The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310/CSSE7231 — Semester 2, 2022
Assignment 1 (version 1)

Marks: 75 (for CSSE2310), 85 (for CSSE7231)
Weighting: 15%
Due: 6:00pm Friday 26 August, 2022

Introduction

The goal of this assignment is to give you practice at C programming. You will be building on this ability in the remainder of the course (and subsequent programming assignments will be more difficult than this one). You are to create a program (called `wordle-helper`) which helps users play the Wordle game (`www.nytimes.com/games/wordle/`). More details are provided below but you may wish to play the game to gain a practical understanding of how it operates. The assignment will also test your ability to code to a particular programming style guide, and to use a revision control system appropriately.

Student Conduct

This is an individual assignment. You should feel free to discuss **general** aspects of C programming and the assignment specification with fellow students, including on the discussion forum. In general, questions like "How should the program behave if ⟨this happens⟩?" would be safe, if they are seeking clarification on the specification.

You must not actively help (or seek help from) other students or other people with the actual design, structure and/or coding of your assignment solution. It is **cheating to look at another student's assignment code** and it is **cheating to allow your code to be seen or shared in printed or electronic form by others**. All submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct actions will be initiated against you, and those you cheated with. That's right, if you share your code with a friend, even inadvertently, then **both of you are in trouble**. Do not post your code to a public place such as the course discussion forum or a public code repository, and do not allow others to access your computer – you must keep your code secure.

You must follow the following code referencing rules for all code committed to your SVN repository (not just the version that you submit):

Code Origin	**Usage/Referencing**
Code provided to you in writing **this semester** by CSSE2310/7231 teaching staff (e.g. code hosted on Blackboard, posted on the discussion forum, or shown in class).	May be used freely without reference. (You must be able to point to the source if queried about it.)
Code you have personally written this semester for CSSE2310/7231 (e.g. code written for A1 reused in A3)	May be used freely without reference. (This assumes that no reference was required for the original use.)
Code examples found in man pages on `moss`.	May be used provided the source of the code is referenced in a comment adjacent to that code. (Code you have *taken inspiration from* must not be directly copied or just converted from one programming language to another.)
Code you have personally written in a previous enrolment in this course or in another ITEE course and where that code has not been shared or published.	
Code (in any programming language) that you have taken inspiration from but have not copied.	
Other code – includes: code provided by teaching staff only in a previous offering of this course (e.g. previous A1 solution); code from websites; code from textbooks; any code written by someone else, or partially written by someone else; and any code you have written that is available to other students.	May **not** be used. If the source of the code is referenced adjacent to the code then this will be considered code without academic merit (not misconduct) and will be removed from your assignment prior to marking (which may cause compilation to fail and zero marks to be awarded). Copied code without adjacent referencing will be considered misconduct and action will be taken.

Uploading or otherwise providing the assignment specification or part of it to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of these sites and many cooperate with us in misconduct investigations.

The course coordinator reserves the right to conduct interviews with students about their submissions, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process. If you are not able to adequately explain your code or the design of your solution and/or be able to make simple modifications to it as requested at the interview, then your assignment mark will be scaled down based on the level of understanding you are able to demonstrate.

In short - **Don't risk it!** If you're having trouble, seek help early from a member of the teaching staff. Don't be tempted to copy another student's code or to use an online cheating service. You should read and understand the statements on student misconduct in the course profile and on the school web-site: `https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`

Specification

Your program is to suggest a list of words that might be suitable for playing in a Wordle game, given some constraints. The length of the word may be set on the command line (between 4 and 9 characters inclusive) but defaults to 5 if no length is specified. The constraints on the word might include particular letters that have to be in particular locations in the word, particular letters that must be found somewhere in the word, or particular letters that must not be found in the word.

Full details of the required behaviour are provided below.

Command Line Arguments

Your program (`wordle-helper`) is to accept command line arguments as follows:

```
./wordle-helper [-alpha|-best] [-len word-length] [-with letters] [-without letters] [pattern]
```

In other words, your program should accept 0 to 8 arguments after the program name. The square brackets (`[]`) indicate optional arguments. The pipe symbol (`|`) indicates a choice. The *italics* indicate placeholders for user-supplied arguments.

