Project 1: Sorting C-Strings

Order properties and sorting of character strings

Revision dated 01/02/18

Educational Objectives: After successfully completing this assignment, the student should be able to accomplish the following:

- Use command-line arguments in a C++ program.
- Use a loop structure to read input of unknown size through `std::cin` and store it in an array.
- Use conditional branching to selectively perform computational tasks.
- Declare (prototype) and define (implement) functions.
- Declare and define functions with arguments of various types, including pointers, references, const pointers, and const references.
- Call functions, making appropriate use of the function arguments and their types.
- Make decisions as to appropriate function call parameter type, from among: value, reference, const reference, pointer, and const pointer.
- Create, edit, build and run multi-file projects using the Linux/Emacs/Make environment announced in the course organizer.

Operational Objectives: Create a project that reads and sorts a file of character strings received via standard input.

Deliverables: Files: `cstringdiff.h`, `cstringdiff.cpp`, `cstringsort.h`, `cstringsort.cpp`, `main.cpp`, and `log.txt`. Note that these files, together with the supplied makefile, constitute a self-contained project.

Assessment Rubric: The following will be used as a guide when assessing the assignment:

```
builds:
 diffcalc.x                          [0..5]:   x
 test.x                              [0..5]:   x
 ssort.x                             [0..5]:   x
tests:
 diffcalc.x string1 string2          [0..5]:   x
 diffcalc.x string3 string4          [0..5]:   x
 ssort.x < data1.in                  [0..5]:   x
 ssort.x < data2.in                  [0..5]:   x
code quality                        [-20..5]:  xx  # note negative points awarded during assessment
dated submission deduction      [(2) pts per]: (xx) # note negative points awarded during assessment
                                                --
total                                [0..40]:  xx
```

Please self-evaluate your work as part of the development process.

Background

One of the most common procedures done with a computer is to sort a collection of character strings. This is more common even than sorting numbers, but it is often left unmentioned in textbook discussions of sorting because of the technical difficulties of dealing with strings. It is not even clear what we mean by "sorting strings" because there are at least two reasonable concepts of order among strings: *ascii order* and *dictionary order*.

Lex Order

The "ascii" character set is essentially what you see on a standard keyboard (lower and upper register), plus some invisible control characters. In ancient times, the control characters were used to manipulate mechanical printers known as "teletype" machines. Each ascii character is associated with an integer in the range [0,128). There are 32 control characters (numbers 0-31) and 96 visible/printable characters (numbers 32-127). See asciitable for more details.

"Ascii" order is more technically called *lexicographical order* ("lex order" for short) and is defined for any character set, including **EXTENDED_ASCII** [2^8 = 256 characters] and **UNICODE16** [2^{16} = 65,536 characters]. (See Section 1.1 of Strings for more about modern character sets used in computing. The more advanced material in Section 1.2 is optional, and Section 2 is definitely beyond the scope of this class.)

Lexicographical order between two strings of characters is determined as follows: compare the characters in the two strings one at a time, starting with the first character. If the two characters are the same, proceed to the next character, stopping at the first index where the strings differ. Then the character set order of these two characters determines the lexicographical order of the two strings. If the characters in one of the strings are exhausted before finding a difference, the shorter string is considered to come before the longer one.

Determination of lexicographical order between two strings `s1` and `s2` is facilitated by a "Diff" function that takes on integer values. The value returned by `LexDiff(s1,s2)` has these properties:

LexDiff	
return value	*condition*
0	strings are identical
negative	s1 comes before s2 in lex order
positive	s1 comes after s2 in lex order

Examples:

```
 -1 = LexDiff(abc,acz) #  -1  = (int)'b' - (int)'c'
-23 = LexDiff(abc,abz) # -23  = (int)'c' - (int)'z'
 +3 = LexDiff(abf,abc) #  +3  = (int)'f' - (int)'c'
  0 = LexDiff(abc,abc)
-99 = LexDiff(ab,abc)  # -99  = (int)'\0' - (int)'c'
```

```
+99 = LexDiff(abc,ab)   # +99  = (int)'c' - (int)'\0'
+31 = LexDiff(abc,abD)
 -1 = LexDiff(abc,abd)
-32 = LexDiff(aBc,abc) # -32  = (int)'B' - (int)'b'
```

The absolute value returned by Diff is not specified in the C++ standard, so a satisfactory Diff function could restrict return values to three possibilities: -1, 0, +1. *We will however call for two distinct implementations to provide experience with both options.* Most modern programming languages use the Diff function technique, including C, C++, and Java.

