Curtin University — Department of Computing
UNIX and C Programming (COMP1000)
Semester 2, 2016

Assignment

Due: Week 13 – Friday 28 October, 10:00 am
Weight: 20% of the unit mark.

Your task for this assignment is to create, in C89, a stand-alone sorting program.

1 Documentation

You must thoroughly document your code using C comments (/* ... */).

For each function you define and each datatype you declare (e.g. using struct or typedef), place a comment immediately above it explaining its purpose, how it works, and how it relates to other functions and types. Collectively, these comments should explain your design. (They are worth substantial marks – see Section 7.)

2 Program Outline

Your program will deal with one input file: a .csv text file to sort (the "user file"), and a single output file containing the sorted text.

Your program must do the following:

a) Take two command-line parameters – the names of the input and output files. these can be in either order, so you will also need command line switches (`i and o`)

   Examples:
   ```
   sortingAssignment i infile o outfile
   sortingAssignment o outfile i infile
   ```

   Invalid command line arguments should cause the program to output a usage message, and exit.

b) Read the input file and store its contents in a (generic) linked list. The format of the input file is specified in section 2.1

c) Output a menu to the user requesting how the user would like the contents of the file sorted. The user must be able to sort on any of the columns, in ascending or descending order.

d) Sort the rows of the input file using a generic insertion sort (in-place). Your insertion sort function MUST be generic - that is it must import a generic linked list and a compare function, export the sorted linked list, and return an integer specifying if anything went wrong - 0 for success, or a negative value for any specific errors.
   Note. Insertion sort is fully described in appendix a

e) Write the sorted data to the specified output file, using the same format as the input file.

(continues next page)

2.1 The input file

The input file will be in .csv format, with the column names specified on the first line, and its data type (string or integer) specified after the name in brackets. There could be any number of columns, and any number of rows.

This is an example of a valid input file:

```
name (string), age (integer), address (string)
mark, 21, at home
jordan, 19, his place
etc
```

Sorted on age, ascending:

```
name (string), age (integer), address (string)
jordan, 19, his place
mark, 21, at home
etc
```

and another:

```
loc (string), desc (string), lat (integer), long (integer)
shark bay, nice and warm, 12543, 34216
esperance, cold and windy, , 4287
etc
```

Sorted on loc, ascending:

```
loc (string), desc (string), lat (integer), long (integer)
esperance, cold and windy, , 4287
shark bay, nice and warm, 12543, 34216
etc
```

Note: You must state any valid assumptions you make about the input file.
If a value is missing from the file, the sorting algorithm must place it last in the sorted list.
If there is an unrecognised datatype name, or the file is otherwise not in the expected format, your program must issue an error message detailing the problem. When this happens, your program must not attempt to sort anything.

3 Makefile (or “How to Actually Get Marks, part 1”)

You must create a makefile, and it must actually work. That is, the marker will type:

```
[user@dir]$ make
```

This is the only way the marker will attempt to compile your code. If your makefile does not work, then, according to the marking guide, your code does not compile or run. (The marker will delete all existing .o files and executable files beforehand.)

Your Makefile must be written by you, and not automatically generated. It must be structured properly in accordance with the lecture notes.

(continues next page)

4 Testing (or “How to Actually Get Marks, part 2”)

Your program must work on either:

- The lab machines in building 314, and/or
- One of the saeshell0Xp.curtin.edu.au machines (where “X” is 1, 2, 3 or 4).

To begin with, construct a small-scale test file containing only a few columns and lines. As you begin to fix bugs, try larger test files.

All user input must be "sanity" checked. Invalid user input should not cause the program to stop (other than invalid command line arguments).

Ensure there are no memory leaks - you must not rely on the program closing to free your memory. You will lose marks for any memory issues.

5 README.txt

Prepare a text file called README.txt (not using Word or any other word processor), that contains the following:

- A list of all files you’re submitting and their purpose.
- A statement of how much of the assignment you completed; specifically:
  - How much of the required functionality you attempted to get working, and
  - How much actually does work, to the best of your knowledge.
- A list of any bugs or defects you know about, if any.
- A statement of which computer you tested your code on. This must be either:
  - One of the four saeshell0Xp machines, or
  - One of the building-314 lab machines. In this case, specify the room number (e.g. 314.220) and the “Service Tag” – a unique 7-character ID found on a black sticker on the top of the each machine.

