Programming Assignment 4
Array of Sunshine

Time due: 9:00 PM Thursday, November 5

Before you ask questions about this specification, see if your question has already been addressed by the Project 4 FAQ. And read the FAQ before you turn in this project, to be sure you didn't misinterpret anything.

As you gain experience with arrays, you'll discover that many applications do the same kinds of things with them (e.g., find where an item is in an array, or check whether two arrays differ). You'll find that it's helpful to have a library of useful functions that manipulate arrays. (For our purposes now, a library is a collection of functions that developers can call instead of having to write them themselves. For a library to be most useful, the functions in it should be related and organized around a central theme. For example, a screen graphics library might have functions that allow you to draw shapes like lines and circles on the screen, move them around, fill them with color, etc. In this view, the Standard C++ library is really a collection of libraries: a string library, a math library, an input/output library, and much more.)

Your assignment is to produce a library that provides functions for many common manipulations of arrays of strings. For example, one function will find where a string occurs in an unordered array of strings. Another will change the order of strings in an array. For each function you must write, this specification will tell you its interface (what parameters it takes, what it returns, and *what* it must do). It's up to you to decide on the implementation (*how* it will do it).

The source file you turn in will contain all the functions and a main routine. You can have the main routine do whatever you want, because we will rename it to something harmless, never call it, and append our own main routine to your file. Our main routine will thoroughly test your functions. You'll probably want your main routine to do the same. If you wish, you may write functions in addition to those required here. We will not directly call any such additional functions. If you wish, your implementation of a function required here may call other functions required here.

The program you turn in must build successfully, and during execution, no function (other than main) may read anything from `cin` or write anything to `cout`. If you want to print things out for debugging purposes, write to `cerr` instead of `cout`. When we test your program, we will cause everything written to `cerr` to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.

All of the functions you must write take at least two parameters: an array of strings, and the number of items the function will consider in the array, starting from the beginning. For example, in

```
string people[5] = { "hillary", "jeb", "rand", "ben", "john" };
int i = lookup(people, 3, "john");  // should return -1 (not found)
```

even though the array has 5 elements, only the first 3 had values we were interested in for this call to the function; the function must not examine the others.

Notwithstanding each function's behavior described below, all functions that return an int must return −1 if they are passed any bad arguments (e.g. a negative array size, or a position that would require looking at the contents of an element past the last element we're interested in). Unless otherwise noted, passing 0 to the function as the array size is not itself an error; it merely indicates the function should examine no elements of the array.

The one error your function implementations don't have to handle, because they can't, is when the caller of the function lies and says the array is bigger than it really is. For example, in this situation, the function `lookup` can't possibly know that the caller is lying about the number of interesting items in the array:

```
string people[5] = { "hillary", "jeb", "rand", "ben", "john" };
int i = lookup(people, 25, "bernie");  // Bad call of function, but
               // your lookup implementation doesn't have to check for
               // this, because it can't.
```

To make your life easier, whenever this specification talks about strings being equal or about one string being less than or greater than another, the case of letters matters. This means that you can simply use comparison operators like == or < to compare strings. Because of the character collating sequence, all upper case letters come before all lower case letters, so don't be surprised. The FAQ has a note about string comparisons.

Here are the functions you must implement:

```
int appendToAll(string a[], int n, string value);
```

Append `value` to the end of each of the `n` elements of the array and return `n`. [Of course, in this and other functions, if `n` is negative, theparagraph above that starts "Notwithstanding" trumps this by requiring that the function return −1. Also, in the description of this function and the others, when we say "the array", we mean the `n` elements that the function is aware of.] For example:

```
string people[5] = { "hillary", "jeb", "rand", "ben", "john" };
int j = appendToAll(people, 5, "!!!");  // returns 5
    // now cand[0] is "hillary!!!", cand[1] is "jeb!!!", ...,
```

```
        // and cand[4] is "john!!!"
int lookup(const string a[], int n, string target);
```

Return the position of a string in the array that is equal to `target`; if there is more than one such string, return the smallest position number of such a matching string. Return −1 if there is no such string. As noted above, case matters: Do not consider "JEb" to be equal to "jEB".

```
int positionOfMax(const string a[], int n);
```

Return the position of a string in the array such that that string is >= every string in the array. If there is more than one such string, return the smallest position in the array of such a string. Return −1 if the array has no elements. For example:

```
string cand[6] = { "bernie", "hillary", "donald", "marco", "carly",
"ben" };
int k = positionOfMax(cand, 6);   // returns 3, since  marco  is latest
                                  // in alphabetic order
int rotateLeft(string a[], int n, int pos);
```

