CS 246—Assignment 1

M. Godfrey B. Lushman J. Schmitz

Fall 2024

Due Date 1: Friday, September 20, 2024, 5:00pm (30% marks)

(If you joined the course late, this due date is automatically extended to 48 hours after you join. It will show up as late on Marmoset, but submit anyway.)

Due Date 2: Friday, September 27, 2024, 5:00 pm (70% marks)

Learning objectives:

- C++ I/O (standard, file streams) and output formatting
- C++ `string`s and `stringstream`s
- Processing Command Line Arguments

Allowed headers: `iostream`, `fstream`, `sstream`, `string`

- **Questions 1, 2a, and 3a are due on Due Date 1; questions 2b and 3b are due on Due Date 2.**
- All code is to be written in C++20 and all library components must be imported into your program (using `import`). **Do not use file inclusion (i.e., `#include`).** You may **only** import the headers listed above. In particular this means you must also use C++-style I/O and memory management facilities. Any attempt to use libraries other than those listed above, or to use C facilities instead of the equivalent C++ facility, or to use file inclusion rather than `import`, will be **rejected** by Marmoset.
- On this and subsequent assignments, you will take responsibility for your own testing. This assignment is designed to get you into the habit of thinking about testing *before* you start writing your program. For each question you will be given a compiled executable program that is a program representing a solution to each question. You should use these provided executables to help you write your test cases, as they can show you the resultant output for given inputs. If you look at the deliverables and their due dates, you will notice that there is *no* C++ code due on Due Date 1. Instead, you will be asked to submit test suites for C++ programs that you will later submit by Due Date 2.

  Test suites will be in a format compatible with the `runSuite` script that you used in CS136L. We have also provided binaries for `runSuite` and `produceOutputs` in the `a1` directory compiled in the student environment if you need them.
- Design your test suites with care; they are your primary tool for verifying the correctness of your code. Note that test suite submission `zip` files are restricted to contain a maximum of 40 tests. The size of each input (`.in`) file is also restricted to 500 bytes. This is to encourage you not to combine all of your testing into one giant test. There is also a limit

for each output file (`.out`), but none of your tests should be able to create such a large output file that you would normally encounter it.

- If we do not explicitly tell you how to handle a particular type of invalid input, then you do not need to worry about that case since it is considered to be *undefined behaviour*. Such cases (invalid inputs that fall under the umbrella of undefined behaviour) **should not be submitted as part of a test suite**. This applies to all assignments and questions in the course.
- There will be a hand-marking component in this assignment, whose purpose is to ensure that you are following an appropriate standard of documentation and style, and to verify any assignment requirements not directly checked by Marmoset. Please code to a standard that you would expect from someone else if you had to maintain their code. **We will not answer Piazza questions about coding style; use your best judgment.** Further comments on coding guidelines can be found here: `https://www.student.cs.uwaterloo.ca/~cs246/F24/codingguidelines.shtml`
- We have provided sample executables under the appropriate `a1` subdirectories. **These executables have been compiled in the CS student environment and will not run anywhere else.**
- **Before asking on Piazza, see if your question can be answered by the sample executables we provide. You may ask private questions on Piazza about the programs and assignment requirements. You are not permitted to ask any public questions on Piazza about what the programs that make up the assignment are supposed to do.** A major part of this assignment involves designing test cases, and questions that ask what the programs should do in one case or another will give away potential test cases to the rest of the class. Questions found in violation of this rule will be marked private or deleted; repeat offences could be subject to discipline.

Question 1

(33% of DD1; 0% of DD2) Note: there is no coding associated with this problem. You are given a *non-empty* array $a[0..n-1]$, containing n integers. (It is an invalid test case to provide integer values that are either too large or too small to be read in, or to provide non-integer input.) The program `q1sort` sorts the array in ascending order by reading the integer values for the array from standard input, such that the first item in the input is defined to be the smallest, with the input wrapping around modularly as necessary. The output of the program is a print of the sorted array starting from the largest element, with one element printed per line.

For example, if the input is

```
4 -9 5 -1 3 10
```

then `q1sort` prints

```
4
5
10
-9
-1
3
```

Your task is not to write this program, but to design a test suite for this program. Your test suite must be such that a correct implementation of this program passes all of your tests, but a buggy implementation will fail at least one of your tests. Marmoset will use a correct implementation and several buggy implementations to evaluate your test suite in the manner just described.

Your test suite should take the form described for `runSuite` in CS136L; each test should provide its input in the file `testname.in`, and its expected output in the file `testname.out`. You have been provided with a sample implementation of the program (in compiled form). You may use this executable along with the `produceOutputs` script to generate the `testname.out` files for the `testname.in` files that you create. The collection of all `testnames` should be contained in the file `suiteq1.txt`.

a) **Due on Due Date 1**: Zip all of the `.in`, `.out`, and `suiteq1.txt` files that make up your test suite into the file `a1q1.zip`, and submit to Marmoset.

b) **Due on Due Date 2**: Nothing.

