The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310/CSSE7231 — Semester 1, 2022
Assignment 1 (version 1.1 - 14 March 2022)

Marks: 65 (for CSSE2310), 75 (for CSSE7231)
Weighting: 15%
Due: 4:00pm Friday 1 April, 2022

Specification changes since version 1.0 are shown in red and are summarised at the end of the document.

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`) which plays the Wordle game in a terminal window. See `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.

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 they cooperate with us in misconduct investigations.

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 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; 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). Copied code without adjacent referencing will be considered misconduct and action will be taken.

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 play the Wordle game via terminal interaction. Your program will call the supplied library function `get_random_word()` (outlined on page 6) to determine a random word of the desired length that is unknown to the user (the *answer*). Your program will then repeatedly prompt the user to guess the answer until they get it right or they run out of attempts. For incorrect guesses, if the guess is valid (i.e. is the right length and appears in the nominated dictionary) then your program will then report on which letters matched those in the answer (if any) – i.e. which letters were in the right place, which were in the wrong place, and which weren't in the answer at all. Invalid guesses don't count as an attempt – the user will be prompted to enter another guess.

Full details of the required behaviour are provided below.

Command Line Arguments

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

`./wordle [-len` *`word-length`*`] [-max` *`max-guesses`*`] [`*`dictionary`*`]`

In other words, your program should accept 0 to 5 arguments after the program name: zero, one or two option arguments (`-len` and/or `-max`, in either order if both present) each followed by an integer (in the range of 3 to 9 inclusive), with the option argument(s) optionally followed by a dictionary file name [1].

Some examples of how the program might be run include the following[2]:

```
./wordle
./wordle -len 6
./wordle -max 7
./wordle ./my-dictionary
./wordle -len 6 /usr/share/dict/words
./wordle -max 7 -len 6
```

The meaning of the arguments is as follows:

- `-len` – if specified, this option argument should be followed by an integer between 3 and 9 inclusive that indicates the length of the word to be used in game play. If the argument is not specified, a default word length of 5 shall be used.
- `-max` – if specified, this option argument should be followed by an integer between 3 and 9 inclusive that indicates the maximum number of guesses that will be permitted in game play. If the argument is not specified, a default value of 6 shall be used.
- *`dictionary`* – the last argument (if present) is the path name of the dictionary file, i.e. the name of a file containing words – one per line. Each line (including the last) is assumed to be terminated by a newline character (`\n`) only. You may assume there are no blank lines and that no words are longer than 50 characters, although there may be words that contain characters other than letters, e.g. "1st" or "don't" The filename can be relative to the current directory or absolute (i.e. begin with a /). Some examples are:

  `/usr/share/dict/words`
  `../dictname.txt`
  `5` (refers to a file with this name in the current directory).

  If a filename is not specified, the default should be used (`/usr/share/dict/words`).

[1] The square brackets (`[]`) indicate optional arguments. The *italics* indicate placeholders for user-supplied arguments.
[2] This is not an exhaustive list and does not show all possible combinations of arguments.

Prior to playing the game (or even checking the dictionary file exists) 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 [-len word-length] [-max max-guesses] [dictionary]
```

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 `-len` nor `-max`.
- The `-len` argument is given but it is not followed by a positive integer between 3 and 9 inclusive.
- The `-max` argument is given but it is not followed by a positive integer between 3 and 9 inclusive.
- Either the `-len` or `-max` argument is given twice.
- Any argument is the empty string.

Note that it is valid for a dictionary name to begin with '`-`' but it is not valid for the dictionary name argument to begin wtih '`-`'. The dictionary name argument can be prefixed with `./` if necessary, e.g., a dictionary file named `-len` can be passed as an argument by writing `./-len`.

Dictionary File Name Checking

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

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

to standard error (with a following newline), and exit with an exit status of 2. (The italicised *filename* is replaced by the actual filename i.e. the full argument given on the command line. The double quotes must be present.) This check happens after the command line arguments are known to be valid.

Game Play

If the checks above are successful, then your program should print the following to standard output (with a following newline):

```
Welcome to Wordle!
```

Your program should then determine a random word (the "answer") by calling `get_random_word()` – see details of this provided library function on page 6.

Your program should then repeatedly prompt the user for a guess by printing the following message to standard output (with a following newline):

