## *Introduction­*

Most of the data encountered in everyday life occurs in groups rather than in individual pieces. Data items tend to be collected into conceptual units with names assigned to them. In the structured data types lab we examined two *heterogeneous* data types, the **struct** and the **class** used to represent groups of data. These two data types were used to store many data items of different types as a conceptual unit. For example, the struct or class could be used to store information about a student -- like name, classification, etc..

In this lab, we will introduce the **array**, a *homogeneous*  data type also used to represent a group of data. This data type might be used to store many items of the same type as a conceptual unit. For example, all of the scores made on a test by students in a biology class might be stored in an array, or the names of students in a biology class might be stored in an array.

***What is an array?***

To understand what an array is, we first review what a variable is. A variable is a symbolic name for a specific location in main memory. For example, the statements:

**int x;**

**x = 25;**

instruct the computer to set aside a memory location that will hold an integer, giving the memory location the symbolic name ***x***.  The second statement instructs the computer to store the integer 25 in the memory location with the symbolic name ***x.***

How can we store twenty integers in memory? We could dream up twenty different symbolic names, such as ***x1, x2, x3, ..., x20***, for twenty places in memory. Obviously, this is not convenient. C++ has a data type called an **array** that addresses this particular problem.

An array is a group of locations in memory such that

1. the locations are usually consecutive,
2. all of these locations will contain the **same type** of data, and
3. all of these locations are referenced by the same symbolic name.

Arrays are used anytime we wish to store and manipulate more than a few pieces of data of the same type. Consider the following examples:

* Suppose that we wish to compute the average pay for 30 people. We do **NOT** require an array for this application, since we do not need to remember the data for all 30 people. In this problem we input data for one person and add it to an accumulator. We can use the same variable for the next person, since we no longer need to remember data for the previous person.
* Suppose we wish to print out 30 scores in reverse order. We need an array for this application because we **must** remember all 30 numbers in order to print them out in reverse.

***Initialize an Array at Declaration Time***

An array can be initialized in the declaration by writing a comma-separated list of values enclosed in braces following an equal sign.

**C++ Syntax of an array declaration with initialization:**

**DataType   ArrayName [numberOfElements] = {val1, val2, ...};**

* There must be at least one initial value between the braces.
* If too many initial values are specified, you get a syntax error message.
* If too few are specified, the remaining array elements are initialized to zero.

**Example:**

int scores[6] = {98, 76, 68, 44, 99, 100};

Although this looks like an assignment, assignment statements with arrays are **NOT** allowed, and this syntax is legal only in a declaration.

***Accessing an Array Element***

You may need to access an array element for

* assigning a value to an array element,
* reading a value to an array element, or
* using the value of an array element for printing or copying the value.

In order to access/store information in an array, one must:

1. Specify the name of the array and
2. Specify exactly which location in the array is to be used to access/store the information.

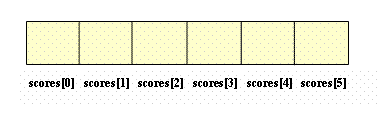
We specify locations in the array by using **an index**.

**ArrayName [ index ]**

The index can be a constant, a variable, or an expression which results in ranges from 0 to the number of elements -1.

**Example:** An array declaration:

**int scores[6];**

creates an integer array scores with six elements as following:    
Valid references are scores[0], scores[1], scores[2], scores[3], scores[4], and scores[5]. Any other references to the score array will be invalid.

## *Assign a Value to an Array Element*

**Example:** Reconsider the array *scores* declared before. We can assign a value to an element of the array score by using an assignment statement such as scores[i]=20; It means that look at the location designated by *i* in the array list and store the value 20 in this position. The index of an array element can be a constant, variable, or expression as shown below:

**scores[5] = 100; //index = 5**

**scores[x] = 80; //index = the value of the variable x**

**scores[3\*x] = 75; //index = three times x**

To determine valid values for the index, reconsider the definition of the array:

**int scores[MAX\_SCORES];**

The upper limit of the index is calculated from the value in brackets, namely *MAX\_SCORES - 1.*In C++, the lower limit is **always 0.** Thus the index may vary from 0 to *MAX\_SCORES - 1*. Thus the first memory location assigned to ***scores*** is designated by the first index value 0, the second by index value 1, and so on, up to the last one which is designated by the index value *MAX\_SCORES - 1*. If, instead, *scores* were declared as follows: 

**int scores[MAX\_SCORES + 1];**

then we could access scores[0], scores[1],..., scores[MAX\_SCORES].

***Accessing Whole Array***

Usually a loop of some kind is used when one is working with an array. Since the number of elements is known, a for loop or a count-controlled while loop is appropriate to use for accessing a whole array. Typical situations for accessing whole array is

* initializing an array,
* filling an array with data,
* accessing an array for printing, copying, or reversing the values in the array.

**Example 1:** The following code could be used to initialize all the elements in the array anArray with 100 elements to zeros.

**int anArray[100];**

**for (int i = 0; i < 100; i++)**

**{**

**anArray[i] = 0;**

**}**

**Example 2:** The following code could be used to read in a (howMany) number of values from the keyboard and store them into the array scores.

**//read the number of scores**

**cout << "How many scores?\n";**

**cin >> howMany;**

**//now read the scores**

**for (int i = 0; i < howMany; i++)**

**{**

**cout << "Input scores[" << i << "]: ";**

**cin >> scores[i];**

**}**

Note that the for loop iteratively reads a value from the keyboard and stores the value into the corresponding element in the array.

## *Aggregate Array Operations*

Some programming language allow aggregate operation on arrays, but C++ does not. If array1 and array2 are declared as

**int array1[100];**

**int array2[100];**

followings are **not** allowed.

**array1 = array2; // assignment not allowed**

**if (array1 == array2) // comparison not allowed**

**cin >> array1; // aggregate input not allowed**

**cout << array2; // aggregate output not allowed**

**array1 = array1 + array2 // arithmetic not allowed**

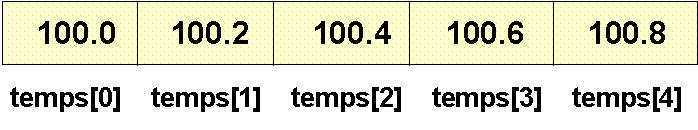
**return array1; // use as a function's return value not allowed**

Passing an array as an argument provides the function access to the array. However, only call-by-reference is allowed.

## *Computation on Arrays*

Consider the following array declaration and the contents of the array shown in the figure:

**float temps[5] = {100.0, 100.2, 100.4, 100.6, 100.8};**



Following statements compute the average value in the temps array:

**int sum = 0;**

**sum += temps[0];**

**sum += temps[1];**

**sum += temps[2];**

**sum += temps[3];**

**sum += temps[4];**

**average = (float) sum / 5;**

The above statements can be rewritten using a for loop structure. The following code performs the same task:

**int sum = 0;**

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

**{**

**sum += temps[i];**

**}**

**average = (float) sum / 5;**

Notice that the value of i is used as the index of the array element during the iteration. The initialization of the sum variable has to be done before the iteration starts, while the computation of the average is done after the iteration.