C++ Lecture 8

* Array Concepts, Basic Array Syntax, and Arrays Used with Functions
* CIS 251 • Shelby-Hoover Campus

Variables Reviewed

* A variable must be declared (with a name and data type) before it can be used in a program
* For a program with few variables, this is usually not a problem
* Having unique names for a large number of variables is problematic
  + The program would contain a large number of declarations
  + Selecting the correct variable may require complex decision logic

Arrays

* An **array** allows a programmer to declare a set of variables using a single statement
  + The **base type** of the array determines the type of each variable in the array (all variables in an array have the same type)
  + The variables in an array all have the same name associated with them
  + The **declared size** of the array (how many variables it contains) must be an integer constant or literal value
* The variables in an array are called **subscripted variables**, **indexed variables**, or **elements**
* Each element is accessed using a position value known as a **subscript** or **index**
  + In C++, the subscript of the first element is zero
  + The subscript of the last element is one less than the array’s declared size

Declaring an Array

* Pattern: baseType arrayName[size];
* Example: int score[5];
  + An array of five int variables
  + The first element in the array is at subscript 0
  + The last element in the array is at subscript 4
* An array may be declared in the same statement as individual variables:  
    
   int next, score[5], max;
* In order to maintain the size of the array for later processing, you may declare a named constant:  
    
   const int SIZE = 5;  
   int score[SIZE];

Using Array Elements

* Access each array element individually by putting the subscript of the desired element in square brackets after the name
  + This is NOT the same as the declaration statement
  + Do NOT put the size (by itself) in the square brackets in any statement dealing with the array other than the array declaration
* Each array element can be used as any variable of the array’s base type can be used:  
    
   score[0] = 88;  
   score[1] = score[0] + 7;  
   cin >> score[2];

Variables as Subscripts

* A variable (or expression involving a variable) can be used as a subscript as long as it is an integer and it stays within the bounds for the array (0 to size – 1)
* The variable could also be a loop control variable, applying the statements in the loop body to each element in the array:  
    
  const int STUDENTS = 12;  
  int stuCount, stuGrades[STUDENTS];  
    
  cout << "Enter twelve scores: ";  
    
  for (stuCount = 0; stuCount < STUDENTS; stuCount++)  
   cin >> stuGrades[stuCount];

Array Loop Example

* Note that when prompting the user for each value separately, the count displayed to the user may need to be one greater than the index (the user starts counting at one rather than zero):  
    
  #include <iostream>  
  using namespace std;  
    
  int main()  
  {  
   const int STUDENTS = 12;  
   int stuCount, stuGrades[STUDENTS];  
    
   for (stuCount = 0; stuCount < STUDENTS; stuCount++)  
   {  
   cout << "Enter score #" << stuCount + 1 << ": ";  
   cin >> stuGrades[stuCount];  
   }  
    
   cout << "Here are the scores you entered: " << endl;  
    
   for (stuCount = 0; stuCount < STUDENTS; stuCount++)  
   cout << "Score #" << stuCount + 1 << ": "  
   << stuGrades[stuCount] << endl;  
    
   return 0;  
  }

Arrays in Memory

* When you declare an array, the array variable stores the **address** (location in memory) of the first element of the array
* The subscript then describes how far beyond the first address to look for an element (the first element is 0 element-sized locations past the starting address)
* Be careful not to use a subscript that is **out of bounds** (**out of range**) for an array
  + C++ will attempt to use the memory location even if the subscript is out of bounds
  + This may cause a runtime error by damaging memory allocated to another variable

Initialization & Assignment

* All of the elements in an array can be initialized at once using an initialization list when the array is declared
* The values to be assigned to the array elements are separated by commas in a set of curly brackets:  
    
   int exams[4] = { 95, 88, 92, 90 };
* The size of the array may be omitted if the array should be sized exactly the same as the number of values in the initialization list:  
    
   int exams[] = { 95, 88, 92, 90 };
* You may choose only to initialize the first few elements:  
    
   int exams[4] = { 95, 88 };
* You cannot skip an element in an initialization list; values are assigned to the elements in the order in which they are listed
* You cannot perform this type of assignment at any other point in the program

Elements as Arguments

* Each element of an array can be used as an argument to a single parameter (pass by value or by reference) of the same type
  + The header of the function does not need to change
  + The call to the function should include the subscript of the element to be sent
* Such a function could be called in a loop to process each element

Elements Argument Example

* Example function definition (notice that it does not have any array-related syntax):  
    
  int curveGrade(int oneGrade)  
  {  
   return (oneGrade + 10);  
  }
* Example function call (with the subscript of the array element included as part of the argument):  
    
  int main()  
  {  
   const int SIZE = 5;  
   int grades[SIZE] = { 77, 83, 72, 65, 84 };  
    
   for(int x = 0; x < SIZE; x++)  
   {  
   cout << "Student #" << (x + 1) << ":\n";  
   cout << "Grade before curve: " << grades[x] << endl;  
   cout << "Grade after curve: "  
   << curveGrade(grades[x]) << endl << endl;  
   }  
    
   return 0;  
  }

Array Parameters

* An entire array can be sent as an argument to an **array parameter**
  + The parameter in the function header must include empty square brackets (don’t include the size there)
  + A second int parameter may be used for the size
  + The argument in the function call should be the array name by itself (no brackets)
* An array parameter works similar to a reference parameter
  + The array variable stores the address of the first element
  + When an array is an argument to a parameter, the argument’s address is copied to the parameter
  + Changes in the array parameter apply to the argument

