The University of Queensland
School of Electrical Engineering and Computer Science

CSSE2310/CSSE7231 — Semester 2, 2023
Assignment 1 (version 1.1)

Marks: 75
Weighting: 15%
Due: 4:00pm Thursday 24 August, 2023

This specification was created for the use of Adnaan BUKSH (s4743556) only.
Do not share this document. Sharing this document may result in a misconduct penalty.
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 `uqwordladder`) which allows users to play a word ladder game – i.e. converting one word into another by changing one letter at a time, with each intermediate word being a valid word. For example, `HEAD` can be turned into `TAIL` by the sequence `HEAD` → `HEAL` → `TEAL` → `TELL` → `TALL` → `TAIL`[1]. 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. (Code in private posts to the discussion forum is permitted.) You must assume that some students in the course may have very long extensions so do not post your code to any public repository until at least three months after the result release date for the course (or check with the course coordinator if you wish to post it sooner). Do not allow others to access your computer – you must keep your code secure. Never leave your work unattended.

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, found in `/local/courses/csse2310/resources` on `moss`, posted on the discussion forum by teaching staff, provided in Ed Lessons, or shown in class).	May be used freely without reference. (You must be able to point to the source if queried about it – so you may find it easier to reference the code.)
Code you have personally written this semester for CSSE2310/7231 (e.g. code written for A1 reused in A3) – provided you have not shared or published it.	May be used freely without reference. (This assumes that no reference was required for the original use.)

[1] This sequence is from Lewis Carroll's original example – see `https://lewiscarrollresources.net/doublets/index.html`.

Code Origin	Usage/Referencing
Code examples found in man pages on `moss`.	May be used provided you understand the code AND the source of the code is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.
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 learned from or that you have taken inspiration from but have not copied[2].	
Code written by, modified by, or obtained from, or code based on code written by, modified by, or obtained from, a code generation tool (including any artificial intelligence tool) that you personally have interacted with, without the assistance of another person.	May be used provided you understand that code AND the source of the code is referenced in a comment adjacent to that code (in the required format) AND an ASCII text file (named `toolHistory.txt`) is included in your repository and with your submission that describes in detail how the tool was used. The file must be committed to the repository at the same time as any code derived from such a tool. If such code is used without appropriate referencing and without inclusion of the `toolHistory.txt` file then this will be considered misconduct.
Other code – includes (but may not be limited to): code provided by teaching staff only in a previous offering of this course (e.g. previous assignment one solution); code from public or private repositories; code from websites or based on code from websites; code from textbooks; any code written or partially written or provided by or written with the assistance of 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.

You must not share this assignment specification with any person, organisation, website, etc. 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 many of these sites and many cooperate with us in misconduct investigations. You are permitted to post small extracts of this document to the course Ed Discussion forum for the purposes of seeking or providing clarification on this specification.

The teaching staff will conduct interviews with a subset of 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 legitimately use code from other sources (following the usage/referencing requirements in the table above) then you are expected to understand that code. If you are not able to adequately explain the design of your solution and/or adequately explain your submitted code (and/or earlier versions in your repository) 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 and/or your submission may be subject to a misconduct investigation where your interview responses form part of the evidence. Failure to attend a scheduled interview will result in zero marks for the assignment. The use of referenced code in your submission (particularly from artificial intelligence tools) is likely to increase the probability of your selection for an interview.

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. Don't help another CSSE2310/7231 student with their code no matter how desperate they may be and no matter how close your relationship. You should read and understand the statements on student misconduct in the course profile and on the school website: `https://eecs.uq.edu.au/current-students/guidelines-and-policies-students/student-conduct`.

Specification

The `uqwordladder` game will print out a starting word and a target word (of the same length) and ask the user to enter one word at a time where each word is (a) a valid word (i.e. can be found in a given dictionary or is the target word), and (b) is the right length and (c) is only one letter different from the previous word, and (d) is not one of the words entered previously. The goal is to come up with a sequence of words ("steps") that ends

[2]Code from elsewhere must not be directly copied or just converted from one programming language to another.

with the target word[3] and to do so within some limit on the number of steps.

