CSCI103 Programming Midterm: Word Search Game

April 9, 2020

1 Introduction

In this exam you will complete the word search game that we had introduced as the last problem of the written midterm exam. However, now you will implement a complete stand alone program that can create a game and permit a user to play interactively.

Let's recall what a word search puzzle is. A **word search puzzle** is a 2-D array of characters. Your program will permit users to create a word search puzzle by giving a list of words and the positions where they will be placed in the puzzle in an input file. Once a 2-D array word search puzzle is created, the program will take input words from users to search the puzzle for the words until the user decides to quit by entering Ctrl-D. In the next section, we will guide you through the main driver program in the file, **main.cpp**, including what must be fixed in it. The remaining sections will discuss the functions that are declared in the file **wordsearch.h** that you need to implement in the file **wordsearch.cpp** to have a successfully working program. We have designed the exam so that you can complete the exam in the order given and do not have to worry if you have problems with earlier functions. All functions are tested in a way that only tests one function at a time even if that function relies on other functions.

2 The Driver Program: main()

We have included the code in the main function in the file, **main.cpp** below, so that we can discuss what the functions are that you need to write to have a completely successful program and what code you need to add to main().

```
int main(int argc, char const *argv[])
{
    if(argc < 4){
        cout << "Usage: wordsearch <random seed> <puzzle size>";
        cout << "<file to load words>" << endl;
        return 1;
    }
```

```
    // Use the second argument, random seed, as random seed for this program
    srand(atoi(argv[1]));
    // Set the size of the puzzle to be a <puzzle size>
    // where n = <puzzle size>  for an n by n 2-D array
    int n = atoi(argv[2]);

// Create a new empty, uninitialized puzzle
    char** my_puzzle = new_puzzle(n);
    // Initialize the puzzle to random characters
    random_puzzle(my_puzzle,n);
    // Using the <file to load words> given as an argument,
    // load each word in the file into the puzzle at given location
    // in puzzle specified in file.
    // No error checking is fine
    load_words(my_puzzle, n, argv[3]);
    // Print the puzzle
    print_puzzle(my_puzzle,n);

// Prompt the User to enter words until the user
// enters Ctrl-D. When the user enters a word,
// search the puzzle for the word. If found, tell
// user where word is found. Otherwise, tell user
// word not found.
    string look;
    cout << "Enter a word: " << endl;
    while(cin >> look)
    {
        int x=0;
        int y=0;
        char d='X';
        if(search_word(my_puzzle, n, look, &x, &y, &d))
        {
            cout << "Word " << look << " found @ " << x << " " << y << endl;
        }
        else
        {
            cout << "Word " << look << " not found." << endl;
        }
    }
    /* add code to delete dynamic memory*/
    return 0;
}
```

You must add code to main() to prevent memory leaks by deleting any dynamic memory you had allocated to create the puzzle.

You must implement the following eight functions declared in **wordsearch.h** for main() to work:

1. char random_letter();
2. char** new_puzzle(int);

3. void random_puzzle(char**, int);
4. void print_puzzle(char**, int);
5. bool place_word(char**, int, string, char, int, int);
6. bool load_words(char**, int, const char*);
7. bool find_word(char**, int, string, char, int, int);
8. bool search_word(char**, int, string, int *, int*, char*);

These functions are all described in the following sections and you can use this as a checklist of your progress.

3 Creating and randomly initializing the puzzle

To create a puzzle, you will need to allocate a 2-D array of size n by n where n is the third parameter of the program. To do so, you must implement the following functions in the file, **wordsearch.cpp**. The function, **new_puzzle()**, takes as its only parameter the integer dimension, n, and returns a pointer to a pointer to a char that is the address of where the 2-D array puzzle: **char** new_puzzle(int n);**. We will refer to this pointer to a pointer to a char as the 2-D array puzzle itself.

Once the 2-D array for the puzzle has been created, it should be initialized to random capital English letters. You must write two functions to complete this task. First, you must implement the function **random_letter()** that takes no parameters and returns a character that is a random capital English letter: **char random_letter();**

Then you must implement the function, **void random_puzzle()** that uses the function **random_letter()** to fill the puzzle board with random capital English letters. The function, **random_puzzle()** takes as its parameter a pointer to a pointer to a char that is the puzzle board itself and the integer dimension, n and returns nothing: **void random_puzzle(char** puzzle, int n);** For both **random_letter()** and **random_puzzle()** do not call **srand()**, assume it has already been called properly.

4 Updating puzzle from input file

In order to create a word puzzle as specified by the user, the wordsearch program takes as its fourth argument a text file, <file to load words>. The text file to load words has on its first line the number of subsequent lines and words to place in the puzzle. Then each subsequent line has the format:

row col dir word

In this format row and col are integers for the indices for the position of where to place the word in the 2-D array for the puzzle, dir is one of three characters for the direction of placement that will be discussed further below, and word is the word to place in capital English letters.

The direction, dir, can be one of three characters, {L,D,T}. If the direction character is **'L'**, the word should be placed from **left to right in row** from the starting position. If the direction character is **'T'**, the word should be placed **down the column** from the starting position. If the direction character is **'D'**, the word should be placed **down the right diagonal** from the starting position. A sample word search file follows. In this file there are 5 words to place in the puzzle. The first is the word, HELLO, to place starting at position (0,0) in the puzzle from left to right. The second is the word, TROJAN, to place starting at position (1,0) and down the right diagonal. And the third is the word, TOMMY, to place starting at position (4,6) down the column.

```
5
0 0 L HELLO
1 0 D TROJAN
4 6 T TOMMY
3 7 T HORSE
8 1 L FOOTBALL
```

You will need to complete the implementations of two functions to place the words from the given input file in the puzzle.

