EECS 281 – Winter 2012
Programming Assignment 2
Chicken Sort

Due 14 Feb 2012, 11:59pm

Overview

Drew keeps meticulous records of his backyard chickens. The database is stored in a comma-separated value (CSV) text file on his laptop which contains the name, breed, date of birth, and other information about each bird. Your task is to write a program to sort the database.

As it turns out, CSV is a common file format, suitable not just for chicken records, but for other data as well. For example, the IMDB collection of movie titles is available as a CSV file. In order to be more generally useful, your sort utility should work with CSV files of any kind. The sorts should be efficient, and able to process large input files.

Your program will be called `csort`, and will function much like the standard UNIX `sort` utility. Information about UNIX sort is available on its man page: type `man sort` at the command line. The next section describes the functionality of `csort` in standard UNIX man page format.

UNIX Man Page for `csort`

```
NAME
       csort - sort lines of a CSV text file

SYNOPSIS
       csort [OPTION]... [FILE]

DESCRIPTION
       Write sorted lines of FILE to standard output, with header as
       first line.

       Mandatory arguments to long options are mandatory for short
       options too.

       -a, --algorithm ALGORITHM
              sort using specified ALGORITHM

       -r, --reverse
              reverse the result of comparisons

       -k, --key KEYN[,KEYM...]
              change sort key (column): compare lines by KEYN, then by KEYM
```

```
-o, --output FILE
       write result to FILE instead of standard output

-h, --help
       display this help and exit

-V, --version
       output version information and exit

ALOGORITHM is the sorting algorithm to use, and shall be one of:
bubble, insertion, merge, quick, heap, or bucket.

KEY is the sort key, indicating the column(s) used to compare two
lines.  Columns begin at 1.

With no FILE, read standard input.
```

Sorting Algorithms

For this project, you need to implement 6 sorting algorithms: the four algorithms discussed in class, and two additional algorithms.

1. Bubble sort
2. Insertion sort
3. Merge sort
4. Quick sort
5. Heap sort
6. Bucket sort
7. The algorithm of your choice (*optional*)

The default algorithm used by `csort` (without any `-a` option) may be any of the specified of the algorithms, or any sorting algorithm of your choice. This algorithm may be used to help your chances on the leader board.

Your implementation of insertion sort should be adaptive, like the example from the text book. For merge sort, you may do bottom-up or top-down. Pivot selection for quick sort is your choice.

It is up to you to come up with an implementation of bucket sort, which will include choosing a bucket size. Radix Sort, as a subsort of Bucket Sort, is permitted. For the sorting inside each bucket, you may use any other sort. In addition to the text, bucket sort resources may found at:
http://en.wikipedia.org/wiki/Bucket_sort
http://www.ics.uci.edu/~eppstein/161/960123.html
http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/bucketSort.htm

The comparison between two fields should be determined by ASCII order. Regardless of whether the field contains letters, numbers, decimals, white space or is empty, fields should be sorted in ASCII order. This can be accomplished with the `string` class comparison functions.

You must implement the sorting algorithms on your own. You are allowed to refer to the course lecture notes and any published, publicly available materials (such as your textbook). Do not copy code verbatim from any source: look at the published materials, think about them and learn from them, then write your own code. You may not refer to the programming projects of other students (past or current, here or elsewhere) or any other unpublished work. You are encouraged to use STL to complete this project, however, use of `std::list::sort`, `std::priority_queue`, `std::algorithm`, and `stdlib::qsort` is forbidden. The text "`qsort`" should not appear anywhere in your code. To be clear, we intend for you write the sorting algorithms for this assignment on your own. Finally, you are required to use the `getopt_long` library to do option parsing. A complete example demonstrating `getopt_long` can found at the end of this document.

