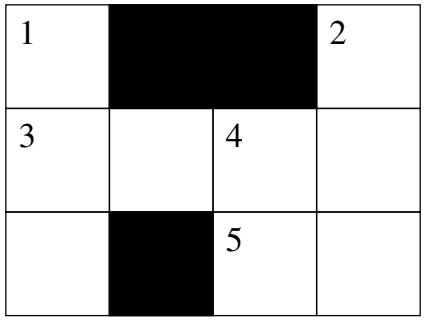
CSCI-1200 Data Structures — Fall 2022
Homework 1 — Crossword Checker

Before starting this homework, make sure you have read and understood the Collaboration Policy & Academic Integrity statement and completed the associated quiz on Submitty.

In this homework you will work with command line arguments, file input and output, and the C++ STL `string` and `vector` classes to check the words in a classic crossword puzzle board. Please read the entire handout for the assignment before starting to program. In addition to the lecture notes, you will also want to refer to the "Helpful C++ Programming Information" and "Good Programming Practices" sections of the course webpage.

Here's a very small example of a classic crossword puzzle. The board consists of a grid of squares with some of the squares blacked out. The remaining white squares need to be filled with 1 letter per square forming *words* of 2 or more letters that run in the horizontal (left to right) and vertical (top to bottom) directions. The traditional crossword puzzle provides clues/definitions for each of these words. Each white square that is the first letter of one or more words is numbered. White squares that are only the middle or end of words are not numbered. Can you solve this crossword puzzle by hand?



ACROSS
3 lacking intensity of color
5 a published statement informing the public of a matter of general interest

DOWN
1 a member of the primate order
2 a place to sleep
4 a note to follow so

Your task for this assignment is to write a tool to help confirm that a filled in crossword puzzle contains real English words by picking out the letters from the board and checking them against a list of allowed words. This tool could be useful both to crossword puzzle solvers and also to crossword puzzle designers.

NOTE: We will not be tackling the much harder problem of matching words to clues or generating clues for words, which would requires a large vocabulary, clever use of puns, knowledge of pop culture, etc. and is an active area of Artificial Intelligence (AI) and Natural Language Processing (NLP) research.

Command Line Arguments

To check the validity of a filled-in crossword puzzle board, your program will expect two required arguments and one optional argument. The first argument is the puzzle filename containing the grid of characters in the proposed board. The second argument is the dictionary filename containing all of the allowed words. If provided, the third argument will specify how to display the puzzle using ASCII art. Here are a few example command lines:

```
./crossword_checker.out puzzle6.txt linux_dict_words.txt
./crossword_checker.out puzzle6.txt linux_dict_words.txt --print
./crossword_checker.out puzzle6.txt linux_dict_words.txt --print_coordinates
./crossword_checker.out puzzle6.txt linux_dict_words.txt --print_numbered
```

You must exactly follow the specifications for the command line and output to ensure you receive full credit for your work. We provide sample input and output files on the course website, and the automated testing and autograding on Submitty will also help you check your work. We recommend starting with the lower numbered puzzles first, and working to higher numbered puzzles as you debug your work.

The file `puzzle6.txt` contains the filled-in crossword puzzle for the example above. The black squares of the board are represented with the number character, '`#`'.

```
a##b
pale
e#ad
```

The file of allowed words used in the example command lines is the contents of the Ubuntu Linux American English dictionary – a version of this file may already be on your computer in `/usr/share/dict/words`. This file is simple plaintext with one word per line. You are encouraged to copy and add or remove words from this file or create a new file with alternate words as you test your program. Note that the allowed words file may contain words with lowercase and uppercase letters. Crossword puzzles are usually case *insensitive*, so your program should ignore capitalization when checking if a word is on the accepted list. The allowed words file may include words containing punctuation characters – you should ignore those words.

