School of Computing and Information Systems

comp10002 Foundations of Algorithms
Semester 2, 2019
Assignment 1

Learning Outcomes

In this project you will demonstrate your understanding of arrays, strings, and functions. You may also use `typedefs` and `structs` if you wish (see Chapter 8) – and will probably find the program easier to assemble if you do – but you are not required to use them in order to obtain full marks. You should not make any use of `malloc()` (Chapter 10) or file operations (Chapter 11) in this project.

Text Formatting

We all want our written work to look good. But when preparing text with a text editor it is annoying to (for example) try and adjust the line-lengths in a document every time we alter it, or if we decide that the text width needs to be narrower. So we use approaches that let us describe the *layout* of documents, and apply a compiler-like tool to generate the exact formatting that we are after. For example, this document (and most other things you will see in this subject, including the textbook) was prepared using a typesetting tool called LaTeX, and you can see the source code for it linked from the LMS/FAQ page. Use of LaTeX allows math expressions like $\lim_{n\to\infty} \sum_{i-1}^{n}(1/i^2) = \pi^2/6$ (plus far more complex expressions too) to be elegantly presented, and allows careful and consistent formatting to be applied throughout long documents. A WYSIWYG[1] (or more usually, WYCSIAYCG[2]) tool like Word might be employed to write a one-page reference letter, but for technical work academics in the maths, physics, and computing sciences almost always prefer LaTeX. (It was originally invented by one of the most famous computer scientists of all, Stanford University's Donald E. Knuth, and now has a world-wide community of developers and language extenders.) Similarly, html tags in conjunction with a style sheet file describe how the contents of a web page are to be rendered and presented.

In this assignment we are going to journey back to the 1970s, and implement a simple text formatting tool based entirely on fixed-width terminal fonts and character-based output.[3] The formatting "commands" will be indicated by input lines that start with a period, "`.`", with the command indicated by the letter in the next character position. For example, the characters "`.p`" in the first two positions of a line are (in this simple language) the signal to start a new paragraph in the output.

Stage 1 – Filling Lines (8/15 marks)

The first version of your program should:

- read text input from `stdin` (you may assume that input lines will be at most 999 characters long);
- completely discard the content of all lines that commence with a period character;
- replace all instances of (multiple) whitespace characters (blanks, tabs, newlines) by single blanks;
- re-insert newline characters in a greedy line-by-line manner, so that each output line emitted is as long as possible, but not longer than 50 characters;
- with the exception that if there is an input token/word that is greater than 50 characters, it goes on an output line by itself, and that line can be greater than 50 characters long.

[1] What you see is what you get.

[2] What you can see is *all* you can get.

[3] Alistair wrote his 1985 PhD thesis using a tool like this, see `http://hdl.handle.net/10092/7926`; Artem is younger, he used LaTeX for his thesis in 2012, see `https://publishup.uni-potsdam.de/frontdoor/index/index/docId/5705`.

The lines in the output should be arranged so that the left margin has a default of 4 initial blanks, and hence that the first non-blank character in each line is in column 4 (counting the columns from zero), and the last character in any line (except as noted in connection with very long tokens) is never beyond column 53 in each output line. For example, if the file `test0.txt` contains

```
one two three four five six seven eight

nine


ten eleven twelve thirteen fourteen
                        fifteen         sixteen
                        seventeen
 eighteen nineteen twenty.
.b
Enough of the numbers already, what about:
101, 102, 103, 104, 105, 106, 107, 108, and 109?
Plus 110, 111, 112, 113, 114 and all of 115 116 117 118 and 119.
```

the output should be

```
0----5---10---15---20---25---30---35---40---45---50---55---60
mac: ass1-soln < test0.txt
    one two three four five six seven eight nine ten
    eleven twelve thirteen fourteen fifteen sixteen
    seventeen eighteen nineteen twenty. Enough of the
    numbers already, what about: 101, 102, 103, 104,
    105, 106, 107, 108, and 109? Plus 110, 111, 112,
    113, 114 and all of 115 116 117 118 and 119.
mac:
0----5---10---15---20---25---30---35---40---45---50---55---60
```

where the two "rulers" are *not* part of the output and are provided here to help you count character positions. (But, hint hint, they *were* part of the output while the sample solution was being debugged.) See the FAQ page linked from the LMS for more example input files and the output that is expected. Note that the values 4 and 50 should be held in variables assigned via initial `#define` starting points – you will need to be able to change their values in Stage 2.

The `getword()` function that was discussed in class may look like a tempting starting point. But you need to be able to examine the first character of each line without necessarily consuming it, and so a main loop that reads and processes lines is probably a better bet. You can then break that line up into tokens as required. You should also pay close attention to the item on the FAQ page that discusses the issues that may arise in connection with newline characters.