Some examples of how the program might be run include the following[1]:

```
./wordle-helper
./wordle-helper -alpha
./wordle-helper -best
./wordle-helper -len 6
./wordle-helper -with abB
./wordle-helper -without xyZ
./wordle-helper R___E
./wordle-helper -alpha -len 6 -with os -without Xyz r_p__e
./wordle-helper r_P__e -alpha -without XyZ -len 6 -with OS
```

Arguments can be in any order, as shown in the last example, but handling such arguments will only be assessed as part of advanced functionality – see below.

The meaning of the arguments is as follows:

- `-alpha` – if specified, this argument indicates that the output words must be sorted in alphabetical order[2] and all duplicates must be removed. If this option is specified, then the `-best` option must not be specified.
- `-best` – if specified, this argument indicates that the output words must be sorted in decreasing order of likelihood (as determined by the provided `guess_compare()` function – described later). Words with the same likelihood must be sorted alphabetically. Duplicates must be removed. If this option is specified, then the `-alpha` option must not be specified.
- `-len` – if specified, this argument must be followed by an integer between 4 and 9 inclusive that indicates the length of the word to be used. If the argument is not specified, a default word length of 5 shall be used.
- `-with` – if specified, this argument is followed by a set of letters that must be present in matching words. Letters can be listed in any order and with any case (i.e. uppercase and/or lowercase). If a letter is

[1]This is not an exhaustive list and does not show all possible combinations of arguments.
[2]Alphabetical order means the order as determined by `strcasecmp()` – or `strcmp()` for words of the same case.

listed more than once then it must be present at least that number of times in matching words. The set of letters may be empty – which is the same as not specifying the `-with` argument. All matching is case-independent – e.g. an 'A' or 'a' in this set will match an 'A' or 'a' in the word.

- `-without` – if specified, this argument is followed by a set of letters that must not be present in matching words. Letters may be listed more than once but this has the same effect as listing them once. Letters may be listed in any order or with any case. The set of letters may be empty – which is the same as not specifying the `-without` argument. All matching is case-independent – e.g. an 'A' or 'a' in this set means that neither 'A' nor 'a' should be present in the word.
- ***`pattern`*** – if specified, this argument must be a string of the same length of words to be matched. The string must contain only letters (any case) and underscores. Words must match the pattern – i.e. must contain letters in the same position as letters that are present in the pattern, and any letter may match an underscore. A pattern of the right length consisting only of underscores is the same as not specifying a pattern (i.e. everything of the right length is a match). All matching is case independent.

Prior to doing anything else, your program must check the command line arguments for validity. If the program receives an invalid command line then it must print the message:

```
Usage: wordle-helper [-alpha|-best] [-len len] [-with letters] [-without letters] [pattern]
```

to standard error (with a following newline), and exit with an exit status of 1.

Invalid command lines include (but may not be limited to) any of the following:

- An argument begins with the character '`-`' but it is not one of `-alpha`, `-best`, `-len`, `-with`, or `-without`.
- The `-len` argument is given but it is not followed by a positive integer between 4 and 9 inclusive.
- The `-with` or `-without` argument is given but is not followed by an argument that contains only uppercase or lowercase letters (or is the empty string[3].)
- Both the `-alpha` and `-best` arguments are given
- Any argument is present more than once (e.g. `-len` is repeated, or there is more than one pattern argument)

Checking whether the *pattern* argument (if present) is valid is not part of the usage checking – that is checked after command line validity – see the next section. The pattern argument can be assumed to be the first argument that does not begin with '-' (and is not the argument immediately following `-len`, `-with` or `-without`).

Pattern Checking

If the command line arguments are valid, then the pattern argument (if supplied) must be checked for validity. If the pattern is invalid, then your program must print the message:

```
wordle-helper: pattern must be of length N and only contain underscores and/or letters
```

to standard error (with a following newline), and exit with an exit status of 2. The placeholder '*N*' in the message must be replaced by the expected word length.

Invalid patterns are:

- those whose length does not match the expected word length (the value given after `-len` or default 5), and/or
- those that contain characters other than underscores and/or letters (uppercase and/or lowercase).

Dictionary File Name Checking

By default, your program is to use the dictionary file `/usr/share/dict/words`. However, if the `WORDLE_DICTIONARY` environment variable is set then your program must use the dictionary file whose name is specified in that environment variable.

