are n elements in the array.

Bubble Sort – Worked out example

PASS 1:

10	5	17	11	-3	12
5	10	17	11	-3	12
5	10	17	11	-3	12
5	10	11	17	-3	12
5	10	11	-3	17	12
5	10	11	-3	12	17

PASS 2:

5	10	11	-3	12	17
5	10	11	-3	12	17
5	10	11	-3	12	17
5	10	-3	11	12	17
5	10	-3	11	12	17

Bubble Sort – Worked out example

PASS 3:

5 10 -3 11 12 17
5 10 -3 11 12 17
5 -3 10 11 12 17
5 -3 10 11 12 17

PASS 4:

5 -3 10 11 12 17
-3 5 10 11 12 17
-3 5 10 11 12 17

PASS 5:

-3 5 10 11 12 17
-3 5 10 11 12 17

Sorted

Bubble Sort – Code

```c
#include <stdio.h>
int main() {//Bubble sort
   int array[] = {10, 5, 17, 11, -3, 12};//array to be sorted
   int size = 5, step, i, temp;
   for(step= 0;step<size - 1;step++) {// loop to access each array element
     for(i=0; i<size - step - 1;i++) {// loop to compare array elements
       if(array[i] > array[i + 1]) {// compare two adjacent elements
         temp = array[i]; // swap if elements not in the intended order
         array[i] = array[i + 1];
         array[i + 1] = temp;
       }//end if
     }//end inner for loop
   }//end outer for loop
   printf("Sorted Array in Ascending Order:\n");
   for (i = 0; i < size; i++)
     printf("%d  ", array[i]);
   return 0;
}
```

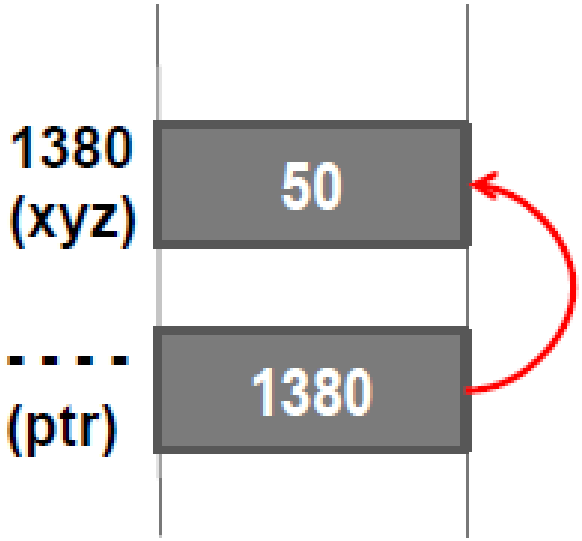
ARRAYS and POINTERS

Topics

1. Pointers revisited
2. Arrays and Pointers
3. Passing array elements/whole arrays to functions

Recap of Pointers

1. A pointer is a C variable whose value is the address of another variable.
2. Example:




```
int xyz = 50;

int *ptr; // Here ptr is a pointer to an integer

ptr = &xyz;
```

How to Access Array Elements – Method 1

```
main( )
{
    int  num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int  i ;

    for ( i = 0 ; i <= 5 ; i++ )
    {
        printf ( "\naddress = %u ", &num[i] ) ;
        printf ( "element = %d", num[i] ) ;
    }
}
```

1. **Using subscripted variables**
2. **What is the address of an array element? Say num[3]?**
3. **So how will I declare a pointer to this array element?**

Recap of Pointers – Illegal usages

1. Pointing at constant.
    a) &235
2. Pointing at array name
    a) int arr[20]; &arr;
3. Pointing at expression
    a) &(a+b)

Pointer Arithmetic

1. **Before we learn about pointers and arrays, we need to learn pointer arithmetic.**

Pointer Operations

1. Addition of a number to a pointer
2. Subtraction of a number from a pointer
3. Subtraction of one pointer from another
4. Comparison of two pointer variables