Note on Error Checking

You should implement simple error checking to ensure that the arguments provided are appropriate. You should also check to make sure that the files exist and your program can successfully open and read the contents. Your program should exit gracefully with a useful error message sent to `std::cerr`, Standard Error (STDERR), if there is a problem with the arguments or the filenames.

Note on Viewing ASCII Art & Plaintext Files

Make sure you're using a good file viewer/editor to look at these files. It should correctly display the UNIX/GNU Linux '`\n`' line ending. Use one of the "Plaintext & Code Viewers/Editors" listed on the "C++ Development" page. Don't attempt use the Windows line ending character '`\m`' or '`\r`' because this will fail validation tests on Submitty.

Basic Output

If all of the horizontal or vertical sequences of 2 or more contiguous white box letters are present in the acceptable words file and the optional third argument is not provided, then your program should print this simple success message to `std::cout`, Standard Output (STDOUT):

```
valid crossword puzzle
```

If one or more of the letter sequences is not present in the acceptable words file, then your program should print all of those non-words to `std::cout`, Standard Output (STDOUT). The lines of your output may be in a different order, but the output otherwise must match exactly. Here is the expected output for the provided input file `puzzle2.txt`:

```
'abcd' is not a word
'aei' is not a word
'bfj' is not a word
'cgk' is not a word
'dhl' is not a word
'efgh' is not a word
'ijkl' is not a word
```

ASCII Art Output

If a third argument is specified, you will print an ASCII art representation of the empty puzzle board. The output for the `--print` option is shown on the right. For full credit, your output must match this sample output exactly.

```
+----+----+----+----+
|    |####|####|    |
|    |####|####|    |
+----+----+----+----+
|    |    |    |    |
|    |    |    |    |
+----+----+----+----+
|    |####|    |    |
|    |####|    |    |
+----+----+----+----+
```

```
+----+----+----+----+
|    |####|####|    |
|    |####|####|    |
+----+----+----+----+
|    |    |    |    |
|    |    |    |    |
+----+----+----+----+
|    |####|    |    |
|    |####|    |    |
+----+----+----+----+

(1,0) ACROSS pale
(2,2) ACROSS ad
(0,0) DOWN ape
(0,3) DOWN bed
(1,2) DOWN la
```

The next step is to prepare a template for the list of word clues. When the `--print_coordinates` option is specified each clue is listed with the position (row and column) of the starting letter of the word and the word direction. This output is shown to the left. Note that the upper left corner of the puzzle grid is position (0,0). It's ok if your words appear in a different order than this sample output, but the output must otherwise match exactly.

This output can be used by a crossword puzzle designer, who would take this output and replace the words with clever clues (something probably still best done by humans).

Extra Credit: Numbering the Start of Each Word

The final (optional) step for this assignment is to use the traditional crossword puzzle numbering scheme instead of coordinates. The numbers should both be placed in the puzzle grid and used in the word clues list. The output when the user specifies "`--print_numbered`" mode on the command line is shown on the right:

```
+----+----+----+----+
|1   |####|####|2   |
|    |####|####|    |
+----+----+----+----+
|3   |    |4   |    |
|    |    |    |    |
+----+----+----+----+
|    |####|5   |    |
|    |####|    |    |
+----+----+----+----+

ACROSS
3 pale
5 ad

DOWN
1 ape
2 bed
4 la
```

Additional Instructions & Submission Details

You should use the C++ STL `string` and `vector` classes in your implementation. You should *not* create any new custom classes or structs in your solution – we'll practice those techniques in Homework 2.

Use good coding style when you design and implement your program. Review the "Good Programming Practices" section on the course webpage to be sure that the TAs will be able give you credit for your hard work. Organize your program into functions: don't put *all* the code in `main`! Use good variable and function names. Be sure to make up new test cases and don't forget to comment your code!

Download and fill out the provided template `README.txt` file, adding any notes you want the grader to read. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you discuss the assignment, your program design, or your debugging process, with anyone (including teaching staff for this course), please list their names in your `README.txt` file. Also list all textbook or internet references you consulted in your `README.txt`.