Assignment 1 - Word Ladder

Please `git pull` frequently to get the latest changes.

Change Log

- (2022-06-02): Initial Release

Overview

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

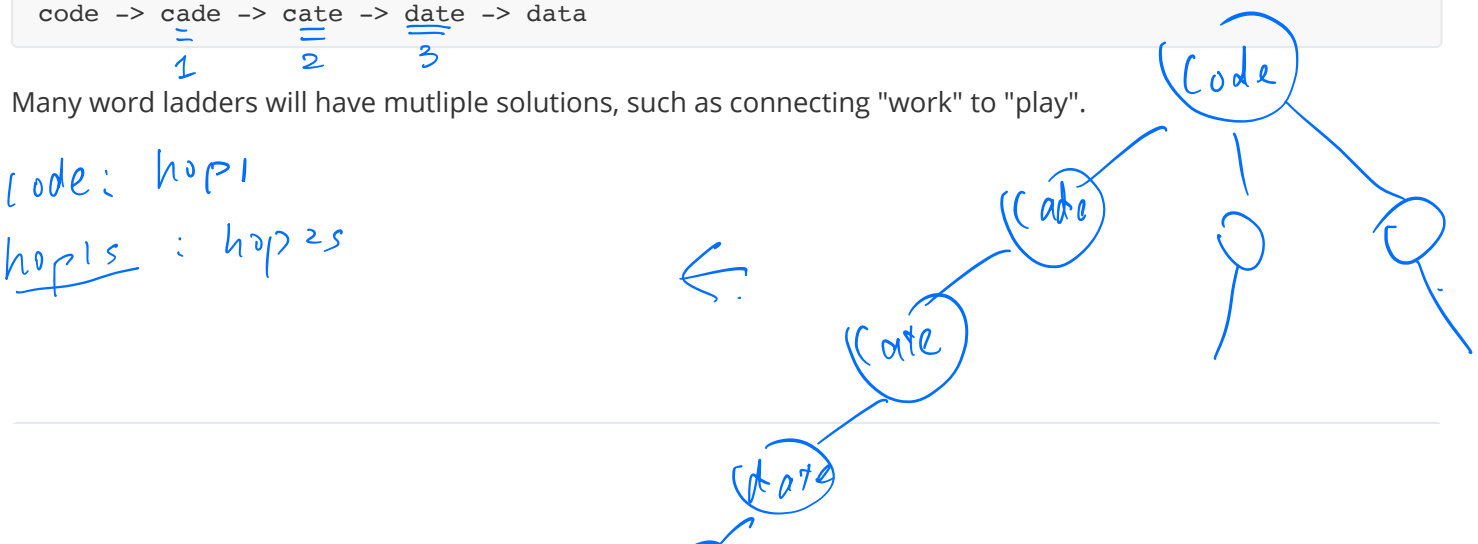
Now that you've been introduced to the C++ standard library, it's time to put that knowledge to use. In the role of client, the low-level details have already been dealt with, and you can focus your attention on solving more pressing problems. Having a library of thoroughly tested and reviewed types designed by field experts vastly broadens the kinds of tasks you can "easily" tackle. In this first assignment, you'll write the back-end for a program that heavily leverages the standard library to do nifty things. It might sound a little daunting at first; but given the power tools in your arsenal, you can focus on solving the problem at hand, instead of worrying about how to implement everything from scratch. Let's hear it for abstraction!

This assignment has several purposes:

1. To explore C++'s value-semantics.
2. To stress the notion of abstraction as a mechanism for managing data and providing functionality without revealing the implementation.
3. To become more familiar with C++'s type system.
4. To gain familiarity with the C++ standard library.

Leveraging `std::vector`, `std::queue`, and `std::unordered_set` are critical for writing a word ladder builder. A word ladder is a connection from one word to another, formed by changing one letter at a time, with the constraint that each transformation yields a new valid word. For example, here is a word ladder connecting "code" to "data".

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



Many word ladders will have mutliple solutions, such as connecting "work" to "play".

```
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
```

"work", "pork", "perk", "peak", "pean", "plan", "plat", "play", "peat"

The back-end that you will write takes a start word, a destination word, and a lexicon, which returns valid word ladders. By using a breadth-first search, you're guaranteed to find the shortest such sequence. Here are some more examples.

```
::word_ladder::generate("awake", "sleep", english_words);
// returns {
//   {"awake","aware","sware","share","sharn","shawn","shewn","sheen","sheep","sleep"}
//   {"awake","aware","sware","share","shire","shirr","shier","sheer","sheep","sleep"}
// }
::word_ladder::generate("airplane", "tricycle", english_words);
// returns {}
```