What happens here?

```
main( )
{
    int  i = 3, *x ;
    float  j = 1.5, *y ;
    char  k = 'c', *z ;

    printf ( "\nValue of i = %d", i ) ;
    printf ( "\nValue of j = %f", j ) ;
    printf ( "\nValue of k = %c", k ) ;
    x = &i ;
    y = &j ;
    z = &k ;
    printf ( "\nOriginal address in x = %u", x ) ;
    printf ( "\nOriginal address in y = %u", y ) ;
    printf ( "\nOriginal address in z = %u", z ) ;
    x++ ;
    y++ ;
    z++ ;
    printf ( "\nNew address in x = %u", x ) ;
    printf ( "\nNew address in y = %u", y ) ;
    printf ( "\nNew address in z = %u", z ) ;
}
```

Value of i = 3
Value of j = 1.500000
Value of k = c
Original address in x = 65524
Original address in y = 65520
Original address in z = 65519
New address in x = 65526
New address in y = 65524
New address in z = 65520

Addition/Subtraction of number to a pointer

```
int i = 4, *j, *k ;
j = &i ;
j = j + 1 ;
j = j + 9 ;
k = j + 3 ;

j = j - 2 ;
j = j - 5 ;
k = j - 6 ;
```

1. **Every time a pointer is incremented/decremented it points to the immediately next/previous <u>location of its type</u>.**

Addition/Subtraction of number to a pointer

```c
#include <stdio.h>
int main() {
    int i=10;
    int * j;
    j = &i;
    printf("\n j = %u", j);
    j++;
    printf("\n j = %u", j);
    j++;
    printf("\n j = %u", j);
    j--;
    printf("\n j = %u", j);
    j=j+9;
    printf("\n j = %u", j);
    j=j-9;
    printf("\n j = %u", j);
    return 0;
}
```

Output

```
/tmp/kUq8tyZUJS.o
j = 3056920588
 j = 3056920592
 j = 3056920596
 j = 3056920592
 j = 3056920628
 j = 3056920592
```

Subtraction of one pointer from another

```
main( )
{
    int  arr[ ] = { 10, 20, 30, 45, 67, 56, 74 } ;
    int  *i, *j ;

    i = &arr[1] ;
    j = &arr[5] ;
    printf ( "%d %d", j - i, *j - *i ) ;
}
```

When possible?

One pointer variable can be subtracted from another provided both variables point to elements of the same array

1. **j – i ?? → 4. why?**
2. **Because j and i are pointing to locations that are 4 integers apart.**
3. ***j – *i ?? → 36.**
4. **Since *j and *i return the values present at addresses contained in the pointers j and i**

Comparison of two Pointer Variables

```
main( )
{
    int  arr[ ] = { 10, 20, 36, 72, 45, 36 } ;
    int  *j, *k ;

    j = &arr [ 4 ] ;
    k = ( arr + 4 ) ;

    if ( j == k )
        printf ( "The two pointers point to the same location" ) ;
    else
        printf ( "The two pointers do not point to the same location" ) ;
}
```

1. **Pointer variables can be compared provided both variables point to objects of the same data type.**

Pointer Operations – Not Allowed

1. Addition of two pointers
2. Multiplication of a pointer with a constant
3. Division of a pointer with a constant

Try and see what errors you get!

How to Access Array Elements – Method 2

```c
#include <stdio.h>
int main() {
    int A[]={1,2,3,4};
    int i, *j;
    j=&A[0];
    for(i=0;i<6;i++){
        printf("\nIndex = %d",i);
        printf("\nAddress = %u",j);
        printf("\nValue = %d",*j);
        j++;
    }
    return 0;
}
```

1. **Using Pointers**

Base Address of an Array

```c
#include <stdio.h>
int main() {
    int A[]={1,2,3,4};
    int *i, *j, *k; //Pointers to base address
    //Different ways to access base address of an array
    i = &A[0];
    j = A;
    k = (A+0);
    printf("\n Base Address = %u",i);
    printf("\nBase Address = %u",j);
    printf("\nBase Address = %u",k);
    int *l, *m; //Pointers to address of element at index 2
    //Different ways to access address of element at index 2
    l = &A[2];
    m = (A+2);
    printf("\nAddress of element at index 2 = %u",l);
    printf("\nAddress of element at index 2 = %u",m);
    return 0;
}
```

Which method to use?

