School of Computing and Information Systems

comp10002 Foundations of Algorithms
Semester 2, 2024
Assignment 1

Learning Outcomes

In this assignment you will demonstrate your understanding of arrays, strings, functions, and the `typedef` facility. You may also (but are not required to) use `structs` (see Chapter 8) if you wish. You must *not* make any use of `malloc()` (Chapter 10) or file operations (Chapter 11) in this project.

TSV Files

The *tab separated format* is a common and convenient way of storing structured data in a text file. The first line of the file stores a set of c "column header" entries all represented as text strings that may contain alphanumeric, blank characters and punctuation characters, but no tab (`'\t'`) or newline (`'\n'`) characters, stored with $c-1$ tab characters between the c entries to denote the breaks between the strings, and with a final newline character at the end of the line. A set of r following rows each also store c data items, represented in the same format. For example, consider the data file `test0.tsv`, with tabs and newline characters shown explicitly:

```
Year\tEvent\tGender\tCountry\tMedal\n
2024\tSwimming\tWomens\tNew Zealand\tfirst: gold\n
2024\tSwimming\tWomens\tChina\tsecond: silver\n
2024\tSwimming\tWomens\tIndonesia\tthird: bronze\n
2024\tCycling\tWomens\tChina\tfirst: gold\n
2024\tCycling\tWomens\tNew Zealand\tsecond: silver\n
2024\tCycling\tWomens\tNew Zealand\tthird: bronze\n
```

This test file has $c = 5$ columns, headed "`Year`", "`Event`", "`Gender`", "`Country`", and "`Medal`" respectively; and contains $r = 6$ data rows that show (for example) that in "`2024`" the country "`New Zealand`" is associated with the medals "`second: silver`" and "`third: bronze`" in the sport "`Cycling`" in connection with the gender "`Womens`". Note that r and c are not stored, and are implicit via the appearance of the newline and tab characters.

Before doing anything else, you should copy the skeleton program `ass1-skel.c` from the LMS Assignment 1 page, and read through the code. Check that you can compile it via either `Grok` or a terminal shell and `gcc`. Once you have it compiled, you should see this when you run it:

```
mac: ./ass1-skel
ta daa!
```

You are now ready to start adding functionality.

Stage 1 – Reading and Printing (12/20 marks)

Extend the skeleton program so that it that reads a TSV input stream from `stdin` and builds a corresponding internal data structure using a 2d array of strings. You may assume that at most $1{,}000$ input lines will be presented, that each input line contains at most 30 columns of information, and that each entry contains at most 50 characters.

The required output for this first stage is a summary of what was read, here is an example:

```
mac: ./ass1-soln < test0.tsv
Stage 1
input tsv data has 6 rows and 5 columns
row 6 is:
   1: Year       2024
   2: Event      Cycling
   3: Gender     Womens
   4: Country    New Zealand
   5: Medal      third: bronze
```

To be eligible for full marks your output must *exactly* match. Other input files and example output files are linked from the LMS page. Note that the columns are labeled from 1, and that the data rows are also labeled starting with 1 – this information is for non-computer scientists!

You may assume throughout that all input files you will be provided with will be "correct", with all rows having the same number of entries; the number of rows and columns being within the specified bounds; the length of each entry being within the specified bound; and with no additional (or missing) newline or tab characters. The last character of all valid input files will be a `'\n'` character, and if you create your own test files, make sure you do the same.

Note that the `getfield()` function handles the PC-versus-Unix newline differences. These difference have the potential to be very frustrating if you don't allow for them (see the section about this in the "Programming Tips" page linked from the LMS).

Temptations to avoid: use the `getfield()` function provided in the skeleton (rather than try and do it yourself with `scanf()` or etc); do *not* try and use `malloc()` or the other similar functions from Chapter 10; do *not* use any of the file operations described in Chapter 11 (except for `fflush(stdout)` when debugging, see the FAQ page); and do *not* print any prompts of any sort.

Stage 2 – Sorting (16/20 marks)

Now modify your program so that it accesses a sequence of integers from the command-line, each of them a value between 1 and c, indicating which column(s) should be used as sort keys. For example, for `test0.tsv`, if columns 4 and then 2 are specified, the r data rows in your internal format are to be reordered so that they are sorted first by `Country` (as a string, using `strcmp()`), and then where the `Country` fields are equal, using `Event` as a secondary key. Where two rows have the same values in the selected columns, they should be retained in the same order as they were originally; that is, your sort should be *stable*. The header row should not be sorted. For `test0.tsv`, sorting by the program arguments "4 2" gives:

```
Year    Event          Gender         Country        Medal
2024    Cycling        Womens         China          first: gold
2024    Swimming       Womens         China          second: silver
2024    Swimming       Womens         Indonesia      third: bronze
2024    Cycling        Womens         New Zealand    second: silver
2024    Cycling        Womens         New Zealand    third: bronze
2024    Swimming       Womens         New Zealand    first: gold
```