You can choose to play the game with a particular start and/or end word, or the game will choose a random word or words for you. An example game play session is shown below. Lines typed by the user (i.e. the command and words entered on standard input) are shown in **bold green** for clarity. Other lines are output from the program to standard output. Note that the `$` character is the shell prompt – it is not entered by the user nor output by `uqwordladder`. Note that players can enter the question mark (?) character at any stage to get suggestions on valid words they can enter. Note also that words entered by the user are case insensitive – they can be upper case, lower case or a mixture of the two.

Example 1: Example `uqwordladder` game session

```
$ ./uqwordladder
Welcome to UQWordLadder!
Your goal is to turn 'NORM' into 'FARE' in at most 13 steps
Enter word 1 (or ? for help):
?
Suggestions:-----------
 CORM
 DORM
 FORM
 FORM
 FORM
 NOAM
 NORA
 NORD
 NORE
 NORI
 NORK
 NORK
 NORN
 NORN
 NORW
 NPRM
 WORM
 WORM
-----End of Suggestions
Enter word 1 (or ? for help):
form
Enter word 2 (or ? for help):
FARM
Enter word 3 (or ? for help):
faRE
You Well done - you solved the ladder in 3 steps
$
```

Full details of the required behaviour are provided below.

Command Line Arguments

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

`./uqwordladder [--init` *`initialWord`*`] [--target` *`toWord`*`] [--max` *`stepLimit`*`] [--length` *`len`*`] [--dict` *`dictfilename`*`]`

The square brackets (`[]`) indicate optional arguments. The *italics* indicate placeholders for user-supplied arguments. Any or all of the options can be specified (at most once each). Options can be in any order.

Some examples of how the program might be run include the following[4]:

```
./uqwordladder
```

[3] Note that the start and end words need not be valid dictionary words themselves, but all intermediate words must be valid words.

[4] This is not an exhaustive list and does not show all possible combinations of arguments.

```
./uqwordladder --init head
./uqwordladder --length 5
./uqwordladder --dict mywords --init TREE
./uqwordladder --max 20 --init car --length 3 --target Bus
```

The meaning of the arguments is as follows:

- `--init` – if specified, this option argument is followed by the starting word (combination of letters, in uppercase, lowercase or mixed-case).
- `--target` – if specified, this option argument is followed by the end or target word (combination of letters, in uppercase, lowercase or mixed-case).
- `--length` – if specified, this option argument specifies the length of the words in the word ladder.
- `--max` – if specified, this option argument specifies the maximum number of words that can be entered in the word ladder before the game is terminated. The default value is 13. The minimum value that can be specified is the same as the length of the word; the maximum value is 40.
- `--dict` – if specified, this option argument is followed by the name of a file that is to be used as the dictionary of valid words.

If more than one of `--init`, `--target` or `--length` is specified, then the lengths must be consistent (i.e. the same), and the length must be between 2 and 9 inclusive.

If the `--init` argument is not supplied, then the program must choose a random initial word using the supplied `get_uqwordladder_word()` function – see the Provided Library section on page 9. If the `--length` argument is given then the random starter word must have that length. If the `--length` argument is not given, but the `--target` argument is given, then the random starter word must have the same length as the target word. If none of `--init`, `--target` or `--length` is specified, then the word length to be used is 4.

If the `--target` argument is not supplied, then the program must choose a random target word using the supplied `get_uqwordladder_word()` function – see the Provided Library section on page 9. If the `--length` argument is given then the random target word must have that length. If the `--length` argument is not given, but the `--init` argument is given, then the random target word must have the same length as the initial word. If none of `--init`, `--target` or `--length` is specified, then the word length to be used is 4.