The function, **place_word()**, places a single word from its input parameters into the puzzle starting at the given location given as a row and column and in the correct direction as specified by the direction parameter. If the word to be placed cannot be placed from the starting position in the given direction because it would exceed the dimensions of the puzzle, the function must return false. *This function must do proper error checking not to exceed the dimensions of the 2-D array. The game does not permit words to wrap around the puzzle.* Otherwise upon successfully updating the puzzle board to include the word, the function should return true. The function, **place_word()**, takes as its parameters a pointer to a pointer to a char that is the puzzle, the integer dimension of the puzzle, n, a string that is the word to be placed from the input file, a char from the set, {L,T,D}, that is a direction to place the word in the board as described above, an integer r that is the starting row position to place the word in the puzzle and an integer c that is the starting column position to place the word in the puzzle: **bool place_word(char** puzzle, int n, string word, char dir, int r, int c);**

If the word cannot be placed in the given direction, dir, at the current location given by the row, r, and column, c, **place_word()** should print the following error message before returning false:

```
cout << "Invalid word placement: " << word << " ";
cout << dir << "@" << r << "," << c << endl;
```

Next you will implement **load_words()**. The function, **load_words()** will open the word file, reading each line and placing each word in the appropriate location in the puzzle by calling **place_word()** properly. It must parse each line in the file after the first to get the position, direction, and word to be passed to the function **place_word()**. The function, **load_words()**, will return false if the input file could not be opened, the file is malformed, or a word could not be placed in the puzzle in the position and direction specified in the input file. The function, **load_words()** takes as its parameters a pointer to a pointer to a char for the 2-D array for the puzzle, the integer n for the dimension, and a pointer to a char that is the name of the input file and returns a bool that is true if and only if all words in the input file are successfully placed in the puzzle and false otherwise: **bool load_words(char** puzzle, int n, const char* fname)**

There are four sample word files for testing distributed with the exam. These are not exhaustive. The files, bad1.txt and bad2.txt, are not formatted correctly and the function **load_words** should fail on these files and return false. The files, word.txt and big.txt, are properly formatted and **load_words()** may or may not fail depending on the size of the puzzle. The function, **load_words()** does not need to check if words in the file would overwrite each other. The function only needs to check if the words in the file will fit in the puzzle with the specified starting location and direction by properly using the return value from **place_word()**.

To print the puzzle you must implement the function, **print_puzzle()**. This function returns nothing and takes as its parameters a pointer to a pointer to a char that is the puzzle and an integer that is the dimension of the puzzle: **void print_puzzle(char**, int);** In order to format your printed puzzle, use the function **setw(3)** before printing a character in the puzzle.

5 User playing the word search game

The last part of the program is to permit the user to play the game by searching the puzzle for words. Inside the main driver program, the user will be prompted to enter a word to search for in the puzzle. Given that input word from the user, the program must search the puzzle board for the word. The user will not enter any starting position or direction. For this reason, you must implement the functions, **find_word()** and **search_word()**, to search for the given input word from the user from every possible starting position in the board in every possible direction. (No, this isn't efficient in the least, but it is correct and there's no need to optimize this during the exam although you can do so later if you'd like.)

In order to search for a specific word in the puzzle at a particular location with a given direction, you will implement the function, **find_word()**. This function is nearly identical to the last problem on the written midterm and it is fine if you start your implementation from that solution. The only differences are that *the directions use different letters, the diagonal is down to the right,*

and your code must include error checking not to exceed the dimensions of the board.

The function, **find_word()**, takes as its parameters a pointer to a pointer to a char that is the puzzle, the integer dimension of the puzzle, n, a string that is the input word to search, a char from the set, {L,T,D}, that is a direction to search, an integer r that is the starting row position in the puzzle and an integer c that is the starting column position in the puzzle. If the input word would exceed the dimensions of the puzzle in the direction or is not found in the given direction from the starting position, the function should return false. If the word is found in the given direction from the starting position, the function should return true. The function, **find_word**, has the following prototype: **bool find_word(char** puzzle, int n, string word, char dir, int r, int c);**

Finally, to search the entire puzzle you will implement the function, **search_word()**. This function will return true at the first starting location and direction in which the input word is found and false only if the word is not found after searching all starting positions in the puzzle in all directions. You will use **find_word()** along with some loops to try all possible starting positions and all possible directions. The function **search_word()** takes as its parameters a pointer to a pointer to a char that is the puzzle, the integer dimension of the puzzle, n, and a string that is the input word entered by the user and three parameters that are pointers. The parameters passed by pointers are to be used to return the location and direction in which the word is found in the puzzle. The pointer to ints x and y should be used to store the row and column positions respectively and the pointer to char should be used to store the direction. The search function has the following prototype : **bool search_word(char** puzzle, int n, string word, int *x, int* y, char* d);**

6 Building code, Testing code, and Tips

1. To build the code on the command line: type "make". When you do so, the executable file, wordsearch, will be created.

2. Usage to run the code on the command line:
   ./wordsearch <random seed> <puzzle size> <file to load words>
   Examples:
   ./wordsearch 1234 5 words.txt
   ./wordsearch 1234 10 big.txt

3. There are eight separate testing files all with the name testX.h where $1 \leq X \leq 8$ and a file to run all of the tests run_tests.sh. To run a single test on the command line: type "sh testX.h" where $1 \leq X \leq 8$.
   For example to run the first test on the command line: type "sh test1.sh"
   To run all the tests on the command line: type "sh run_tests.sh"

4. **Do not change or delete any preprocessor directives such as #defines and #ifdefs in the files!** Doing so will break the test cases.

5. **Do not delete solution.o!** Doing so will cause all tests to stop working.

6. Please remember to get rest and have some fun with this. You have plenty of time to challenge yourself and see what you can produce. Good luck!