If the environment variable is set but the given dictionary filename does not exist or can not be opened for reading, your program should print the message:

[3]Note that an empty string is not the same as a missing argument. An empty string can be passed from the shell command line as `""`. Running `./wordle-helper -with ""` is valid; running `./wordle-help -with` is not.

```
wordle-helper: dictionary file "filename" cannot be opened
```

to standard error (with a following newline), and exit with an exit status of 3. (The italicised *filename* is replaced by the actual filename i.e. the value of the `WORDLE_DICTIONARY` environment variable. The double quotes must be present.) This check happens after the command line arguments and pattern (if supplied) are known to be valid.

The dictionary file is a text file where each line contains a "word". (Lines are terminated by a single newline character. The last line in the file may or may not have a terminating newline.) You may assume there are no blank lines and that no words are longer than 50 characters (excluding any newline)[4], although there may be "words" that contain characters other than letters, e.g. "1st" or "don't". The dictionary may contain duplicate words and may be sorted in any order. The dictionary may contain any number of words (including zero).

Program Operation

If the checks above are successful, then your program should output all words in the given dictionary file that satisfy the following:

- the word length matches that required (see the description of the `-len` command line argument above);
- the word contains only letters – which can be uppercase or lowercase or a combination of both;
- the word must contain those letters specified after the `-with` argument (if specified, see description above);
- the word must not contain those letters specified after the `-without` argument (if specified, see description above); and
- the word must match the given pattern from the command line (if specified, see description above).

Program Output

Your program must output (to standard output) all matching words in uppercase (i.e. any lowercase letters in matching words must be printed as uppercase letters).

By default, if neither the `-alpha` nor `-best` arguments are present, then words must be output in the order they are found in the dictionary file. This may include duplicate words if the word is found more than once in the dictionary file.

If either the `-alpha` or `-best` argument is present, then the output must be sorted as described in the command line arguments section above.

If at least one matching word is found and output then your program must exit with exit status 0, indicating a successful run. If no matching words are found then your program must exit with exit status 4 – but nothing should be printed to standard output or standard error.

Your program must output nothing to standard output or standard error other than the output/messages described above. Your program will never print a message to both standard output and standard error during the same run. (You may print debugging information to standard output or standard error when developing your program, but make sure you delete all such print statements prior to submission. Extraneous output text will cause you to lose marks.)

Advanced Functionality

Some of the behaviour described above is more advanced and it is recommended that you delay implementation of this functionality until you have other functionality implemented:

- Implementation of sorting (using the `-alpha` or `-best` arguments) will require the use of dynamically allocated memory to store all matching words so they can be sorted before being output. Without this sorting functionality, you can just process words from the dictionary line by line and output them immediately if they match (or not, if they don't). If you do not implement the sorting functionality, you should still check for and accept the `-alpha` and `-best` command line arguments (in order to get marks for command line argument handling) – but just output the words in dictionary order.

[4]A statement that you may assume these things means that your program does not have to handle situations where these assumptions are not true – i.e. your program can behave in any way it likes (including crashing) if there are blank lines present in the dictionary and/or any words in the dictionary are longer than 50 characters.

- Allowing command line arguments to appear in any order complicates command line argument processing so this is considered more advanced functionality. For most functionality marks, testing will only take place with command line arguments (if present) in the order given in the usage strings above. For the advanced command line argument processing marks, your program must accept command line arguments in any order (parameters associated with option arguments, e.g. `-len 6` will still be together with that option argument).

Example Runs

Consider a dictionary with the following content:

```
raise
rogue
ridge
range
Range
rafted
RANGER
reaper
rented
redeem
```

Assume that this content is in the file `$HOME/example-dictionary`. The user can use this dictionary by setting the `WORDLE_DICTIONARY` environment variable as shown below. Note also that in the examples below, the `$` character is the shell prompt – it is not entered by the user.