Given a Diff function for strings over a character set, it is simple to determine the relative order of two strings `s1` and `s2`: if `Diff(s1,s2) < 0` then `s1` comes before `s2` in lex order, otherwise not.

Dictionary Order

The "dictionary" order between two strings is special to the ASCII character set. It is essentially the lex order except that the case of letters is ignored, so that upper and lower case letters are considered equal when determining order.

Dictionary order is also facilitated by a DictionaryDiff function, with modified properties:

DictionaryDiff	
return value	*condition*
0	strings are identical *ignoring case*
-1	s1 comes before s2 in lex order *ignoring case*
+1	s1 comes after s2 in lex order *ignoring case*

Examples:

```
 -1 = DictionaryDiff(abc,acz) # 'b' comes before 'c' in ascii order
 -1 = DictionaryDiff(abc,abz) # 'c' comes before 'z' in ascii order
 +1 = DictionaryDiff(abf,abc) # 'f' comes after 'c' in ascii order
  0 = DictionaryDiff(abc,abc)
 -1 = DictionaryDiff(ab,abc)
 +1 = DictionaryDiff(abc,ab)
 -1 = DictionaryDiff(abc,abD) # 'c' comes before 'd' in ascii order
 -1 = DictionaryDiff(abc,abd)
  0 = DictionaryDiff(aBc,abc)
```

Relative dictionary order of two ascii strings is determined in the same way as lex order, except using the dictionary Diff function.

Arithmetic v. Logical Diff

In the examples above (and *required* in your implementations) we have used an "arithmetic" approach to implement LexDiff and a "logic" approach to implement DictionaryDiff. Both methods follow the same ideas, and in fact can follow the same code, up to the place where the first "difference" character is found. The arithmetic approach then returns the calculated difference between the characters at this place, whereas the logic approach returns -1, 0, or +1 based on the relative positions of these characters in ascii order (after converting to lower case for DictionaryDiff).

Use an arithmetic approach to implement LexDiff and a logic approach to implement DictionaryDiff.

Insertion Sort

This sort was introduced in COP3014 as a sort of an array of `int`. (See the COP3014 Chapter 6 notes linked from our course organizer.) That code works, but it has undesirable aspects:

1. It uses array index and loop control variables of type `int`, which is the same type as the data being sorted. Arrays never have negative indices, and the loops in the algorithm never have negative control values. Therefore the preferred type for these variables is `size_t`. This change acknowledges the distinction between the control type (`size_t`) and the data type (`int`).
2. The names of variables are cumbersome at best. (Admittedly, this is a personal choice.)

Note that `size_t` is defined in `<cstdlib>`. Here is a direct translation of the code taking into account 1 and 2:

```
// Data is passed using pointers defining a range in memory: [beg,end)
void IntegerInsertionSort (int* beg, int* end)
{
  size_t size = end - beg; // size of array obtained with pointer arithmetic
  if (size < 2) return;    // nothing to do
  size_t i; // outer loop control
  size_t j; // inner loop control
  int    t; // value holder
  for (i = 0; i < size; ++i)
  {
    t = beg[i];
    for (j = i; j > 0 && t < beg[j-1]; --j) // copy values up until t >=beg[j-1]
      beg[j] = beg[j-1];
    beg[j] = t; // copy t into vacated slot
  }
}
```

The only real change is the use of two pointers to define the range of values to be sorted, rather than the beginning of the range and its size. If `A` is an array, then `A` is the begin pointer and `A + size` is the end pointer.

