Lab6 Array I

Objectivities

1. Using `rand` to generate random numbers and using `srand` to seed the random-number generator.
2. Declaring, initializing and referencing arrays.
3. The follow-up questions and activities also will give you practice:
4. Remembering that arrays begin with subscript 0 and recognizing off-by-one errors.
5. Preventing array out-of-bounds errors.
6. Using two-dimensional arrays.

Experiment 1

Description of the Problem

Write a program that simulates the rolling of two dice. The program should call `rand` to roll the first die, and should call `rand` again to roll the second die. The sum of the two values should then be calculated. [Note: Each die has an integer value from 1 to 6, so the sum of the two values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 being the least frequent sums.] Figure L 7.1 shows the 36 possible combinations of the two dice. Your program should roll the two dice 36,000 times. Use a one-dimensional array to tally the numbers of times each sum appears. Print the results in a tabular format. Also, determine if the totals are reasonable (i.e., there are six ways to roll a 7, so approximately one sixth of all the rolls should be 7).

<table>
<thead>
<tr>
<th>Sum</th>
<th>Total</th>
<th>Expected</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1000</td>
<td>2.778%</td>
<td>2.778%</td>
</tr>
<tr>
<td>3</td>
<td>1958</td>
<td>5.556%</td>
<td>5.439%</td>
</tr>
<tr>
<td>4</td>
<td>3048</td>
<td>8.333%</td>
<td>8.467%</td>
</tr>
<tr>
<td>5</td>
<td>3979</td>
<td>11.111%</td>
<td>11.053%</td>
</tr>
<tr>
<td>6</td>
<td>5007</td>
<td>13.889%</td>
<td>13.908%</td>
</tr>
<tr>
<td>7</td>
<td>6087</td>
<td>16.667%</td>
<td>16.908%</td>
</tr>
<tr>
<td>8</td>
<td>4996</td>
<td>13.889%</td>
<td>13.878%</td>
</tr>
<tr>
<td>9</td>
<td>3971</td>
<td>11.111%</td>
<td>11.031%</td>
</tr>
<tr>
<td>10</td>
<td>2996</td>
<td>8.333%</td>
<td>8.322%</td>
</tr>
<tr>
<td>11</td>
<td>2008</td>
<td>5.556%</td>
<td>5.578%</td>
</tr>
<tr>
<td>12</td>
<td>950</td>
<td>2.778%</td>
<td>2.639%</td>
</tr>
</tbody>
</table>

Problem-Solving Tips

1. Remember that array subscripts always begin with zero. This is also true for each dimension of a multiple-subscripted array (which this lab does not use).

2. The actual percentage is the likelihood, based on the results of your program, that a dice roll produced a certain result. In other words, if you roll the dice 36,000 times the actual percentage will be the (number of times a result occurred / 36000) * 100.
3. The expected percentage is the statistical probability that a dice roll will produce a certain result. This can be calculated from the diagram “36 possible outcomes of rolling two dice,” shown in the problem description. For example, there is only one combination that will produce the sum of 2 and there are 36 total combinations. Therefore, the expected percentage of rolling a 2 is 1/36 or 2.778%.

Follow-Up Questions and Activities
1. What happens if the elements of array sum are not initialized to zero? Try to run the program without initializing the array. Show your results.

2. Modify the program to use a two-dimensional array similar to the diagram in Figure L.7.1. Now, rather than counting the number of times each sum appears, increment the correct cell in the array. Print this array with the number of times each dice combination occurred. A sample output may look like the following:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1011</td>
<td>971</td>
<td>1027</td>
<td>1025</td>
<td>971</td>
<td>1015</td>
</tr>
<tr>
<td>2</td>
<td>1013</td>
<td>968</td>
<td>990</td>
<td>968</td>
<td>1081</td>
<td>993</td>
</tr>
<tr>
<td>3</td>
<td>993</td>
<td>1014</td>
<td>983</td>
<td>973</td>
<td>1019</td>
<td>977</td>
</tr>
<tr>
<td>4</td>
<td>980</td>
<td>1004</td>
<td>974</td>
<td>1022</td>
<td>946</td>
<td>1046</td>
</tr>
<tr>
<td>5</td>
<td>1003</td>
<td>1021</td>
<td>1019</td>
<td>979</td>
<td>1004</td>
<td>1056</td>
</tr>
<tr>
<td>6</td>
<td>1026</td>
<td>1015</td>
<td>931</td>
<td>989</td>
<td>1014</td>
<td>979</td>
</tr>
</tbody>
</table>