Example 1: Matching all 5 letter words

```
$ WORDLE_DICTIONARY=$HOME/example-dictionary ./wordle-helper
RAISE
ROGUE
RIDGE
RANGE
RANGE
```

Example 2: Specifying that a letter must be present or not

```
$ WORDLE_DICTIONARY=$HOME/example-dictionary ./wordle-helper -with g -without d
ROGUE
RANGE
RANGE
```

Example 3: Sorting the output – note the removal of the duplicate word

```
$ WORDLE_DICTIONARY=$HOME/example-dictionary ./wordle-helper -alpha -with g -without d
RANGE
ROGUE
```

Example 4: Specifying a pattern – note the case insensitive pattern matching

```
$ WORDLE_DICTIONARY=$HOME/example-dictionary ./wordle-helper r__Se
RAISE
```

Example 5: Specifying a different length, and using repeated letters in the `-with` argument

```
$ WORDLE_DICTIONARY=$HOME/example-dictionary ./wordle-helper -len 6 -with eE
REAPER
RENTED
REDEEM
```

Example 6: Conflicting constraints – no output, but not an error

```
$ WORDLE_DICTIONARY=$HOME/example-dictionary ./wordle-helper -len 6 -with eE -without e
```

Provided Library: libcsse2310a1

A library has been provided to you with the following function which your program must use if you implement the `-best` sorting functionality:

```
        int guess_compare(const char* word1, const char* word2);
```

The function is described in the `guess_compare(3)` man page on moss. (Run `man guess_compare`.)

To use the library, you will need to add `#include <csse2310a1.h>` to your code and use the compiler flag `-I/local/courses/csse2310/include` when compiling your code so that the compiler can find the include file. You will also need to link with the library containing this function. To do this, use the compiler arguments `-L/local/courses/csse2310/lib -lcsse2310a1`.

Style

Your program must follow version 2.2.0 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site.

Hints

1. The string representation of a single digit positive integer has a string length of 1.
2. You can use the function `getenv()` to retrieve the value of an environment variable.
3. You **may** wish to consider the use of the standard library functions `isalpha()`, `islower()`, `isupper()`, `toupper()` and/or `tolower()`. Note that these functions operate on individual characters, represented as integers (ASCII values).
4. There is no expectation that you write your own sorting routine. It is recommended that you use the `qsort()` library function.
5. Some other functions which **may** be useful include: `strcasecmp()`, `strcmp()`, `strlen()`, `exit()`, `fopen()`, `fprintf()`, and `fgets()`. You should consult the man pages for these functions.
6. The style guide shows how you can break long string constants in C so as not to violate line length requirements in the style guide.
7. Removal of duplicate items from a list is best done after sorting a list – duplicate items will be adjacent to each other.
8. It is strongly recommended that you do not try to use `getopt()` to parse command line arguments as the form of arguments specified here is not amenable to parsing by `getopt()`, e.g. short forms of arguments are not to be supported, duplicate arguments are not permitted, arguments beginning with `--` are not permitted, etc.

Suggested Approach

It is suggested that you write your program using the following steps. Test your program at each stage and commit to your SVN repository frequently. Note that the specification text above is the definitive description of the expected program behaviour. The list below does not cover all required functionality.

1. Write a program that outputs the usage error message and exits with exit status 1. This small program (just a couple of lines in `main()`) will earn marks for detecting usage errors (category 1 below) – because it thinks everything is a usage error![5]

[5]However, once you start adding more functionality, it is possible you may lose some marks in this category if your program can't detect all the valid command lines.

2. Detect the presence of the `-len` command line argument and, if present, check the presence/validity of the argument that follows it. Exit appropriately if not valid.
3. Detect the presence of the `-with` command line argument and, if present, check the presence/validity of the argument that follows it. Exit appropriately if not valid.
4. Detect the presence of the `-without` command line argument and, if present, check the presence/validity of the argument that follows it. Exit appropriately if not valid.
5. Detect the presence of a sort method argument (`-alpha` and `-best`) and make sure only one is present. Exit appropriately if not valid.
6. Detect the presence of a pattern argument and check whether it is valid or not. Exit appropriately if not valid.
7. Check whether the dictionary can be opened for reading or not (both the default dictionary and one specified using the environment variable). Exit appropriately if not. (If it can be opened, leave it open – you'll need to read from it next.)
8. Read the dictionary line by line, convert it to uppercase and just print out each word that is the right length (i.e. initially assume all words of the right length match).
9. Check the words to make sure they only contain letters.
10. If a pattern is given, check each word matches the pattern before outputting it.
11. If the `-with` argument is given, check each word contains only those letters. It is easiest to do this by counting the number of each letter in the word and comparing that with the number of each letter in the `-with` set.
12. If the `-without` argument is given, check each word does not contain those letters (you can use the count of each letter that you've already determined).
13. Implement remaining functionality as required . . .

