# Program 5: The Scrambler

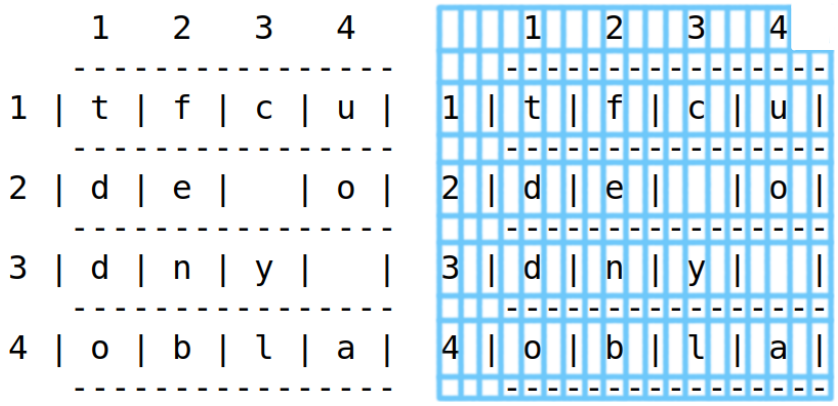
## Contents

# 1 Introduction

We will write a program that plays a game called "Scrambler". The goal of the game is to de-scramble words, presented on a grid. The player is limited in that they can only shift one row or column of the grid at a time. For this project we will design more classes, manage some memory (depending on your approach), and manage a larger project with separate source files. There is no starter code provided this time.

# 2 The Board

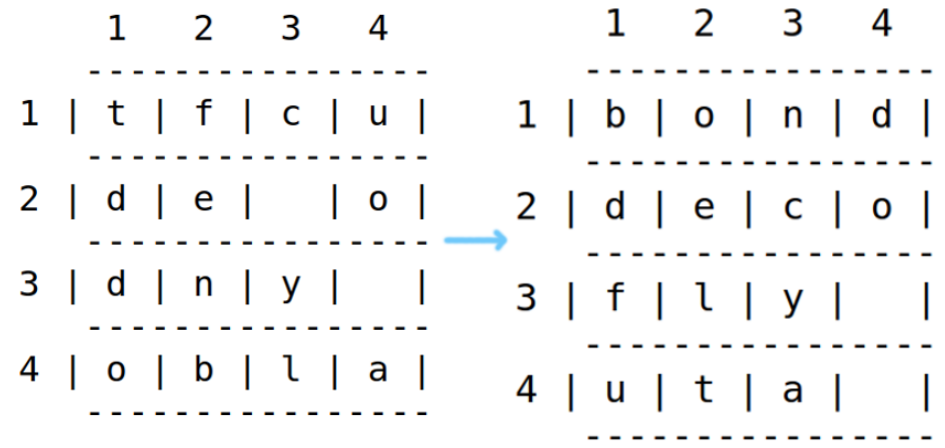
We will start by discussing the board, because displaying and modifying the board is central to this program. Below is an example of what the board will look like when displaying it to the user, on the right is the same board but with guidelines drawn to help you see the individual characters. Note the board below is in the start state.



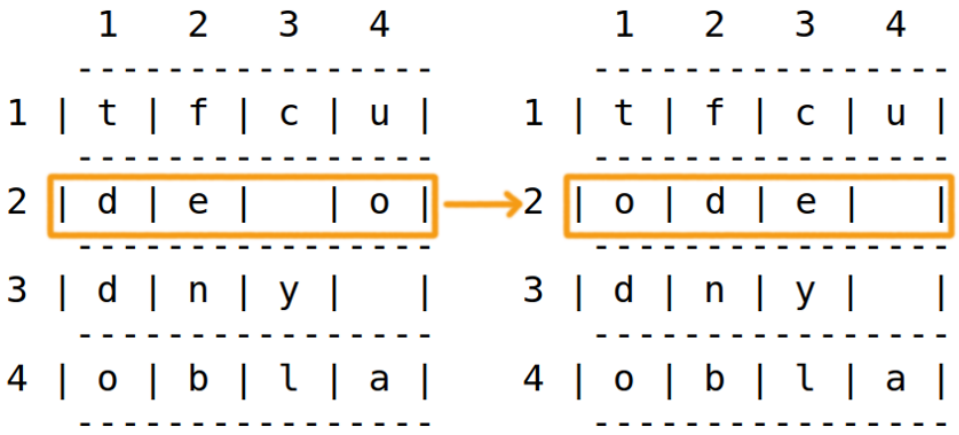

The board is a square grid, in this case 4x4, and each element in the grid holds a single character. In the example above, the first row contains the characters 't', 'f', 'c', 'u' and the first column contains the characters 't', 'd', 'd', 'o'. To begin designing the program, and deciding how to store the grid in memory, you will need to know how the user plays the game.