***************

Experiment 2

The following program in this section does not run properly. Fix all the compilation errors so that the program will compile successfully. Once the program compiles, compare the output with the sample output, and eliminate any logic errors that may exist. The sample output demonstrates what the program’s output should be once the program’s code has been corrected.

```cpp
#include <iostream>
using std::cout;
using std::endl;
#include <iomanip>
using std::setw;
#include <ctime>
const int NUM_GRADES = 10;
const int NUM_SUDENTS = 3;
int findHighest( int );
int findLowest( int *);
void printDatabase( const int [][], const char [][ 20 ] );
int main()
{
    int student1[ NUM_GRADES ] = { 0 };
    int student2[ NUM_GRADES ] = { 76, 89, 81, 42, 66, 93, 104, 91, 71, 85, 105 };
    ```
int student3[NUM_GRADES] = { 65, 69, 91, 89, 93, 72, 76, 79, 99 };  
char names[NUM_SUDENTS][20] = { "Bob", "John", "Joe" }; 

int database[NUM_SUDENTS][NUM_GRADES]; 
int i = 0; 
srand(time(0)); 
// initialize student1 
for (i = 0; i < NUM_GRADES; i++) 
student1[NUM_GRADES] = rand() % 50 + 50; 
// initialize database 
for (i = 1; i < NUM_GRADES; i++) { 
database[0][i] = student1[i]; 
database[1][i] = student2[i]; 
database[2][i] = student3[i]; 
} // end for 
printDatabase(database, studentNames); 
for (i = 0; i < NUM_SUDENTS; i++) { 
cout << studentNames[i] << "'s highest grade is: " << findHighest(student1) << endl 
cout << studentNames[i] << "'s lowest grade is: " << findLowest(database[i]) << endl; 
} // end for 
return 0; 
} // end main 
// determine largest grade 
int findHighest(int[]) 
{ 
int highest = a[0]; 
for (int i = 1; i <= NUM_GRADES; i++) 
if (a[i] > highest) 
highest = a[i]; 

return highest; 
} // end function findHighest 

// determine lowest grade 
int findLowest(int a[]) 
{ 
int lowest = a[0]; 
for (int i = 1; i < NUM_GRADES; i++) 
if (a[i] < lowest) 
lowest = a[i];
return lowest;
// end lowestGrade
// output data
void printDatabase( int a[NUM_GRADES], char names[NUM_SUDENTS] )
{
    cout << "Here is the grade database\n\n"
    << setw( 10 ) << "Name"
    << setw( 4 ) << n
    << endl;
    for ( int i = 0; i < NUM_SUDENTS; i++ )
    {
        cout << setw( 10 ) << names[i]
        << setw( 4 ) << a[i]
        << endl;
    }
}

Sample Output

<table>
<thead>
<tr>
<th>Name</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>56</td>
<td>67</td>
<td>83</td>
<td>81</td>
<td>70</td>
<td>84</td>
<td>94</td>
<td>64</td>
<td>68</td>
<td>86</td>
</tr>
<tr>
<td>John</td>
<td>76</td>
<td>89</td>
<td>81</td>
<td>42</td>
<td>66</td>
<td>93</td>
<td>104</td>
<td>91</td>
<td>71</td>
<td>85</td>
</tr>
<tr>
<td>Joe</td>
<td>65</td>
<td>69</td>
<td>91</td>
<td>89</td>
<td>82</td>
<td>93</td>
<td>72</td>
<td>76</td>
<td>79</td>
<td>99</td>
</tr>
</tbody>
</table>

Bob's highest grade is: 94
Bob's lowest grade is: 56
John's highest grade is: 104
John's lowest grade is: 42
Joe's highest grade is: 99
Joe's lowest grade is: 65
Lab7 Array II

Objectives
This lab was designed to reinforce programming concepts from Chapter 7 of C++ How To Program: Sixth Edition. In this lab, you will practice:
- Sorting data using the bubble sort algorithm.
The follow-up question and activity also will give you practice:
- Optimizing a program to be more efficient.

Experiments
In the bubble sort algorithm, smaller values gradually “bubble” their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom. The bubble sort makes several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.
Write a program that sorts an array of 10 integers using bubble sort.

<table>
<thead>
<tr>
<th>Data items in original order</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 6 4 8 10 12 89 68 45 37</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data items in ascending order</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 4 6 8 10 12 37 45 68 89</td>
</tr>
</tbody>
</table>

Template
```cpp
// Lab 2: bubblesort.cpp
// This program sorts an array's values into ascending order.
#include <iostream>
using std::cout;
using std::endl;
#include <iomanip>
using std::setw;