If the maximum number of steps is not specified (i.e. no `--max` option is specified), then the default value of 13 must be used. If this argument is specified, then the value must be between the word length and 40 inclusive – i.e. if the word length being used is 4 then the minimum acceptable limit on the number of steps is 4 (and the maximum is 40).

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

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 (single line) message:

```
Usage: uqwordladder [--init initialWord] [--target toWord] [--max stepLimit]
[--length len] [--dict dictfilename]
```

to standard error (with a following newline), and exit with an exit status of 11.

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

- A valid option argument is given (i.e. `--init`, `--target`, `--max`, `--length` or `--dict`) but it is not followed by an associated value argument.
- The `--length` argument is given but the associated value is not a positive decimal integer.
- The `--max` argument is given but the associated value is not a positive decimal integer.
- Any of the option arguments are listed more than once (with associated values). Note that the command line arguments like `--init --init` would not be an invalid command line – this would be an invalid word error – see below.
- An unexpected argument is present.

Checking whether the *initialWord*, *toWord*, *stepLimit*, *len* and/or *dictfilename* arguments (if present) are themselves valid is not part of the usage checking (other than checking that the *stepLimit* and/or *len* arguments are positive decimal integers). Validity of values is checked after command line validity as described in the

following sections – and in the same order as these sections are listed, i.e. length validity is checked before word validity, which is checked before the *stepLimit* option validity, which is checked before dictionary filename validity.

Length Validity Checking

If more than one of `--init`, `--target` or `--length` is specified (with associated values) then your program must check that the word lengths are consistent (i.e. the same). If they are not the same, then your program must print the message:

```
uqwordladder: Word length conflict - lengths must be consistent
```

to standard error (with a following newline), and exit with an exit status of 2.

If the lengths are consistent, or only one of these arguments is specified, then the length must be between 2 and 9 inclusive. If not, then your program must print the message:

```
uqwordladder: Word length should be between 2 and 9 (inclusive)
```

to standard error (with a following newline), and exit with an exit status of 15.

Command Line Word Checking

If either or both of the `--init` or `--target` arguments are specified (with associated option values) then the given words must be checked for validity. Valid words contain only letters (lowercase and/or uppercase). If either or both of these words are invalid (i.e. contain characters other than letters) then your program must print the message:

```
uqwordladder: Words must not contain non-letters
```

to standard error (with a following newline), and exit with an exit status of 10.

It is not required that the word(s) given on the command line be found in the dictionary. Only the "intermediate" words need to be in the dictionary.

If both of the `--init` and `--target` arguments are specified (with associated values that are valid) then the given words must also be checked to make sure they are different. If the two words are the same (when ignoring the case of the letters) then your program must print the message:

```
uqwordladder: Start and end words must not be the same
```

to standard error (with a following newline), and exit with an exit status of 4.

If one or both words are generated by calling `get_uqwordladder_word()` your program can assume that the initial and end words are different, i.e. it does not have to handle the case that they are the same.

stepLimit Checking

If the `--max` argument is specified (with an associated positive decimal integer option value), then the value must be checked for validity. Values must be between the word length (as determined above) and 40 (inclusive). If the value is outside the permitted range, then your program must print the message:

```
uqwordladder: Limit on steps must be word length to 40 (inclusive)
```

to standard error (with a following newline), and exit with an exit status of 13.

Dictionary File Name Checking

By default, your program is to use the dictionary file `/usr/share/dict/words`. If the `--dict` argument is supplied, then your program must instead use the dictionary whose filename is given as the value associated with that option argument. If the given dictionary filename does not exist or can not be opened for reading, your program must print the message:

```
uqwordladder: File "filename" cannot be opened
```

to standard error (with a following newline), and exit with an exit status of 1. (The italicised *filename* is replaced by the actual filename, i.e. the value from the command line. The double quotes must be present.) This check happens after all of the checks above.