# 3 Rules Of The Game

The goal of the game is to rearrange the elements of the grid to the reveal the hidden words. Below is the same board from Section 2, in the original state (left) and in a solved state (right).




The player is restricted to only moving one row, or column at a time, with a wrapping mechanic. For example, the player might chose to move the second row to the right.



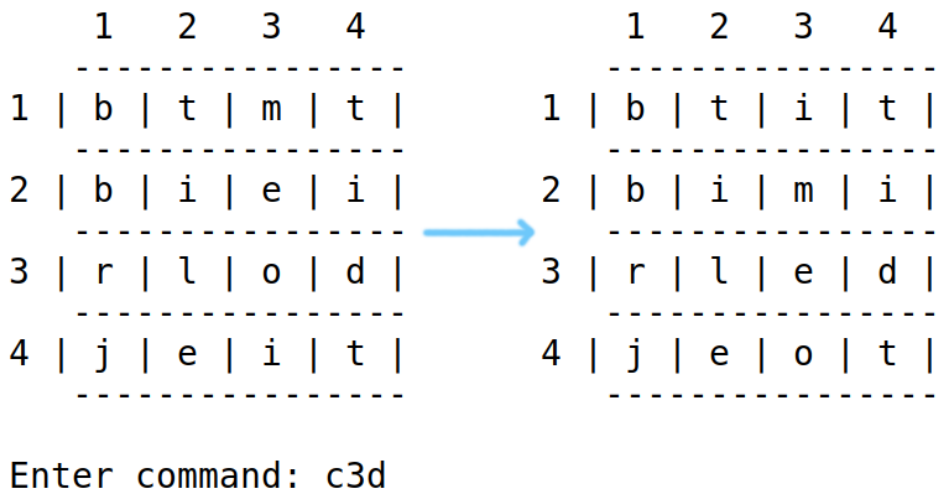

Notice how the character 'o' moved from the last column to the first, and the rest simply shifted to the right by one column. The player is also allowed to shift an entire column. Columns can be moved up or down, as opposed to right or left for rows.

# 4 How The Player Will Enter Commands

When the user wants to move a certain row or column, we need to have a way for them to communicate that to our program. We will use a simple 3 character command for this purpose, because there are only 3 pieces of information we need. Those are ...

- row or column by typing `r` or `c`
- a number indicating which row or column to select
- the direction to shift, `l` or `r` (left right) for rows – `u` or `d` (up down) for columns.

Below is an example of how that will work in your program when the user shifts the *third column down*.




# 5 Setting Up The Board With Random Words

Now that you have seen how to play the game in Sections 3 and 4, we need to discuss an important detail about setting up the board. The words on the grid will be chosen randomly, from a data set of words.

The data set will consist of words of length 3-5 characters. Here is an example of what the format will look like.

not
had
his
but
her
one
has
...

Since the board has a limited size, being an $X$ by $X$ grid, we will need to pay attention to what random words will be selected. Follow this algorithm when populating the board, or there could be issues with the autograder expecting different random words.

SCRAMBLER-INIT

```
1  for each row, starting from the top.
2      while row has more than 2 spaces remaining
3          draw a random word from list of all words.
4          if row has space for word and remaining space ! = 3
5              Insert word into row, with space in front if following another word
6  run JUMBLE-BOARD procedure
```

JUMBLE-BOARD

```
1  Randomly pick an integer x between [0, 7]
2  nmoves = x + 3
3  for i = 1 → nmoves
4        pick a random index idx
5        if i is even
6              pick 'u' or 'd' at random (50/50) = dir
7              shift column with index idx and direction dir
8        else
9              pick 'r' or 'l' at random (50/50) = dir
10             shift row with index idx and direction dir
```

# 6 Requirements

We will require certain class and function names, or else autograding wont be feasible. Therefore, to complete this assignment you need to meet the following requirements.