Forbidden Functions

You must not use any of the following C functions/statements. If you do so, you will get zero (0) marks for the assignment.

- `goto`
- `longjmp()` and equivalent functions
- `system()`
- `popen()`
- `execl()` or any other members of the exec family of functions
- POSIX regex functions

Submission

Your submission must include all source and any other required files (in particular you must submit a `Makefile`). Do not submit compiled files (eg `.o`, compiled programs) or dictionary files.

Your program (named `wordle-helper`) must build on `moss.labs.eait.uq.edu.au` with:
`make`

Your program must be compiled with `gcc` with at least the following options:
`-pedantic -Wall -std=gnu99`

You are not permitted to disable warnings or use pragmas to hide them. You may not use source files other than `.c` and `.h` files as part of the build process – such files will be removed before building your program.

If any errors result from the `make` command (i.e. the `wordle-helper` executable can not be created) then you will receive 0 marks for functionality (see below). Any code without academic merit will be removed from your program before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality).

Your program must not invoke other programs or use non-standard headers/libraries.

Your assignment submission must be committed to your subversion repository under

```
https://source.eait.uq.edu.au/svn/csse2310-sem2-sXXXXXXX/trunk/a1
```

where sXXXXXXX is your moss/UQ login ID. Only files at this top level will be marked so **do not put source files in subdirectories**. You may create subdirectories for other purposes (e.g. your own test files) but these will not be considered in marking – they will not be checked out of your repository.

You must ensure that all files needed to compile and use your assignment (including a Makefile) are committed and within the `trunk/a1` directory in your repository (and not within a subdirectory) and not just sitting in your working directory. Do not commit compiled files or binaries. You are strongly encouraged to check out a clean copy for testing purposes.

To submit your assignment, you must run the command

```
2310createzip a1
```

on `moss` and then submit the resulting zip file on Blackboard (a GradeScope submission link will be made available in the Assessment area on the CSSE2310/7231 Blackboard site)[6]. The zip file will be named

`sXXXXXXX_csse2310_a1_`*`timestamp`*`.zip`

where `sXXXXXXX` is replaced by your moss/UQ login ID and *timestamp* is replaced by a timestamp indicating the time that the zip file was created.

The `2310createzip` tool will check out the latest version of your assignment from the Subversion repository, ensure it builds with the command '`make`', and if so, will create a zip file that contains those files and your Subversion commit history and a checksum of the zip file contents. You may be asked for your password as part of this process in order to check out your submission from your repository.

You must not create the zip file using some other mechanism and you must not modify the zip file prior to submission. If you do so, you will receive zero marks. Your submission time will be the time that the file is submitted via GradeScope on Blackboard, and **not** the time of your last repository commit nor the time of creation of your submission zip file.

We will mark your last submission, even if that is after the deadline and you made submissions before the deadline. Any submissions after the deadline[7] will incur a late penalty – see the CSSE2310/7231 course profile for details.

Marks

Marks will be awarded for functionality and style and documentation. Marks may be reduced if you are asked to attend an interview about your assignment and you are unable to adequately respond to questions – see the **Student conduct** section above.

Functionality (60 marks)

Provided your code compiles (see above) and does not use any prohibited statements/functions (see above), and your zip file has been generated correctly and has not been modified prior to submission, then you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Partial marks will be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. **If your program does not allow a feature to be tested then you will receive 0 marks for that feature**, even if you claim to have implemented it. For example, if your program can never open a file, we can not determine if your program can determine word validity correctly. If your program takes longer than 10 seconds to run any test[8], then it will be terminated and you will earn no marks for the functionality associated with that test. The markers will make no alterations to your code (other than to remove code without academic merit).

[6]You may need to use scp or a graphical equivalent such as WinSCP, Filezilla or Cyberduck in order to download the zip file to your local computer and then upload it to the submission site.

[7]or your extended deadline if you are granted an extension.