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)[5], 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 must determine start and target words if one or both of these is not given on the command line. It does this by calling `get_uqwordladder_word()` – see details of this provided library function on page 9. Note that if neither `--init` nor `--target` is specified then the *initialWord* must be determined by the first call to `get_uqwordladder_word()` and the *toWord* must be determined by the second call to `get_uqwordladder_word()`. Your program may assume that the word(s) returned by `get_uqwordladder_word()` are upper case words of the right length containing only letters (and that the words are different if your program calls the function twice).

Your program must then print the following to standard output (with a newline at the end of each line):

```
Welcome to UQWordLadder!
Your goal is to turn 'initialWord' into 'toWord' in at most stepLimit steps
Enter word 1 (or ? for help):
```

where

- *initialWord* is replaced by the initial word (printed in upper case);
- *toWord* is replaced by the target word (printed in upper case);
- *stepLimit* is replaced by the limit on the number of steps in the ladder.

These values are as specified on the command line, or, if not specified there, as returned by `get_uqwordladder_word()` in the case of the words, or the default value for the maximum number of steps (13). Note that the single quotes (') must be present around the words.

Your program must repeatedly prompt the user for the next word by printing the following message to standard output (with a following newline):

```
Enter word N (or ? for help):
```

where *N* is replaced by the step number (starting from 1). Note that invalid words (see below) are not counted.

Words are entered on standard input and are terminated by a newline (or pending EOF). The word must then be checked for validity – with checks happening in the following order.

If the user enters a question mark (`?`) on a line by itself then the program must show some word suggestions (see the "Providing Suggestions" section on page 7). No further validity checks are performed on the question mark.

If the word entered by the user is not the expected length, then your program must print the following message to standard output (followed by a newline) and then re-prompt for that word:

```
Word should have num characters - try again.
```

where *num* is replaced by the expected word length. This includes the case where the user just presses Return/Enter, i.e. when the length of the "word" is zero.

If the word contains characters other than letters A to Z (can be any case, i.e. lowercase or uppercase), then your program must print the following message to standard output (followed by a newline) and then re-prompt for that word:

```
Word should contain only letters - try again.
```

If the word is not exactly one character different (when ignoring the case of the letters) to the last valid word entered (or the initial word, if this is the first word entered) then your program must print the following message to standard output (followed by a newline) and then re-prompt for that word:

```
Word must differ by only one letter - try again.
```

If the word is the same (when ignoring the case of the letters) as one of the valid words entered previously (or the initial word) then your program must print the following message to standard output (followed by a newline) and then re-prompt for that word:

```
You can't return to a previous word - try again.
```

If the word is not the target word and the word can't be found in the dictionary (with letters of any case), then your program must print the following message to standard output (followed by a newline) and then re-prompt for that word:

[5] Being told that you may assume something means that your program does not have to handle situations where this assumption is not true – i.e. in this case, 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.

`Word not found in dictionary - try again.`

If an invalid word is entered, then the program will prompt again for the same word by printing the following to standard output (with a trailing newline):

`Enter word N (or ? for help):`

The N value is unchanged from the last prompt.

If the word entered is valid (i.e. passes the checks above) and it matches the target word then the game is over – the user has won. The program must print the following to standard output (with a trailing newline):

`Well done - you solved the ladder in N steps.`

where N is replaced by the number of steps it took to solve the ladder. Your program must then exit with exit status 0.

If the word is valid but the user has run out of steps (i.e. the *stepLimit* limit has been reached, then the game is over – the user has lost. The program must print the following to standard output (with a trailing newline):

`Game over - no more steps remaining.`

Your program must then exit with exit status 12.

If the word is valid, but not the target word, and the user has steps remaining then the program will prompt for a new word, i.e. will print

`Enter word N (or ? for help):`

to standard output, where N is incremented.

