EECS 281 – Spring 2022
Programming Project 1
Letterman Reboot
(Path Finding)

Due Tuesday, May 10 11:59 PM

Overview

The evil Spell Binder is loose, and it's up to Letterman to save us! Letterman hasn't been very active lately, and his power of changing one word into another by changing only one letter needs upgrading. Yes, in the old days he could change "tickle" into "pickle", but can this new Letterman 2.0 change "evil" into "good"? Only you can say!

Your program will be given a dictionary of words, a "beginning" word and an "ending" word, and which types of conversions you are allowed to perform (such as changing one letter to another, adding or deleting a letter, etc.). Your goal is to convert the beginning word to another word, to another word, etc., eventually leading to the ending word, making one change at a time. You must use the given rules of conversion, and only use valid words from the dictionary along the way. For example, you could change "chip" into "chop", but you couldn't change "chip" into "chiz" because the word "chiz" doesn't exist in the dictionary.

Program Input

You must help Letterman navigate through the Spell Binder's word traps. You will be given a beginning and ending word, and a dictionary to search. The beginning and ending words, and any other necessary options, will be given on the command line when the program is run.

Input file format (The Dictionary)

The program gets its dictionary from standard input (`cin`). The dictionary can be in a file, and you redirect that file to `cin` when you run the program (details later). There are two different types of dictionaries that the program needs to be compatible with: complex (`C`) and simple (`S`).

For both dictionaries, the first line will be a single character specifying the dictionary type '`C`' or '`S`'. **Unlike the output mode, which is given on the command line (see below), this is part of the file.** The second line will be a single positive integer `N` indicating the number of lines in the dictionary not counting the first line and lines that are comments (i.e. for simple dictionaries, the number of words, and for complex dictionaries, the number of word-generating lines).

We do not place a limit on the magnitude of `N` and neither should your code.

Comments may also be included in any input file. Comment lines begin with `"//"` (without quotes) in column 1, and are allowed anywhere in the file after the second line. When developing your test files, it is good practice to place a comment on line 3 describing the nature of the dictionary in the test file. Any dictionaries with noteworthy characteristics for testing purposes should also be commented. *You should discard all existing comments from the input file; do not save them in memory as part of your data structures*.

Additionally, there may be extra blank/empty lines at the end of any input file: your program should ignore them. If you see a blank line in the file, you may assume that you have hit the end.

Simple Dictionary
The first type of dictionary that your program needs to handle is the simple dictionary. This is a simple text file specifying the words in the dictionary, one word per line. Each "word" will be a sequence of alphabetic characters.

Each word in the dictionary is unique; there will never be two copies of the same word.

Here is a valid input file:
```
S
10
// Just a short example dictionary.  Although these words
// are in alphabetical order, that is not required.
chip
chop
junk
leet
let
shin
ship
shop
shot
stop
```

Complex Dictionary
The second type of dictionary that your program needs to handle is a complex dictionary. Like the simple dictionary, there will be one string per line. However, in this dictionary, each line could be a simple alphabetic string, like the simple dictionary, or it could contain special characters. If a line contains special characters, then it will be used to generate alphabetic words that are a part of the dictionary. **Each line will contain at most one special character** (except in the

case of insert-each, where a pair of square brackets counts as one special character).

Here are the special characters that may be included:

- **Reversal (&)**: If an ampersand appears at the end of the word, then both the word and the reversal of the word are generated, in that order. An ampersand will not appear in the middle of a word.
  - Example: “desserts&” - “desserts” and “stressed” are generated, in that order
- **Insert-each (`[]`):** If a set of characters appears inside square brackets, each character is inserted into the word, generating N words in the order of the letters, where N is the number of characters within the square brackets. There will not be square brackets without letters within them and there will not be duplicate letters.
  - Example: “tre[an]d” - “tread” and “trend” are generated, in that order
  - Example: “c[auo]t” - “cat”, “cut”, and “cot” are generated, in that order
- **Swap (`!`):** If an exclamation point appears after two characters, then the original string and the string with the two previous characters swapped are generated, in that order. An exclamation point will only occur if at least two characters precede it.
  - Example: “bar!d” - “bard” and “brad” are generated
- **Double (`?`):** If a question mark appears after one character, then the original string and the string with the one previous character doubled are generated, in that order. A question mark will only occur if at least one character precedes it.
  - Example: “le?t” - “let” and “leet” are generated

Here is an example complex dictionary, with words similar to the previous simple dictionary:

```
C
7
//This generates the dictionary:
//chip, chop, junk, star, tsar, ship, shop,
//shot, stop, pots, let, leet
ch[io]p
junk
st!ar
sh[io]p
shot
stop&
le?t
```

Morph Modes

There are a few ways that Letterman can convert words to other words.

