International University of Technology Twintech
جامعة تونتك الدولية للتكنولوجيا

Faculty of Computer Science and Information Technology
Department of Business Information Technology
Sana'a - Yemen

Lab Manual # 11&12

Prepared by Eng. Enas Aldahbali

Student Name			
Department			
Roll #			
Section			
Lab Date	-11-2023		
Submission Date			
Lab Grade:	10	**Obtained Grade**	
Instructor's Signature:			

A. *Title:* **Arrays**

B. ***Objectives of this lab:***

- Learn to declare, initialize, and use arrays
- Learn about Types of Array
- Learn about Array Index Out of Range
- Learn about two dimensional arrays.

Activity 8.1 - **Learn how to declare Arrays**

So far all of our variables have been able to hold only one value at any one point in time. Such variables are called *scalar* variables. Now it is time for our first non-scalar variable, an *array*.

An array is a variable capable of storing multiple values. When we declare an array we tell the compiler how many values we want the array to hold. We also tell the compiler what type of values the array can store. All of the values in an array must be of the same type.

Here is a declaration of an array called *numlist* that will be used to store 8 integers:

```
int numlist[8];       // declaring an integer array that can store 8
values
```

Each of the integers in the array is stored in the same number of bytes as a scalar integer, which on most machines is 4 bytes. Thus, the entire array will occupy 32 bytes of memory. The compiler always stores an array in contiguous memory locations (all of the elements of the array are stored in one chunk of memory with no gaps). Here is one way you may visualize the above array numlist when it stores the following 8 integers: 12, 8, 10, 123, 1000, 23, 4, 10

Imaginary Memory Address	1023	1024	1025	1026	1027	1028	1029	1030
Array Index	0	1	2	3	4	5	6	7
Indexed *numlist* Variable	numlist[0]	numlist[1]	numlist[2]	numlist[3]	numlist[4]	numlist[5]	numlist[6]	numlist[7]
Array Content	12	8	10	123	1000	23	4	10

The individual values stored in an array are called the *elements* of the array. You will also hear them called *indexed variables* or *subscripted variables*. Each of the elements of an array is assigned an *index*. An index is a natural number in the range {0,1,2,...}. Note that the array index started from 0.

As it is shown in the above table, to access one of the elements of an array, you put the index of that element in square brackets after the name of the array. The 0th element in the array called numlist is numlist[0], the next one is numlist[1], and so forth. Since we start the numbering with 0, the last element in numlist is numlist[7].

To put the value of 12 into the 0th element of numlist, we will use:

```
numlist[0] = 12;
```

If we wanted to store a value that is entered from the keyboard into element numlist[1] we use:

```
cin >> numlist[1];
```

An array element like numlist[4] can be used in any way that a scalar variable can be used. All of the following statements are legal:

```
if(numlist[2] > numlist[1]) // Compares the third element of the
array with
                            // the second element of the array
cout << numlist[5];         // Displays the sixth element of the
array

sum = sum + numlist[7];     // Adds the 8th element to sum
```

The index inside the square brackets does not have to be an integer constant such as 3 or 4. It can be any integral expression that evaluates to an integer within the permitted range of the array's index. So an expression such as this:

for(i = 0; i < 3; i++)
*numlist[2*i+1] = 0; // set the odd elements of the array to 0*

If you wish to fill the array numlist with the integers typed from the keyboard, you can use a *for loop* too. Here is a *for loop* that will allow you to enter 8 values from the keyboard and will store them in the array numlist. Notice that we have used the variable i as an index for array numlist.

```
for (i=0; i<8; ++i)
{
        cout << "Enter the next value: ";
        cin >> numlist[i];
}
```

It might be easier for our user to keep up with different values that need to be entered if we display something more helpful than "Enter the next value: ". Since users typically number items in a list starting from 1, we will say "Enter value #1: " when asking for numlist[0], "Enter value #2: " when asking for numlist[1], and so forth. Here is the improved vrsion of the loop:

```
for (i=0; i<8; ++i)
{
        cout << "Enter value #" << i+1 << ": ";
        cin >> numlist[i];
}
```

By asking for value 1, then value 2, etc., we are allowing our user to count in a more natural way than C++ forces us to count. That is the most confusing part of working with arrays. It is natural to think that an array of size 8 will keep 8 values, thus, assuming that the indices would be 1 through 8. For an array of size 8, index 1 is a valid index, but index 8 is invalid and will cause a run-time error if it is used.