If your program detects end-of-file (EOF[6]) at the start of the line when attempting to read a word, then the game is also over – the user has given up[7]. The program must print the following to standard output (with a trailing newline):

`Game over - you gave up.`

Your program must then exit with exit status 19.

Providing Suggestions

If the user enters a question mark (`?`) when prompted for a word then your program must help the user by suggesting possible words. The words to be suggested are (a) the target word if it is one character different from the last word entered (or one character different from the first word if no words have been entered) ~~and~~ followed by (b) all words in the supplied dictionary that:

- are the right length;
- contain only letters;
- are one character different from the last word entered (or the initial word if no intermediate words have yet been entered);
- have not been previously entered;
- are not the initial word; and
- are not the target word.

All words must be shown in uppercase. If the target word is to be listed, then it must be listed first and will only be listed once. Words from the dictionary must be listed in the same order as they are present in the dictionary. If a word is present more than once in the dictionary then it will be listed more than once here.

If suggestions are able to be made, then your program must output the following line to standard output (followed by a newline):

`Suggestions:-----------`

(there are 11 dashes after the colon).

The suggestions must be printed to standard output in uppercase – one per line, each with a single leading space and a trailing newline. After all the suggestions have been printed, your program must output the following line to standard output (followed by a newline):

[6] EOF is "end of file" – the stream being read (standard input in this case) is detected as being closed. Where standard input is connected to a terminal (i.e. a user interacting with a program) then the user can close the program's standard input by pressing Ctrl-D (i.e. hold the Control key and press D). This causes the terminal driver which receives that keystroke to close the stream to the program's standard input.

[7] Note that a word might be terminated by EOF rather than newline. Such an entry must be treated as if the word was followed by a newline. EOF will be detected on the subsequent attempt to read a word.

`-----End of Suggestions`

(there are 5 dashes present before the word "End").

If no suggestions are available (i.e. there are no words that satisfy the requirements above), then your program must output the following to standard output (followed by a newline):

`No suggestions available.`

If no suggestions are available then the user will not be able to make any valid guesses – but it is up to them to finish the game by closing standard input (i.e. by pressing `Ctrl-D` if playing interactively).

Other Requirements

Your program must open and read the dictionary file only once[8] and store its contents (or a subset of its contents, e.g. only words of the right length that contain only letters) in dynamically allocated memory. Your program must free all allocated memory before exiting.

Example Game Sessions

Example 2: Example `uqwordladder` game session. Lines entered by the user are shown in **bold green**. Some suggestions from the dictionary have been omitted to save space.

```
$ ./uqwordladder --init bugs --target code
Welcome to UQWordLadder!
Your goal is to turn 'BUGS' into 'CODE' in at most 13 steps
Enter word 1 (or ? for help):
?
Suggestions:-----------
 BAGS
 BEGS
 ...
 RUGS
 TUGS
 VUGS
-----End of Suggestions
Enter word 1 (or ? for help):
bags
Enter word 2 (or ? for help):
bars
Enter word 3 (or ? for help):
hello
Word should have 4 characters - try again.
Enter word 3 (or ? for help):
bars
Word must differ by only one letter - try again.
Enter word 3 (or ? for help):
code
Word must differ by only one letter - try again.
Enter word 3 (or ? for help):
1234
Word should contain only letters - try again.
Enter word 3 (or ? for help):
baGS
You can't return to a previous word - try again.
Enter word 3 (or ? for help):
barz
Word not found in dictionary - try again.
Enter word 3 (or ? for help):
CARS
Enter word 4 (or ? for help):
```

[8] `rewind()`ing the file or `fseek()`ing to the beginning are not permitted.

```
Care
Enter word 5 (or ? for help):
core
Enter word 6 (or ? for help):
code
Well done - you solved the ladder in 6 steps.
$
```

The example below shows a blank line being input at the first prompt (i.e. no characters were present before the newline). The first valid word (`eval`) has been terminated by an EOF – e.g. the user typed Ctrl-D Ctrl-D (i.e. Ctrl-D twice)[9] after typing `eval` rather than typing a newline. This pending EOF terminated the word, and when an attempt was made to read the next line (word 2), the EOF was detected.