```
Enter a N letter word (M attempts remaining):
```

where `N` is replaced by the word length for this game (e.g. 5) and `M` is replaced by the number of attempts remaining (e.g. initially 6 by default, or whatever value is specified with `-max`). When there is only one attempt remaining, the program should instead print:

```
Enter a N letter word (last attempt):
```

Guesses are entered on stdin and are terminated by a newline (or pending EOF). If the user enters a guess that is not the correct length then your program should print the following to standard output (with a following newline), followed by another prompt:

```
Words must be N letters long - try again.
```

`N` is replaced by the expected word length. (Note that the attempt number will not be incremented when the prompt is printed – only valid attempts are to be counted.)

If the user enters a guess of the correct length that contains characters other than the letters A to Z (upper case or lowercase), e.g. the guess contains numbers or spaces or punctuation characters, then your program should print the following to standard output (with a following newline), followed by another prompt:

```
Words must contain only letters - try again.
```

If the user enters a guess that is the right length and contains only letters and matches the answer then your program should print the following to standard output (with a following newline) and then exit with exit status 0 (success):

```
Correct!
```

Matching must be done on a case insensitive basis, e.g. the guess `HelLo` will match the word `hello`.

If the user enters a guess that is the correct length and contains only letters but it is not the answer and the guess cannot be found in the dictionary file then your program should print the following to standard output (with a following newline), followed by another prompt:

```
Word not found in the dictionary - try again.
```

Matching against words in the dictionary file must also be done on a case insensitive basis.

A guess that is the correct length, contains only letters, is not the answer, and is found in the dictionary is considered a valid attempt. When a user enters a guess that is considered a valid attempt, then your program will echo that word back to the user with the characters modified in the following way:

- If the letter in that position isn't found anywhere in the answer, then the '`-`' (hyphen) character is printed.
- If the letter in that position is found in that position in the answer, then the letter is printed in uppercase. (The is the same as a green letter in Wordle.)
- If the letter in that position is not found in that position in the answer but is found elsewhere in the answer, then the letter is printed in lowercase. (This is the same as a yellow letter in Wordle.)

If the user runs out of attempts (i.e. the number of valid guesses they've made reaches the maximum but they haven't guessed the answer) OR end-of-file (EOF) is detected on standard input (e.g. the user presses Ctrl-D when prompted for a guess), then your program should print the following to standard error (with a following newline) and exit with exit status 3:

```
Bad luck - the word is "WORD".
```

where *WORD* is replaced by the answer. The double quotes must be present.

If the last unsuccessful attempt is a valid guess then your program should report the letters that match in the normal way before printing the message above and exiting. If the program detects EOF when expecting a guess then no matching information is reported – the program just prints the message above and exits.

Other Functionality Notes

It is permissible for the answer to not appear in the dictionary of valid words – whether the guessed word is the answer or not must be checked before checking whether the guess is in the dictionary or not.

It is possible that the dictionary does not contain any words of the correct length (and indeed may not contain any words at all). In this case all guesses will be considered invalid unless the user happens to guess the answer.

If EOF (end of file) is detected on standard input when attempting to read a new guess, then this is to be interpreted as the user giving up and your program should print the "`Bad luck ...`" message.

If a library call fails unexpectedly (e.g. `malloc()` returns `NULL`) then your program may behave in any way you like. This will not be tested.

Your program may assume that guesses entered by the user are at most 50 characters (excluding the newline) – the same assumption that can be made about dictionary entries. We will not test your program using guesses longer than 50 characters.

Advanced Functionality 1

The matching behaviour described above applies when letters appear only once in the guess. If one or more letters appears more than once in a guess then the reported match is slightly more complicated:

- If a letter appears more than once in the guess but only once in the answer, then
  - If one of those repeated letters in the guess is in the correct position, then it will be reported in uppercase in that position (right letter, right location) and the other of those letters in the guess will be reported as not matching any letters (i.e. shown with a hyphen '`-`'). This makes it clear to the user that the letter only appears once in the answer and that they have found that position.
  - If neither of the letters in the guess is in the correct position, then the first of those letters in the guess will be reported in lower case (right letter, wrong location) and the second will be reported as not matching any letters in the answer (i.e. shown with a hyphen '`-`'). This makes it clear to the user that the letter only appears once in the answer.
- Similar principles apply where there are three repeated letters in the guess and one or two of those letters in the answer. The number of matches reported can never exceed the number of those letters in the answer – and if a letter is in the correct location then it must be shown in uppercase (i.e. right letters in the right location are considered a higher priority match then right letters in the wrong location).

Advanced Functionality 2

The program functionality described above can be implemented without using `malloc()`. If you read the dictionary file each time that you need to check the validity of a guess, then you don't need to store the dictionary in memory. Advanced Functionality 2 requires you to store the dictionary (which may have an arbitrary number of words) in memory for checking the validity of guesses. (You don't have to store the whole dictionary – you can just store valid words of the correct length – this is up to you.) If you implement this functionality then your program may only read the dictionary file once.

You must also ensure that your program frees all allocated memory before exiting[3]. This will be checked using `valgrind`.

Example Game Sessions

In the examples below, guesses entered on standard input are shown in **bold green** for clarity. Other lines are output to standard output except for the "`Bad luck ...`" message at the end of example 2 which is sent to standard error. Note that the `$` character is the shell prompt – it is not entered by the user.

Example 1: The word to be guessed is "tense". Five letter words are the default with a maximum of six guesses.