It is useful to refactor this code, in two steps. The first step is to use a 3rd control variable `k` that tracks one index ahead of the inner loop variable `j` as it decrements, so that `k == j - 1`

```
void IntegerInsertionSort (int* beg, int* end)
{
  size_t size = end - beg;
  if (size < 2) return;
  size_t i; // outer loop control
  size_t j; // inner loop control
  size_t k; // k is always j - 1
  int    t; // value holder
  for (i = 0; i < size; ++i)
  {
    t = beg[i];
    for (k = i, j = k--; j > 0 && t < beg[k]; --j, --k)
      beg[j] = beg[k];
    beg[j] = t;
  }
}
```

The second step is to convert the control structure from indices to pointers:

```
void IntegerInsertionSort (int* beg, int* end)
{
  if (end - beg < 2) return;
  int * i; // outer loop control
  int * j; // inner loop control
  int * k; // k is always j - 1
  int   t; // value holder
  for (i = beg; i != end; ++i)
  {
    t = *i;
    for (k = i, j = k--; j != beg && t < *k; --j, --k)
      *j = *k;
    *j = t;
  }
}
```

Any of these implementations can be re-worked to sort an array of C-strings. We recommend the third one. Whichever one you use as a starting point for the string sorts, be sure that you understand all three refactorings.

InsertionSort is actually a very useful sort algorithm, even though it is "slow": It runs in quadratic time when input is random data. However, it runs in linear time when the data is pre-sorted, and proportionally more efficient when data is somewhere "between" random and sorted. InsertionSort is also *stable*, meaning that relative position of equal keys is not changed. This is evident in your Dictionary sort, which should not interchange `aaa` and `AAA` no matter which comes first in the data. And finally, "Sort by Insertion" is a higher level concept that can lead to more efficient sorts as well as serve as a model for analysis of QuickSort (in a later course).

Procedural Requirements:

1. Begin your log file named `log.txt`. (See Assignments for details.)
2. Create and work within a separate subdirectory `cop3330/proj1`. Review the COP 3330 rules found in Introduction/Work Rules.
3. Copy all of the files from `LIB/proj1`. These should include:

   ```
   makefile
   deliverables.sh
   main.start    # contains functions CopyString and PrintStrings, plus the command line argument processing
   diffcalc.cpp  # program calculates Diffs for two input strings
   ```

   In addition you should have the script `submit.sh` in either your `.bin` or your `proj1` as an executable command. Be sure you have version 3.0.
4. Create five more files

   ```
   cstringdiff.h
   cstringdiff.cpp
   cstringsort.h
   cstringsort.cpp
   main.cpp
   ```

   complying with the Technical Requirements and Specifications stated below.
5. Turn in six files `cstringdiff.h`, `cstringdiff.cpp`, `cstringsort.h`, `cstringsort.cpp`, `main.cpp`, and `log.txt` using the submit script.

   Warning: *Submit scripts do not work on the `program` and `linprog` servers. Use `shell.cs.fsu.edu` or `quake.cs.fsu.edu` to submit projects. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.*
6. After submission, take Quiz 1 in Blackboard. This quiz covers these areas:

   i. Casting; integer and floating point arithmetic.
   ii. Function calls
   iii. Loops
   iv. This assignment
   v. Course Syllabus

Note that the quiz may be taken two times. The last of the grades will be recorded and count as 10 points (20 percent of the assignment).

Technical Requirements and Specifications

1. The project should compile error- and warning-free on `linprog` with the command `make`.
2. All test output should be identical to that of the area51 executables.
3. The number of strings in the file to be sorted is not known in advance, except that it will not exceed a parameter entered at the command line (default = 1000).
4. Once the input strings have been read, the program should sort them and display the results to standard output.
5. One command line argument is required: (1) either 'A' or 'D'. 'A' indicates that the sort should use ascii order, whereas 'D' indicates that the sort should use dictionary order. (This argument may be entered either upper or lower case.)

   Two command line arguments are optional: (2) the max number of strings to that can be read and (3) the max length (number of characters) of the individual strings. These two arguments have default values of 1000 and 200, respectively.

