ROB 502 FA24: Homework 2

See Canvas for due dates.

Rules:

1. All HW must be done individually, but you are encouraged to post questions on Piazza and discuss during OHs (recall: do not email assignment questions to teaching staff!)
2. Late work adheres to the late grading policy (see syllabus, or use late-day tokens)
3. Submit your code on autograder.io
4. Remember that copying-and-pasting code from other sources is not allowed

Code Download:

The code is available on the ROB 502 FA24 Git repo in the appropriate HW folder.

We prefer that you use the '`fetch`'/'`pull`' Git methods for the HWs, but we know that Git can be hard/confusing; if necessary, you can always download the code from the repo link. However, learning to use Git will be to your benefit for multiple reasons:

1. You can get some easy extra credit if you properly pull and version-control (and comment) your work for the HW assignments!
2. ROB 550 will require you to use Git in a much more complex capacity, and without much learning time (large code databases, lots of teamwork with multiple branches, etc.).
3. If you ever need to setup a new VM or use your code on another device (new computer, VM crashes, etc.) you can easily restore your code progress with Git.
4. Many robotics-related industries and research labs rely on Git for robust version-controlling of their work… you can put this on your resumé!

Open the directory `hw2` in VSCode. You can do this by `cd`-ing to the `hw2` directory and running `code .` . Do not run VSCode from subdirectories of `hw2` .

Instructions:

Each problem will give you a file with some template code, and you need to fill in the rest. We use the autograder, so if you are ever wondering "will I get full points for this?" just upload your code in the autograder to check. There is no limit to how many times you can upload to autograder. The autograder is set to ignore whitespace and blank lines, so there is some forgiveness on formatting. In most of these questions, the input/output and printing is handled for you in the template code, but in some you are asked to write it yourself. The autograder may test your problem with multiple different inputs to make sure it is correct. We will give you examples of some of these inputs, but the autograder will also try your code with some other inputs. The autograder will only show you if you got it right/wrong, so if you don't get full points, try to test some other inputs and make sure your program works.

Sample input and output are given as `input.txt` and `output.txt` respectively. Getting your program to successfully compile and correctly reproduce the sample output will get you some points, while the remaining points will be from hidden inputs that test some edge cases. Think about valid inputs that could break your logic. **Note that the templates given will read from `input.txt` and output to `output.txt`, so it might be a good idea to make a copy of the sample output to avoid overriding it.**

Problem 1: Declarations [2 points]

Refer to Lab 3 for a summary of the concepts.

This problem introduces the concepts of *declarations* and *scope*. A program has already been written for you, but it uses variables that are not declared. Your task is to add declarations in the file `hw2/declarations/declarations.cpp` where necessary, or include the standard library header file that includes them. Note that you will not have to implement any of the functions with names starting with `std::` . Also, beware of type signed-ness and size.

You do not have to add any additional outputs. Ensuring that the file compiles and when run with no arguments produces expected behavior is sufficient for full marks.

Problem 2: Pointers, References, Memory, and Data Management [11 points]

Refer to Lab 4 for a summary of the concepts.

This problem introduces file I/O and memory management in C++. In this problem, you are to edit `hw2/memory/memory.cpp` to read a file `input.txt` into memory, and accomplish the following:

- replacing all lowercase ‘e’ with ‘3’, ‘l’ with ‘1’, ‘t’ with ‘7’
- reverse the line order (so first line becomes last line, second line becomes second to last, and so on, until the last line becomes the first)
- output the replaced text to a file `output.txt`
- keep track of all the replaced numbers (only the ones that were replaced to become numbers; if they were numbers already you could ignore them) and
    - multiply each by the line number they appeared in (first line is line 0)
    - sum them together
    - output the total sum as the last line in `output.txt`

For example, our `input.txt` is

```
This is the first line 214
Here is the second line 569
There goes the third line
Finally the last line
```

and the output will be:

```
Fina11y 7h3 1as7 1in3
Th3r3 go3s 7h3 7hird 1in3
H3r3 is 7h3 s3cond 1in3 569
This is 7h3 firs7 1in3 214
155
```

The total sum comes from:

- line 0 * (7 + 3 + 7 + 1 + 3) = 0
- line 1 * (3 + 3 + 7 + 3 + 3 + 1 + 3) = 23
- line 2 * (3 + 3 + 3 + 7 + 3 + 7 + 1 + 3) = 60
- line 3 * (1 + 1 + 7 + 3 + 1 + 7 + 1 + 3) = 72
- 0 + 23 + 60 + 72 = 155

Look into `std::ifstream` and `std::ofstream` for file I/O. Since you need to reverse the lines, you need to store the processed lines in memory using `std::vector<std::string> lines` which is defined for you. **Do not use any global variables** and remember to initialize numbers (no `int x;` nonsense!). Note that `createCounter` should return a pointer to a heap allocated value. If you return a pointer to a local value (stack allocated), that value will become invalid after returning, after which accessing it via the returned pointer will lead to undefined behavior. `counter` simulates a memory intensive resource that you must often write to in called functions, and which we do not have total ownership over. If we had total ownership, we could create it directly instead of only retrieving a pointer or reference to it. In the function called `processLine`, we pass it by pointer because we want to modify the counter outside of the function (you could also do this with references, but for this question we will use pointers). Passing by value would result in the function creating a copy of it, the edits on which would be lost as soon as the function returns.

