**ICS 103: Computer Programming in C**

**Lab 9: How to Use 1-D array with Functions**

**Objectives:**

Learning how to declare 1D-arrays

Learning how to use Partial arrays

Learning how to use Parallel arrays

Learning some 1D-array applications:

* 1. Learn how to search arrays (using linear search).
  2. Learn how to sort arrays (using selection sort).
  3. Generating random numbers

Practicing how to use 1D arrays with functions

**1. 1D-Array declarations**

Arrays are collections of elements all of the same data type. The elements are arranged in consecutive memory locations.

The syntax of one-dimensional array declaration is:

**dataType arrayName[integralExpression];**

where the **integralExpression ≥ 0** specifies the size or length of the array i.e., the number of elements in the array.

**Note:** zero length arrays are allowed in C.

Example: The declaration **int b[10];**

declares an array, named **b**, consisting of ten elements, each of type **int**

An individual array element is accessed by using an int expression subscript or index. C uses zero-based indexing: The index (subscript) starts from ***0*** to ***ArraySize - 1***.

The array **b** declared above has the elements: b[0], b[1], b[2], b[3], and b[4].

The elements of an array behave like normal variables:

**b[0] = 10;**

**b[1] = 20;**

**b[2] = b[0] + b[1];**

**printf("Enter value of element at index 3: ");**

**scanf("%d", &b[3]);**

**printf("d", b[2] + b[3] \* 2);**

Note:

* **Starting from C99 it is possible to define the size of an array at runtime. Example:**

**int arraysize;**

**printf("How big do you want your array to be?\n");**

**scanf("%d",&arraysize);**

**int array2[arraysize];**

however; once an array is declared, **its size never changes**.

* In C Array accesses are not checked; an attempt to use an index that is less than zero or greater than or equal to the size of the array does not generate compile error; however it may lead to run-time or logic errors.

**2. Aggregate operations on arrays**

C does not provide aggregate operations on arrays (manipulation of entire array as one unit.)

Example: Given: **int x[5] = {1, 2, 3, 4, 5} , y[5] = {10, 45, 72, 13, 35};**

• NO aggregate assignment of y to x

**x = y; // illegal**

**x = {7, 8, 12, 15, 60}; // illegal**

• NO aggregate comparison of arrays

**if (x = = y) // illegal**

**printf("Arrays are equal");**

• NO aggregate I/O on arrays [Except **char** arrays]

**scanf("%d", x); // illegal**

**printf("%d", x); // illegal**

• NO aggregate arithmetic on arrays

**x = x + y; //illegal**

• NO returning an array as the value of a function

**return x; // illegal**

• The only thing we can do to an array as a whole is pass it as a parameter to a function (discussed in Part 7, below)

In each of the illegal cases mentioned above except **return x;** , an appropriate loop has to be used. Example, to copy the contents of array **y** to **x** we use:

**int i;**

**for(i = 0; i < 5; i++)**

**x[i] = y[i];**

**3. Partial arrays**

It is possible to use a 1D-array of size **n** in which only the first **m** contiguous elements of the array are initialized, where **m < n**. Such an array is called a **partial array**. In such a case, it is necessary to know the actual number of initialized elements.

Example: Consider the loop given in Section 2.2:

**double grades[100], grade;**

**int index = 0;**

**printf("Enter grade [-ve value or value > 100 to terminate]: ");**

**scanf("%lf", &grade);**

**while(grade >= 0 && grade <= 100){**

**grades[index] = grade;**

**index++;**

**printf("Enter grade [-ve value or value > 100 to terminate]: ");**

**scanf("%lf", &grade);**

**}**

If the user enters fewer than 100 grades, then the resulting array will be a partial array.

**4. Parallel arrays**

Two or more full or partial1D-arrays of the same size where index-corresponding elements store related data.