int main()
{
    const int arraySize = 10; // size of array a
    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
    int hold; // temporary location used to swap array elements
    cout << "Data items in original order\n";
    // output original array
    for ( int i = 0; i < arraySize; i++ )
        cout << setw( 4 ) << a[ i ];
    // bubble sort
    // loop to control number of passes
    /* Write a for header to loop for one iteration less than the size
    of the array */
```
Problem-Solving Tips

1. Each “bubbling” pass through the array brings one element, the $i^{th}$ up to its correct position. This means that the program will require arraySize - 1 passes through the array to sort the entire array.

2. Each bubbling pass will look at each pair of adjacent elements and swap them if they are not already in sorted order.

3. To swap two elements, the value of one element will have to be stored in a temporary storage variable while the value of the other element is placed in the first, and then the second element can be replaced with the temporary storage value.

Follow-Up Question and Activity

1. This bubble sort algorithm is inefficient for large arrays. Make the following simple modifications to improve the performance of the bubble sort:
   a) After the first pass, the largest number is guaranteed to be in the highest-numbered element of the array; after the second pass, the two highest numbers are “in place,” and so on. Instead of making nine comparisons on every pass, modify the bubble sort to make eight comparisons on the second pass, seven on the third pass, and so on.
   b) The data in the array may already be in the proper order or near-proper order, so why make nine passes if fewer will suffice? Modify the sort to
check at the end of each pass if any swaps have been made. If none have 
been made, then the data must already be in the proper order, so the 
program should terminate. If swaps have been made, then at least one more 
pass is needed.

Program Challenge

A prime integer is any integer greater than 1 that is evenly divisible only by itself and 
1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as 
follows:

1. Create an array with all elements initialized to 1 (true). Array elements with 
prime subscripts will remain. All other array elements will eventually be set 
to zero. You will ignore elements 0 and 1 in this exercise.

2. Starting with array subscript 2, every time an array element is found whose 
value is 1, iterate through the remainder of the array and set to zero every 
element whose subscript is a multiple of the subscript for the element with 
value 1. For array subscript 2, all elements beyond 2 in the array that are 
multiples of 2 will be set to zero (subscripts 4, 6, 8, 10, etc.); for array 
subscript 3, all elements beyond 3 in the array that are multiples of 3 will be 
set to zero (subscripts 6, 9, 12, 15, etc.); and so on.

When this process is complete, the array elements that are still set to one indicate that 
the subscript is a prime number. These subscripts can then be printed. Write a program 
that uses an array of 1000 elements to determine and print the prime numbers between 
2 and 999. Ignore elements 0 and 1 of the array.