```
$ ./wordle
Welcome to Wordle!
Enter a 5 letter word (6 attempts remaining):
Hi
Words must be 5 letters long - try again.
Enter a 5 letter word (6 attempts remaining):
wrdle
Word not found in the dictionary - try again.
Enter a 5 letter word (6 attempts remaining):
vOwEl
---e-
Enter a 5 letter word (5 attempts remaining):
erase
e--SE
Enter a 5 letter word (4 attempts remaining):
these
T-eSE
Enter a 5 letter word (3 attempts remaining):
tense
Correct!
$
```

Example 2: A 4 letter word with a maximum of 4 guesses – the word to be guessed is "byte".

```
$ ./wordle -len 4 -max 4
Welcome to Wordle!
Enter a 4 letter word (4 attempts remaining):
2310
Words must contain only letters - try again.
Enter a 4 letter word (4 attempts remaining):
ease
---E
Enter a 4 letter word (3 attempts remaining):
TIME
t--E
Enter a 4 letter word (2 attempts remaining):
Vote
--TE
Enter a 4 letter word (last attempt):
```

[3]This includes the memory allocated by `get_random_word()`.

```
cute
--TE
Bad luck - the word is "byte".
$
```

Provided Library: libcsse2310a1

A library has been provided to you with the following function which your program must use:

```
char* get_random_word(int wordLength);
```

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

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`.

For testing purposes, if you set the environment variable `WORD2310` prior to running `./wordle` then `get_random_word()` will return the value in that variable (provided it is the correct length). This allows you to test your program like this:

Example 3: Running `wordle` with a known word ("break") for testing purposes.