Question 2

(33% of DD1; 40% of DD2)

In this question, you will create a program to play the "word ladder" game. A word ladder has a starting word and an ending word. The player's goal is to find a chain of words that link the two, where each word differs from the previous by *exactly* one character. The shorter the chain, the better! For example a chain to go from "cold" to "warm" would be:

1. "cold"
2. "co**r**d"
3. "c**a**rd"
4. "**w**ard"
5. "warm"

This chain is of length 3 i.e. it took 3 words to go from "cold" to "warm".

Your program should take three command line arguments: the starting word, the ending word, and a filename for the `words` file (a file that contains a list of words valid to use in the game).

The lengths of both the starting and ending words must be greater than 0 and equal to each other. The starting word and the ending word must not be the same. Since these are strings, they are case-sensitive i.e. "Cat" is not the same as "cat".

Before your program begins playing the ladder game it should check to ensure that the starting word and ending word can be found as words in the `words` file. If either the starting word or ending word cannot be found, print to standard error `Error: Starting or ending word not found in words file` and return from the program with a code of 1.

Your program begins by printing `Starting Word: <the word>` to standard output. It will then continually accept words via standard input until end-of-file is detected. A word is valid if and only if it belongs to the words file, has the same length as the starting/ending words, and differs from the previous word in the ladder by exactly one character.

If a word is invalid, your program should print a message to standard error. You should print `Error: <the word> does not belong to word file`, and/or `Error: <the word> does not`

`differ from <previous word> by exactly one character`. After printing the error message, ignore the invalid word the user entered, and wait for the user to enter a valid word.

If a word is valid, your program should simply continue playing the game: accept the user supplied word, and wait for the next word in the ladder.

Your program should also be on the lookout for suboptimal play. If at any point, a validly played word would have been a validly played word at an earlier point in the game, your program will accept the word, but will also print the warning message `This word could have been played earlier` to standard error.

There is a limit of 8 steps between the starting word and ending word (excluding both the starting and ending word). If, after entering 8 words, the player cannot validly enter the ending word, the player loses, and you should display `You lose` and end the game.

When a user validly enters the end word, your program should print `Congratulations! Your Score: <score>, Best Score: <best score>` to standard output. The score is given by the 8 minus the chain length of the word ladder. Once this is printed, the user should be re-prompted with `Starting Word: <the word>` on standard output and allowed to play again. The best score (which is the highest score) is kept track of for one execution of the program. When your score is less than the best score, the best score should be updated to match your score.

For testing purposes, `/usr/share/dict/words` contains a full dictionary of words, and could be a useful `words` file. However, you may wish to include some words like CS246, C++, OOP, or eepy, or you may wish to exclude other words. Hence, it may be useful for you to create your own `words` files that only contains words that you like.

a) **Due on Due Date 1:** Submit a file called `a1q2.zip` that contains the test suite you designed, called `suiteq2.txt`, and all of the `.in`, `.out`, `.err`, `.ret`, and `.args` files, as well as any word files used in your tests.

b) **Due on Due Date 2**: Write the program in C++. Save your solution in a file named `a1q2.cc`.

Question 3

(33% of DD1; 60% of DD2)

In this problem, you will write a program to calculate the page width of columnar data. You will be given a series of lines of text from standard input. We will guarantee that no line will contain more than 10 words, where a word is defined as a sequence of non-whitespace characters, surrounded by any non-zero number of whitespace characters (or the beginning or end of the line). Your task is to report the number of characters a line would occupy if the words on each line were arranged into columns, with no leading or trailing whitespace, and a single space between columns. For example, if the input was as follows:

```
zero one two
three four
```

then the correct output would be `14` (5 + 1 + 4 + 1 + 3).

What was described above is the default behaviour of your program. It can be overridden with a single command-line argument denoting a number `n`, which must be at least 1 and at most 10. If a number `n` is supplied on the command-line, then this forces the output to have `n` columns, and you should do the calculation accordingly. For example, for the input given above if `n` was 1, the correct output would be `5` (because all of the words would be in one column, which would need a width of 5), and if `n` was 2, the

correct output would be `10` (the first column would contain `zero two four` and the second column would contain the other words). If `n` was 5 or greater, the correct answer would be `23`.

a) **Due on Due Date 1:** Submit a ZIP file called `a1q3.zip` that contains the test suite you designed, called `suiteq3.txt`, and all of the `.in`, `.out`, and `.args` files.

b) **Due on Due Date 2**: Write the program in C++. Save your solution in a file named `a1q3.cc`.

Submission

The following files are due at Due Date 1: `a1q1.zip`, `a1q2.zip`, `a1q3.zip`,
The following files are due at Due Date 2: `a1q2.cc`, `a1q3.cc`,