CSV Input File: The Chicken Database

The chicken database is a plain text file in comma-separated value (CSV) format. The first line is a header containing a label for each column, and the following lines are database entries. Commas separate the fields on each line and a field may contain any characters except commas or newlines. For example, the following header line contains 5 fields where the first column of data (or "field") contains a chicken's name, the second column contains the bread, etc. Note that when splitting a line into fields, the commas are not considered part of the field.

```
name,breed,date of birth,date of death,avg eggs per day
```

Lines following the header contain one database entry each. The lines should have the same number of fields as the header: any line that contains a differing number of fields should generate an error. There may be whitespace (spaces or tabs); do not remove or alter whitespace. A field will not contain a comma. Fields may vary in length, and some may be empty, but files will contain only printable text and whitespace (no arbitrary binary data). Duplicate lines may be present. For example:

```
name,breed,date of birth,date of death,avg eggs per week
Marilyn,Isa Red,2011-04-24,N/A,6
Mildred,Araucana,2011-05-01,N/A,3
Mabel,Isa Red,2011-04-26,N/A,5
Myrtle,Araucana,2011-08-01,N/A,0
Myrtle,Araucana,2011-05-01,2011-07-13,0
```

The exact contents of the input file may vary widely: the number of columns, the header fields, and the number of lines may be different in different files.

Output and Examples

Your program should first print the header line, unmodified, followed by the remaining input lines in sorted order. No input line should be modified, only reordered and printed. The default behavior of `csort` will read from standard input and print to standard output, sorting the lines. By default, lines are compared by every field in ascending order. For example, if the above input example were named

`chickens.csv`, the output of "`./csort < chickens.csv`" would print the following to standard output:

```
name,breed,date of birth,date of death,avg eggs per week
Mabel,Isa Red,2011-04-26,N/A,5
Marilyn,Isa Red,2011-04-24,N/A,6
Mildred,Araucana,2011-05-01,N/A,3
Myrtle,Araucana,2011-05-01,2011-07-13,0
Myrtle,Araucana,2011-08-01,N/A,0
```

Since the program is designed to work like a standard UNIX utility, the same results could be obtained by running "`./csort chickens.csv`". In this case, the program would open the file and read it, whereas in the case of "`./csort < chickens.csv`", the "<" character tells the UNIX shell to open the file and *redirect* the file contents to csort's standard input.

In some cases, duplicate keys may exist in a column. In the above example, there are two chickens named Myrtle, one that was born in May and brutally murdered by a raccoon in July. A second chicken replaced Myrtle in August and was named in Myrtle's honor. By default, csort will first sort by column 1 (in this case, name), then by column 2, etc. Columns 1 and 2 are identical for both Myrtle entries, but column 3 differs in date of birth. Thus, the Myrtle born earlier (i.e., whose birthday comes first in ASCII order) is printed first.

With the "-k" option, csort is able to use any column or even multiple columns, as the sort key(s). For example, "`./csort -k 2 chickens.csv`" will sort the file by column 2. The ordering among fields whose second column matches is undefined. In other words, some sorts may re-order lines whose second field matches, while others maintain the initial ordering among such lines. This is dependent on the stability of the sort. For stable sorts, there is one correct answer, while different implementations of an unstable sort may yield different correct answers. Once again, note that ONLY the specified columns are compared to determine the sorting order. For example, the correct result of "`./csort -k 2 -a insertion chickens.csv`" for a stable sort (like insertion sort) is:

```
name,breed,date of birth,date of death,avg eggs per week
Mildred,Araucana,2011-05-01,N/A,3
Myrtle,Araucana,2011-08-01,N/A,0
Myrtle,Araucana,2011-05-01,2011-07-13,0
Marilyn,Isa Red,2011-04-24,N/A,6
Mabel,Isa Red,2011-04-26,N/A,5
```

With the "-k" option, multiple columns maybe specified, separated by commas. In this case, sort the lines, using the fields to compare in the specified order. `./csort -k 2,1 -a insertion chickens.csv` :

```
name,breed,date of birth,date of death,avg eggs per week
Mildred,Araucana,2011-05-01,N/A,3
Myrtle,Araucana,2011-08-01,N/A,0
Myrtle,Araucana,2011-05-01,2011-07-13,0
Mabel,Isa Red,2011-04-26,N/A,5
Marilyn,Isa Red,2011-04-24,N/A,6
```