Eliminate the item at position `pos` by copying all elements after it one place to the left. Put the item that was thus eliminated into the last position of the array. Return the original position of the item that was moved to the end. Here's an example:

```
string running[5] = { "carly", "mike", "ted", "bernie", "jeb" };
int m = rotateLeft(running, 5, 1);  // returns 1
    // running now contains:  "carly", "ted", "bernie", "jeb", "mike"
int countRuns(const string a[], int n);
```

Return the number of sequences of one or more consecutive identical items in `a`.

```
string d[9] = {
    "ben", "chris", "marco", "marco", "donald", "donald", "donald",
"marco", "marco"
};
int p = countRuns(d, 9);  //  returns 5
     //  The five sequences of consecutive identical items are
     //    "ben"
     //    "chris"
     //    "marco", "marco"
     //    "donald", "donald", "donald"
     //    "marco", "marco"
int flip(string a[], int n);
```

Reverse the order of the elements of the array and return `n`. For example,

```
string folks[6] = { "chris", "marco", "", "ben", "donald", "john" };
int q = flip(folks, 4);  // returns 4
    // folks now contains:  "ben"  ""  "marco"  "chris"  "donald"
"john"
int differ(const string a1[], int n1, const string a2[], int n2);
```

Return the position of the first corresponding elements of `a1` and `a2` that are not equal. `n1` is the number of interesting elements in `a1`, and `n2` is the number of interesting elements in `a2`. If the arrays are equal up to the point where one or both runs out, return the smaller of `n1` and `n2`. For example,

```
string folks[6] = { "chris", "marco", "", "ben", "donald", "john" };
string group[5] = { "chris", "marco", "john", "", "carly" };
```

```
      int r = differ(folks, 6, group, 5);  //  returns 2
      int s = differ(folks, 2, group, 1);  //  returns 1
int subsequence(const string a1[], int n1, const string a2[], int n2);
```

If all `n2` elements of `a2` appear in `a1`, consecutively and in the same order, then return the position in `a1` where that subsequence begins. If the subsequence appears more than once in `a1`, return the smallest such beginning position in the array. Return −1 if `a1` does not contain `a2` as a contiguous subsequence. (Consider a sequence of 0 elements to be a subsequence of any sequence, even one with no elements, starting at position 0.) For example,

```
      string names[10] = { "ted", "hillary", "rand", "bernie", "mike",
      "jeb" };
      string names1[10] = { "hillary", "rand", "bernie" };
      int t = subsequence(names, 6, names1, 3);  // returns 1
      string names2[10] = { "ted", "bernie" };
      int u = subsequence(names, 5, names2, 2);  // returns -1
int lookupAny(const string a1[], int n1, const string a2[], int n2);
```

Return the smallest position in `a1` of an element that is equal to any of the elements in `a2`. Return −1 if no element of `a1` is equal to any element of `a2`. For example,

```
      string names[10] = { "ted", "hillary", "rand", "bernie", "mike",
      "jeb" };
      string set1[10] = { "carly", "mike", "bernie", "hillary" };
      int v = lookupAny(names, 6, set1, 4);  // returns 1 (a1 has "hillary"
      there)
      string set2[10] = { "ben", "donald" };
      int w = lookupAny(names, 6, set2, 2);  // returns -1 (a1 has none)
int split(string a[], int n, string splitter);
```

Rearrange the elements of the array so that all the elements whose value is < `splitter` come before all the other elements, and all the elements whose value is > `splitter` come after all the other elements. Return the position of the first element that, after the rearrangement, is not < `splitter`, or `n` if there are no such elements. For example,

```
      string cand[6] = { "bernie", "hillary", "donald", "marco", "carly",
      "ben" };
      int x = split(cand, 6, "chris");  //  returns 3
        // cand must now be
        //      "bernie"  "carly"  "ben"  "hillary"  "marco"  "donald"
        // or   "carly"  "ben"  "bernie"  "donald"  "hillary"  "marco"
        // or one of several other orderings.
        // All elements < "chris" (i.e., "carly", "ben", and "bernie")
        //   come before all others
        // All elements > "chris" (i.e., "marco", "hillary", and "donald")
        //   come after all others
      string cand2[4] = { "donald", "hillary", "jeb", "ben" };
      int y = split(cand2, 4, "donald");  //  returns 1
        // cand2 must now be either
        //      "ben"  "donald"  "hillary"  "jeb"
        // or   "ben"  "donald"  "jeb"  "hillary"
        // All elements < "donald" (i.e., "ben") come before all others.
        // All elements > "donald" (i.e., "jeb" and "hillary") come after all
        // others.
```

For each of the functions `rotateLeft`, `flip`, and `split`, if the function is correctly implemented, you will earn one bonus point for that function if it does its job without creating any additional array.

