C++ Lecture 9

* Programming with Arrays, Multidimensional Arrays
* CIS 251 • Shelby-Hoover Campus

Partially-Filled Arrays

* An array’s size must be set to a constant or literal value at the beginning of the program
* Predicting exactly how many values a user will enter in a sentinel-terminated list is virtually impossible
* Rule of thumb:
  + Declare the array to hold at least as many elements as you’d expect the longest list to require
  + In the program, use a variable to track how many elements are actually used
  + When looping through the array, keep the subscript less than the number of elements used instead of less than the size of the array itself

Partially-Filled Example

* Printing a series of integers after entry (adapted from the function fill\_array in Display 7.9):  
    
  const int SIZE = 20;  
  int values[SIZE], current, used = 0, sub;  
    
  cout << "Enter up to " << SIZE << " positive integers." << endl;  
  cout << "Enter a negative value to stop." << endl;  
    
  cin >> current;  
  while ((current >= 0) && (used < SIZE))  
  {  
   values[used] = current;  
   used++;  
   cin >> current;  
  }  
    
  cout << "Here are the values you entered: ";  
  for (sub = 0; sub < used; sub++)  
   cout << values[sub] << " ";  
    
  cout << endl;

Element Count in Functions

* The example functions with an array parameter in the previous lecture had a second parameter for the size of the array
* When dealing with a partially-filled array, a function may need to know the number of elements that contain good values, not the size of the array
* Some functions may need to know both: a function that fills an array with values needs a value parameter for the size of the array and a reference parameter for the number of elements filled (unless that value is returned)

Searching an Array

* An array can hold a set of values among which the program can search for a specific value
* If an array is unsorted, the search must begin at the first element in the array
* Compare each element to the value to be found
* If the program finds the element, the process terminates
* If the program reaches the end of the array (or the end of the filled elements) without finding a match, the value does not exist in the array

Array Search Code

* The syntax to search the array can be placed in its own function (Display 7.10) that returns the subscript of where the item is found (or -1 if not found)
* A bool variable serves as a flag terminating the loop if the element matches the value; it also allows the function to determine whether to return the index or the error value  
    
  int search(const int a[], int number\_used, int target)  
  {  
   int index = 0;  
   bool found = false;  
   while ((!found) && (index < number\_used))  
   {  
   if (target == a[index])  
   found = true;  
   else  
   index++;  
   }  
    
   if (found)  
   return index;  
   else  
   return -1;  
  }

Sorting an Array

* There are several algorithms that can be used to sort the elements in an array
* The textbook uses the selection sort algorithm
  + Suppose that the variable used stores the number of elements in the array that contain values
  + First, find the smallest item in the entire array (from subscript 0 to used – 1), and swap it with the value currently at subscript 0
  + Next, find the smallest item from subscript 1 to used – 1, and swap it with the value currently at subscript 1
  + Overall: for each subscript from 0 to used – 2 (the largest value will make its way to subscript used – 1 through other swaps), find the smallest item from that subscript to used – 1, and swap it into that subscript
* Implementation Example: Display 7.12

Sorting: A Swap Function

* The following function swaps the values in the arguments (which could be array elements) to two reference parameters
* Swapping requires a third temporary variable to hold one of the values as it is swapped  
    
  void swapValues(int& v1, int& v2)  
  {  
   int temp;  
   temp = v1;  
   v1 = v2;  
   v2 = temp;  
  }

Sorting: Find the Minimum

* This algorithm must find the subscript of the smallest value that has yet to be sorted to know which two elements need to swap values
* This function assumes that the first element could be the minimum, keeping the subscript of any smaller value that is encountered through the end of the used array elements  
    
  int indexOfSmallest(const int a[], int start, int used)  
  {  
   int minIndex = start;  
    
   for (int index = start + 1; index < used; index++)  
   {  
   if (a[index] < a[minIndex])  
   minIndex = index;  
   }  
    
   return minIndex;  
  }

Sorting: The Entire Array

* For each element except the last one, the program must find the subscript of the minimum value and swap its value with that of the first unsorted element
* A function could contain the loop to call these functions accordingly  
    
  void sort(int a[], int used)  
  {  
   int indexOfNextMin;  
    
   for (int index = 0; index < used – 1; index++)  
   {  
   indexOfNextMin = indexOfSmallest(a, index, used);  
   swapValues(a[index], a[indexOfNextMin]);  
   }  
  }

Multidimensional Arrays

* Arrays with one size (in a single set of square brackets after the name) are considered one-dimensional; they can be visualized as a row of boxes, with each box being a variable
* A program can contain arrays with multiple dimensions
  + Each dimension’s size should be written in a separate set of square brackets
  + Each element should be accessed with a subscript for each dimension
* A two-dimensional array is like a table with rows and columns
  + Example: int examGrades[30][4]; /\* a total of 120 array elements – 30 students, 4 exam grades each \*/
  + examGrades[0][0] = 92; /\* grade for first student, first exam \*/
  + examGrades[29][3] = 75; /\* grade for last student, last exam \*/