**Requirement**: You must have `class Scrambler`, which will be an abstraction for the game.

```
class Scrambler
{
    // members here
};
```

The autograder is going to test your code by creating objects of this class. It will call various member functions, to make changes to the board. Then it will validate those changes are made correctly, by calling other member functions. If you do not have this class defined, it is unlikely you will pass any tests.

**Requirement**: class Scrambler needs a constructor that accepts two arguments, first `string` which represents the path to the file containing *all* the words that could appear in a game, second is the size of the grid which is assumed to be $3 \leq X \leq 9$.

```
Scrambler::Scrambler(const string& path, const int size)
{
    // your code here
}
```

The autograder will create instances of your class using this constructor. It will pass, as an argument, the path to the data file. We discussed the details of the data set in Section 5. You cannot hard-code the path to the data anywhere in the Scrambler class. After construction, the board should be set with scrambled words. This is the initial board state.

**Requirement**: class Scrambler must have a member function called `str` which returns the entire board as a string. We will be using this to verify that you are updating the board correctly. So you need to define a member function like so.

```
string
Scrambler::str() const
{
    // your code here
}
```

The autograder will test that your board is being updated correctly by checking the returned string. It is expecting to see the grid, as shown in Section 2, including the numbers indicating row and column number. Use the guidelines we added, for the grid on the right, to make sure you are adding whitespace correctly. You can assume that the board will never be larger than $9 \times 9$, so you don't have to worry about how two digit column numbers affect the margins. Your string should not end in a newline character.

**Requirement**: class Scrambler needs a member function called `try_move` which accepts a single `string` as an argument, and that string represents the command the user gave. Some examples of this could be *c2d*, *r1r*, or *c3u*. The user can also type $R$ or $r$ to restart the game from the intial jumbled board state.

```
void
Scrambler::try_move(const string& cmd)
{
    // your code here
}
```

The autograder will call this member function with commands that the user would type. It needs to be *case-insensitive*, and *validate* that the user gave a valid command. If `cmd` is not valid, then `try_move` should have no affect on the board.

**Requirement**: class Scrambler needs a member function called `is_over`, which accepts no arguments and returns true or false. It will return true if the game is over, which happens if the user has won the game or quit the game.

```
bool
Scrambler::is_over() const
{
    // your code here
}
```

The way you win the game is by unscrambling the words, *not* by re-arranging the board into the state it was in *before* you ran the JUMBLE-BOARD procedure. What this means is that you have to check the whole board to see if the unscrambled word is present. If all words are unscrambled, then the game is won. For example, if we reference the solution board in Section 3, the word `bond` could appear in any of the rows (it just has to appear once. To make things simpler, words are only considered unscrambled if they appear horizontally. If the first column happened to be 'b' 'o' 'n' 'd' then it is not

considered unscrambled.

**Requirement**: class Scrambler needs a member function `display_solution`, which accepts no arguments and returns a string showing the solution to the game. This is the solution *from the start*, not from the current state of the board, by basically reversing the JUMBLE-BOARD algorithm.

```
string
Scrambler::display_solution() const
{
    // your code here
}
```

This is a source of trouble for the autograder, because the output is long. We describe the details of what the output should look like in Section 10. Note that this needs to be output correctly or the autograder might not be able to run tests, where it needs to know the solution beforehand.

**Requirement**: class Scrambler needs a member function `get_words` which returns a vector of strings that contain all the words that are present on the jumbled board.

```
vector<string>
Scrambler::get_words() const
{
    // your code here
}
```

The autograder will need to know the words you have on the board, because they are jumbled at game start. That way it can confirm that your `display_solution` and `is_over` work correctly.

