# Assignment 1 - Word Ladder

## Overview

In Week 2 we are learning about the C++ STL, and this assignment is your chance to practise those skills.

## Change Log

- 09/06: Clarification on give command: Able to submit more than the core files if desired (but certainly not necessary)
- 09/06: First examples of word ladders now contain all of their possible outputs to avoid confusion.
- 09/06: Added line "In this assignment you may use any STL containers or algorithms that you see fit"
- 09/06: 5% linter mark instructions given more clearly
- 09/06: Reference solution & time limit section added
- 10/06: Directory in which to call reference solution updated
- 10/06: "You are allowed to have a trailing space at the end of each line." for each ladder you output. This now reflects the reference solution. Previously it said no trailing whitespace. If your program already accounted for having no trailing whitespace, you will not be penalised at all.
- 15/06: Input and output words will be the same length during testing
- 18/06: Start and destination words will always be in the lexicon

## Introduction

Your task is to write a program to build **word ladders**. You will leverage the C++ standard containers `std::vector`, `std::queue`, and `std::set` to complete this task.

A word ladder is a connection from one word to another word formed by changing one letter at a time with the constraint that at each step the sequence of letters still forms a valid word. For example, here is a word ladder connecting "code" to "data":

```
code -> cade -> cate -> date -> data
```

You will ask the user to enter a start and a destination word, and then your program should find a word ladder between them if one exists. By using an algorithm known as breadth-first search, you are guaranteed to find the shortest such sequence.

Here is some sample output of the word ladder program:

```
Enter start word (RETURN to quit): work
Enter destination word: play
Found ladder: work fork form foam flam flay play
work pork perk peak pean plan play
work pork perk peak peat plat play
work pork perk pert peat plat play
work pork porn pirn pian plan play
work pork port pert peat plat play
work word wood pood plod ploy play
work worm form foam flam flay play
work worn porn pirn pian plan play
work wort bort boat blat plat play
work wort port pert peat plat play
work wort wert pert peat plat play

Enter start word (RETURN to quit): awake
Enter destination word: sleep
Found ladder: awake aware sware share sharn shawn shewn sheen sheep sleep
awake aware sware share shire shirr shier sheer sheep sleep

Enter start word (RETURN to quit): airplane
Enter destination word: tricycle
No ladder found.
```

## Exploring the problem

Finding a word ladder is a specific instance of the shortest path problem, where the challenge is to find the shortest path from a starting position to a goal. Shortest path problems come up in a variety of situations such as packet routing, robot motion planning, social networks, studying gene mutations, and more. One approach for finding a shortest path is the classic algorithm known as breadth-first search. A breadth-first search searches outward from the start in a radial fashion until it hits the goal. For word ladder, this means first examining those ladders that represent "one hop" (i.e. one changed letter) from the start. If any of these reaches the destination, we're done. If not, the search now examines all ladders that add one more hop (i.e. two changed letters). By expanding the search at each step, all one-hop ladders are examined before two-hops, and three-hop ladders only considered if none of the one-hop nor two-hop ladders worked out, thus the algorithm is guaranteed to find the shortest successful ladder.

Breadth-first is typically implemented using a queue. The queue is used to store partial ladders that represent possibilities to explore. The ladders are enqueued in order of increasing length. The first elements enqueued are all the one-hop ladders, followed by the two-hop ladders, and so on. Due to FIFO handling, ladders will be dequeued in order of increasing length. The algorithm operates by dequeueing the front ladder from the queue and determining if it reaches the goal. If it does, you have a complete ladder, and it is the shortest. If not, you take that partial ladder and extend it to reach words that are one more hop away, and enqueue those extended ladders onto the queue to be examined later. If you exhaust the queue of possibilities without having found a completed ladder, you can conclude that no ladder exists.

A few of these tasks deserve a bit more explanation. For example, you will need to find all the words that differ by one letter from a given word. A simple loop can change the first letter to each of the other letters in the alphabet and ask the lexicon if that transformation results in a valid word. Repeat that for each letter position in the given word and you will have discovered all the words that are one letter away.

Another subtle issue is the restriction that you may not re-use words that have been included in a previous (and shorter) ladder. This is an optimization that avoids exploring redundant paths. For example, if you have previously tried the ladder `cat -> cot -> cog` and are now processing `cat -> cot -> con`, you would find the word `cog` one letter away from `con`, so `cog` looks like a potential candidate to extend this ladder. However, `cog` has already been reached in an earlier and shorter ladder, and there is no point in re-considering it in a longer ladder.

Finally, you need to avoid unwanted loops or redundant paths, such as in a circular ladder like:

```
cat -> cot -> cog -> bog -> bat -> cat
```

Since you need linear access to all of the items in a word ladder when time comes to print it, it makes sense to model a word ladder using a `vector<string>`. And remember that you can make a copy of a `vector<string>` by just assigning it to be equal to another via traditional assignment (i.e., `vector<string> wordLadderClone = wordLadder`).

### If there is more than one shortest path

In this case, your implementation must print all solutions as follows:

- Each solution appears on a separate line (terminated by a newline). The very first one is printed after "Found ladder: ".
- The words in each solution are separated by exactly one blank space. You are allowed to have a trailing space at the end of each line.
- All solutions (considered as a string each) are printed in their lexicographic order

Sample output:

```
Enter start word (RETURN to quit): con
Enter destination word: cat
Found ladder: con can cat
con cot cat
```

Please ensure all words output from the word ladder are lowercase.

