comp10002 Assignment 1, 2019s2

Last updated: September 20, 2019

Submission Instructions

Submissions have been opened. The submission process will be shown in class on 9 September. Be sure to watch that segment carefully when you are making your own submission, and be sure to carry out **all** the steps. Best bet is to make submissions all the way through the development process, both to document the progress that you are making, and as a routine to familiarize yourself with the mechanics of the process that is required.

Note that the submission process is set up to test "Stage 3" programs. Until you have reached that point, the `submit` process will report that your program is not generating the required output.

Information and Resources -- Read from the bottom up

1.
2. *Tears coming soon*: Looks like it is going to be a long weekend for some of you -- there are now 480 of you who have made one or more submissions (plus another ten or so helloworld programs), but around 300 more are still expected. Don't say I didn't warn you. Ontime submissions will close at 10am Monday morning. Once that happens you will need to use:
   ```
   submit comp10002 ass1.late myassignment.c
   ```
3. `verify comp10002 ass1.late > my-receipt-ass1.txt`
4.

   to make your submission. You'll then have until 10am Tuesday to make a submission that incurs a two-mark late penalty. Note that if you do make a late submission at all, any on-time submission you have made (via project `ass1`) will *not* be marked.
5. *Common submission problems*: I have had all of these arise within the last 24 hours via student emails telling me that "they can't submit":
   - Not running the VPN before trying to copy files to `dimefox`.
   - Not appending the "`:`" required at the end of the `scp` command.
   - Not then realizing that you have to `ssh` to `dimefox` to do the actual submission.
   - Not typing the user name in all lower-case letters.
   - Using the wrong user name.
   - Using the wrong password.

- Typing the wrong subject name or assignment name, it has to be `submit comp10002 ass1 myassignment.c`
-

  where `myassignment.c` is the name of the file containing your program (it can be any filename).
- Not actually reading any error messages that `submit` gives, and thinking it worked when it didn't (wrong subject name, or wrong assignment name, for example).
- Not doing a `verify` and hence not noting that their program didn't even compile on `dimefox`.

6. In other words, you need to *follow the instructions carefully* to have it all work correctly.
7. *Corrupted strings*: Note that the submission process includes a test for non-printable non-ASCII characters in the output that you generate. If you see things like `(0xad)(0xfe)` popping up in your `submit` output that you can look at via `verify`, it means that you are corrupting your string characters somehow (perhaps via a rogue pointer) and have a bug. To see what characters you are actually writing from your program, pipe the output into `od -a`.

8. *Only Your Last Submission Will Be Marked*: When it is getting marked, the markers will decide whether it includes code to handle all three stages, or only stages 1+2, or only Stage 1. If it is only Stage 1 code, then the "Stage 2 initial allocation" and "Stage 3 initial allocation" lines in the marking rubric will all get removed, thereby reducing the maximum mark that can be attained. And if the last program that is submitted prior to the deadline is only a Stage 1 program, then part of the assessment done by the markers will be in regard to whether or not the starting point was a good basis to have (potentially) gone on to do Stage 2 and Stage 3 by adding more functions. Because otherwise Stage 1 can be done with a ten line program (approx), and I'm not going to allow such programs to get 8/15 marks.

9. *Some Detailed Questions*: *In test2, should the space at the end of the "Heading Three" line really be there?*
   No, my program should have removed it. My bad. I won't run any tests that rely on you retaining or removing traling spaces in centered

and in heading lines.
In test3, should there be a paragraph break inserted before the first centered line, as a result of the initial setting of the margins?
No, paragraph breaks should only appear if the last thing that happened was a text output line. The test3 output files are correct.
If we get a sequence of .p .b .w .l commands, how many blank lines should be generated?
One. Any sequence of .b commands generates just a single break. And any sequence of .p or .l or .w commands, even if it include .b commands, generates just a single paragraph break.
How should we handle hyphenated words like good-looking?
Treat hyphens as you would any other non-blank character. (Or course, in a sophisticated system, we might also break lines at hyphens, or even introduce hyphens at sensible points within words, when it was necessary to do so. We are not writing a sophisticated program here.)

10. *First Warning*: As of Monday afternoon, 13 September, there are (only!) 336 submissions. That means that more than 450 students have yet to make their first submission. Every year I plead for students to **submit early and submit often**. Every year you ignore me.
Every year I say **there are going to be tears**, and **please please please I beg of you, practice the submission process**. Every year you ignore me.
And then, every year, in the end, on the day that the project is due, **there are mistakes made, and great dripping pools of tears**. And *I* ignore *you*.

11. *Example Program*: As an example of the standard of work that is expected (including the type and degree of commenting), here are:
    - The 2013 Assignment 1 specification
    - A sample solution to the 2013 Assignment 1.
12. Note that you are **not** required to use `structs` in your Assignment 1 submission, but may do so if you wish to. Note also that in 2013 students were given some code sections as a starting point, and only wrote a part of this program.