Example 3: Example `uqwordladder` game session – with early termination.

```
$ ./uqwordladder --target Good --init evil --max 35
Welcome to UQWordLadder!
Your goal is to turn 'EVIL' into 'GOOD' in at most 35 steps
Enter word 1 (or ? for help):

Word should have 4 characters - try again.
Enter word 1 (or ? for help):
?
Suggestions:-----------
 EMIL
 EPIL
 EVAL
 EVIE
 EVIN
-----End of Suggestions
Enter word 1 (or ? for help):
evalEnter word 2 (or ? for help):
Game over - you gave up.
$
```

Provided Library: libcsse2310a1

A library has been provided to you with the following function which your program must use (when no starter word is provided on the command line):

```
const char* get_uqwordladder_word(unsigned int wordLen);
```

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

Style

Your program must follow version 2.4 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site. Your submission must also comply with the *Documentation required for the use of AI tools* if applicable.

[9] If the cursor is not at the start of a line, then a single Ctrl-D causes the text entered so far to be flushed to the program's standard input (i.e. available to be read by the program if desired). An immediately following Ctrl-D causes standard input to be closed.

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()`, `isdigit()`, `islower()`, `isupper()`, `toupper()` and/or `tolower()`. Note that these functions operate on individual characters, represented as integers (ASCII values).
3. Some other functions which **may** be useful include: `strcmp()`, `strlen()`, `strdup()`, `exit()`, `fopen()`, `fprintf()`, and `fgets()`. You should consult the man pages for these functions.
4. 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.
5. You may wish to consider using a function from the `getopt()` family to parse command line arguments. This is not a requirement and if you do so your code may not be any simpler or shorter than if you don't use such a function. You may wish to consider it because it gives you an introduction to how programs can process more complicated combinations of command line arguments. See the `getopt(3)` man page for details. Note that short forms of arguments are not to be supported, e.g. the arguments "`-d filename`" (for specifying a dictionary filename) shouldresult in a usage error. To allow for the use of a `getopt()` family function, we will not test your program's behaviour with the argument `--` (which is interpreted by `getopt()` as a special argument that indicates the end of option arguments). We also won't test abbreviated arguments, e.g. `--di`, `--dic`, 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 11. 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![10]
2. Detect the presence of the `--init` command line argument and associated *initialWord*.
3. Detect the presence of the `--target` command line argument and associated *toWord*.
4. Detect the presence of the `--length` command line argument and check the presence/validity of the argument that follows it. Exit appropriately if not valid.
5. 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.
6. Detect the presence of the `--dict` command line argument and associated *dictfilename*.
7. Check for word length validity and consistency and exit appropriately if not.
8. Check whether any words supplied on the command line are valid and exit appropriately if not.
9. Check that the *stepLimit* argument is valid, if specified. Exit appropriately if not.
10. Check whether the dictionary can be opened for reading or not. Exit appropriately if not. (If it can be opened, leave it open – you'll need to read from it next.)
11. Have your program print out the welcome message
12. Read the dictionary line by line, saving the words (or an appropriate subset of words) into dynamically allocated memory.
13. Have your program repeatedly prompt for words.
14. Check the words for validity.
15. Implement remaining functionality as required ...

[10] 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 invalid command lines.

Forbidden Functions, Statements etc.

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

- `goto`
- `#pragma`
- `gcc` attributes

You must not use any of the following C functions. If you do so, you will get zero (0) marks for any test case that calls the function.

