Lecture 4.3

Topics

1. Derived Data Types – Arrays
2. Data Storage and Initialization – Automatic, Global, Static

---

1. Array Initialization and Access

Arrays are declared and should be defined (i.e., assigned with known values) before using. The declaration asks the system to allocate **one contiguous block of memory** that has the size of the array times (i.e., the number of elements) the size of the array type.

The general syntax is given as follows,

```
DataType arrayName[ CONSTANT_SIZE ]; //Local array
```

Or, for dynamic array,

```
DataType* pointerToArray;
pointerToArray = new DataType[ integralExpression ];
```

Thus, the total number of bytes for some array is found as follows,

Local Array: `CONSTANT_SIZE * sizeof( DataType )`

Dynamic array: `integralExpression * sizeof( DataType )`

With this given memory, values then can be stored in the array and the array is subsequently defined.

Let's look at the value assignment or initialization again.

1.1 Initialization

For local arrays, the initialization can be done in several ways as follows,

- They can be left undefined,
- They are completely specified,
- They are partially defined, or
- They are forced with completion.

For examples,

```
int iA1[3]; /*No initialization: values are unknown*/

int iA2[4] = {1, 2, 3, 4};  /* Complete initialization
                                    for all four members*/

int iA3[] = {5, 6, 7};  /* Complete initialization for
                              an array of three members*/

int iA4[5] = {8, 9};  /* Partially initialized for the
                            first two members; the remaining
                            three members will have value 0*/

int iA5[10] = {0};  /* Partially initialize the first member
                         with 0 and fill in the remaining members
                         with 0's. That means all values are 0*/
```

1.2 Accessing Arrays

- Array members can be accessed by the array name and a valid index.
- The index should be in the range from **0** to **(CONSTANT_SIZE – 1)**.
- Any index that is outside of this range may produce an unknown value and may also alter data belonging to other objects or programs.
- C++ has no index or boundary checking. The user must keep track of this range of indexes and make sure the individual index is valid.
- Array members can be either **lvalue** or **rvalue**. That means one can either retrieve an array member value or assign new value to an array member.

For examples,

```
iA1[0] = 100;
iA2[0] = iA1[0];
```

Note that the brackets are actually called the **index** operator. It has very high precedence and thus its evaluation is almost immediate in many expressions.

1.3 Array Example – Finding The Smallest Value

In previous problem of finding the smallest value from a sequence of numbers being entered (one as a time), the values were being discarded and only the smallest value of interest was kept. What if one would like to find out the average value of the same sequence? Largest value? etc.

Certainly, one can modify code to progressively deal with one value at a time. This is possible but not that pretty, and the previous data would be lost forever while the program is running.

Another possible approach is to use an array to store the values and then perform the operations on the array. This approach has its own disadvantage because the number of data points must be known before creating the array.

Assuming that the array size is given, consider the finding smallest value problem in the code below.

Example 1

```
/**
 *Program Name: cis25L0431.cpp
 *Discussion:   Array Application - Finding Smallest Value
 */
#include <iostream>
#include <cfloat>
using namespace std;

#define ARRAY_SIZE 10

int main() {
  int iSize = ARRAY_SIZE;
  double dArray[ARRAY_SIZE];
  int iDataSize;
  int i;
  double dSmallestVal;
  int iQuit;

  //Assigning DBL_MAX to all elements

  for (i = 0; i < ARRAY_SIZE; i++) {
```

```
    dArray[i] = DBL_MAX;
  }

  //Receiving actual data
  cout << "\nWill work with at most "
       << ARRAY_SIZE << " numbers.\n" << endl;

  cout << "\nEnter the number of data points : " << flush;
  cin >> iDataSize;

  for (i = 0; i < iDataSize; i++) {
    cout << "\nEnter value for number -- " << i << " : ";
    cin >> dArray[i];
  }

  dSmallestVal = DBL_MAX;
  for (i = 0; i < iDataSize; i++) {
    if (dSmallestVal > dArray[i]) {
      dSmallestVal = dArray[i];
    }
  }

  cout << "\nThe smallest value is " << dSmallestVal << endl;

  cout << "\nEnter a number to quit: ";
  cin >> iQuit;

  return 0;
}
```

OUTPUT

```
Will work with at most 10 numbers.


Enter the number of data points :
5

Enter value for number : 0 : 2

Enter value for number : 1 : 8

Enter value for number : 2 : -5

Enter value for number : 3 : 6

Enter value for number : 4 : 9

The smallest value is -5

Enter a number to quit: q
```

How do we put the above code in a function?

2. Data Storage and Initialization – Automatic, Global, Static

The initialization of an array and variable depends on how the array/variable was stored. There are three kinds of storage in C++ in which a variable's value can be stored under.

2.1 Automatic (Local) Storage

An automatic or local storage will be given to variables and arrays that were defined inside functions (including the formal arguments). These variables and arrays will be available only within the functions and they will no longer exist as soon as the functions are done with their execution.

Example

```
int main() {
  int iA1[3];
  int iA2[2] = {1, 2};

}
```

2.2 External (Global) Storage

External (global) storage is given to variables and arrays that were defined outside of any functions. Global variables and arrays are available everywhere after they are defined. They last through out the program execution.

Global variables and arrays are initialized with **`zero(s)`** unless other specific values were given with the declarations.

Example

```
int common;
const int MAXSIZE = 5;
int iExtArray[3] = {1, 2, 3};

int main() {
  int iA1[3];
  int iA2[2] = {1, 2};

  return 0;
}
```

2.3 Static Storage

A declaration that was modified with the **`static`** keyword will force the system to provide a **static storage**. This static storage reserves a common block of memory for a static variable and its value will be shared by any invocation that would require access to this static member.

Static variables are defined inside a method and will have the values initialized to **zero(s)** unless other values were given.

Example

```
int test() {
  static int iCommonValue;
  int iATemp[2] = {1, 2};

  return 0;
}
```

2.4 Example – Bubble Sort

```
/**
```

```
 *Program Name: cis25L0432Sorting.cpp
 *Discussion:   Array Application - Sorting
 */
#include <iostream>
using namespace std;

#define ARRAY_SIZE 10
#define MAX_ARY_SIZE 15

int main() {
  int i, j;
  int iTemp;
  int iSize;
  int iAry[MAX_ARY_SIZE] = {89, 72,  3, 15, 21,
                            57, 61, 44, 19, 98,
                            5, 77, 39, 59, 61};


  iSize = MAX_ARY_SIZE;
  for (i = 0; i < iSize; i++) {
    for (j = i + 1; j < iSize; j++) {
      if (iAry[i] < iAry[j]) {
        iTemp = iAry[ i ];
        iAry[ i ] = iAry[ j ];
        iAry[ j ] = iTemp;
      }
    }
  }

  return 0;
}
```