Array Parameter Example

* Function Definition (notice the empty square brackets after the array parameter name):  
    
  void getGrades(int gradeArray[], int size)  
  {  
   cout << "Enter " << size << " grades: " << endl;  
   for (int sub = 0; sub < size; sub++)  
   cin >> gradeArray[sub];  
  }
* Function Call: The address of the array grades is also used as the address of the array parameter gradeArray in the function getGrades (notice the lack of square brackets on the argument)  
    
  const int SIZE = 10;  
  int grades[SIZE];  
  getGrades(grades, SIZE);

Constant Array Parameters

* You may not want a function to have the ability to change the value of any of the array elements
* The key word const can be used to “lock” the contents of an array parameter’s elements, making it a **constant array parameter**:  
    
  void displayGrades(const int gradeArray[], int size)  
  {  
   int index;  
   cout << "The grades are: ";  
   for (index = 0; index < size; index++)  
   cout << gradeArray[index] << " ";  
   cout << endl;  
  }
  + The function still receives the address of the array argument (not a copy), but it cannot make any changes
  + The compiler will generate an error message if any changes are attempted
  + If a function with a constant array parameter passes the parameter as an argument to another function, the parameter in the second function must also be a constant array parameter (see the Pitfall section in section 7.2)

Array Program Example

* The Case Study (end of section 7.2) walks through the process of writing a program to draw a graph indicating how many thousands of units are produced by each of four plants (production levels are stored in an array)
* Directives, a global constant, and prototypes:

#include <iostream>  
#include <cmath>  
using namespace std;  
  
const int NUMBER\_OF\_PLANTS = 4;  
void input\_data(int a[], int last\_plant\_number);  
void scale(int a[], int size);  
void graph(const int asterisk\_count[], int last\_plant\_number);  
void get\_total(int& sum);  
int round(double number);  
void print\_asterisks(int n);

* The main function, with calls to three functions that accept the array and its size as arguments:

int main()  
{  
 int production[NUMBER\_OF\_PLANTS];  
  
 cout << "This program displays a graph showing\n"  
 << "production for each plant in the company.\n";  
  
 input\_data(production, NUMBER\_OF\_PLANTS);  
 scale(production, NUMBER\_OF\_PLANTS);  
 graph(production, NUMBER\_OF\_PLANTS);  
  
 return 0;  
}

* The function input\_data gets input for all plants by calling get\_total in a loop (notice the adjustment between the displayed value plant\_number and the subscript plant\_number – 1):

void input\_data(int a[], int last\_plant\_number)  
{  
 for (int plant\_number = 1; plant\_number <= last\_plant\_number;   
 plant\_number++)  
 {  
 cout << endl << "Enter production data for plant number "  
 << plant\_number << endl;  
 get\_total(a[plant\_number – 1]);  
 }  
}

* The function get\_total gets input to total the units produced by each department in a plant (the argument to this function is a single array element, but since the parameter is pass by reference, the function is allowed to change the value of the argument):

void get\_total(int& sum)  
{  
 cout << "Enter number of units produced by each department.\n"  
 << "Enter a negative number when you are finished.\n";  
  
 sum = 0;  
 int next;  
 cin >> next;  
 while (next >= 0)  
 {  
 sum = sum + next;  
 cin >> next;  
 }  
  
 cout << "Total = " << sum << endl;  
}

* The function scale converts the value in each array element from single units to thousands of units:

void scale(int a[], int size)  
{  
 for (int index = 0; index < size; index++)  
 a[index] = round(a[index]/1000.0);  
}

* The function round makes sure the production level is rounded to the nearest thousand:

int round(double number)  
{  
 return static\_cast<int>(floor(number + 0.5));  
}

* The function graph ensures that each plant’s production level is displayed with labels (notice that the first parameter is a constant array parameter):

void graph(const int asterisk\_count[], int last\_plant\_number)  
{  
 cout << "\nUnits produced in thousands of units:\n";  
 for (int plant\_number = 1; plant\_number <= last\_plant\_number;  
 plant\_number++)  
 {  
 cout << "Plant #" << plant\_number << " ";  
 print\_asterisks(asterisk\_count[plant\_number – 1]);  
 cout << endl;  
 }  
}

* The function print\_asterisks prints enough asterisks to represent the thousands of units produced by one plant:

void print\_asterisks(int n)  
{  
 for (int count = 1; count <= n; count++)  
 cout << "\*";  
}