- `longjmp()` and equivalent functions
- `system()`
- `popen()`
- `mkfifo()` or `mkfifoat()`
- `fork()`
- `pipe()`
- `rewind()`
- `fseek()`
- `execl()`, `execvp()` or any other members of the exec family of functions
- Functions described in the man page as non standard, e.g. `strcasestr()`

Submission

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

Your program (named `uqwordladder`) must build on `moss.labs.eait.uq.edu.au` and in the Gradescope environment 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 `uqwordladder` 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)[11]. 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 will be asked to confirm references in your code and also to confirm your use (or not) of AI tools to help you.

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[12] will incur a late penalty – see the CSSE2310/7231 course profile for details.

Note that Gradescope will run the test suite immediately after you submit. When complete[13] you will be able to see the results of the "public" tests.

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 dictionary file then we can not determine if your program can determine word validity correctly. Your tests must run in a reasonable time frame, which could be as short as a few seconds for usage checking to many tens of seconds when `valgrind` is used to test for memory leaks. If your program takes too long to respond, 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 invalid and/or inconsistent word lengths (4 marks)
3. Program correctly handles invalid words specified on the command line (3 marks)
4. Program correctly handles invalid *stepLimit* value (2 marks)
5. Program correctly handles dictionary files that are unable to be read (3 marks)
6. Program correctly prints the welcome message (with a variety of valid command lines) (3 marks)
7. Program correctly handles games with only invalid or blank attempts – with EOF on stdin to terminate game (8 marks)

[11] 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.

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

[13] Gradescope marking may take only a few minutes or more than 30 minutes depending on the functionality/efficiency of your code.

8. Program correctly plays game with only valid attempts made (which may include early termination via EOF on stdin) (8 marks)
9. Program correctly handles playing games with a variety of valid and invalid attempts (5 marks)
10. Program correctly provides suggestions in response to entry of question mark (`?`) with a variety or valid and invalid attempts (8 marks)
11. Program behaves correctly, reads dictionary only once, and frees all memory upon exit (8 marks)

Tests for categories 6 and higher will take place with a variety of valid command lines, including a variety of dictionary files. If your program doesn't detect a particular set of command line options as valid, or doesn't handle a particular option (e.g. `--length`) or can't open a dictionary other than the default dictionary then you will lose marks in multiple categories. There are other dependencies between categories. For example, if your program doesn't correctly print the welcome message then no marks will be possible for categories 6 to 11.

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 2.4 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[14].

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)
- 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[15].

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`. You can check the result of Gradescope style marking soon after your Gradescope submission – when its test suite completes running.

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

[15] 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.

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 Marking (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:

- 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 – only one or two commits OR all commit messages are meaningless.
1	Some progressive development evident (three or more commits) AND at least one commit message is meaningful.
2	Progressive development is evident (multiple commits) AND at least half the commit messages are meaningful.
3	Multiple commits that show progressive development of ALL functionality (e.g. no large commits with multiple features in them) AND at least half the commit messages are meaningful.
4	Multiple commits that show progressive development of ALL functionality AND meaningful messages for all but one or two of the 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.

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).
- V is the scaling factor (0 to 1) determined after interview(s) (if applicable – see the Student Conduct section above) – or 0 if you fail to attend a scheduled interview without having evidence of exceptional circumstances impacting your ability to attend.

Your total mark for the assignment will be:

$$M = (F + \min\{F, S + H\} + \min\{F, C\}) \times V$$

out of 75.

In other words, you can't get more marks for style or SVN commit history 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 clarifications or updates to the assignment specification will be summarised here. Any updated versions of the specification will be 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 suggestions must not include the initial word and that the target word (if to be listed) will only be listed once and must be listed first.
- Clarified that the program does not have to deal with start and initial words being the same if one (or both) of them is generated with `get_uqwordladder_word()`

- Clarified that comparisons “same” and “one character different” are considered when ignoring the case of the letters
- Corrected example 1 to include duplicated dictionary words that should be output if the default dictionary is used.
- Correct default maximum number of steps in example game sessions
- Fixed success message in example one to match the required text