Adding the "-r" option reverses the order of the sort, for example `./csort –k 2,1 –a insertion -r chickens.csv`:

```
      name,breed,date of birth,date of death,avg eggs per week
      Marilyn,Isa Red,2011-04-24,N/A,6
      Mabel,Isa Red,2011-04-26,N/A,5
      Myrtle,Araucana,2011-05-01,2011-07-13,0
      Myrtle,Araucana,2011-08-01,N/A,0
      Mildred,Araucana,2011-05-01,N/A,3
```

The following example are intended to clarify functionality csort's command line options:

```
./csort chickens.csv
```
[sort with the algorithm of your choice]

```
./csort –-key 2 chickens.csv
```
[same output as `./csort -k2 chickens.csv`]

```
./csort –k 2, chickens.csv
Error

./csort –k ,1 chickens.csv
Error

./csort –k 0 chickens.csv
Error

./csort –k 2,1,0 chickens.csv
Error

./csort –k 6 chickens.csv
Error

./csort –z chickens.csv
Error

./csort file_that_does_not_exist.csv
open failed: file_that_does_not_exist.csv

./csort –-version
csort [ANY VERSION NUMBER]
written by [YOUR NAME]
[ exit(0) ]

./csort –h -V -o FILE chickens.csv
```
[Print help message and exit(0). Do not print version. Do not read file. Do not create file. Do not sort.]

```
./csort -V -h
```
[Print version message and exit(0). Do not print help message.]

```
./csort --algorithm insertion --reverse chickens.csv
```
[same as `./csort --algorithm insertion -r chickens.csv`]
[same as `./csort -a insertion -r chickens.csv`]
[same as `./csort -a insertion --reverse chickens.csv`]
[same as `./csort -a insertion -r -k 1,2,3,4,5 chickens.csv`]

It is possible to force an unstable sort to be stable by specifying every column, in order, using the "-k" option to specify every column in the file. For example, the output of "`./csort --algorithm quicksort -k 1,2,3,4,5 chickens.csv`" will be the same as the output of "`./csort -a insertion -k 1,2,3,4,5 chickens.csv`"

Since csort may operate with input files, output files, standard input or standard output, you may use filenames or shell redirection for input and output.

```
./csort chickens.csv
```
[same output as `./csort < chickens.csv`]
[same output as `cat chickens.csv | ./csort`]

When the "`-o`" option is used, do not print anything (except errors) to standard output.

```
./csort chickens.csv -o out.csv
```

Redirecting standard output using the shell will yield the same result as using the "`-o`" option.

```
./csort chickens.csv > out.csv
```

When a different input file with different fields is used, your program should still work correctly:

[puppies.csv]

```
type,size
weimaraner,medium
mastiff,large
chihuahua,small
```

```
./csort puppies.csv
type,size
chihuahua,small
mastiff,large
weimaraner,medium
```

Testing

Part of this project is to prepare several test cases that will expose defective sorting algorithms. Each test case is a text file that will be sorted on the autograder using a set of sorting routines that we have prepared. One of our sorting routines is correct and the others have bugs. If your test case causes the correct routine and the incorrect routine to produce different output, the test case is said to expose that bug. Each test case must be in a file with a name of the form `testN-OPTIONS.csv`, where N is the test number (0-9) and OPTIONS are the command line options to be used when calling `csort` on the test, with the spaces removed. For example, a test with file name `test0-aquicksort-k2,1-r.csv` would be run as ./`csort -a quicksort -k 2,1 -r test0-aquicksort-k2,1-r.csv` For the purpose

of testing, only short options (options that begin with a single "-") should be used. Additionally, only file input will be used for testcases (no standard input).

Please do not submit unnecessarily large test files. We ask that your complete, uncompressed code and testcases not exceed 10MB in total. To determine the size of your code, clean up your directory, remove your old submission tarball, and use the `du` command:

```
make clean
rm -f submit.tgz
du -sh .
```

Test with no command line options, using all defaults.

```
test1.csv
./csort test1.csv
```

Test reverse sorting with default algorithm.

```
test5-r.csv
./csort -r test5-r.csv
```

Invalid test due to long option format. Don't do this.

```
test0--algorithmquicksort-k2,1-r.csv
```