1. Accessing array elements by pointers is always faster than accessing them by subscripts.
2. Convenience matters:
    a) Array elements should be accessed using pointers if the elements are to be accessed in a fixed order
    b) Access the elements using a subscript if there is no fixed logic in accessing the elements

Passing Array Elements to a Function

1. **Array elements can be passed to a function by calling the function by value, or by reference.**
2. **Call by value: pass values of array elements to the function**
3. **Call by reference: pass addresses of array elements to the function – what is the address of an array element? – Similar to variable addresses.**

Passing Array Elements to a Function

```
main( )
{
      int  i ;
      int  marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;

      for ( i = 0 ; i <= 6 ; i++ )
            display ( marks[i] ) ;

       for ( i = 0 ; i <= 6 ; i++ )
            disp ( &marks[i] ) ;
 }
```

```
display ( int  m )
{
      printf ( "%d ", m ) ;
}
```

```
disp ( int  *n )
{
      printf ( "%d ", *n ) ;
}
```

Passing an Entire Array to a Function

```
display ( &num[0], 6 ) ;
display ( num, 6 ) ;
```

1. **Just pass the address of the first (base) element of the array**
2. **And the number of elements in the array**

Passing an Entire Array to a Function

main.c

```c
// Online C compiler to run C program online
#include <stdio.h>
void display ( int *j, int n ) {
    int i ;
    for ( i = 0 ; i <= n - 1 ; i++ ) {
        printf ( "\nelement %d = %d",i, *j ) ;
        j++ ; /* increment pointer to point to next element */ }
}
int main(){
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    display (&num[0], 6 ) ;
    return 0;
}
```

Passing an Entire Array to a Function

```c
// Online C compiler to run C program online
#include <stdio.h>
void display ( int *j, int n ) {
    int i ;
    for ( i = 0 ; i <= n - 1 ; i++ ) {
        printf ( "\nelement %d = %d",i, *j ) ;
        j++ ; /* increment pointer to point to next element */ }
}
int main(){
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    display (num, 6 ) ;
    return 0;
}

```

Passing an Entire Array to a Function

```c
#include <stdio.h>
int search(int array[], int n, int x) {
  for (int i = 0; i < n; i++)
    if (array[i] == x)
      return i;
  return -1;
}
int main() {
  int array[] = {2, 4, 0, 1, 9};
  int x = 1;
  int n = sizeof(array) / sizeof(array[0]);
  int result = search(array, n, x);
  (result == -1) ? printf("Element not found") : printf("Element found at
      index: %d", result);
  return 0;
}
```

Pointer Expressions

1. If p1 and p2 are two pointers, the following statements are valid:
   a) sum = (*p1) + (*p2); BUT NOT sum = p1 + p2
   b) prod = (*p1) * (*p2);
   c) *p1 = *p1 + 2;
   d) x = *p1 / *p2 + 5;

What is happening here?

```
int x[ 5 ] = { 10, 20, 30, 40, 50 };
int *p;

p = &x[1];
printf( "%d", *p);

p++;
printf( "%d", *p);

p = p + 2;
printf( "%d", *p);
```

Last Slide!

```
int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
```

Base address of an array is given by the name of the array.

So what is *num?

***num and *(num + 0) both refer to 24.**

So what is *(num + 1)

When we say, num[i], the C compiler internally converts it to *(num + i). So following all are same:

```
num[i]
*( num + i )
*( i + num )
i[num]
```