CSCI-1200 Data Structures — Fall 2019
Homework 4 Debugging & Grocery Lists

This assignment consists of two independent parts that can be completed in either order. *Please carefully read through the whole handout before deciding which part you will tackle first.*

PART 1: Using a Traditional Step-by-Step Debugger

In this assignment we provide legacy source code for an existing, complete program that decrypts an input text file. While the program is "complete", it is in need of some serious debugging. Your objective is to get lots of practice using a step-by-step, command line debugger (either `gdb` or `lldb`) while investigating and fixing the errors in this program. You will make minimal edits to the provided source code to fix the errors and decrypt your secret message.

We've prepared multiple versions of the decryption program with errors on different line numbers. At the top of the main "Gradeables" page (https://submitty.cs.rpi.edu/f19/csci1200) for Data Structures on Submitty, you'll find a .zip file for your assigned version of the decryption program. Download those files.

Follow the instructions in the "Homework 4 PART 1" quiz format gradeable on Submitty. Answer the questions as you go and copy-paste information from your debugging session into the webpage as directed.

https://submitty.cs.rpi.edu/f19/csci1200/gradeable/hw04_part1/

The "Rules" for PART 1 of this assignment:

- **Pretend this is a huge, complex project.** Pretend it's so big you can't possibly read and understand all of the code. When you encounter a bug, you can study the relevant surrounding code as needed to understand the situation and propose a solution.
- **Try to avoid falling back on "Print Debugging".** Don't add `std::cout` or `printf` statements to the code. Don't comment out blocks of code. Instead, use the debugger to navigate through the code and examine specific values in the code. Find a good reference or tutorial on your chosen debugger and learn how to use its amazing features.
- **"If it ain't broke, don't fix it."** When working with code written by another developer, we are often tempted to make major logic or style changes because we personally would have written things differently. Resist the urge to do so. Only make changes to the code if it is necessary to fix an error or address a compilation warning.
- **Propose the smallest change to fix the bug.** Once you've identified the bug, if there is more than one way to correct the problem, chose a solution that is the smallest edit. When you edit a line, you should change/add/remove the fewest characters on that line.
- **Don't add or remove any lines or newlines in the file.** Changing the source code line numbers will break the secret code values printed by the program and will change the line numbers you use to set breakpoints. The autograding for PART 1 depends on these line numbers, so make sure your file has the same total number of lines after you finish each step of editing.

After completing PART 1, you should have plenty of practice with all (or most) of these tasks in `gdb` or `lldb`:

- Launch/run the program within the debugger (and specify any necessary command line arguments).
- Manage (create, list, & delete) breakpoints and watchpoints, including conditional watchpoints.
- Control execution of the program one line at a time, either stepping into or over helper functions (using `step`, `next`, and `continue`).
- Examine values of arguments and local variables in the current frame.
- View the overall call stack, and examine different frames on the stack.

The Secret Message Decryption Program

The provided code for PART 1 is organized into four separate *operations* functions with additional helper functions. There are many intentionally-placed bugs within this code. Each operation function is designed to return a specific value that contributes to the decryption process; however, bugs are causing these functions to return the wrong value or crash! By finding and fixing all the bugs, you will obtain secret codes and ultimately decrypt the file. The comments left behind by the original developers will tell you how the program is supposed to work.

For example, consider the first function, `arithmetic_operations`. It starts by initializing 19 different variables with various mathematical operations, and its comments indicate what values they should hold to pass all the tests and return the correct value. However, not all of these variables are calculated correctly, since the program will crash with an assertion on one its own tests. Your job is to fix the arithmetic operations to ensure that the variables get the intended values.

PART 2: Using STL Lists to manage Grocery Lists

For the second part of the assignment, you will practice using STL lists, which are a *sequential container*, just like STL vectors. But vectors and lists have different memory layouts, and thus they have different core functionality and different performance on some operations they have in common.