6 Submission

You must keep a copy of all required files in your $HOME/UCP/assignment directory, including files that you used for testing. These must not be touched after the due date and time for the assignment. You must also submit the following electronically, via the assignment area on Blackboard, inside a single .tar.gz file (not .rar or other formats):

- Your makefile and README.txt.
- All .c and .h files (everything needed for the make command to work).
- a completed “Declaration of Originality” (appendix b)

You are responsible for ensuring that your submission is correct and not corrupted (download and check it!). You may make multiple submissions, but only your newest submission will be marked. The late submission policy (see the Unit Outline) will be strictly enforced. A submission 1 second late, according to Blackboard, will be considered 1 day late. A submission 24 hours and 1 second late will be considered 2 days late, and so on.

(continues next page)

7 Mark Allocation

Every valid assignment will be initially awarded 100 marks. Marks will then be deducted for the following:

up to -50% Commenting.
You will not lose any marks if you have provided good, meaningful explanations of all the files, functions and data structures needed for your implementation.

up to -50% Coding practices
You will not lose any marks if you have followed good coding practices, and your code is well-structured, including being separated into various, appropriate .c and .h files.

up to -100% functionality.

– You have correctly implemented the required functionality, according to a
visual inspection of your code by the marker.

up to -100% working product.

You will not lose any marks if your program compiles, runs and performs the required tasks without unexpected error. The marker will use test data, representative of all likely scenarios, to verify this. You will not have access to the marker's test data.

You may lose this entire component of your mark by either (a) not having a working makefile, OR (b) failing to solve memory issues.

Once your mark has been calculated, the practical signoff result will be applied as follows:

Your total practical signoffs will be awarded a mark out of 10, which will have the following effect on the assignment mark.

Total	Effect
8 - 10	100
7 - 7.9	90
6 - 6.9	80
5 - 5.9	70
3 - 4.9	50
0 - 2.9	30

(continues next page)

8 Academic Misconduct – Plagiarism and Collusion

If you accept or copy code (or other material) from other people, websites, etc. and submit it, you are guilty of plagiarism, unless you correctly cite your source(s). Even if you extensively modify their code, it is still plagiarism.

Exchanging assignment solutions, or parts thereof, with other students is collusion.

Engaging in such activities may lead to a grade of ANN (Result Annulled Due to Academic Misconduct) being awarded for the unit, or other penalties. Serious or repeated offences may result in termination or expulsion.

You are expected to understand this at all times, across all your university studies, with or without warnings like this.

End of Assignment

(appendices follow)

Appendix a - insertion sort

Inspired from the idea of adding items to a list in sorted order.
Every time a new item is added, insert it in sorted position.

Can also be applied to sorting an existing list.
Maintain a marker and insertion-sort the element at the marker into the items to the left of the marker.

> ie: take the next item and insert it in sorted order into the sub-list that precedes the item.
>
> Start the marker at the front of the list and move it up by one after each inserted item. Then items before marker will be sorted.
>
> Searches for the insert position backwards so that we can take advantage of semi-sorted lists.

Pseudo-code for insertion sort using integers in an array.

```
METHOD InsertionSort IMPORT array EXPORT array
FOR nn ← 1 TO array.length–1 DO                   ← Start inserting at element 1 (0 is pointless)
  ii ← nn                                          ← Start from the last item and go backwards
  temp ← array[ii]
  WHILE (ii > 0) AND (array[ii-1] > temp) DO      ← Insert into sub-array to left of nn
                                                     Use > to keep the sort stable
    array[ii] ← array[ii-1]                       ← Shuffle until correct location
    ii ← ii – 1
  ENDWHILE
  array[ii] ← temp
ENDFOR
```

Note carefully - this algorithm is for integer arrays - you will need to convert it to sort LinkedLists of strings/integers. Remember that a LinkedList is of unknown length, so you should NOT use a for loop.