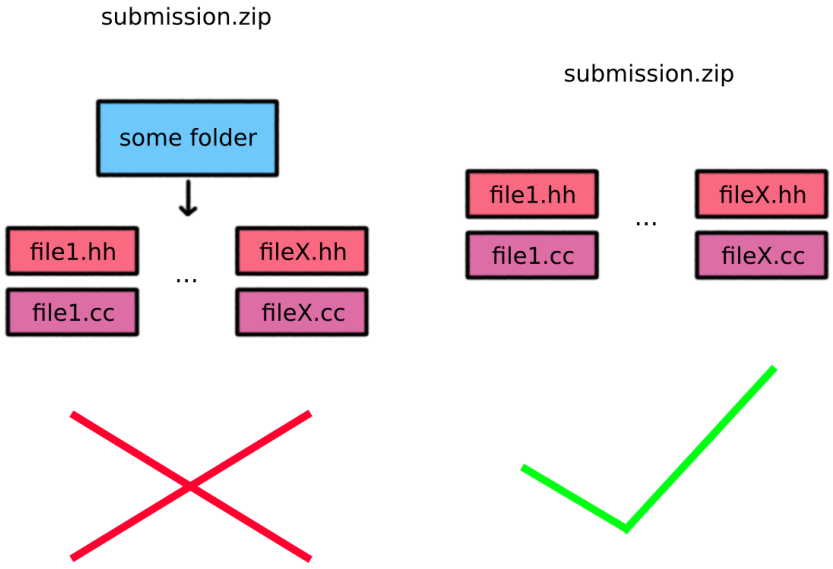
# 7 Allowances

We set up the requirements in Section 6 because of limitations in the autograder, but we also want to minimize the number of requirements so that you have some freedom in designing your code. While we require `class Scrambler`, we encourage you to create other classes to represent important objects in this program. For example, our solution makes use of a separate class for managing the board's memory and performing low-level operations on it. The Scrambler class uses an object of this separate class in its own algorithms. Below is a list of what we allow.

- you can create extra classes
- you can have more than one source file in your submission
- you can make use of strings, vectors, or manage your own memory
- you can write your own main

# 8 Multiple Source Files

We encourage you to organize your submission with multiple source files. That is a good way to keep your project clean, and easy to understand. Gradescope will allow you to submit a zip file of your project, but you must follow our format for submissions or there will be issues with compilation. Submitting your work correctly is your responsibility. The format will be a flat zip file, so do not zip a folder containing your code.




The autograder will search your submission for all source files (.cc or .cpp), compile them separately, and finally link them together with our test cases into one executable. It is important to point out that **your main function is ignored**. The autograder will use it's own main to test your code.

# 9 How to Submit

Like previous homework assignments, you will submit your work to Gradescope. Navigate to the Assignments tab and you will find an assignment called *Program 5*. There will be an autograder to give you fast feedback, but keep in mind that it is not a replacement for your own testing. It is not the autograder's job to tell you what is wrong with your code, but at the very least it should tell you what input caused it to deduct points. We will try to program it to be robust, but keep in mind that if your program has infinite loops or causes premature crashes (on certain inputs) then you might get no feedback at all because the autograder crashed with your program. At the time of release of this assignment, the autograder might not be available. It is our intention to have it available with enough time before the deadline. The link to submit will be available with the autograder.

# 10 Display Solution

The function `display_solution` should return a string, such that if we printed it to the player, they could see how to solve the game from the initial scrambled board. We will keep the format as simple as possible, to minimize issues with the autograder. The string should contain a sequence of boards, and between those boards will be a short code indicating what move was done to transition from one board to another. See the example on the right, which shows the initial board, then prints one move, then shows the board afterwards.

The line `*** Move 1 (r, 1, r)` represents moving the first row to the right. Note that the newline should be after the `)`. If there is another command needed, to get to the solution, another `*** Move ...` line would be printed after the second board and then we would display the board again. As a final note, the entire string should not end in a newline character.

```
    1   2   3   4   5
  ---------------------
1 | a | p |   | y | p |
  ---------------------
2 | c | r | a | a | w |
  ---------------------
3 | s | s | o | s | s |
  ---------------------
4 | p | o | n | a | k |
  ---------------------
5 | o | w | a | y | n |
  ---------------------
*** Move 1 (r,1,r)
    1   2   3   4   5
  ---------------------
1 | p | a | p |   | y |
  ---------------------
2 | c | r | a | a | w |
  ---------------------
3 | s | s | o | s | s |
  ---------------------
4 | p | o | n | a | k |
  ---------------------
5 | o | w | a | y | n |
  ---------------------
```