Example: The following three parallel arrays store Quiz01 and Quiz02 grades for 5 students:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 3 | 4 | 5 |
| IDs | 90012350 | 90012351 | 90012352 | 90012353 | 90012354 |
|  |  |  |  |  |  |
| Quiz01 | 10.0 | 7.5 | 6.4 | 9.6 | 8.2 |
|  |  |  |  |  |  |
| Quiz02 | 5.4 | 6.0 | 8.3 | 4.0 | 7.6 |

1. **Generating Random Numbers:**

To generate a pseudo-random integer in the interval [k, m] where **k** and **m** are integers such that **k < m**, use the standard function **rand** in the expression:

**k + rand()%(m - k + 1)**

Examples:

* To generate pseudo-random integers in the interval [5 , 30], use the call **5 + rand( ) % 26**
* To generate pseudo-random integers in the interval [1 , 30], use the call **1 + rand( ) % 30**
* To generate pseudo-random integers in the interval [0 , 40], use the call  **rand( ) % 41**

To generate different pseudo-random sequences whenever the program is executed use the following call

before the loop that generates the pseudo-random numbers:

**srand(time(NULL));**

The required header files are:

|  |  |
| --- | --- |
| Function or constant | Header file |
| rand | stdlib.h |
| srand | stdlib.h |
| time | time.h |
| NULL | Defined in several header files: stdio.h, stdlib.h, and stddef.h |

**Note:**

* The call **time(NULL)** returns the number of milliseconds that have elapsed since 00:00 hours, Jan

1, 1970 UTC.

* **rand()** returns a pseudo random integer in the range 0 to RAND\_MAX inclusive, where RAND\_MAX is an integer constant defined in **stdlib.h** with a value of at least 32767
* Choose appropriate values for **k** and **m**

**6. Passing an individual array element to a function:**

You can pass individual array elements or their addresses to a function that has normal or pointer variables as arguments respectively.

* An array element can be passed to a function both call by value and call by reference.
* Modifying an array element that is passed by value within the called function does not modify that element in the calling function.
* Modifying an array element that is passed call by reference within the called function modifies that element in the calling function.

|  |  |
| --- | --- |
| Call by value | Call by reference |
| **#include <stdio.h>**  **void modify1(int x);**  **int main(void) {**  **int a[] = {10, 5, 7, 12, 45, 50};**  **printf("a[3] = %d\n", a[3]);**  **modify1(a[3]);**  **printf("a[3] = %d\n", a[3]);**  **system("pause");**  **return 0;**  **}**  **void modify1(int x){**  **x = 60;**  **}** | **#include <stdio.h>**  **void modify2(int \*x);**  **int main(void) {**  **int a[] = {10, 5, 7, 12, 45, 50};**  **printf("a[3] = %d\n", a[3]);**  **modify2( &a[3]);**  **printf("a[3] = %d\n", a[3]);**  **system("pause");**  **return 0;**  **}**  **void modify2(int \*x){**  **\*x = 60;**  **}** |
| Output:  a[3] = 12  a[3] = 12 | Output:  a[3] = 12  a[3] = 60 |

**7. Passing an entire array to a function:**

We can also have a function that takes the whole array as argument. Such a function does not create a local array when called; instead it will just receive the address of the actual array (the one used in the call) and work directly on the actual array elements. Thus array addresses are passed in a function call. **An array is passed call by reference**. Modifying any element of the array in the called function modifies that element in the calling function.

We may specify the size of a 1D array parameter or we may use square brackets without specifying the size. In addition to the array parameter, we need to pass the number of elements of the array to the function, if that size is not declared globally. When calling such a function, we use the name of the actual array without square brackets for the first argument. For the second argument, we use the size of the actual array if it is full; otherwise we use the actual number of the elements present in the array if it is partially filled.

Example:

|  |
| --- |
| **#include <stdio.h>**  **void modify3(int x[], int size);**  **int main(void) {**  **int a[] = {10, 5, 7, 12, 45, 50};**  **printf("a[3] = %d\n", a[3]);**  **printf("a[5] = %d\n", a[5]);**  **modify3(a, 6);**  **printf("a[3] = %d\n", a[3]);**  **printf("a[5] = %d\n", a[5]);**  **system("pause");**  **return 0;**  **}**  **void modify3(int x[], int size){**  **x[3] = 60;**  **x[size – 1] = x[0] \* 2;**  **}** |
| Output:  a[3] = 12  a[5] = 50  a[3] = 60  a[5] = 20 |