   To be reminded of these arguments, enter the executable with no arguments.
6. The source code should be structured as follows:

   a. Implement separate functions with the following prototypes:

   ```
   int  LexDiff              (const char* s1, const char* s2);
   int  DictionaryDiff       (const char* s1, const char* s2);
   bool LexComp              (const char* s1, const char* s2);
   bool DictionaryComp       (const char* s1, const char* s2);
   void LexStringSort        (char* *beg, char* *end);    // see hint on this topic
   void DictionaryStringSort (char* *beg, char* *end);    // see hint on this topic
   ```

   b. I/O is handled by function `main()`; no other functions should do any I/O
   c. Function `main()` calls `LexStringSort` and `DictionaryStringSort` conditionally, depending on the required command line argument.
   d. Function `LexStringSort` calls `LexComp`
   e. Function `DictionaryStringSort` calls `DictionaryComp`
   f. Function `LexComp` calls `LexDiff`
   g. Function `DictionaryComp` calls `DictionaryDiff`
7. The source code should be organized as follows:

   a. Prototypes for `LexDiff`, `LexComp`, `DictionaryDiff`, and `DictionaryComp` should be in file `cstringdiff.h`
   b. Prototypes for `LexStringSort` and `DictionaryStringSort` should be in file `cstringsort.h`
   c. Implementations for `LexDiff`, `LexComp`, `DictionaryDiff`, and `DictionaryComp` should be in file `cstringdiff.cpp`
   d. Implementations for `LexStringSort` and `DictionaryStringSort` should be in file `cstringsort.cpp`
   e. Function `main` should be in file `main.cpp`
8. The `LexDiff` and `DictionaryDiff` functions should comply with the specs discussed above under Background.
9. The `LexComp` function should use the values returned by `LexDiff` to return a bool true or false.
10. The `DictionaryComp` function should use the values returned by `DictionaryDiff` to return a bool true or false.
11. The `LexSort` function should implement the Insertion Sort algorithm, using `LexComp` to determine order.
12. The `DictionarySort` function should implement the Insertion Sort algorithm, using `DictionaryComp` to determine order.
13. When in doubt, your program should behave like the distributed executables `ssort_i.x` and `diffcalc_i.x` in `area51`.
14. Behavior of your executables should be ***identical*** to that of the `area51` executables. In particular, the data input loop in `main.cpp` should not be interupted by prompts - this will make file redirect cumbersome. No prompting for data is necessary.
15. Your functions should cross-compile with our function main and the resulting program should produce output identical to `ssort.x`.

Hints

- **Development Strategy Hints.**

  a. Develop main.cpp without calls to the sorts so the I/O is tested and debugged. (`c3330 main`) Then add the sort calls.
  b. Develop and test the Diff functions with `diffcalc` before working on the sorts. (`make diffcalc.x`)
  c. If you are in doubt about the sort implementations for strings, you can create a parallel set of code that sorts files of integers, just to debug the sort algorithm and get the processing right.
  d. Once you know that main is working properly, the Diff and Comp functions are correct, and the Sort algorithm is working, put it all together.

- Example executables are distributed as `[LIB]/area51/ssort_i.x` and `[LIB]/area51/diffcalc_i.x`. The suffix indicates it is compiled to run on the Intel/Linux architecture (linprog machines).
- To run a sample executable, follow these steps: (1) Copy the appropriate executable into your space where you want to run it: log in to `linprog` and enter the command "`cp [LIB]/area51/ssort_i.x .`". (2) Change permissions to executable: "`chmod 700 ssort_i.x`". (3) Execute by entering the name of the executable, the required argument ('a' or 'd'), and redirect a file to the command. If you want

to run it on file "data1", use input redirect as in: "ssort_i.x A < data1". If you want the output to go to another file, use output redirect: "ssort_i.x D < data1 > data1.out".

