Assignment #1: Basic C++

Due Date 1: Friday, September 22, 2023, 5:00 pm EST (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 29, 2023, 5:00 pm EST (70% marks)

Learning objectives:

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

- **Questions 1, 2a, and 3a are due on Due Date 1; questions 2b and 3b are due on Due Date 2.**
- 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 eggs in one basket. 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.
- You must use the standard C++ I/O streaming and memory management (MM) facilities on this assignment; you may **not** use C-style I/O or MM. More concretely, you may `import` the following C++ libraries (and no others!) for the current assignment: `iostream`, `fstream`, `sstream`, `iomanip`, and `string`. Marmoset will be setup to **reject** submissions that use C-style I/O or MM, or libraries other than the ones specified above.
- 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/F23/codingguidelines.shtml`
- We have provided some code and 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 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.

Coding Assessment

Questions 2 and 3 are part of the coding assessment, and may be publicly discussed on Piazza so long as solutions are neither discussed nor revealed.

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. The program `sort` sorts the array in descending order by reading the integer values for the array from standard input. 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

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

then `sort` prints

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

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 (what we believe to be) a correct implementation of the `sort` program (in compiled format). 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 up all of the 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) For this question: use C++20 imports. Compile the system headers with `g++20h` and compile your program with `g++20m`.

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**ard"
4. "**w**ard"
5. "warm"

This chain is of length 3: 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).

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 the starting word or ending word cannot be found, print to cerr `Error: Starting or ending word not found in words file` and quit the program with an exit code of 1.

Your program will begin by printing `Starting Word: <the word>`. It will then continually accept words via standard input. A word is valid if and only if it belongs to the words file, and differs from the previous word in the ladder by one character.

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.

If a word is invalid, your program should print a message to `cerr`. You should print `Error: <the word> does not belong to the 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.

When a user validly enters the end word, your program should print `Congratulations! Your Score: <score>, Best Score: <best score>`. The score is given by the chain length of the word ladder. Once this is printed, the user should be reprompted with `Starting Word: <the word>` and allowed to play again. The best score (which is the lowest 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, it doesn't contain many good words like CS246, C++, OOP, or eepy. It also contains some bad words, like racket, python, shoot, and darn. 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`, 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) For this question: use #include. Compile your program with `g++20i`.

In this question we will implement an election that uses *cumulative voting*. In this voting scheme, a voter has $(X > 0)$ number of votes that they can choose to distribute among the candidates which are numbered from 1 to n, where n is at most 10 (the edge case of 0 candidates is possible and is considered valid input, however, X must be specified at such a case). The value for X may be provided as an optional positive command line argument. If a value of X is not provided as an argument, the default value of n, calculated by reading in candidate names, is used (i.e., $0 < X = n \leq 10$). Input begins with the names of candidates, one full name per line. (You may assume that there is no leading or trailing whitespace.) The first name is considered as candidate 1, the second as candidate 2, and so on. A candidate's name will never contain a numeral and consists of at least 1 and at most 15 characters (including any spaces), so you do not need to test for that. The list of candidates is followed by some number of lines where each line indicates one voter's distribution of votes, which we call a ballot. For a ballot, the i^{th} column indicates the number of votes allocated to the i^{th} candidate. A ballot is considered invalid (*spoilt*) if it does not consist of n columns or the sum of the votes in the ballot exceeds X. In addition, the votes allocated to a specific candidate within a ballot are always non-negative (≥ 0). The number of voters is unknown beforehand, but is, of course, non-negative (≥ 0). Votes are terminated by end-of-line and that the overall set of all votes from all voters is terminated by end-of-file (Ctrl+D).

As an example, given the following data ($X = 7$):

```
Victor Taylor
Denise Duncan
Kamal Ramdhan
Michael Ali
Anisa Sawh
Carol Khan
Gary Owen
3 0 1 0 0 1 2
1 1 1 1 0 1 2
1 1 1 1 1 1 1
2 1 3 1
7 0 0 0 0 0 0
1 1 1 1 1 1 2
```

your program should produce the following output:

```
Number of voters: 6
Number of valid ballots: 4
Number of spoilt ballots: 2

Candidate: Score

Victor Taylor: 12
Denise Duncan: 2
Kamal Ramdhan: 3
Michael Ali: 2
Anisa Sawh: 1
Carol Khan: 3
Gary Owen: 5
```

Note: Do not copy-paste the example above to create a test file. The formatting might NOT be correct. Use the test cases provided to you within the repository.

a) **Due on Due Date 1:** Submit a 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`,