The following program allows you to enter 8 integers from the keyboard and will store those values in array numlist.

Code Answer:-

```
// P10_1.cpp - A program that uses an array of integers
#include <iostream>
using namespace std;
int main(void)
{
        int numlist[8], i;

        // Read 8 integers from the keyboard
        for (i = 0; i<8; i++ )
        {
                cout << "Enter value #" << i+1 << ": ";
                cin >> numlist[i];
        }
        // Display the numbers in a reverse order
        for (i = 8; i > 0; i-- )
        {
                cout << "Value #" << i << ": ";
                cout << numlist[i-1] << endl;  //Pay attention to i-1!
        }

        return 0;
}
```

Activity 10.2:-Array Index Out of Range

A common error when working with an array is attempting to access an element of the array using an index that is out of range. In the above program, array numlist has 8 elements. The final value is called numlist[7]. If we try to access numlist[8], most C++ compilers will not give us an error message at run time. C++ does not verify that your index is within the proper range when you compile the program. If you print the value of numlist[8], the compiler will grab the 4 bytes following numlist[7], interpret whatever is stored there as an integer and print the value of that integer. If you store a value at numlist[8], the compiler will place that value into the first 4 bytes following numlist[7], even if those bytes happen to be storing a different variable! That is what happens in the following program.

Your Code:-

Here is the output of this program:

```
     i          numlist[i]
    =====     ========
     0          0
     1          2
     2          4
     3          6
     4          8
     5          10
     6          12
     7          14
     16         1      // Do you see any thing wrong here?
```

Activity 10-3 - Initializing Arrays

A scalar variable can be initialized when it is declared, like this:

```
int num = 4;
```

An array can also be initialized when it is declared. Here we put the value 0 into numlist[0], the value 1 into numlist[1], etc.:

int numlist[8] = {12, 8, 10, 123, 1000, 23, 4, 10};

If you list fewer values within the braces { and } than the **declared size** (8 in the above example) of the array, our C++ compiler will initialize all the rest of the elements to 0. However, not all C++ compilers will do this. If you initialize an array when it is declared, you can omit the size of the array. C++ will use the number of initializers in your list as the size of the array. Here is an example:

`char vowels[] = {'a', 'e', 'i', 'o', 'u'};` This declared a character array of size 5 which stores the lowercase vowels, *a*, *e*, *i*, *o*, and *u*.

Activity 10-4 - Creating Arrays of Flexible Size

One way to create an array with a particular size is to use a global variable to define the size of that array. Thus, every time one can change that number to increase or decrease the size of an array. This requires you to recompile the program for the new size to take effect. Let's modify program 10_1.cpp to work on flexible size arrays.

Code:-

```
// P10_1b.cpp - A program that uses a flexible size array of integers
#include <iostream>
using namespace std;
const int SIZE = 8;  // Set the maximum size for the array
int main(void)
{
        int numlist[SIZE];
        // Read SIZE integers from the keyboard
        for (int i = 0; i<SIZE; i++ )
        {
                cout << "Enter value #" << i+1 << ": ";
                cin >> numlist[i];
        }
        // Display the numbers in a reverse order
        for (int i = SIZE; i > 0; i-- )
        {
                cout << "Value #" << i << ": ";
                cout << numlist[i-1] << endl;  //Pay attention to i-1!    }
        return 0;
}
```

This produces the same result as P10_1.cpp. Now, you are limited to array of 8 integers. By changing the value for SIZE, you can read as many numbers as you wish

Exercise 10.1:

Modify program P10_1b.cpp such that it reads some number of integers as defined by SIZE, stored them in array numlist, displays the array numlist, then reverses the contents of the array, and at last displays the contents of that array again.