Your program must *not* use any function templates from the algorithms portion of the Standard C++ library. If you don't know what the previous sentence is talking about, you have nothing to worry about. If you do know, and you violate this requirement, you will be required to take an oral examination to test your understanding of the concepts and architecture of the STL.

Your implementations must not use any global variables whose values may be changed during execution.

Your program must build successfully under both Visual C++ and either clang++ or g++.

The correctness of your program must not depend on undefined program behavior. Your program could not, for example, assume anything about `t`'s value in the following, or even whether or not the program crashes:

```
int main()
{
    string top[3] = { "hillary", "ben", "donald" };
    string t = top[3];   // position 3 is out of range
    …
```

What you will turn in for this assignment is a zip file containing these two files and nothing more:

1. A text file named **array.cpp** that contains the source code for your C++ program. Your source code should have helpful comments that explain any non-obvious code.
2. A file named **report.doc** or **report.docx** (in Microsoft Word format), or **report.txt** (an ordinary text file) that contains:
    a. A brief description of notable obstacles you overcame.
    b. A list of the test data that could be used to thoroughly test your functions, along with the reason for each test. You must note which test cases your program does not handle correctly. (This could happen if you didn't have time to write a complete solution, or if you ran out of time while still debugging a supposedly complete solution.) Notice that most of this portion of your report can be written just after you read the requirements in this specification, before you even start designing your program.

How nice! Your report this time doesn't have to contain any design documentation.

As with Project 3, a nice way to test your functions is to use the `assert` facility from the standard library. As an example, here's a very incomplete set of tests for Project 4:

```
        #include <iostream>
        #include <string>
        #include <cassert>
        using namespace std;

        int main()
        {
            string h[7] = { "bernie", "hillary", "donald", "jeb", "", "carly",
"ben" };
            assert(lookup(h, 7, "carly") == 5);
            assert(lookup(h, 7, "donald") == 2);
            assert(lookup(h, 2, "donald") == -1);
            assert(positionOfMax(h, 7) == 3);

            string g[4] = { "bernie", "hillary", "jeb", "carly" };
            assert(differ(h, 4, g, 4) == 2);
            assert(appendToAll(g, 4, "?") == 4 && g[0] == "bernie?" && g[3] ==
"carly?");
            assert(rotateLeft(g, 4, 1) == 1 && g[1] == "jeb?" && g[3] ==
"hillary?");

            string e[4] = { "donald", "jeb", "", "carly" };
            assert(subsequence(h, 7, e, 4) == 2);

            string d[5] = { "hillary", "hillary", "hillary", "ben", "ben" };
            assert(countRuns(d, 5) == 2);

            string f[3] = { "jeb", "donald", "marco" };
            assert(lookupAny(h, 7, f, 3) == 2);
            assert(flip(f, 3) == 3 && f[0] == "marco" && f[2] == "jeb");

            assert(split(h, 7, "carly") == 3);

            cout << "All tests succeeded" << endl;
        }
```

The reason for the one line of output at the end is to ensure that you can distinguish the situation of all tests succeeding from the case where one test silently crashes the program.

Make sure that if you were to replace your main routine with the one above, your program would build successfully under both Visual C++ and either clang++ or g++. (This means that even if you haven't figured out how to implement some of the functions, you must still have *some* kind of implementations for them, even if those implementations do nothing more than immediately return, say, 42.) If the program

built with that main routine happens to run successfully, you'll probably get a pretty good correctness score.

By November 4, there will be links on the class webpage that will enable you to turn in your zip file electronically. Turn in the file by the due time above. Give yourself enough time to be sure you can turn something in, because we will not accept excuses like "My network connection at home was down, and I didn't have a way to copy my files and bring them to a SEASnet machine." There's a lot to be said for turning in a preliminary version of your program and report early (You can always overwrite it with a later submission). That way you have something submitted in case there's a problem later.