13. *Access Denied on dimefox*: Note that you won't have an account yet on dimefox if you have never logged in to a lab machine so far this semester, for example, if you have been using your own computer for all the workshops, or if you simply haven't attended any workshops. The symptoms of this will be an "access denied" message when you try and connect with scp/puttyscp or ssh/putty when you want to copy/submit your program. You will need to login to a lab machine, get you account initialized, and then wait a few hours for everything to percolate through the various processes involved with transferring those account details on to dimefox. Then you'll be in a position to ssh/scp and eventually submit. If you still have problems after you have taken this step, and are sure you are using the right password (and can log in to other University services using it), send me an email confirming that you have taken all of these steps and **still** can't get access to `dimefox`. *Don't leave this until the last minute. It is a problem that can't be fixed in a minute!*

14. *Marking Rubric*: The marking rubric is linked here. Lines that do not apply to your program will be removed during the marking process; your mark will then be the sum of the lines that remain, positives and negatives. Marks won't go below zero in each section, and won't go below zero overall either. Note carefully the deductions that will apply in the Academic Honesty Section. Note also that you are expected to copy the assignment skeleton ass1-skel.c and include the Authorship Declaration at the top of your program.

15. *Attribution for Re-Used Code*: It is ok to make use of code (for example, insertionsort, and/or getword(), and etc) from the book or from the lecture slides or from other published/public sources, but you should remember to add an attribution as a comment to each relevant function, saying where you got it from, what modifications you added to make it suit your purpose, and so on -- exactly as you would when quoting some other author when writing an essay. Of course, the expectation is that the assembly and "glueing together" of these bits to make a final program will all be your own work, and that the "quoted" bits will be a relatively small fraction of the "new" output you are being asked to generate. So it is **not** ok to take a whole solution from somewhere else, even if it appears on the web;

and it is **not** ok to solicit or commission a solution by posting the specification to a forum or web site and asking for "assistance" or "guidance" or "suggestions". Just as it wouldn't be ok to submit something you found online in response to an assignment that involved writing an essay.
And a reminder of what it says in the specification: we **will** be using similarity-checking software across all the submissions, and we **will** be referring cases of suspected academic misconduct for disciplinary hearings run by the School of Engineering, and in the past those hearings **have** resulted (including multiple times in **this** subject) in students being awarded penalties including final marks of **zero** for the subject, regardless of their other components of assessment.

16. *Debugging (more)*: If a C program encounters a run-time error and exits, there might still be pending output that has not been written. This is a particular problem in the `submit` environment, because it can look like the program is failing before it generates any output at all. If in doubt, add `fflush(stdout)` function calls after each of your debugging `printf()`'s (or even, add it to the macro), to force all pending output to be written immediately. You'll then be able to get a much clearer idea of how far the program is getting before it fails. And it is certainly a good idea to routinely put an `fflush(stdout)` call at the end of each of the program's main stages, to ensure that errors in one stage don't prevent the output from earlier stages being marked.

17. *Debugging*: Try putting this at the top of your program: `#define DEBUG 1`
18. `#if DEBUG`
19. `#define DUMP_DBL(x) printf("line %d: %s = %.5f\n", __LINE__, #x, x)`
20. `#else`
21. `#define DUMP_DBL(x)`
22. `#endif`
23.

    and then, later in your code, where you have a `double` variable (say) `score`, try `DUMP_DBL(score);`
24.

    Then change `DEBUG` to `0` at the top of the program, compile it again, and then run it again. Get it? You can then add `DUMP_INT` and `DUMP_STR`, and get the extra output turned on whenever you need it to understand what your program is doing. Then turn it all off again with one simple edit.

25. *Trouble with newline characters*: Text files that are created on a PC, or copied to a PC, edited and then saved again on the PC, may end up with PC-format two-character (`CR+LF`) newline sequence, see the Wiki page for details. If you have compiled your program on a PC, and it receives a `CR+LF` sequence, then `getchar()` will consume them both, and hand a single-character `'\n'` newline to your program. So in that sense, everything works as expected. Likewise, on a PC when you write a `'\n'` to `stdout`, a `CR+LF` pair will be placed in to the output file (or passed through the pipe to the next program in the chain).
The problems arise when you copy your program and a PC-format test file to a Unix system and then try compiling and executing your program there. Now the `CR` characters get in the way and arrive via `getchar()` into your program as stand-alone `'\r'` characters.
The easiest way to defend against these confusions is to write your program so that it looks at every character that it reads, and if it ever sees a `CR` come through, it throws it away. That way, if you do accidentally get `CR` characters in your test files on the Unix server (or on your Mac) your program won't be disrupted by them. Here is a function that you should use to do this:

```
    int
mygetchar() {
    int c;
    while ((c=getchar())=='\r') {
    }
    return c;
}