Test is missing the "-a" option, but specifies quicksort. Don't do this.

```
test0-quicksort-k2,1-r.csv
```

Test tries to write to FILE. Don't do this.

```
test0-oFILE.csv
```

Filename does not start with the word "test". Don't do this.

```
file.csv
```

Test number exceeds 9. Don't do this.

```
test10-r.csv
```

Submission and Grading

Your grade will be based on code correctness, testcase effectiveness, algorithm efficiency, and code style. Part of your grade will be determined by the performance of your algorithms. Fast running algorithms will receive all possible performance points. Slower running algorithms may receive only a portion of the performance points. A leader board will be posted on the course website, and will be updated frequently during the project. You may track your progress there.

A small portion of your grade is dependent on coding style. Good coding style has not changed since project 1.

For your code, put all the files you want to turn in (all needed files, as well as test cases) in some directory other than your home directory. This will be your "submit directory". Before you turn in your code be sure that:

- Your makefile is called `Makefile`
- Typing the command make will generate an executable called `csort`
- You have deleted all .o files, executable(s), and tilde (~) – appended files, which are generated automatically by emacs. Typing "`make clean`" accomplishes this.
- Your test cases (up to 10) have names of the form `testN-OPTIONS.csv`
- The total size of your program and test cases does not exceed 10MB, uncompressed
- If you created your files on a Windows machine, run your files through the `dos2unix` utility
- Your code compiles and runs correctly using the g++ compiler on CAEN Linux machines. Even if everything seems to work on another operating system or with different versions of g++, the course staff will not support anything other than the g++ running on CAEN Linux
- The total size of your program and test cases does not exceed 10MB
- In order to speed up your code, be sure to omit g++ debugging flags (e.g., `-g`), and include the `-O3` optimization flag in your final submission

Turn in all of the following files:

- All your `.h`, `.cpp and .cc` files for the project
- Your `Makefile`
- Your test case files

You must prepare a compressed `tar` archive (a `.tgz` file) of all your files to submit to the autograder. Put all your files for submission (and nothing else) in one directory. Go into this directory and run:
`tar czvf submit.tgz *.cpp *.cc *.h Makefile test?*.csv`

This will prepare a suitable file in your working directory. Submit your `submit.tgz` to the autograder at https://grader8.eecs.umich.edu/. You can safely ignore and override any warnings about an invalid security certificate.

Once the autograder is available, you can submit your code to it as many times are you like. However, you will only receive feedback for the first three submissions each day. Correctness and timing feedback for these submissions will come by email at a later time, following submission. The last submission will be graded.

Grades will be assigned using the following point distribution:

- 12% correctly sort our test cases with bubble sort
- 12% correctly sort our test cases with insertion sort
- 12% correctly sort our test cases with merge sort
- 12% correctly sort our test cases with heapsort
- 12% correctly sort our test cases with quicksort
- 12% correctly sort our test cases with bucket sort
- 10% provide test cases that expose our buggy sorts
- 10% efficiency of bucket sort on large test cases
- 8% coding style

Option Parsing Example Code

Parsing the options in a real command line program can be complicated. Luckily, there are libraries to do this for you. The following example C++ code demonstrates the use of the `getopt` library. It takes care of identifying which arguments are options and which are files. It generates an error message for omitted option arguments and incorrect options.