- **Change**: Letterman can change a single letter of a word
  - Example: he can turn "pun" into "fun"
- **Swap:** Letterman can swap any single pair of adjacent letters
  - Example: he can turn "brad" into "bard"

- **Insert:** Letterman can add a letter
  - Example: he can turn "stun" into "stunt"
- **Delete:** Letterman can remove a letter
  - Example: he can turn "boar" into "bar"

The modifications that Letterman is allowed to make will be determined by arguments on the command line.

Command line arguments

Your program should take the following case-sensitive command line options (when we say a switch is "set", it means that it appears on the command line when you call the program):

- **`--stack`, `-s`**: If this switch is set, use the stack-based routing scheme.
- **`--queue`, `-q`:** If this switch is set, use the queue-based routing scheme.
- **`--change`, `-c`:** If this switch is set, Letterman is allowed to change one letter into another.
- **`--swap`, `-p`:** If this switch is set, Letterman is allowed to swap any two adjacent characters.
- **`--length`, `-l`:** If this switch is set, Letterman is allowed to modify the length of a word, by inserting or deleting a single letter.
- **`--output (W|M)`, `-o (W|M)`:** Indicates the output file format by following the flag with a `W` (word format) or `M` (modification format). If the --output option is not specified, default to word output format (W). If --output is specified on the command line, the argument (either `W` or `M`) to it is required. See the examples below regarding use.
- **`--begin <word>`, `-b <word>`:** This specifies the word that Letterman starts with. This flag must be specified on the command line, and when it is specified a word must follow it.
- **`--end <word>`, `-e <word>`:** This specifies the word that Letterman must reach. This flag must be specified on the command line, and when it is specified a word must follow it.
- **`--help`, `-h`:** If this switch is set, the program should print a brief help message which describes what the program does and what each of the flags are. The program should then `exit(0)` or `return 0` from `main()`.

When we say --stack, or -s, we mean that calling the program with --stack does the same thing as calling the program with -s. See **`getopt_long()`** for how to do this.

Legal command line arguments must include exactly one of --stack or --queue (or their respective short forms -s or -q). If none are specified or more than one is specified, the program should print an informative message to standard error (`cerr`) and call `exit(1)`. A legal command line must specify at least one of --change, --length, and --swap (or their respective short forms).

Examples of legal command lines:

- `./letter --stack -b ship -e shot --length < infile`
    - This will run the program using a the stack algorithm and word output mode. The only modifications allowed on words are inserting/deleting letters, NOT changing one letter into another. The file "infile" on disk is redirected to `cin`.
- `./letter -b ship -e shot -c --queue --output W < infile | more`
    - This will run the program using the queue algorithm, word output mode, and letters can only be changed into other letters.
    - Output is sent to the `more` program; press the spacebar after each page.
- `./letter --stack --output M -b ship -e shot --length --change < infile > outfile`
    - This will run the program using the stack algorithm, modification output mode, and letters can be changed, inserted, or deleted.
    - Output is saved on disk in "outfile", destroying that file if it already exists.

Examples of illegal command lines:

- `./letter --queue -s -b ship -e shot -c < infile > outfile`
    - Contradictory choice of routing
- `./letter -b ship -e shot -c < infile | more`
    - You must specify either stack or queue
- `./letter -s -b ship -e shot < infile > outfile`
    - You must specify at least one of change, length, and swap.

Routing schemes

You are to develop two routing schemes to help Letterman get from the beginning word to the ending word:

- A queue-based routing scheme
- A stack-based routing scheme

In the routing scheme use a data structure (queue or stack, or better yet a deque) of words to check, which we will refer to as the "search container". First, initialize the algorithm by adding the beginning word into the search container. Mark this word as already discovered. Then loop through the following steps:

1. Remove the next word from the search container: this becomes the "current" word.
2. Investigate the current word: add all words to the search container that are sufficiently similar to (as defined by the command line) the current word that are available (not already discovered). **Add any such words in the following order: beginning of dictionary to the end. Do not add words that have already been discovered. Mark each word added to the search container as discovered.**
3. As you add these words to the search container, check to see if any of them is the ending word; if so, stop; else go back to step 1.

