COMSC-165 Lab 6
Arrays and Vectors in C++

In this lab session you will exercise arrays. The purpose of this lab is to familiarize you with the ways that arrays can be used in C++. You will reproduce some coding exactly as it appears in this lab sheet, and modify it as directed in the instructions. You will also get to write your own code, based on specifications provided.

As you complete this assignment, post the required file(s) to the COMSC server. For assignments that involve more than one file, you may post them all at once, or one-at-a-time as you complete them, as you prefer. For assignments that involve more than one separate "lab" (like lab 1a, 1b, and 1c), the individual labs are graded in order, starting with "a", which must be fully correct before "b" is graded, and so on. This assignment is worth 50-points, to be awarded after all labs are successfully completed. Use the "Submit your FA2014 work" link on the class website to post your file for this lab, using the **lab6** folder provided.

LAB 6: Working With Arrays `[ Array.cpp ]`
Create a console application named **Array**. Do NOT submit until you complete the last step.

STEP 1 of 5: Create And Fill An Array
Write the **Array.cpp** program as listed below. It uses an array to store *up to* 50 test scores from the keyboard. Scores are integer numbers as low as 0, and up to and including 100.

Note that the program reads console input directly into a numeric variable. Run this, and see what happens if you enter text when prompted for numeric input -- it's not pretty! Remember that our convention is to not read numerics directly into `cin`. Be sure to follow all of the programming conventions for this course, even if they are not followed in the sample code provided in this writeup!

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main()
{
  const int MAX_SCORE = 50; // the maximum #of scores that a user can enter
  int score[MAX_SCORE]; // create storage for up to 50 scores
  int nScores = 0; // count the number of scores entered

  // read the scores from the keyboard, space and/or newline delimited
  for (int i = 0; i < MAX_SCORE; i++)
  {
    cin >> score[i];
    if (score[i] < 0)
      break; // enter no more scores after the 1st negative is found
    else
      nScores++; // count the score if it is non-negative
  }
  cout << nScores << endl; // say how many scores entered
}
```

In addition to fixing the `cin` code block, and adding your identifying comments and `cout`'s, add the following:

1. Include **`cout`** statements to guide the user -- say e.g., "Enter up to 50 numbers...", and "Enter a Q after the last score is entered.".

2. Include a **`cout`** statement to identify the last output -- say, e.g., "The number of scores entered is:".

3. If a user enters a ***negative*** number or a number greater than 100, skip it. so it's `break;` for Q; `continue;` for numbers outside the range of 0-100; count anything else.

4. Beware of "holes" in the array! These can happen if you `continue;` after a number outside the range gets entered. Note that the for-loop already positions `i` on the array element to be filled next. Continuing will leave that position unused, and move `i` to the *next* element. Think of a way to "rewind" `i` before `continue;` to avoid these "holes".

Compile and run. Note that multiple inputs can be entered on a single line, separated by spaces. Also note that the user can enter a **Q** to terminate the data entry before 50 entries are made.

STEP 2 of 5: Create And Fill An Array Using a Function
Modify **Array.cpp** from step 1, so that a function is used to read data from the keyboard into an array.

1. Use this prototype for the added function: **`int readArray(int, int[]);`**. The meaning of the return value and each of the parameters is outlined below in instructions 2-4.

2. The function is to *return the number of numbers read* into the array. It is passed as an array without the size specified, because the array can be of any size, depending on the number of values to be averaged. *(The actual size is passed in the first parameter.)*

3. The *first function parameter* is the maximum size of the array.

4. The *second function parameter* is the array to be filled. Note the syntax for the array parameter, using [ ]'s instead of an asterisk, as you may have learned previously. It's an alternate syntax for sharing arrays in parameters lists, and so we are learning this way here.

5. While the coding that fills the array in step 1 has score-specific variable names and comments, use generic names and comments in the function. Remember that the function does not know that it is collecting scores -- it just sees them as non-negative numbers.

Use the following code for the **`int main()`** function. Put all other code in a function, to be written by you.

```
int main()
{
  const int MAX_SCORE = 50; // the maximum #of scores that a user can enter
  int score[MAX_SCORE]; // create storage for up to 50 scores
  cout << readArray(MAX_SCORE, score) << endl; // say how many scores entered
}
```

STUDENT'S OPTION
At your option, you may write the `const int` as a global constant instead of a local constant in main. If you elect this option, do *not* include it as a parameter in any function calls, either in this step or the ones to follow.

STEP 3 of 5: Compute Averages Using Arrays
Modify **Array.cpp** from step 2, by adding a function to compute the average of the scores in the keyboard-entered array.

1. Use this prototype for the added function: **`int avg(int, const int[]);`**. The meaning of the return value and each of the parameters is outlined below in instructions 2-4.

2. The function is to *return the simple integer average* of the numbers read into the array.

3. The *first function parameter* is the number of numbers in the array to be averaged together.

4. The *second function* parameter is the array that was filled. It is passed as an array without the size specified, because the array can be of any size, depending on the number of values to be averaged. *(The actual size is passed in the first parameter.)*

5. To avoid run-time errors as a result of dividing by zero, check the number of numbers to be averaged -- if it is zero, return 0 for the average.

Use the following code for the **`int main()`** function. Put all other code in the functions, to be written by you.