- Source code for a "diff calculator" is given as diffcalc.cpp, and its build is included in the makefile. Please read this source code - note that after error checking, it has only two lines of code!

```
int main(int argc, char* argv[])
{
  std::cout << " LexDiff(s1,s2) = " << LexDiff(argv[1],argv[2]) << '\n'
            << " DicDiff(s1,s2) = " << DictionaryDiff(argv[1],argv[2]) << '\n';
}
```

  This is a useful calculator to help check your work coding the two Diff functions. It also illustrates how command-line arguments work: Note the use of argv[1],argv[2]. These are passed in as C-strings by the operating system. char* argv[] ("argv" stands for "argument vector") is an array of C-strings and int argc ("argc" stands for "argument count") is the size of the array. Note that the first argument argv[0] is always the name of the executable itself.

- The less-than character in the command:
  ssort.x a < data1
  is a Unix/Linux operation that redirects the contents of data1 into standard input for ssort.x. Using > redirects program output. For example, the command:
  ssort.x a < data1 > data1.out
  sends the contents of data1 to standard input and then sends the program output into the file data1.out. These are very handy operations for testing programs and building easy-to-use command-line tools.

- It is sometimes simpler to develop the code in a single file (such as project.cpp) that can be edited in one window and test-compiled with a single command (such as c3330 project.cpp) and split the file up into the deliverables after the initial round of testing and debugging.

- **Hint on Prototypes.** The official prototype signature for the two sort functions may take some thinking to understand. Taking the Lex case, what we have listed above is:

```
void LexStringSort (char* *beg, char* *end);
```

  This is stated in a way that reads easily. The parameter char* *beg is interpreted as a pointer named beg that is pointing to a C-string, which of course is technically just a pointer to type char. A couple of alternatives may make more sense, or at least help clarify the nature of beg. We have used colors to separate out the array element type from the array:

```
void LexStringSort (char* *beg, char* *end);
void LexStringSort (char** beg, char** end);
void LexStringSort (char ** beg, char ** end);
void LexStringSort (char **beg, char **end);
void LexStringSort (char* beg[], char* end[]);
```

  The last one emphasizes that beg and end are array variables. These all work, so you may use the one that you like best.

  As you may have noted, the code standard doesn't address the notation for pointer-to-pointer, so we are allowing personal choice here.

- **Hint on Range of Sort.** It is probably not clear why the Sorts require two arguments of the same type (maybe you had been expecting an array followed by a number of elements). The notation we are using just points to the places where the sort should begin and end. A typical call would be something like this:

```
LexStringSort (stringarray, stringarray + count);
```

  where stringarray is the name of the array being used to store C-strings and count is the number of C-strings currently being stored in the array (that is, the number read from the file).

  Pointers and pointer arithmetic are used to make the call: The name of the array is the base address of the array, and the name + count is the address "one past the end" of the data under consideration. So, we want the sort to start at address stringarray and end count slots further down the array.

- It is worth taking a look at the sort algorithm you are implementing. Ideally you want to be moving pointers around, not pointees. In other words, remember that what you are sorting is an array of "handles" (pointers) for strings. When you swap or otherwise need to change the array location of a string based on comparision with another one, you want to just swap or move the pointer to it. This is much simpler than attempting to swap the actual string data because the strings have different lengths and cannot be just copied to one another. This is also more efficient, because a pointer is essentially just an integer, whereas a string is an entire array of data and much more time-consuming to copy.

- To test your functions for correct signature, make sure they cross-compile with OUR function main. There is a target "test.x" in the supplied makefile that does this. You want test.x and ssort.x to compile and have identical behaviour.

- The following may help to visualize the storage array used in main(). This is the state of the array after reading the file containing "Chris Lacher Dalton Bohning":

a[0]->	C	h	r	i	s	\0	
a[1]->	L	a	c	h	e	r	\0
a[2]->	D	a	l	t	o	n	\0
a[3]->	B	o	h	n	i	n	g	\0

- The project and makefile are set up so you can test individual components as you complete them:

  1. Test-compile a component by "making" that target. For example, "make cstringsort.o" to debug cstringsort.cpp.

2. Test functionality of the Diff functions by "`make diffcalc.x`" and then executing "`diffcalc.x`".
3. Test functionality of the Sort functions by "`make test.x`" and then executing "`test.x`".
4. Finally debug and test your own main.cpp.

Have fun!