```

Then just call `mygetchar()` whenever you would ordinarily call `getchar()`, on both PC and Mac. Because most of you work on PCs (including in the labs), the test files that are provided have been created **with** the PC-style `CR+LF` newlines, and should work correctly when copied (use right-click->"Save as") to a PC. With `mygetchar()` they can also be used on a Mac, but won't interact sensibly using the

standard `getchar()` function.
To be consistent, the final post-submission testing will also be done using PC-style input files but will be executed **on a Unix machine**, meaning that **all** submitted programs will need to make use of `mygetchar()`.
You can use the "Preferences->Encodings" menu ("screwdriver/ hammer Options->Encodings" in the PC version) in jEdit to select whether to use Unix (`LF`) or DOS/Windows (`CR+LF`) encodings in any test files that you create with jEdit. Note that this only applies to newly created files. jEdit will by default respect the formatting in any current files.
Note also that jEdit doesn't automatically add a newline after the last line of text files, you need to put it there explicitly yourself (just press enter one more time, so that jEdit thinks there is an empty line at the end of the file). Watch out for this problem if you are creating your own test files on Mac or PC. All the test files that are supplied will have a newline at the end of the last line of the file, including the ones used during the post-submission re-testing.
If in any doubt, use `od -a <file>` in a Unix shell to look at the byte-by-byte contents of a file, and check which format is being used, and whether there is a final newline character (or final `CR+LF` pair). You can do this on a PC by starting the MinGW shell and then using `cd` to reach the right directory. There is an `od` version available within the MinGW shell on the PCs in the labs. On a Mac, `Terminal` is a Unix shell.

33. *Tabs*: The default in jEdit is for tabs to be aligned every 8 character positions. Some of you have altered that to four (Preferences->Editing->Tab width), to reflect the layout that the programs in the book have. Then, on submission, the tabs have "appeared" in the output as being 8 again, which can make your program spill past the 80-character RH boundary. When I run the programs for marking, they'll **all** get formatted with tabs reflecting 4 character positions, not 8. But don't use any fewer than 4 in your jEdit (or other editor) settings.
34. *Magic numbers*: Here is a summary of the rules about magic numbers:

- Where a number is totally self-defining, I'm happy for it to be used any number of times without a hash-define, provided the code is commented each time and/or explicitly sensible variable names are used. For example, in `/* compute percentage */`
- `pcent = 100.0*count/totcount;`
-

  I wouldn't expect 100 to have been hash-defined, since the comment explains the role of the 100, and it isn't going to change, ever, even if other percentages are calculated in the program using 100 too.
- This rule also allows 0 and 1, of course, unless they represent something other than the additive and multiplicative identities, in which case they should be hash-defined.
- This rule also allows `while (scanf("%lf%lf", &x, &y)==2)`, since the 2 is immediately obvious from the adjacent context (two variables to be read).
- Where a constant is one that is a fact that is in no way ever going to be varied, then provided it only appears once in the program and is explained with a comment, then it need not be hash-defined. The example here is the `-32.0` in the temperature conversion computation, assuming that it is entirely within a function called `Cels2Fahr` or etc and that it isn't used in other places scattered through the program. Anything of this type that appears even twice should be hash-defined.
- Where a factual constant is used more than once in a program, even if all occurrences are in a single function, it should be hash-defined.
- Where a constant is one that is clearly an artifact of the problem description or the program that implements the solution (for example, numbers like `MAXINT`, or the number of variables), then they must be hash-defined, even if only used once in the program.

35. Make sense?
36. *Skeleton Program*: You ***must*** start your program using the skeleton linked here. Then, before you make your final submission, add your name, student number, and the date, to "sign" the submission and certify it as being your own work. Submissions that do not include

the declaration, or in which it has not been completed, will suffer large mark penalties.

37. *Test data*: Test data for Assignment 1, and some sample outputs:
    - test0.txt and test0-out-dos.txt (PC version) and test0-out-mac.txt (Unix version), as a **Stage 1** output example
    - test1.txt and test1-out-dos.txt (PC version) and test1-out-mac.txt (Unix version), as a **Stage 2** output example
    - test2.txt and test2-out-dos.txt (PC version) and test2-out-mac.txt (Unix version), as a **Stage 3** output example
    - test3.txt and test3-out-dos.txt (PC version) and test3-out-mac.txt (Unix version), as a second **Stage 3** output example
38. See #5 above to understand why there are two different versions of the output files. You should copy the right set on to your computer so that you can use the Unix/MinGW command `diff` to compare your output with the required output. Your output is expected to exactly match what is shown in these examples.

39. *Specification*: The specification for the project was provided here on Friday 6 September.
40. (And if you are interested, the source LaTeX file is available here.)

Alistair Moffat, ammoffat@unimelb.edu.au