### Implementation Hints

Please leverage STL libraries, such as:

- The linear, random-access collection managed by a vector is deal for storing a word ladder
- A queue object is a FIFO collection that is just what's needed to track those partial ladders under consideration. The ladders are enqueued (and thus dequeued) in order of length so as to find the shortest option first.
- The start and destination words are always taken from the lexicon

## The Task

Here is a development plan to help you if you're struggling to figure out how to approach the problem

### Step 1: Get familiar with the STL

You've seen std::vector and std::queue in lectures during week 2, but **std::set** is new. There's not a lot you'll need to do, but make sure you understand it. **In this assignment you may use any STL containers or algorithms that you see fit.**

### Step 2: Dictionary handling

In the starter code, we set up a std::set with the large dictionary read from our data file. Write a function that will iteratively construct strings that are one letter different from a given word and run them by the dictionary to determine which strings are words. You will be required to write tests for your code, but we are specifically not telling you what to test and what not to -

we will leave that up to you. You may do as little or as much testing as you like (testing marking scheme will be in the marking criteria).

### Step 3: Implement breadth-first search

The code is not long, but it can be dense, and definitely requires understanding how a BFS works.

## Getting Started

If you haven't done so already, clone the repository:

```
$ git clone https://github.com/cs6771/comp6771 comp6771
```

Then navigate to the **assignments/wl** directory

All of the files you need are in this directory. Here is a list of files that and a description of their purpose:

| File | Description |
|---|---|
| **main.cpp** | **The binary that will be ran** |
| word_ladder.cpp/h | Your "library" of word ladder functions |
| word_ladder_test.cpp | Your tests for the library of word ladder functions |
| lexicon.cpp/h | The function to generate the set from the file |
| words.txt | Full list of valid words in the lexicon |
| BUILD | Build file containing build, test, dependency instructions |

## Reference Solution & Time Limits

A reference solution has been included in the word ladder folder (named "reference_solution").

Your program must solve the problem in a time no greater than 10 times the time taken for the reference solution to complete it.

For a given test file (e.g. test1, shown below) you can test the timing of the reference solution as follows:

```

# Note: This must be run in the project directory
time ./assignments/wl/reference_solution < test1
```

Where **test1** is:

```
cat
dog
```

## Running your code

From your project directory:

### Running

```
$ bazel build //assignments/wl:main
```

```
$ bazel run //assignments/wl:main
```

### Testing

```
$ bazel build //assignments/wl:word_ladder_test
```

```
$ ./bazel-bin/assignments/wl/word_ladder_test
```

## Assessment

This assignment will contribute 15% to your final mark.

The assessment for the assignment recognizes the difficulty of the task, the importance of style, and the importance of appropriate use of programming methods (e.g. using while loops instead of a dozen if statements).

| | |
|---|---|
| 55% | **Correctness**<br>The correctness of your program will be determined automatically by tests that we will run against your program. You will not know the full sample of tests used prior to marking. Your program must also find the word ladder(s) for a given input in the time limit specified above. |

| | |
|---|---|
| 20% | **Your tests**<br>You are required to write your own tests to ensure your program works. You will write tests in `word_ladder_test.cpp`. At the top of this file you will also include a block comment to explain the rational and approach you took to writing tests. Please read the Catch2 tutorial (https://github.com/catchorg/Catch2/blob/master/docs/tutorial.md) or review lecture/tutorial content to see how to write tests. Tests will be marked on several factors. These include, but are not limited to<br>• Correctness - an incorrect test is worse than useless.<br>• Coverage - your tests might be great, but if they don't cover the part that ends up failing, they weren't much good to you.<br>• Brittleness - If you change your implementation, will the tests need to be changed (this generally means avoiding calling functions specific to your implementation where possible - ones that would be private if you were doing OOP).<br>• Clarity - If your test case failed, it should be immediately obvious what went wrong (this means splitting it up into appropriately sized sub-tests, amongst other things). |
| 20% | **C++ style**<br>Your adherence to good C++ style. This is **not** saying that if you conform to the style guide you will receive full marks for this section. This 20% is also based on how well you use modern C++ methodologies taught in this course as opposed to using backwards-compatible C methods. Examples include: Not using primitive arrays and not using pointers. |
| 5% | **cpplint and clang-format**<br>In your project folder, run the following commands on all C++ files you submit:<br>`$ clang-format -i /path/to/file.cpp && python cpplint.py /path/to/file.cpp`<br>If the program outputs the following, you will receive full marks for this section (5%). Otherwise you will receive no marks.<br>`Done processing assignments/wl/main.cpp`<br>`Total errors found: 0` |

The following actions will result in a 0/100 mark for Word Ladder, and in some cases a 0 for COMP6771:

- Knowingly providing your work to anyone and it is subsequently submitted (by anyone).
- Submitting any other person's work. This includes joint work.
- Submitting another person's work without their consent.

The lecturer may vary the assessment scheme after inspecting the assignment submissions but it will remain broadly similar to the description above.

## Originality of Work

The work you submit must be your own work. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted. The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline. Assignment submissions will be examined both automatically and manually for such submissions.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct.

Do not provide or show your assignment work to any other person — apart from the teaching staff of COMP6771. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, you may be penalized, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.

Note you will not be penalized if your work has the potential to be taken without your consent or knowledge.

## Submission

This assignment is due **Saturday 22nd of June, 14:59:59**. Submit the assignment using this *give* command:

```
give cs6771 wordladder main.cpp word_ladder.cpp word_ladder_test.cpp word_ladder.h *
```

Note: The files above are the minimum required to submit. However, you can submit further files if you would like (e.g. if you create helper files). You can also submit your own BUILD file if varies from the one provided. Do not modify or submit lexicon.(cpp|h])

## Late Submission Policy

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 2%. For example if an assignment worth 76% was submitted 5 hours late, the late submission would have no effect (as maximum mark would be 90%). If the same assignment was submitted 30 hours late it would be awarded 40%, the maximum mark it can achieve at that time.