**Types of array parameters**

Array arguments and parameters can be classified into three types: input, output, and inputOutput:

|  |  |  |
| --- | --- | --- |
| Type | Comment | Example |
| Input | The array is initialized before the function call and the function does not modify the passed array. | **void print\_array (double a[], int n);**  **double get\_average (double a[], int n );** |
| Output | The array is not initialized before the function call. The function modifies the array by initializing it. | **void read\_array (double a[], int n);** |
| inputOutput | The array is initialized before the function call and the function modifies it to possibly different values. | **void reverseArray(int a[], int n);** |

**Note:** In C, since each array is passed by reference, potentially each array is an output argument. To ensure that a called function does not modify a passed array, the array can be declared as a constant array. Example:

**void print\_array (const double a[], int n);**

**8. Applications:**

**8.1 Linear Search:**

**The linear search function** receives a sorted or unsorted array of type **int** and a target to search for. The function returns the first location of the target (its index) if found or -1 if not found.

int linear\_search(int a[], int n, int target){

int i;

for(i = 0; i < n; i++){

if(target == a[i])

return i ;

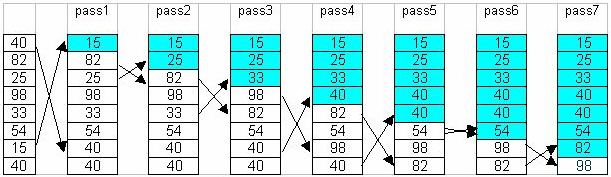
}

return -1;

}

**8.2 Selection Sort:**

**Selection Sort** is a simple algorithm to sort an array of numbers and it works as follows. To sort an array A in increasing order: First, we find the minimum element and swap it with A[0]. Next, we find the minimum of the remaining elements and swap it with A[1]. We continue this way until n -1 passes are completed, where n is the number of elements in A. The following is an example:



The following is the code for selection sort function:

|  |
| --- |
| **void selection\_sort(double x[], int size){**  **int k,j,minpos;**  **double temp;**  **for (k = 0; k <= size - 2; k++){**  **minpos = k; // initialize location of min value**  **// go over the elements to find location of minimum value**  **for(j = k+1; j <= size - 1; j++){**  **if(x[j] < x[minpos])**  **minpos = j;**  **}**  **// bring minimum value which is at minpos to index k**  **temp = x[minpos];**  **x[minpos] = x[k];**  **x[k] = temp;**  **}**  **}** |

**Lab Tasks:**

**Task # 1:**

Write a function maxTwoArrays that takes 2 arrays of integer values and their size (same size) and returns an array maxTwo such that:

maxTwo[i]=maximum (ar1[i],ar2[i])

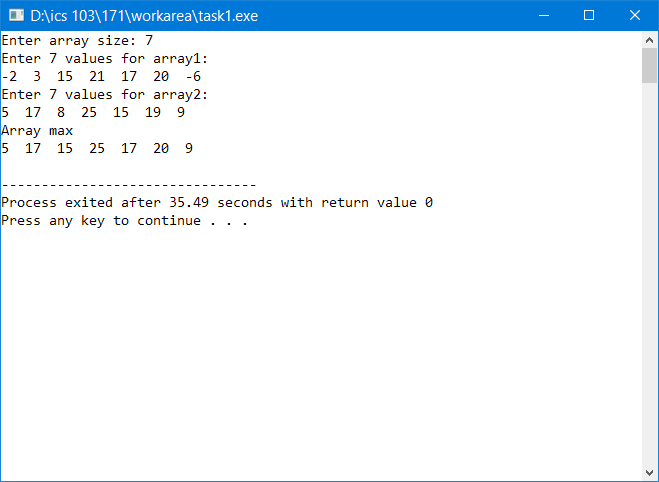
Test your function in a complete C program. Your function must be general, it must work for any valid array size.