Think of that (hint, hint) as being "debugging" output. Then the actual Stage 2 output requires that the first data row, the middle data row (that is, row $\lceil r/2 \rceil$), and the last data row:

```
mac: ./ass1-soln 4 2 < test0.tsv
<<<Stage 1 output here, see above>>>
Stage 2
sorting by "Country",
   then by "Event"
```

```
row 1 is:
   <<<5 output lines for row 1>>>
row 3 is:
   <<<5 output lines for row 3>>>
row 6 is:
   <<<5 output lines for row 6>>>
```

Full examples are provided at the LMS page. If there are no column numbers supplied on the command line, your program should exit after completing Stage 1, without any Stage 2 output. You may assume that each specified integer will be in the range 1 to c inclusive, and that no values will be repeated.

Use insertionsort rather than quicksort (this project is about programming in C, not about algorithm implementation, and there won't be a penalty for using insertionsort). And keep it simple and elegant – use functions sensibly, including a suitable comparison function and a suitable swapping function.

Stage 3 – Tabulation (20/20 marks)

Now use the sorted TSV values to generate a report that shows counts of rows matching the same selected column combination:

```
mac: ./ass1-soln 4 2 < test0.tsv
<<<Stage 1 and 2 output here, see above>>>
Stage 3
------------------
Country
    Event    Count
------------------
China
    Cycling      1
    Swimming     1
Indonesia
    Swimming     1
New Zealand
    Cycling      2
    Swimming     1
------------------
```

The last column starts one space to the right of the longest entry (including the column header) in the last column getting printed, with the counts then printed as five-digit numbers under the word "`Count`". There are no tabs in the Stage 3 output, and the formatting is based solely on spaces. Marks will be deducted for layout discrepancies. The format descriptor `*` will be your friend to help achieve this goal, use `printf("%-*s", num, str)` to print the string `str` left-justified in `num` columns, where `num` is a variable/value calculated while your program is executing.

General Tips...

You will probably find it helpful to include a `DEBUG` mode in your program that prints out intermediate data and variable values. Use `#if (DEBUG)` and `#endif` around such blocks of code, and then `#define DEBUG 1` or `#define DEBUG 0` at the top. Turn off the debug mode when making your final submission, but leave the debug code in place. The FAQ page has more information about this.

The sequence of stages described in this handout is deliberate – it represents a sensible path through to the final program. You can, of course, ignore the advice and try and write final program in a single effort, without developing it incrementally and testing it in phases. You might even get

away with it, this time and at this somewhat limited scale, and develop a program that works. But in general, one of the key things that makes some people better at programming than others is the ability to see a design path through simple programs, to more comprehensive programs, to final programs, that keeps the complexity under control at all times. That is one of the skills this subject is intended to teach you. And if you submit each of the stages as you complete it, you'll know that you are accumulating evidence should you need to demonstrate your progress in the event of a special consideration application becoming necessary.

Boring But Important...

This project is worth 20% of your final mark, and is due at **6:00pm on Friday 13 September**.

Submissions that are made after the deadline will incur penalty marks at the rate of two marks per (part or all) working day. For example, submissions made after 6:00pm Friday and before 6:00pm Monday will lose two marks. Multiple submissions may be made; only the last submission that you make before the deadline will be marked. If you make any late submission at all, your on-time submissions will be ignored, and if you have not been granted an extension, the late penalty will be applied.

A rubric explaining the marking expectations is linked from the LMS, and you should study it carefully. Marks and feedback will be provided approximately two weeks after submissions close.

You need to submit your program for assessment **via the link to GradeScope at the bottom of the LMS Assignment page**. Submission is **not possible through Grok**.

Academic Honesty You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** have any "accidents" that allow others to access your work; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program, pointing out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. In the past students have had their enrolment terminated for such behavior.

The skeleton program includes an Authorship Declaration that you must "sign" and include at the top of your submitted program. Marks will be deducted if you do not include the declaration, or do not sign it, or do not comply with its expectations. A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions. Students whose programs are identified as containing significant overlaps will have substantial mark penalties applied, or be referred to the Student Center for possible disciplinary action.

Nor should you post your code to any public location (`github`, `codeshare.io`, etc) while the assignment is active or prior to the release of the assignment marks.

Special Consideration Students seeking extensions for medical or other "outside my control" reasons must lodge a request following the FEIT process linked from the LMS page, and do so as soon as possible after those circumstances arise. If you attend a GP or other health care service as a result of illness, be sure to obtain a letter from them that describes your illness and their recommendation for treatment. Suitable documentation should be attached to **all** extension requests.

And remember, algorithms are fun!