If the search container becomes empty before you reach the ending word, the search has failed and there is no series of words to foil the Spell Binder's evil plan.

We use different terminology carefully here, and will try to do so in office hours. "Discovered" is when a word is added to the search container, "investigated" is when that word leaves the search container to become the current word, and you check for words that are similar to it. Since every word can be discovered at most once, it can be investigated at most once. Some words might be discovered but never investigated (they were still in the search container when the ending word was discovered).

Output file format

The program will write its output to standard output (`cout`), and there are two possible output formats. The output format will be specified through a command line option '--output', or '-o', which will be followed by an argument of `W` or `M` (`W` for word output and `M` for modification output). See the section on command line arguments below for more details.

For both output formats, you will show the path of words you took from start to finish. In both cases you should first print the number of words in the morph, and include the beginning word (see examples below).

<u>Word output mode (W):</u>

For this output mode, you should print each word. Starting at the beginning word, print the words in the morph until you reach the ending word.

For both the simple and complex dictionary sample inputs specified earlier, using the queue-based routing scheme and word (`W`) style output and trying to change "chip" into "stop", you should produce the following output:

```
Words in morph: 4
chip
chop
shop
stop
```

Using the same input file but with the stack-based routing scheme, you should produce the following output:

```
Words in morph: 4
chip
ship
shop
```

```
stop
```

Modification output mode (M):

For this output mode, instead of printing each word, each line of output should be how to change from one word to the next. The beginning word should always be displayed. The modification lines which follow the beginning word have one of the following forms:

- `c,<position>,<letter>`
- `i,<position>,<letter>`
- `d,<position>`
- `s,<position>`

These four forms correspond to a letter changing (c), a letter being inserted (i),a letter being deleted (d), or two letters being swapped (s). The <position> always indicates the index of the modification (for swaps, the index of the first letter swapped), and the <letter> is the new letter (either changed to or inserted).

If there is one change, but two possible places for the index, always specify where the first difference occurs. For example, if "put" changed into "putt", the characters at positions 0, 1, and 2 are the same, the new letter occurs at index 3. Similarly for changing "putt" into "put", the deletion occurs at index 3 (because indices 0 through 2 are the same characters).

The following are examples of correct output format in (`M`) modification mode that reflect the same solution as the word output format above.

For the queue solution:

```
Words in morph: 4
chip
c,2,o
c,0,s
c,1,t
```

In the above output, it is saying to begin with the word "chip", change the letter at index 2 into o (producing chop), then change the letter at index 0 into s (producing shop), then change the letter at index 1 into t (producing stop).

For the stack solution:

```
Words in morph: 4
chip
c,0,s
c,2,o
c,1,t
```

There is only one acceptable solution per routing scheme for each dictionary and begin/end word pair. If no valid morphing exists (such as trying to change "ship" into "junk"), the program should simply display "`No solution, X words discovered.`" (without quotes) on a line by itself, instead of the "`Words in morph`" line. The `X` represents the number of words in the dictionary that were part of the search. A word has been "discovered" if it was ever added to the search container. For example, if you were trying to change "ship" into "junk", simple dictionary with change mode on, the output would read "`No solution, 7 words discovered.`". This output will be the same for either word or morph output mode, and for either stack or queue mode.

Errors you must check for

A small portion of your grade will be based on error checking. You must check for the following errors:

- More or less than one --stack/-s or --queue/-q on the command line.
- No --change/-c, --length/-l, or --swap/-p on the command line.
- The --output/-o flag is followed by an invalid string.
- Either the begin or end word is not specified, or does not exist in the dictionary.
- The --change/-c and/or --swap/-p flags are specified, but --length/-l is not, and the begin/end words do not match in length (creating an impossible situation).

In all of these cases, print an informative error message to standard error (`cerr`) and call `exit(1)`. Read the file `Error_messages.txt` for standard error messages and why to use them.

You do not need to check for any other errors.