```
int main()
{
  const int MAX_SCORE = 50; // the maximum #of scores that a user can enter
  int score[MAX_SCORE]; // create storage for up to 50 scores
  int nScores = readArray(MAX_SCORE, score); // read array and return count
  cout << avg(nScores, score) << endl; // say the average of the scores
}
```

NOTE
You may *not* write `int nScores` as a global varaible!

STEP 4 of 5: A Statistical Function
Modify **Array.cpp** by replacing the **`avg()`** function with a new **`stat()`** function. Refer to lab 3, for how to pass values by reference, and use that same logic in the **`stat()`** function.

1. Use this prototype for the added function: **`int stat(int, const int[], int&, int&, int&);`**. The meaning of the return value and each of the parameters is outlined below in instructions 2-5.

2. The function is to *return zero*, unless there are no numbers in the array. Return 1 if the array is empty.

3. The *first function parameter* is the number of numbers in the array to be averaged together.

4. The *second function parameter* is the array that was filled by the **`readArray()`** function. It is passed as an array without the size specified, because the array can be of any size, depending on the number of values to be averaged. *(The actual size is passed in the first parameter.)*

5. The *3rd, 4th, and 5th function parameters* are the minimum, maximum, and average of the numbers in the array. They are passed *by reference* so that when their values are reset by the function, they will be reset also in **`main`**.

6. If the number of numbers in the array is zero, do *not* set the min, max, and average (which are specified as the 3rd, 4th, and 5th parameters.)

7. Be sure to add full and proper text to the output statements.

Use the following code for the **`int main()`** function. Put all other code in the functions, to be written by you.

```
int main()
{
  const int MAX_SCORE = 50; // the maximum #of scores that a user can enter
  int score[MAX_SCORE]; // create storage for up to 5 scores
  int nScores = readArray(MAX_SCORE, score); // read array and return count

  int minScore, maxScore, avgScore;
  if (stat(nScores, score, minScore, maxScore, avgScore) == 0)
    cout << "Min=" << minScore << endl
      << "Max=" << maxScore << endl
      << "Average=" << avgScore << endl;
  else
    cout << "no data" << endl;
}
```

Notice how the first `cout` statement is sparated into three lines. This is to demonstrate how an otherwise very long output statement can be separated, to make it more easily readable. It also demonstrates that `endl`'s can appear anywhere in an output statement -- not just at the end.

STEP 5 of 5: Histograms
Modify **Array.cpp** by adding a **`histogram()`** function, that counts the numbers of scores in each grade category -- e.g., a score in the range 90-100 is grade A. Use a compound **`if`** statement in the function to determine which histogram cell to increment.

1. Add a function with this prototype: **`int histogram(int, const int[], int[]);`**. The meaning of the return value and each of the parameters is outlined below in instructions 2-5.

2. The function is to *return zero*, unless there are no numbers in the array. Return 1 if the array is empty.

3. The *first function parameter* is the number of numbers in the array to be averaged together.

4. The *second function parameter* is the array that was filled by the **`readArray()`** function. It is passed as an array without the size specified, because the array can be of any size, depending on the number of values to be averaged. *(The actual size is passed in the first parameter.)*

5. The *third function parameter* is for returning a histogram of the grade counts. (90 and above is A, 80 and above is B, 70 and above is C, and 60 and above is D. It is an array of known size (5) since there are exactly 5 possible grades. E.g., if there are 10 A's among the scores, then the first element of the array will be 10 when the function returns. (By specifying the size of an array in the parameter list, only arrays of the specified size can be passed.)

6. Be sure to add full and proper text to the output statements.

Use the following code for the **`int main()`** function. Put all other code in the functions, to be written by you.

```
int main()
{
  const int MAX_SCORE = 50; // the maximum #of scores that a user can enter
  int score[MAX_SCORE]; // create storage for up to 50 scores
  int nScores = readArray(MAX_SCORE, score); // read array and return count