Make your program as general as possible. Thus, your program should be able to reverse the contents of an array of any size defined by SIZE. Note that we didn't ask you to display the array in reverse, that would be what program P10_2.cpp is doing. We want you to reverse the contents of the array, then display the array itself. Example:

int A = {1, 2, 4, 5, 8, 2, 0, 9};

After you reverse the contents of array A, that array would become: {9, 0, 2, 8, 5, 4, 2, 1}. So, you will display the array A before you reverse the content and after you reversed the contents.

Code Answer:-

Test cases:

32 F is 0 C
212 F is 100 C

Activity 11-1 - Two Dimensional Arrays

Two dimensional arrays are defined in a similar way as the 1-D arrays. Imagine a container with two columns of partitions that one can use to store pills. My grandmother used to have one of those. She had to take two pills a day. So, we would load the container for her every Sunday night and she was good to go for the rest of the week. She only had to know where the pills for each day were and which one of the pills was a day pill and which one was night pill. We would tell her that the left ones were day pills and the right ones were night pills. The container had 7 rows for 7 days of the week and two columns one for the day and the other for the night. If we would view the container as a 2-D array, we would define it as: pill container[7][2].

Here we use two brackets; one to define the number of rows, and the other to define the number of columns. Similarly, a 2-D array of integers with 3 rows and 4 columns would be defined as: int x[3][4];

Now, let me ask you a question. Suppose my grandmother would use the container with 7 rows and 2 columns to keep her pills. Assume that the bottom row is used for Sunday, i.e. row 0. Can you tell me, in terms of rows and columns, where she could get the pill for Tuesday night?
Row: ______ Column:______ OR in an array form: container[___][___]?

Now let's write a program that uses a 2-D arrays.

Code:-

```
//P10_3.cpp - This program will ask a runner for her/his fastest 5 times for 6
// different distances and will display them using a 2-D array.
#include<iostream>
using namespace std;
int find_distance(int j);   //a function that returns a distance based on the choice j
int main( )
{
  int i =0;
  int distance[6];
  double data[6][5];  //This array will keep 30 values in 6 rows and 5 columns
                              // 6 events and 5 times for each one of the events
  for(int j = 0; j < 6; j++)
  {
      distance[j] = find_distance(j);
     cout << "\nEnter 5 of your best running times for \n " <<  distance[j] << " m \n";
     for(i = 0;  i < 5; i++)
    {
      cout << "Enter a time \n";
      cin >> data[j][i];
    }
  }
  cout << "Here is your best 5 times: ";
  for(j = 0;  j < 6; j++)
  {
      cout << "\nDistance : " << distance[j] << " m \n";
      for(i = 0; i < 5; i++)
      {
       cout << data[j][i] << "\t";
      }
      cout << endl;
  }
  return 0;
}
int find_distance(int j)
{
     switch (j)
     {
           case 0: // 100 meter
                return 100;
           break;
          case 1: // 150 meter
                return 150;
           break;
          case 2: // 200 meter
                return 200;
           break;
          case 3: // 400 meter
                return 400;
           break;
          case 4: // 500 meter
                return 800;
           break;
          default: // 1600 meter
            return 1600;
   }
}
```

In the above program, we can access the 3rd time of the 4th event (400 m) in:

```
data[3][2];  // note that the 3rd time is stored in column with index 2
             // and the 4th event is stored in row with index 3
```

The 4th event was 400 meter.
To access the 5 times for 150 m event, we can use:
data[1][0], data[1][1], data[1][2], data[1][3], data[1][4]
Or use a *for* loop to access them:

```
for(i = 0; i < 5; i++)
   data[1][i];
```

Exercise 11.1:-

Modify the above program such that it finds the best, the worst, and the average time for each of the six events. The program should display, for each event, all five times, the worst, the best, and the average.

Code Answer:-