Assumptions you may make

- The first line of the dictionary will contain either a capital letter `C` or `S` and that letter will correctly reflect the dictionary type.
- The second line of the dictionary will contain a number, and the number of words in the file will match that number (the file will contain that many words/word generating lines).
- Comments will begin with `//` as the first two characters on a line, and can have any number of words on that line. No comment will appear before the number of words on line 2.
- The number of words on line 2 does NOT include comment lines.
- A dictionary will not contain any duplicate words.
- Other than comment lines, the dictionary will have one word per line (thus words will never contain blank spaces).
- For complex dictionaries, special characters will not appear incorrectly (for example, the

reversal symbol will not appear in the middle of a word).
- You do not have to consider upper- and lower-case versions of words to be the same (for example “a” and “A” are different words).

Test Files

It is extremely frustrating to turn in code that you are "certain" is functional and then receive half credit. We will be grading for correctness primarily by running your program on a number of test cases. If you have a single bug that causes many of the test cases to fail, you might get a very low score on the project *even though you completed 95% of the work.* Most of your grade will come from correctness testing. Therefore, it is imperative that you test your code thoroughly. To help you do this we will require that you write and submit a suite of test files that thoroughly test this project.

Your test files will be used to test a set of buggy solutions to the project. Part of your grade will be based on how many of the bugs are exposed by your test files. We say a bug is *exposed* by a test file if the test file causes *our* buggy solution to produce different output from our correct solution.

Each test file should be a dictionary input file. Each test file file should be named *test-n-begin-end-flags.txt*, where 0 < n <= 15 for each test file. The *begin* and *end* indicate the begin/end words, and the *flags* portion should include a combination of letters which correspond to command line arguments. Valid letters in the *flags* portion of the filename are:

- `s`: Run stack mode
- `q`: Run queue mode
- `c`: Run in change mode
- `l`: Run in length mode
- `p`: Run in swap mode
- `w`: Produce word output
- `m`: Produce modification output

The *flags* that you specify as part of your test filename should allow us to produce a valid command line. For instance, don’t include both `s` and `q`, but include one of them; include at least one of `c`, `l` and `p`; include at most one of `w` or `m`, but if you leave it off, we’ll run in word output mode. For example, a valid test file might be named `test-1-ship-shot-scw.txt` (change from ship to shot, stack mode, change mode, word output). Given this test file name, we would run your program with a command line similar to the following (we might use long or short options, such as --change instead of -c):

```
./letter --begin ship -e shot --stack -c -o W < test-1-ship-shot-scw.txt > test-1-out.txt
```

Each dictionary may contain no more than 20 words. You may submit up to 15 test files (though

it is possible to get full credit with fewer test files). The dictionaries the autograder runs with your solution are **NOT** limited to 20 words; your solution should not impose any size limits (as long as sufficient system memory is available). Your complex dictionary might start with 20 words but produce more; that is still a valid dictionary.

Input and Output Redirection

We are using input and output redirection in some of the above examples. While we are reading our input from a file and sending our output to another file in this case, we are <u>NOT</u> using file streams! The < *redirects* the file specified by the next command line argument to be the standard input (`stdin/cin`) for the program. This is much easier than retyping the dictionary every time you run the program! The > redirects the output (to `stdout/cout`) of the program to be printed to the file specified by the next command line argument. The | *pipes* the output of your program to the input of the command that follows, such as more (which displays with page breaks). The operating system makes calls to `cin` to read the input file and it makes calls to `cout` to write to the output file. Come to office hours if this is confusing!

Runtime

The program must run to completion within 35 seconds of total CPU time (user + system). This is more time than you should need. See the **`time`** manpage for more information (this can be done in Unix by entering “man time” to the command line). We may test your program on very large dictionaries (up to several hundred thousand words). Be sure you are able to navigate to the end word in large dictionaries within 35 seconds. Smaller dictionaries should run MUCH faster.

Libraries and Restrictions

Unless otherwise stated, you are allowed and <u>encouraged</u> to use all parts of the C++ STL and the other standard header files for this project. You are **not** allowed to use other libraries (eg: boost, pthread, etc). You are **not** allowed to use the C++ smart pointers (shared or unique), or the C++11 regular expressions library (it is not fully implemented in the version of gcc used by the autograder) or the thread/atomics libraries (it spoils runtime measurements).

Submission to the Autograder

Do all of your work (with all needed files, as well as test files) in some directory other than your home directory. This will be your "submit directory". Before you turn in your code, be sure that:
- Every source code and header file contains the following project identifier in a comment at the top of the file:
  `// Project Identifier: 50EB44D3F029ED934858FFFCEAC3547C68768FC9`
- The `Makefile` must also have this identifier (in the first TODO block).