CMPUT 275 - Tangible Computing **Winter 2019**
Weekly Exercise #6

STL Concepts

In this exercise, you will apply the C++ Standard Template Library (STL) to complete two separate tasks: (1) learn how to store a user-defined struct in an unordered set using a custom hash function and (2) use `pair` and `stack` data structures to implement a postfix calculator for complex numbers. The goal of this exercise is for you to gain familiarity with common STL data structures. Through two different applications of the STL, you will see ways in which these concepts can be used in practice. This knowledge will help you in later exercises and the second major assignment, in which you are expected to be very familiar with STL data structures and algorithms.

Your Task #1: Hashing a Custom Struct

In the first part of this exercise, you will learn how to create an `unordered_set` to store a user-defined data structure. Recall that we have been using the following struct as a running example in class:

```
struct StudentRecord {
    string name;
    int id, grade;
};
```

The `unordered_set` implements a hash table and requires a hash function to map each object to its hash value. However, user-defined structs contain multiple data members and so they can have no automatically defined hash function. This is because creating a hash function depends on the data types stored in the struct and on the semantics of the struct itself.

> In order to use an unordered associative container such as `unordered_set` with a user-defined key-type, you need to define two things:
>
> 1. A **hash function**, a struct that overrides **operator()** and calculates the hash value given an object of the key-type. One particularly straightforward way of doing this is to specialize the std::hash template for your key-type.
>
> 2. A **comparison function for equality**; this is required because the hash cannot rely on the fact that the hash function will always provide a unique hash value for every distinct key (i.e., it needs to be able to deal with collisions), so it needs a way to compare two given keys for an exact match. You can implement this either as a struct that overrides **operator()**, or as a specialization of std::equal, or – easiest of all – by overloading **operator==()** for your key type.
>
> - paraphrased from StackOverflow

For example, the following code:

```
// inside main function, same file as StudentRecord definition
unordered_set<StudentRecord> table;
```

will not work because there is no hash function and no equality operator defined for `StudentRecord`.

Your Task: Complete all of the following.

1. Modify the provided file `student_hash.cpp` to create a custom hash function and equality operator for the `StudentRecord` struct. The hash function can be based solely on the field `id`, which is guaranteed to be unique. Some additional resources for implementing the custom hash and equality operator can be found in the Hints section below.
2. Create an instance of an `unordered_set` that stores instances of `StudentRecord`.
3. Write a program to parse input from the command line (see below) and manipulate and query your `unordered_set`. Each query will ask a question about the contents of the `unordered_set` (Are there any students in the set named "Joseph"?) or manipulate the set (delete the student with id = 1503333).

Input:

Each line of input will take one of the following forms:

- `I name id grade`: If the `id` is not already in the set, insert a new StudentRecord into the set with the given values.
- `R id`: If an object with `id` exists, remove it from the set.
- `Q i id`: Query for the given `id`. If an object with `id` exists, print it. There can be at most one such object.
- `Q n name`: Query for and print all objects with the given `name`. There can be many such objects.
- `Q g grade`: Query for and print all objects with the given `grade`. There can be many such objects.
- `S`: Indicates the end of the query session. No input will follow this line.

Output:

For lines containing `Q`, output the StudentRecords that match the provided field in the following format:

```
Name: name, ID: id, Grade: grade
```

Note that there is a single space after every comma and colon. If there are multiple matches, printing order does not matter.

Perform error handling on all lines of input:

- For lines containing `I`, print an error message, "Error: Cannot insert duplicate ID", if the provided `id` already exists in the set. Do not allow duplicate insertion.
- For lines containing `R`, print an error message, "Error: Cannot remove non-existent ID", if the provided `id` does not exist in the set. Do not attempt to remove an `id` that is not present.

- For lines containing `Q`, if no students match the provided field (`id`, `name`, or `grade`), print "Error: No matches found".

None of these error messages should cause the program to quit. Simply print the message and continue reading input. You do not need to perform any other error handling.

Sample Input 1

```
I Jane 1234567 50
I Ruben 7238139 75
Q n Jane
R 1234567
Q n Jane
I Josh 1231234 75
Q g 75
S
```

Sample Output 1

```
Name: Jane, ID: 1234567, Grade: 50
Error: No matches found
Name: Josh, ID: 1231234, Grade: 75
Name: Ruben, ID: 7238139, Grade: 75
```

Explanation: We insert Jane and Ruben and then query for all students named Jane. One is found. Then we remove Jane. Performing the same query results in no matches. Then add Josh with the same grade as Ruben. Both Ruben and Josh are returned when searching for grade = 75. Note that the last two lines could be in either order.