Understanding a word ladder implementation

Finding a word ladder is a specific instance of a 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, and studying gene mutations. One approach for finding a shortest path is the classic breadth-first search algorithm. A breadth-first search searches outward from the start in a radial fashion, until it hits the goal. For our word ladder, this means examining those ladders that represent one hop from the start. A "hop" is a change in letter. One "hop" from the start means one changed letter, two "hops" means two changes in letters, and so on. It's possible for the same position to change letters across multiple non-adjacent hops. If any of these reach the destination, we're done. If not, the search now examines all the ladders that add one more hop. By expanding the search at each step, all one-hop ladders are examined before two-hop ladders, and three-hop ladders are only taken into consideration if none of the one-hop or two-hop ladders work out; thus the algorithm is guaranteed to find the shortest successful ladder.

let q be a queue
label from as explored
q.enqueue(from)
while q is not empty do
    v = q.dequeue()
    if v is to then
        return v
    for all words from v to w in G.adjacent(v) do
        if w is not labeled as explored then
            label w as explored
            q.enqueue(w)

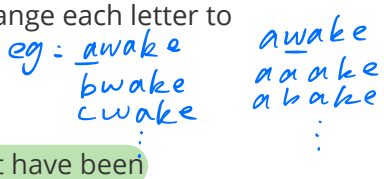
different(cur_word, new_word) == 1

(from,) == hop

const vector<string> hop-lexicon(hopnum, from, smaller_lexicon)

Breadth-first searches are typically implemented using a queue. The queue stores 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 dequeuing the front ladder from the queue and determining if it reaches the goal. If it does, then 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 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'll need to find all the words that differ by one letter from a given word. You might reach for a raw loop to change each letter to all the other letters in the alphabet. Repeat this for each letter position in the word and you will have discovered all the words that are one letter away.



Another, more subtle issue, is the restriction that you shouldn't reuse words that have been included in a previous ladder. This is an optimisation that avoids exploring redundant paths. For example, if you previously tried the ladder `cat->cot->cog` and are now processing `cat->cot->con`, you would find that the word `cog` is one letter away from `con`, which looks like a potential candidate to extend this ladder. However, `cog` has already been reached in an earlier (and thus shorter) ladder, so there is no point in reconsidering it in a longer ladder. The simplest way to ensure this is to keep track of the words that have been used in any ladder, and ignore them when they resurface. This is also necessary to avoid getting trapped in circular, non-terminal ladders such as `cat->cot->cog->bog->bat->cat`. Since you need linear access to all of the items in a word ladder when time comes to return it, it makes sense to model an individual word ladder using `std::vector<std::string>`. Remember that because C++ has value semantics, you're able to copy vectors via copy initialisation (e.g. `auto word_ladder_clone = word_ladder;`) and copy assignment (e.g. `word_ladder_clone = word_ladder`).

If there are multiple shortest paths, your implementation must return all the solutions, sorted in lexicographical order. Thus, the return type for your word ladder generator will be `std::vector<std::vector<std::string>>`.

The Task

This generator might seem like it's a panacea, but it still benefits from a step-by-step development plan to keep things moving along.

- **Task 1** --- *Familiarise yourself with the libraries available.* You don't need to deep-dive, but it would be a smart idea to read up on `std::vector`, `std::queue`, and `std::unordered_set`, from the standard library; and to read up on the range adaptors we've listed in the appendix. You shouldn't worry about their implementation details: focus on the interfaces, and how you use them in practice.
- **Task 2** --- *Test design.* We've provided you with a very simple test case to show you how to use the test library. You should add more `TEST_CASE`s underneath, so you have a suite of checks to

make sure you catch any logic errors in your generator. We adopt Google's Beyoncé rule in this class: "if you liked it, you should have put a test on it". Test words that are short (one or two letters) and test words that are longer.

- **Task 3** --- *Design your algorithm.* Be sure you understand the breadth-first search algorithm on paper, and what types you will need to use.
- **Task 4** --- *Lexicon handling.* Set up an `std::unordered_set` object with the large lexicon, read from our data file. There's a utility function called `word_ladder::read_lexicon` that will read it in from file for you. Please don't modify this function.

Assumptions

- You can assume that the start word and end word have the same length (i.e. number of characters).
- You can assume that both the start and the end word are in the lexicon.
- You can assume the start word and the end word will not be the same word

Implementation hints

Again, it's all about leveraging the libraries at your disposal --- you'll find your job is just to coordinate the activities of your objects to do the search.