You'll complete a program to manage the contents of a kitchen as various recipes are prepared. The program will handle several different operations: adding a recipe, buying ingredients for the kitchen, printing out a recipe, making a recipe (which removes ingredients from the kitchen), and printing out the current contents of the kitchen. The input for the program will come from a file and the output will also go to a file. These file names are specified by command-line arguments. Here's an example of how your program will be called:

```
./grocery_list.out requests.txt results.txt
```

The input format is relatively simple. Each request begins with a single character, indicating one of the following requests, described below. You may assume the input file strictly follows this format (i.e., you don't need to worry about parsing badly-formatted input).

```
r salad
2 tomatoes
1 lettuce
0
a
3 tomatoes
4 lettuce
0
p salad
m salad
k
```

1. The first request, indicated by the letter '`r`', says to add a particular recipe to the list of recipes. The ingredients follow, listed one per line with the quantity and name. The ingredient list ends with a '`0`' (zero) on a line by itself. You should verify that there is not already a recipe with this name in the list of recipes. The output for a successful request is:

   ```
   Recipe for salad added
   ```

   Or, if a recipe with that name was previously entered into in the system:

   ```
   Recipe for salad already exists
   ```

2. The next request, indicated by the letter '`a`', says that various ingredients should be added to the kitchen supplies. Again, the ingredients follow, listed one per line with the quantity & name, and the ingredient list ends with a '`0`' (zero). When you add an ingredient to the kitchen you need to first check if the ingredient is already present. If it is, simply increase the quantity of that item. The output for this request indicates the number of distinct ingredients (not the quantity) that were added:

```
2 ingredients added to kitchen
```

3. The '`p`' command, asks that the recipe for `salad` be printed. The ingredients should be listed in alphabetical order with their quantities. For example:

```
To make salad, mix together:
  1 unit of lettuce
  2 units of tomatoes
```

   Note the user friendly output text distinguishes between singular and plural quantities. If the program does not have the requested recipe, the following message should be output:

```
No recipe for salad
```

4. The next request, indicated by the letter '`m`', asks the program to attempt to make a particular recipe. First the program will check the kitchen to see if the necessary ingredients are available. If all of the ingredients are available in sufficient quantities, the kitchen will be edited to remove the appropriate amounts of each ingredient and this message is output:

```
Made salad
```

   If the ingredients are not available, the output message will list the insufficient ingredients (sorted alphabetically), with the quantities that are missing. In this example, one salad can be made, but if a second salad is requested, the following message will be output:

```
Cannot make salad, need to buy:
  1 unit of tomatoes
```

   If the program does not have the requested recipe, the following message should be output:

```
Don't know how to make salad
```

5. The fifth request, indicated by the letter '`k`', asks for the current contents of the kitchen to be output, sorted by quantity first, and then alphabetically for items with equal quantity. In this example:

```
In the kitchen:
  1 unit of tomatoes
  3 units of lettuce
```

   If a particular ingredient has been "used up", it should be removed from the list of ingredients.

Sample input and output files are posted on the course web site. Please follow these examples exactly to aid in the automatic grading of your work. You can use the UNIX `diff` command to compare your output to the sample output files.

Big O Notation

You should implement the functionality above with efficiency in mind. In your README.txt file, use big O notation to analyze the computation required to process each of the different requests. In your analysis use:

i = # of different ingredients in the kitchen
u = maximum units of a single ingredient in the kitchen
r = # of different recipes
k = maximum # of different ingredients in a single recipe
v = maximum units of single ingredient in a single recipe
a = maximum # of different ingredients added with a single 'a' command
w = maximum units of a single ingredient added with a single 'a' command

Note: You may not need to use all of these variables in your answers.

Additional Requirements, Hints and Suggestions

You may not use vectors or arrays for this assignment. Use the standard library (STL) lists and iterators instead. You may not use maps, or sets, or things we haven't discussed in lecture yet.

You must write at least one new class. We have provided a partial implementation of the main program to get you started. There are member function calls to our versions of the `Recipe` and `Kitchen` classes, so you can deduce how some of the member functions in our solution work. You may use none, a little, or all of this, as you choose, but we strongly urge you to examine it carefully.

Submission

There are 2 gradeables associated with this homework on Submitty. The quiz format multiple choice and short answer for PART 1, and then the normal file upload for all of your finished materials (README, PART 1, and PART 2). Follow the intructions carefully. The combined total assignment will be worth 50 points, the same as other homeworks. Late day usage will be determined from the timestamp on the file upload gradeable. You must submit to the PART 1 quiz format gradeable before or at the same time as the file upload gradeable.

Use good coding style when you design and implement your program for PART 2. Be sure to make up new test cases and don't forget to comment your code! Please use the provided template `README.txt` file for any notes you want the grader to read. **You must do this assignment on your own, as described in the "Academic Integrity for Homework" handout. If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file.**