Example: If the passed array1 is {-2, 3 , 15, 21, 17, 20, -6}

and passed array2 is { 5 , 17, 8 , 25, 15, 19, 9}

then the returned array is {5 , 17, 15, 25, 17, 20, 9};

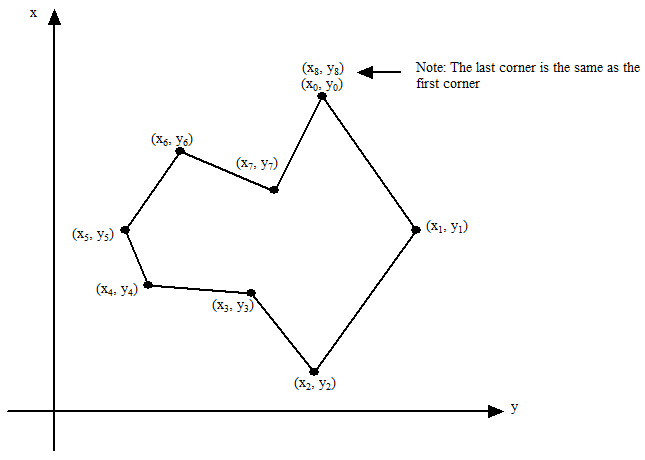
sample run



**Task # 2:**

Consider a closed polygon with **n** distinct corners: **(x0, y0), (x1, y1), . . . , (xn-1, yn-1)**

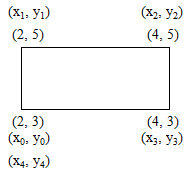
Example:



The area of such a polygon can be computed by the formula:

where **(xn , yn)** is the repeated first point of the polygon, i.e., the corner **(x0, y0)**.

Example: For the rectangle:



The area is 1/ 2.0 \* | (x1 + x0)(y1 – y0) + (x2 + x1)(y2 – y1) + (x3 + x2)(y3 – y2) + (x4 + x3)(y4 – y3)|

= 1/ 2.0 \* | (2 + 2)(5 – 3) + (4 + 2)(5 – 5) + (4 + 4)(3 – 5) + (2 + 4)(3 – 3)|

= 4.0

To implement the formula for a polygon with **n** distinct corners, use two parallel arrays of x- and y-coordinates of the polygon corners, each array of size **n + 1**, and in which the coordinates of the first corner are repeated in the last parallel array elements i.e. (x[n]=x[0] and y[n]=y[0]):

Example: For the above rectangle; n=4, the parallel arrays x and y are of size 5:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4=x0 |
| x-coordinates | 2.0 | 2.0 | 4.0 | 4.0 | 2.0 |
|  |  |  |  |  |  |
| y-coordinates | 3.0 | 5.0 | 5.0 | 3.0 | 3.0 |
|  | y0 | y1 | y2 | y3 | y4=y0 |

Write a C program that prompts for and reads the number **n** of corners of a polygon. If the size is less than 3, the program displays an appropriate error message and terminates; otherwise it prompts for and reads the **n** x- and y-coordinates of the polygon distinct corners into two parallel arrays of size **n + 1**. The program then initializes the last elements of the two arrays to the x- and y-coordinate of the first corner respectively i.e i.e. (x[n]=x[0] and y[n]=y[0]). It then passes the two arrays to a function **calculatePolygonArea** that returns the area of the polygon. Finally the returned area is displayed in the main function.

The prototype of **calculatePolygonArea** function is:

double calculatePolygonArea(int numCorners, double x[], double y[]);

Sample program runs:

|  |
| --- |
|  |
|  |
|  |

**Task # 3:**

Write a logical function **sorted\_inc** that receives an array of int values and **n** representing the number of values. The function will return **1** if the array values are sorted in increasing order, **0** otherwise.

The array is sorted in increasing order if a[0] ≤ a[1] ≤a[2]…….. ≤a[n-1]

Write another version of **selection\_sort**; **selection\_sort\_dec** so that it sorts the array in decreasing order. You need to change the type of the array and variable temp from double to int in the definition of selection\_sort to accommodate the question here.

Write a program that reads the size **n** of an array. Then, it will generate **n** random integer values from 1 to 100 to be stored in the array. print the array on the screen.

Check if it is by chance sorted in increasing order by using the functions **sorted\_inc**. If the array is not sorted (this is what is expected since the values are random), sort the array by using the original and modified selection sort functions i.e. increasing and decreasing. Print your array after each sorting.

Sample run

|  |
| --- |
|  |