- `std::vector` maintains a contiguous sequence of elements, and has random access. Benchmarks have shown that its contiguous storage makes it the fastest and smallest container in many situations (even when computer science theory tells us that a linked list would be better!).
- `std::queue` offers a FIFO interface over a container, which is probably what you'll want to use for tracking your partial ladders. Ladders are enqueued (and thus dequeued) in order of length, so as to find the shortest option first.
- `std::unordered_set` (note 1) is the hash-set that we use for the input lexicon, and is recommended for any internal lexicons you might use.
- [Algorithms] and ranges are powerful tools for describing what you want your program to do.
- As a minor detail, it doesn't matter if the start and destination words are contained in the lexicon or not. You can eliminate non-words from the get-go or just allow them to fall through and be searched anyway. During marking, the start and destination words will always be taken from the lexicon.

Getting Started

If you haven't done so already, clone this repository.

```
$ git clone gitlab@gitlab.cse.unsw.edu.au:COMP6771/22T2/students/z5555555/ass1.git
```

(Note: Replace z5555555 with your zid)

Navigate inside the directory. You can then open vscode with `code .` (not the dot).

Refer to `include/comp6771/word_ladder.hpp` for help starting your `source/word_ladder.cpp` file.

Running the tests

Similar to the first tutorial, you simply to have to run `Ctrl+Shift+P` and then type `Run Test` and hit enter. VS Code will compile and run all of your tests and produce an output.

Adding more tests

Part of your assignment mark will come from the quality and extensiveness of tests that you write.

You can add more test files to the `test/word_ladder/` directory. Simply copy `test/word_ladder/word_ladder_test1.cpp` into another file in that directory.

Note, everytime that you add a new file to the `test/word_ladder/` directory you will need to add another few lines to `test/CMakeLists.txt`. You can once again, simply copy the test reference for `word_ladder_test1.cpp` and rename the appropriate parts. Every time you update `CMakeLists.txt` in any repository, in VSCode you should press `Ctrl+Shift+P` and run `Reload Window` for the changes to take effect.

Measuring your performance

While you aren't assessed on the performance of your code, there is a **15 second time limit** per test that we will enforce when marking your assignment. For the average student, this means that some of the hardest test cases may not run fast enough with your algorithm and you may fail the latter tests due to timeout.

We have provided the hardest test case we will test against in `test/word_ladder/word_ladder_test_benchmark.cpp`.
If you can successfully pass this test within 15 seconds **ON THE CSE MACHINE** then we are providing you a guarantee that you will not fail those tests from a timeout. If you exceed 15 seconds for that test, you may still get the marks, but it's just something we can't guarantee.

Please note: This benchmark test is VERY difficult. If your code takes an hour to run this test that's not something that should make you panic. It's quite easy to fall in the time limit for most of the tests. Don't stress and just make sure earlier tests fall in the 15 second time limit on CSE machines.

To measure your performance:

1. In VSCode, down the very bottom of the window, change your Cmake from `[Debug]` to `[Release]`. This will remove debug symbols and other things, which will mean your code runs faster, but is near impossible to debug for mere mortals. This is usually what is done when you're finished developing and ready for release. We will discuss this more in depth later.

2. In VSCode, press `Ctrl+Shift+P` and select `Build Taret`. Once the next textbox comes up, type `benchmark` and select that test to build
3. In your project directory (the one that has the README.md file in it) run `bash benchmark`. It will output something like this below. The time you should measure yourself against is the "user" time. If this is under 15 seconds then you're all good.

```
real    1m45.089s
user    1m44.497s
sys     0m0.560s
```

4. In VSCode, down the very bottom of the window, change your Cmake from `[Release]` to `[Debug]`. Now that you're done doing a sanity check benchmark, leave debug symbols on so that you can more effectively debug your code.

Compiling with debugging_main.cpp

Two important notes here are:

- You will have to add another file to the "LINK" part in `CMakeLists.txt` to have this compile with `lexicon.cpp`
- When loading the lexicon file, use the path `./test/word_ladder/english.txt`. This is because this is the path that the executable wants if you run it as `build/source/debugging_main` etc.

Marking Criteria

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).

50%	**Correctness** 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.
	Your tests You are required to write your own tests to ensure your program works. You will write tests in the `test/` directory. At the top of each file you will also include a block comment to explain the rational and approach you took to writing tests. Please read the Catch2 tutorial 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: • Correctness — an incorrect test is worse than useless. • Coverage - your tests might be great, but if they don't cover the part that ends up

25%	failing, they weren't much good to you. • 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). • 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). At least half of the marks of this section may be awarded with the expectation that your own tests pass your own code.
20%	**C++ best practices** Your adherence to good C++ best practice in lecture. 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. We will also penalise you for standard poor practices in programming, such as having too many nested loops, poor variable naming, etc.
5%	**clang-format** In your project folder, run the following commands on all cpp/h files in the `source` and `test` directory. `$ clang-format-11 -style=file -i /path/to/file.cpp` If, for each of these files, the program outputs nothing (i.e. is linted correctly), you will receive full marks for this section (5%). A video explaining how to use clang-format can be found HERE.