Note also that while it is possible to achieve the limited functionality required in this stage with a program that is little more than a "`while (getchar())`" loop, you will only receive 8/8 for a Stage 1 submission if your program shows that you have planned it in a way that allows the functionality required for Stage 2 and Stage 3 to be added without needing to completely rewrite it.

Stage 2 – Processing Commands (13/15 marks)

Once you have the Stage 1 program operational (and submitted, so that it is "on the record"), extend your program so that it recognizes and acts upon the following commands, where the "`.`" that indicates a formatting command always appears in the first position of an input line, straight after a newline character:

- `.b` – break the current line, so that the next input token starts at the beginning of the next line;
- `.p` – leave a blank line and start a new paragraph without altering the margins, with the next input token starting the new paragraph;
- `.l` *nn* – alter the left margin from its current value (default initial value of 4) to the new value *nn*, and start a new paragraph;
- `.w` *nn* – alter the width of each line from its current value (default initial value of 50) to the new value *nn*, and start a new paragraph.

There are further examples showing the required output linked from the LMS.

Stage 3 – Adding Structure (15/15 marks)

Now add two further commands:

- `.c` – take the remaining contents of this line and center them within the current output width. If the remaining contents cannot fit within the current output width, then it should be placed so that it starts at the left margin and overflows beyond the current width. When there is an odd number of spaces to be assigned, the rounded-down half should be at the beginning of the line, and the rounded-up half at the end of the line.
- `.h` *nn* – take the remainder of the contents of this line and use them as a section heading at the level indicated by *nn*. The Section/subsection/subsubsection number should be printed, and then the complete heading on a single line, even if it is longer that the current width. Headings at level *nn* reset the numbering for headings level $nn + 1$, $nn + 2$, and so on; and (to avoid non-computing people from being confused) all headings start their counting from one. At most five levels of headings may occur, counting from $nn = 1$ (the top-level heading) to $nn = 5$. All headings are always preceded and followed by a paragraph boundary, and level-1 headings are also preceded by a full-width line of "`-`" characters.

Again, see the FAQ page for linked examples of input and output combinations.

Beyond the Scope of the Project

Bored? Fidgetty? Lost interest in facebook? How about:

- `.i` – make an item in a bulleted list, just like this sentence is part of a bulleted list generated by LaTeX. Use a "`o`" as the "bullet" character, and indent the left margin by three characters and reduce the width by three spaces, until then next "`.p`" or "`.b`" command occurs, which ends the bulleted list. (But please dont submit any such programs for assessment, even if you get them working prior to the submission deadline.)
- Or, if you want to do something even more rewarding, find an on-line LaTeX tutorial, install one of the many free LaTeX packages, and teach yourself how to make professional-looking documents. (And you'll never have to fight with Word again...)

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 though 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 is 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.

The boring stuff...

This project is worth 15% of your final mark. A rubric explaining the marking expectations is provided on the FAQ page. You need to submit your program for assessment; detailed instructions on how to do that will be posted on the FAQ page once submissions are opened. Submission will *not* be done via the LMS; instead you will need to log in to a Unix server and submit your files to a software system known as `submit`. You can (and should) use `submit` **both early and often** – to get used to the way it works, and also to check that your program compiles correctly on our test system, which has some different characteristics to the lab machines. *Failure to follow this simple advice is highly likely to result in tears.* Only the last submission that you make before the deadline will be marked. Marks and a sample solution will be available on the LMS before Tuesday 8 October.

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** "lend" your "Uni backup" memory stick to others for any reason at all; 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**" when they ask for a copy of, or to see, your program, pointing out that your "**no**", and their acceptance of that decision, is the only thing that will preserve your friendship. *A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode. Students whose programs are so identified will either lose marks through the marking rubric, or will be referred to the Student Center for possible disciplinary action without further warning. This message **is** the warning.* See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to online forums or marketplaces, whether or not there is payment involved, is also Academic Misconduct. In the past students have had their enrollment terminated for such behavior.

The FAQ page contains wording for an Authorship Declaration that you **must include as a comment at the top of your submitted program. A significant fraction of the available marks will be deducted if you do not do so.**

Deadline: Programs not submitted by **10:00am on Monday 23 September** will incur penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email `ammoffat@unimelb.edu.au` 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 take a Health Professional Report (HPR) form with you (get it from the Special Consideration section of the Student Portal), you will need this form to be filled out if your illness develops in to something that later requires a Special Consideration application to be lodged. You should scan the HPR form and send it with any non-Special Consideration assignment extension requests.

And remember, algorithms are fun!