  int avgScore, minScore, maxScore;
  if (stat(nScores, score, minScore, maxScore, avgScore) == 0)
  {
    cout << "Average=" << avgScore << endl
      << "Max=" << maxScore << endl
      << "Min=" << minScore << endl;
    int grade[5] = {}; // = {} is for newer compilers only
    histogram(nScores, score, grade);
    cout << "As: " << grade[0] << endl;
    cout << "Bs: " << grade[1] << endl;
    cout << "Cs: " << grade[2] << endl;
    cout << "Ds: " << grade[3] << endl;
    cout << "Fs: " << grade[4] << endl;
  }
  else
    cout << "no data" << endl;
}
```

*Post **Array.cpp** to the COMSC server.*

GRADING POLICY

Lab work will be returned for you to fix and resubmit for any of the following reasons, or for not following stated specifications, unless directed otherwise by your instructor, Point penalties will be assessed per the syllabus for each time a lab is returned for redo for any listed reason below or specification above. Points will be awarded for an assignment upon final, successful completion of all of its labs. Note that you will NOT receive a list of corrections to make, because to do so would remove any incentive for students to do quality work. Instead, grading of your work will stop upon discovery of the first mistake. Grading will not continue until the mistake is corrected. So to avoid multiple cycles of correction and resubmission, and the cumulative point penalties, make sure that your work is as correct as you can make it before submitting it.

Basic checklist, even if your compiler "forgives" doing otherwise:

1. Spell and case filenames *exactly* as specified. To avoid errors, be sure to configure your Windows computer to NOT hide file name extensions -- refer to the Burns Intro to Programming text, chapter 2.
2. Submit files to the specified folder. Do NOT submit archive files, like ZIP, JAR, or RAR.
3. Include identifying information as the first lines in the file AND as the first lines of your output in main.
4. Do NOT use the statement `system("PAUSE");` or any other system-specific code.
5. Do NOT try to access the value stored in an uninitialized variable.
6. CPP and H files must be free of careless typing mistakes that would prevent compilation.
7. Put an `endl` as the last item in the last-executed `cout` statement before the closing curly brace of `main`.
8. Do NOT forget to use the `cin.ignore` statements for non-string console input, unless specifically directed otherwise.

9. Do NOT forget to use the `fin.ignore` statements for non-string file input, unless specifically directed otherwise.
10. For file I/O, NEVER include a path designation. ALWAYS use the "working folder" for files.
11. Use the proper `#includes` and `usings` as listed in the C++ Library PDF.
12. Do NOT #include C-libraries such as `math.h` -- include their C++ equivalents instead, like `cmath`.
13. Do NOT #include libraries that you do not use.
14. Do NOT #include "**stdafx.h**"; do NOT use `pragma`.
15. Do NOT use C/C++ constructs or library items that are not specifically taught in this class, such as `goto`, `scanf` or `fscanf`, `fixed`, `strcmp_c`, `stricmp`, etc.
16. Do NOT use `void main()` -- use `int main()` always.
17. Do NOT use `return 0;` as the last statement in `int main()`, because it is not required by ANSI Standard C++.
18. If you use output statements in your program for debugging purposes, be sure to delete them before submitting your work.
19. Do *not* use "global variables".

Advanced checklist, even if your compiler "forgives" doing otherwise:

1. `NULL` is ONLY a pointer value -- do NOT use it for integer or other numeric values, even if your compiler allows it.
2. Do NOT declare a static array with a variable size specification, such as `int n =...; int a[n];`, even if your compiler allows it.
3. NEVER #include a CPP file, even if the textbook includes examples of such.
4. For console input and text file input with `>>`, use `.ignore` as explained in the Burns Intro to Programming text, chapter 5, unless specifically directed otherwise.
5. NEVER use `cin >>` to read directly into a numeric variable, like `int` or `double` or `float`. Read as a string (C or C++) and convert to a number using `atoi` or `atof`. Include `cstdlib`!
6. Do NOT use the `inline` keyword. To make class member functions inline, write their definitions inside the class definition.
7. Once the `const` keyword is introduced in this course, use it where applicable. Note that class "getter" functions are not really getter functions unless they have a trailing `const` keyword to flag them as such.
8. Make sure that all value-returning functions return a valid value under any logical circumstance.
9. Exit functions with `return` statements, and NOT `exit` -- do NOT use `exit` under any circumstances.
10. In every H and CPP file, ALWAYS #include libraries and H files for ALL functions, classes, and identifiers used in that file. Exception: use a class "forward declaration" instead of #including the class' H file when only pointers and reference variables appear.
11. Include the `cstring` library for char-array-based string ("C string") *functions*, and the `string` library for STL-based-strings ("C++ strings").
12. When using the random number generator seed function `srand`, call it ONCE ONLY in your program, and "sacrifice" the first value, like this: `srand(time(0)); rand();` at the top of main. Be sure to #include both `ctime` and `cstdlib`.
13. In COMSC 210, use `using namespace std;`. In 165 and 200, use `using std::` statements, properly placed under the #includes of the libraries to which they pertain.
14. `std::string` and `std::getline` belong to the `string` library. In 165 and 200, use `using std::` statements for these. Note that `cin.getline` is a different thing, and only requires `using std::cin;`.
15. Do NOT use `using std::` for anything in the C libraries. Use it for items from C++ libraries, only.
16. Test drivers MUST at least include ifndef testing, object copy testing with assignment after

declaration, and const object testing with assignment upon declaration.

17. Test drivers must NOT include console or file input, unless directed otherwise.
18. Do NOT allow memory leaks, unless specifically instructed otherwise.