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.

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.

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

PLEASE NOTE: We have a record of ALL previous submissions of this assignment submitted. If you find a solution from a friend, or online, we will find it and you will receive 0 for the assignment and potentially 0 for the course. Trust me, at least 1 person does it every term and I encourage you not to think you'll get lucky.

Submission

This assignment is due *Friday 17th of June, 19:59:59*.

Our systems automatically record the most recent push you make to your master branch. Therefore,
to "submit" your code you simply need to make sure that your master branch (on the gitlab website)
is the code that you want marked for this task.

It is your responsibiltiy to ensure that your code can be successfully demonstrated on the CSE machines (e.g. vlab)
from a fresh clone of your repository. Failure to ensure this may result in a loss of marks.

Late Submission Policy

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 0.2% up to 120 hours late, after which it will receive 0.

For example if an assignment you submitted with a raw awarded mark of 90% was submitted 5 hours late, the late submission would have no effect (as maximum mark would be 99%).

If the same assignment was submitted 72 hours late it would be awarded
85%, the maximum mark it can achieve at that time.

This late penalty has been amended from the original specification, and you should not assume it will be the same for future assignments.

Appendix

- (note 1) You'll need to consult the lexicon to see if the transformation is a valid word in either case.

FAQ

Lexicon File Path

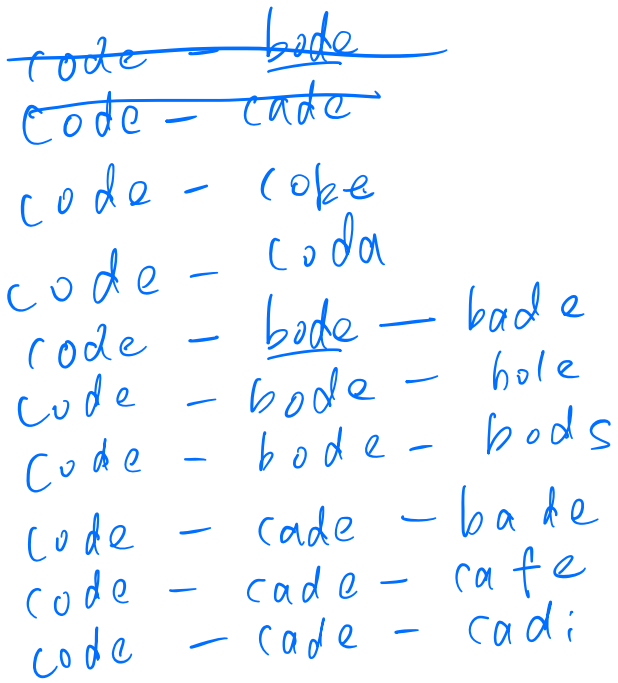
Q. Should be lexicon file path be `./test/word_ladder/english.txt` or `./english.txt`?

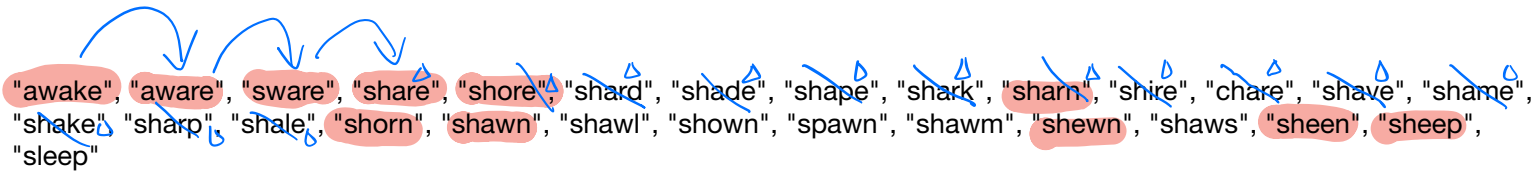
Providing paths to files can be a bit tricky in programs. Generally speaking when you provide a relative path to a program the path is relative to the directlyctory that your script/executable was invoked from (basically the directory of your shell), and NOT of the directory of the source code that is actually opening the file.

The default repository sets the lexicon file path to `./test/word_ladder/english.txt`. This will work when you invoke an executable directly such as `./build/test/word_ladder/word_ladder_test` after building it.

However, if you want to use the `Run Tests` command (which runs all the tests) that we also demonstrated in `tut01`, then you might want to update the file path to `./english.txt` in your test files.

Choose whichever method you prefer - we will ensure that EITHER file path will validly work with our automarkers.





q : ~~awake~~, ~~aware~~

g : Awake = aware aware : Sware

expanded_node : aware