2D Arrays and Functions

* An array parameter for a one-dimensional array in a function uses empty square brackets (the size is omitted)
* With multidimensional array parameters, every size except that of the first dimension must be specified in the square brackets
* A program can contain a mixture of one-dimensional and multidimensional arrays
* The sizes can be declared as global constants for consistency
* Example: Display 7.13  
    
  #include <iostream>  
  #include <iomanip>  
  using namespace std;  
  const int NUM\_STUDENTS = 4, NUM\_QUIZZES = 3;  
    
  void computeStudentAvg(const int grade[][NUM\_QUIZZES], double stuAvg[]);  
  void computeQuizAvg(const int grade[][NUM\_QUIZZES], double quizAvg[]);  
  void display(const int grade[][NUM\_QUIZZES], const double stuAvg[],   
   const double quizAvg[]);

2D Arrays: Declare, Fill

* Each student is in a different row, and each quiz is in a different column:  
    
  int main()  
  {  
   int grade[NUM\_STUDENTS][NUM\_QUIZZES];  
   double stuAvg[NUM\_STUDENTS];  
   double quizAvg[NUM\_QUIZZES];  
    
   for (int stuNum = 1; stuNum <= NUM\_STUDENTS; stuNum++)  
   {  
   cout << "Student #" << stuNum << endl;  
   for (int qNum = 1; qNum <= NUM\_QUIZZES; qNum++)  
   {  
   cout << "Enter grade for quiz " << qNum << ": ";  
   cin >> grade[stuNum – 1][qNum – 1];  
   }  
   cout << endl;  
   }  
    
   computeStudentAvg(grade, stuAvg);  
   computeQuizAvg(grade, quizAvg);  
   display(grade, stuAvg, quizAvg);  
   return 0;  
  }

2D Arrays: Student Average

* To determine a student’s average, sum the values in one row, then divide by the number of quizzes
* The students’ averages are stored in a separate one-dimensional array:  
    
  void computeStudentAvg(const int grade[][NUM\_QUIZZES], double stuAvg[])  
  {  
   // Outer loop: students  
   for (int stuNum = 1; stuNum <= NUM\_STUDENTS; stuNum++)  
   {  
   double sum = 0;  
    
   // Inner loop: quizzes  
   for (int quizNum = 1; quizNum <= NUM\_QUIZZES; quizNum++)  
   sum = sum + grade[stuNum - 1][quizNum - 1];  
    
   stuAvg[stuNum - 1] = sum / NUM\_QUIZZES;  
   }  
  }

2D Arrays: Quiz Average

* To determine a quiz’s average, sum the values in one column, then divide by the number of students
* The quizzes’ averages are stored in a separate one-dimensional array:  
    
  void computeQuizAvg(const int grade[][NUM\_QUIZZES], double quizAvg[])  
  {  
   // Outer loop: quizzes  
   for (int quizNum = 1; quizNum <= NUM\_QUIZZES; quizNum++)  
   {  
   double sum = 0;  
    
   // Inner loop: students  
   for (int stuNum = 1; stuNum <= NUM\_STUDENTS; stuNum++)  
   sum = sum + grade[stuNum - 1][quizNum - 1];  
    
   quizAvg[quizNum - 1] = sum / NUM\_STUDENTS;  
   }  
  }

2D Arrays: Display Results

* Use one loop to display each student’s quiz grades, with the average at the beginning
* Use a separate loop to display the average for each quiz  
    
  void display(const int grade[][NUM\_QUIZZES], const double stuAvg[],   
   const double quizAvg[])  
  {  
   cout.setf(ios::fixed);  
   cout.setf(ios::showpoint);  
   cout.precision(1);  
   cout << setw(10) << "Student" << setw(5) << "Avg"   
   << setw(15) << "Quizzes\n";  
    
   for (int stuNum = 1; stuNum <= NUM\_STUDENTS; stuNum++)  
   {  
   cout << setw(10) << stuNum << setw(5) << stuAvg[stuNum – 1]   
   << " ";  
   for (int quizNum = 1; quizNum <= NUM\_QUIZZES; quizNum++)  
   cout << setw(5) << grade[stuNum – 1][quizNum – 1];  
   cout << endl;  
   }  
    
   cout << "Quiz averages = ";  
   for (int quizNum = 1; quizNum <= NUM\_QUIZZES; quizNum++)  
   cout << setw(5) << quizAvg[quizNum – 1];  
   cout << endl;  
  }