You need to output to a file named `output.txt` in the same directory as the compiled executable. By default, programs will interpret file names as relative to the directory the program is run from, these are called relative paths. For example, if I am running the program `/home/zhsh/rob502/hw2/my_solution` from `/home/zhsh/rob502/hw2/` then the relative path `"output.txt"` corresponds to `/home/zhsh/rob502/hw2/output.txt`; if I run the same program from `/home/zhsh/rob502/` then the same relative path corresponds to `/home/zhsh/rob502/output.txt`.

Problem 3: Classes and Objects [17 points]

This problem introduces classes and objects; refer to Lab 6 for a summary of the concepts.

```
using Coefficient = double;
class UnivariatePolynomial {
public:
    UnivariatePolynomial() = default;
    // constructor from an initializer list of coefficients
    UnivariatePolynomial(std::vector<Coefficient>);

    // the degree of the polynomial; for simplicity, we will let the degree
    // of the zero polynomial be -1
    int degree() const;

    // change the symbol for the polynomial such as 'y' or 'x'
    void setSymbol(char c);
    // get the symbol
    char symbol() const;
```

```
    // return the derivative of the polynomial
    UnivariatePolynomial derivative() const;

    // addition operation with another polynomial
    UnivariatePolynomial operator+(const UnivariatePolynomial&) const;

    // multiplication with a scalar
    UnivariatePolynomial operator*(Coefficient) const;

    // return its coefficients
    // note that we have both a const and non-const version
    // the const version is needed when the object is const
    // this is an example of method overloading
    const std::vector<Coefficient>& coeffcients() const;
    std::vector<Coefficient>& coeffcients();

private:
    std::vector<Coefficient> _coef;
    // default initialization via assignment
    char _symbol = 'x';
};
```

In this problem, you are to fill out the implementations for the declared methods in `hw2/classes/classes.cpp`, which will be used in some simple math operations to produce an output file in `main`. You will also need to construct the `p2` polynomial in `main` using the given constructor, which should correspond to `p2(x) = -0.1 + -1.4x^2`. Input will be given in `input.txt` where each line will have a variable number of numbers corresponding to the coefficients of a polynomial. Each polynomial will be outputted, followed by its derivative on a newline. The second-to-last line will output the sum of all the polynomials from `input.txt` . The last line will output the derivative of the sum of the polynomials from the previous line.

Problem 4: Constness [14 points]

This problem introduces constness; refer to Lab 3 for a summary of the concepts. In this problem you are to implement the `Table` class, some functions related to it, and create some `const Table` objects in `hw2/constness/constness.cpp`. You will need to implement at least `updateTable`, `t.get` , `t.add` , and `filterOutOddKeys`. Feel free to define more methods inside the `Table` class as needed.

Each Table represents a table of int-string entries (like a dictionary lookup), and we use it to build sentences from number sequences. Your implementation needs to satisfy how existing functions use the Table objects, which will involve `const`. A `const` Table object can only call `const` member methods, so keep this in mind when implementing the methods. Note that for `t.get`, you should return just a space `" "` (not empty) if the key is not in the table. Efficiency is not a concern in this problem. Input will be given in `input.txt` where each line will be a pair of `int` and `std::string` that corresponds to an entry in `t2`. Reading the inputs and adding them to `t2` is handled for you.

Problem 5: Function Overloading and Namespaces [11 points]

For this assignment, some overloaded functions have been defined; refer to Lab 4 and Lecture 6 for a summary of the concepts. You need to fill out the missing parts in `hw2/overload/overload.cpp`. The input will be a file `input.txt` with each line being `operation type type value value`, and you need to call the correct overloaded operation on the types, and output the correct result in a line for the output to `output.txt`. The possible operations are `{"add", "sub", "mult"}`, and the possible types are `{"int", "float", "string"}`. See the in-code comments for what is expected of each operator.

Example input-output pairs and explanations:

- input `add float float 2.2 5.5`
  - output `8` 2.2 gets rounded to 2 and 5.5 gets rounded to 6, 2+6=8
- input `mult string float word 2.6`
  - output `wordwordword` 2.6 gets rounded to 3 and `word` is repeated 3 times

We put the operators inside a namespace here because they have very common names like `add`, `sub`, and `mult`, which are at higher risks of conflict. For use as a library, it also better lets the user specify which version of `add` they are using. Note that the operations are only given for `int` and `std::string`, so you need to convert the `float` to `int` by rounding. Be aware that directly casting a `string` to `int` when it is a `float` will floor the number rather than round it. The `if/else` tree for checking the type and operator is written for you. For the string subtraction, if the second string is longer than the first one, return the first string as-is.