```
$ WORD2310=break ./wordle
Welcome to Wordle!
...
```

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 **may** wish to consider the use of the standard library functions `isalpha()`, `islower()`, `isupper()`, `toupper()` and/or `tolower()`. Note that these functions operate on integers (ASCII values). ~~so you will need to cast characters to integers and vice versa as appropriate to avoid compiler warnings. For example, given that variable c is of type `char`, the following code will convert the character stored in c to lower case (without compiler warnings):~~
   ~~`c = (char)tolower((int)c);`~~
3. If a user enters a line of more than the requested number of characters to `fgets()` then the last non-null character in the returned string will not be a newline.
4. Some other functions which **may** be useful include: `strcasecmp()`, `strcmp()`, `strlen()`, `exit()`, `fopen()`, `fgets()`, and `fprintf()`. You should consult the man pages for these functions.
5. Note that implementation of the advanced functionality 2 will require the use of dynamically allocated memory (i.e. with `malloc()` etc.) to store the dictionary in memory. You can implement the majority of the functionality without dynamically allocated memory by opening and reading the dictionary each time you need to check whether a word is valid or not.

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 (two lines in `main()`) will earn marks for detecting usage errors (category 1 below) – because it thinks everything is a usage error![4]

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 `-max` 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 a dictionary file name argument and check whether it can be opened for reading. Exit appropriately if not valid.

5. Determine the answer word using the supplied library. Print the welcome message and print the prompt for input.

6. Repeatedly prompt and read a word (guess) from the user and check the word is the expected length and only contains letters. Print appropriate message if not. Exit appropriately if EOF detected or if number of attempts is exceeded.

7. Check if the guess matches the answer – if so, exit appropriately, otherwise print a message that the word must be found in the dictionary. (Even though you haven't checked the dictionary, pretending that you have will earn marks.)

8. Write a function to read all the words in the dictionary and check whether the guess is in the dictionary. If not, print an appropriate message and prompt again. (Initially, you can read the dictionary for every guess.)

9. If the guess is in the dictionary, report on the match – i.e. which letters in the guess can be found in the answer, etc.

10. Extend matching to deal with case insensitive matching

11. Add code to store the dictionary (or words of the right length from the dictionary) in memory, and scan this copy of the dictionary when checking guesses.

12. Extend the match reporting to deal with answers that have repeated letters.

13. Implement any 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()`
- `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`) must build on `moss.labs.eait.uq.edu.au` with:
`make`

[4] 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.

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` 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-sem1-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)[5]. 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[6] 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

[5] 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.

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

seconds to run any test[7], 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).
Marks will be assigned in the following categories.

1. Program correctly handles invalid command lines (8 marks)
2. Program correctly handles dictionary files that are unable to be read (4 marks)
3. Program initially prompts for input correctly on valid command lines (4 marks)
4. Program correctly handles guesses of the wrong length (5 marks)
5. Program correctly handles guesses containing non-letters (5 marks)
6. Program correctly handles invalid guesses (with both default and supplied dictionary) (6 marks)
7. Program correctly handles incorrect guesses with various capitalisation (5 marks)
8. Program correctly handles running out of attempts (5 marks)
9. Program correctly handles EOF on input (5 marks)
10. Program correctly handles getting the answer right (including varied capitalisation) (5 marks)
11. Program correctly handles valid guesses with repeated letters (including varied capitalisation) (4 marks)
12. Program only reads the dictionary file once and frees all allocated memory (4 marks)

It is unlikely that you can earn marks for categories 7 and later unless your program correctly reports matching letters. It is unlikely that you can earn marks for categories 4 and later unless your program correctly prompts for guesses. Some functionality may be assessed in multiple categories, e.g. the ability to set the word length with `-len`.

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. **For assignment one, there is no weighting associated with human style marking – but you will be given feedback so that you can improve your style for later programming assignments.**

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[8].

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

[7] Valgrind tests for memory leaks (category 12) will be allowed to run for longer.
[8] Make sure you remove any unneeded files from your repository, or they will be subject to style marking.

- W be the total number of distinct compilation warnings recorded when your `.c` files are individually built (using the correct compiler arguments)
- 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[9].

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) – not included in the assignment one total mark

Your human style mark will not be included in your total mark for assignment one, but feedback will be given so that you are better prepared for the later programming assignments.

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

[9] 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.

SVN commit history assessment (5 marks) – not included in the assignment one total mark

Your SVN commit history mark will not be included in your total mark for assignment one, but feedback will be given so that you are better prepared for the later programming assignments.

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:

- 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) – not counted for assignment one.
- C be the SVN commit history mark (out of 5) – not counted for assignment one.
- 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:

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

out of 65 (for CSSE2310 students) or 75 (for CSSE7231 students).

For future programming assignments, your total mark 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`.

Version 1.1

- Clarified that command line validity must be checked prior to checking whether a provided dictionary file can be read.
- Clarified that guesses will be no more than 50 characters longer (excluding any newline) – longer guesses will not be tested.
- Clarified that valgrind tests for memory leaks (marking criteria 12) will be allowed to run for longer than the stated 10 seconds.
- Clarified that marking criteria 7 is about incorrect guesses and that criteria 11 is about valid guesses.
- Fixed hint about `tolower()` etc. – casting is not needed to avoid compiler warnings with the compiler that we are using.
- Clarified when correct guesses are checked for.