[8]Valgrind tests for memory leaks (category 11) will be allowed to run for longer.

Marks will be assigned in the following categories.

1. Program correctly handles invalid command lines (8 marks)
2. Program correctly handles invalid patterns (with various `-len` options) (6 marks)
3. Program correctly handles dictionary files that are unable to be read (3 marks)
4. Program correctly outputs words of the right length from various dictionaries with either no arguments or just a `-len` argument and valid length (4 marks)
5. Program correctly matches specified patterns (with various `-len` options and dictionaries but no other options) (6 marks)
6. Program correctly implements `-with` argument (with various `-len` options and dictionaries but no other options) (6 marks)
7. Program correctly implements `-without` argument (with various `-len` options and dictionaries but no other options) (6 marks)
8. Program correctly matches words with various combinations of arguments (in usage message order if present) and various dictionaries (6 marks)
9. Program correctly matches and sorts words with `-alpha` or `-best` and various dictionaries and combinations of other arguments (in usage message order if present) (6 marks)
10. Program behaves correctly with various arguments in non-standard order (with various dictionaries but no sorting arguments) (6 marks)
11. Program behaves correctly and frees all memory when used with a sorting argument (`-alpha` or `-best`) and various dictionaries and various other arguments in non-standard order (3 marks)

It is unlikely that you can earn many marks for categories 8 and later unless your program correctly matches patterns, including support for the `-with` and `-without` arguments.

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 2.2 of the CSSE2310/CSSE7231 C Programming Style Guide (found on Blackboard). Style marks will be made up of two components – automated style marks and human style marks. These are detailed below.

You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final – it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` tool. Your style marks can never be more than your functionality mark – this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

You are encouraged to use the `style.sh` tool installed on `moss` to style check your code before submission. This does not check all style requirements, but it will determine your automated style mark (see below). Other elements of the style guide are checked by humans.

All `.c` and `.h` files in your submission will be subject to style marking. This applies whether they are compiled/linked into your executable or not[9].

Automated Style Marking (5 marks)

Automated style marks will be calculated over all of your `.c` and `.h` files as follows. If any of your submitted `.c` and/or `.h` files are unable to be compiled by themselves then your automated style mark will be zero (0). (Automated style marking can only be undertaken on code that compiles. The provided `style.sh` script checks this for you.)

If your code does compile then your automated style mark will be determined as follows: Let

- W be the total number of distinct compilation warnings recorded when your `.c` files are individually built (using the correct compiler arguments)

[9]Make sure you remove any unneeded files from your repository, or they will be subject to style marking.

- A be the total number of style violations detected by `style.sh` when it is run over each of your `.c` and `.h` files individually[10].

Your automated style mark S will be

$$S = 5 - (W + A)$$

If $W + A \geq 5$ then S will be zero (0) – no negative marks will be awarded. Note that in some cases `style.sh` may erroneously report style violations when correct style has been followed. If you believe that you have been penalised incorrectly then please bring this to the attention of the course coordinator and your mark can be updated if this is the case. Note also that when `style.sh` is run for marking purposes it may detect style errors not picked up when you run `style.sh` on moss. This will not be considered a marking error – it is your responsibility to ensure that all of your code follows the style guide, even if styling errors are not detected in some runs of `style.sh`.

Human Style Marking (5 marks)

The human style mark (out of 5 marks) will be based on the criteria/standards below for "comments", "naming" and "other". The meanings of words like *appropriate* and *required* are determined by the requirements in the style guide. Note that functions longer than 50 lines will be penalised in the automated style marking. Functions that are also longer than 100 lines will be further penalised here.

Comments (2.5 marks)

Mark	Description
0	The majority (50%+) of comments present are inappropriate OR there are many required comments missing
0.5	The majority of comments present are appropriate AND the majority of required comments are present
1.0	The vast majority (80%+) of comments present are appropriate AND there are at most a few missing comments
1.5	All or almost all comments present are appropriate AND there are at most a few missing comments
2.0	Almost all comments present are appropriate AND there are no missing comments
2.5	All comments present are appropriate AND there are no missing comments

Naming (1 mark)

Mark	Description
0	At least a few names used are inappropriate
0.5	Almost all names used are appropriate
1.0	All names used are appropriate

Other (1.5 marks)