The following example program accepts three options, as well as multiple filename arguments. The options are "-n", which should be followed by a number (e.g. "-n 5"), "-s", which requires a string argument ("-s asdf"), and -V, which does not accept any argument. Any number of file names may follow the options, or be mixed with the options.

```
/*
  getopt_example.cpp
  Example code for using getopt_long

  by Andrew DeOrio
  Jan 2012
 */

#include <iostream>
#include <string>
#include <getopt.h>
#include <stdlib.h>

using namespace std;

int main (int argc, char **argv) {

  // variables for options
  int numopt;
  string stringopt;

  // specify the options
  static struct option long_options[] =
    {
      {"numopt",    required_argument, 0, 'n'},
      {"stringopt", required_argument, 0, 's'},
      {"version",   no_argument,       0, 'V'},
      {0, 0, 0, 0}
    };

  // parse all the options
  while (1) {

    // getopt_long stores the option index here
    int option_index = 0;

    // current option: "n:s:V" string includes a letter for each option.
    // Those options with a required argument are followed by ":"
    int c = getopt_long (argc, argv, "n:s:V", long_options, &option_index);

    // Detect the end of the options
```

```
    if (c == -1)
      break;

    switch (c) {

    case 0:
      // If this option set a flag, do nothing else now
      break;

    case 'n':
      numopt = atoi(optarg);
      cout << "--numopt " << numopt << endl;
      break;

    case 's':
      stringopt = optarg;
      cout << "--stringopt " << stringopt << endl;
      break;

    case 'V':
      cout << "getopt example 0.1\n"
           << "by Andrew DeOrio"
           << endl;
      exit(0);

    default:
      abort ();

    }// switch

  } // while

  // remaining arguments are file names
  for (; optind < argc; optind++) {
    cout << "FILE = " << argv[optind] << endl;
  }

} //main
```

`istream` Example Code

Like real UNIX utilities, `csort` may read from either standard input or from a file. Regardless of the input source, it should produce the same output. For this reason, it can be helpful to write functions that process input data with the input's source abstracted away. In other words, write a function that reads input without prior knowledge of whether it will come from standard input, from a file, or elsewhere.

The `istream` class can help with this task. Both `iostream` class (where `cin` comes from) and `fstream` class (where `ifstream` file input comes from) inherit from `istream`. Thus, a function that accepts an istream input can be called on either standard input or an open file stream. The following example program illustrates this concept. File writing can be accomplished in similar fashion.

```
/* istream_example.cpp
   Example code for using streams to read either files or standard input.

   by Andrew DeOrio
   Jan 2012

   compile with "g++ istream_example.cpp -o istream_example"
   run with "./istream_example < /proc/version"
*/

#include <iostream>
#include <fstream>
#include <string>

using namespace std;


// read a stream and print it to standard output
void read_stream(istream& is) {

  // print lines to standard output
  string line;
  while ( getline(is, line) ) {

    cout << line << endl;

  }

}


int main(int argc, char* argv[]) {

  // read from file
  string filename = "/proc/version";
  ifstream fin( filename.c_str() );
  if ( !fin.is_open() ) cout << "open failed: " << filename << endl;
  cout << "[" << argv[0] << ": reading file " << filename << "]" << endl;
  read_stream( fin );
```

```
  fin.close();

  // read from standard input
  cout << "\n[" << argv[0] << ": reading standard input]" << endl;
  read_stream(cin);

} // main
```

Frequently Asked Questions and Solutions

Q. My project passes all of my test cases, but doesn't pass any of the test cases on the autograder?

A. Make sure the lack of any option in the command line does not make the program error or not work correctly. Specifically, the –k option does NOT need to be present to have the program work correctly.

Q. When I submit to the autograder, it says that the project doesn't exist/I don't have permissions to submit to the project?

A. Make sure all your test cases aren't too big. When you submit a submit.tar.gz file that is too large, it is possible that the autograder is refusing it because of the size. Test cases you submit do not need to be very large to be effective; do not try to stress test the autograder.

Q. I keep getting SIGSEGV on the autograder. What does that mean?

A. Your program is experiencing a segmentation fault on that case. Check your pointers to make sure you aren't trying to dereference data you do not have access to.

Q. I keep getting SIGABRT on the autograder. What does that mean?

A1. An error is occurring in a library function you are calling.

A2. Your program reached an `assert` with a false value.

A3. You are running out of memory, or somehow otherwise calling `new` or `delete` erroneously. This is most likely caused by an infinite loop.

Q. I keep getting a SIGFPE on the autograder. What does that mean?

A. FPE stands for floating point error, which most likely means you are dividing by zero somewhere. You cannot divide by zero.

Document Errata

Version 2012-01-21

Original assignment.

Version 2012-02-01

Added `std::list::sort` and `std::priority_queue` to list of forbidden libraries.

Version 2012-02-02

Added examples for combinations of options that include -h and -V, e.g., `./csort -h -V`. Fixed error in example discussing stable sorts, by removing an extraneous "-r" from the example.