Mark	Description
0	One or more functions is longer than 100 lines of code OR there is more than one global/static variable present inappropriately OR there is a global struct variable present inappropriately OR there are more than a few instances of poor modularity (e.g. repeated code)
0.5	All functions are 100 lines or shorter AND there is at most one inappropriate non-struct global/static variable AND there are at most a few instances of poor modularity
1.0	All functions are 100 lines or shorter AND there are no instances of inappropriate global/static variables AND there is no or very limited use of magic numbers AND there is at most one instance or poor modularity
1.5	All functions are 100 lines or shorter AND there are no instances of inappropriate global/static variables AND there is no use of magic numbers AND there are no instances of poor modularity

SVN commit history assessment (5 marks)

Markers will review your SVN commit history for your assignment up to your submission time. This element will be graded according to the following principles:

[10]Every `.h` file in your submission must make sense without reference to any other files, e.g., it must `#include` any `.h` files that contain declarations or definitions used in that `.h` file.

- Appropriate use and frequency of commits (e.g. a single monolithic commit of your entire assignment will yield a score of zero for this section)
- Appropriate use of log messages to capture the changes represented by each commit. (Meaningful messages explain briefly what has changed in the commit (e.g. in terms of functionality) and/or why the change has been made and will be usually be more detailed for significant changes.)

The standards expected are outlined in the following rubric:

Mark (out of 5)	**Description**
0	Minimal commit history – single commit OR all commit messages are meaningless.
1	Some progressive development evident (more than one commit) OR at least one commit message is meaningful.
2	Some progressive development evident (more than one commit) AND at least one commit message is meaningful.
3	Progressive development is evident (multiple commits) AND at least half the commit messages are meaningful.
4	Multiple commits that show progressive development of all functionality AND meaningful messages for most commits.
5	Multiple commits that show progressive development of all functionality AND meaningful messages for ALL commits.

We understand that you are just getting to know Subversion, and you won't be penalised for a few "test commit" type messages. However, the markers must get a sense from your commit logs that you are practising and developing sound software engineering practices by documenting your changes as you go. In general, tiny changes deserve small comments – larger changes deserve more detailed commentary.

Design Documentation (10 marks) – for CSSE7231 students only

CSSE7231 students must submit a PDF document containing a written overview of the architecture and design of your program. This must be submitted via the Turnitin submission link on Blackboard.

Please refer to the grading criteria available on BlackBoard under "Assessment" for a detailed breakdown of how these submissions will be marked. Note that your submission time for the whole assignment will be considered to be the later of your submission times for your zip file and your PDF design document. Any late penalty will be based on this submission time and apply to your whole assignment mark.
This document should describe, at a general level, the functional decomposition of the program, the key design decisions you made and why you made them. It must meet the following formatting requirements:

- Maximum two A4 pages in 12 point font
- Diagrams are permitted up to 25% of the page area. The diagram(s) must be discussed in the text, it is not ok to just include a figure without explanatory discussion.

Don't overthink this! The purpose is to demonstrate that you can communicate important design decisions, and write in a meaningful way about your code. To be clear, this document is not a restatement of the program specification – it is a discussion of your design and your code.

If your documentation obviously does not match your code, you will get zero for this component, and will be asked to explain why.

Total Mark

Let

- F be the functionality mark for your assignment (out of 60).
- S be the automated style mark for your assignment (out of 5).
- H be the human style mark for your assignment (out of 5)
- C be the SVN commit history mark (out of 5)
- D be the documentation mark for your assignment (out of 10 for CSSE7231 students) – or 0 for CSSE2310 students.

Your total mark for the assignment will be:

$$M = F + \min\{F, S + H\} + \min\{F, C\} + \min\{F, D\}$$

out of 75 (for CSSE2310 students) or 85 (for CSSE7231 students).

In other words, you can't get more marks for style or SVN commit history or documentation than you do for functionality. Pretty code that doesn't work will not be rewarded!

Late Penalties

Late penalties will apply as outlined in the course profile.

Specification Updates

Any errors or omissions discovered in the assignment specification will be added here, and new versions released with adequate time for students to respond prior to due date. Potential specification errors or omissions can be discussed on the discussion forum or emailed to `csse2310@uq.edu.au`.