Sample Input 2

```
R 1111111
I Emilia 1234567 24
Q i 1234560
Q i 1234567
I Henry 1234567 89
R 1234567
I Henry 1234567 88
Q i 1234567
S
```

Sample Output 2

```
Error: Cannot remove non-existent ID
Error: No matches found
Name: Emilia, ID: 1234567, Grade: 24
Error: Cannot insert duplicate ID
Name: Henry, ID: 1234567, Grade: 88
```

Explanation: First, no students have been inserted so the removal results in an error message: Cannot remove non-existent ID. After inserting Emilia, querying for the wrong id (1234560) results in no matches. Henry can only be inserted after the id (already taken) is removed from the table.

Hints:

- Here are several helpful resources:
  - How to use Unordered_set with User defined classes - Tutorial & Example.
  - cppreference.com, std::hash.
  - cppreference.com, std::unordered_set, a reference page. Pay attention to the methods `insert`, `erase`, `find` (under Modifiers).
- To determine if any data members existing in the `unordered_set` have a given field value (e.g., *name= "Zac"*), a constant-time lookup (e.g., using `find`) is not sufficient (e.g., because *name* alone is not the key). You will need to come up with a different way to do this check, especially if there are multiple matches. Your solution must return all possible matches.

Your Task #2: Postfix Calculator for Complex Numbers

In this part of the exercise, you will implement a calculator to evaluate expressions of complex numbers written using postfix notation.

What is Postfix Notation?

Postfix notation, also called reverse polish notation, is a method of writing arithmetic expressions in which operands appear before their operators. It is often used because it is extremely simple: there is no need to learn complex precedence rules and brackets are not required. In elementary school, you learned how to evaluate arithmetic expressions given in *infix notation*. This is the notation you are most familiar with. For example:

$$(12 + 1) * (7 - 3) = 13 * 4 = 52$$

Here is the equivalent expression in postfix notation:

$$12 \quad 1 \quad + \quad 7 \quad 3 \quad - \quad *$$

Reading from left to right, we could evaluate this expression in the following way:

1. *Store* the operand 12. *Store* the operand 1.
2. *See* the operator + and *Retrieve* the previous two stored operands, 1 and 12. Evaluate and *Store* the result 12 + 1 = 13.
3. *Store* the operand 7. *Store* the operand 3.
4. *See* the operator - and *Retrieve* the previous two operands, 3 and 7. Evaluate and *Store* the result 7 - 3 = 4.
5. *See* the operator * and *Retrieve* the previous two operands, 4 and 13. Evaluate and *Store* the result 13 * 4 = 52.

6. There is no more input to process, so *Retrieve* the final result of the entire postfix expression, 52, and print it.

A natural data structure used to perform this computation is a **stack**, where the *Store* operation is equivalent to `push` (add an item onto the top of the stack) and *Retrieve* is simply `pop` (retrieve and remove the top item from the stack).

In this task, you will use STL data structures to evaluate expressions written in postfix notation using an algorithm based on the above example. Some additional pseudocode and explanation can be found here if you need more guidance. To make things more interesting, you will use the `pair` data structure to store complex numbers on the stack.

Background: Complex Numbers

> A **complex number** is any number that can be written as a + bi, where i is the imaginary unit and a and b are real numbers.
>
> The real part of the number, or a, is the real number that is being added to the pure imaginary number.
>
> The imaginary part of the number, or b, is the real number coefficient of the pure imaginary number.
>
> - from Khan Academy, Intro to Complex Numbers

More background and some motivation for complex numbers can be found at Complex Numbers: Introduction. Note that the imaginary part is some multiple of the unit i, where

$$i = \sqrt{-1}$$

This means that $i^2 = -1$.

Operations on Complex Numbers

In your postfix calculator, you are required to implement all of the following operations.

Binary Operations: A binary operation `op` takes two operands, meaning that it is of the form `A op B`. In postfix notation, a binary operation looks like: `A B op`

1. **Addition:** To add two complex numbers together, simply add the real and imaginary parts separately.

```
(2 + 3i) + (5 - 7i) = 7 - 4i
```

2. **Subtraction:** Simply subtract the real and imaginary parts separately.

```
(2 + 3i) - (5 - 7i) = -3 + 10i
```

3. **Multiplication:** Multiply through the brackets (*i.e.*, FOIL) and collect like terms.

```
(2 + 3i) * (5 - 7i) = 2*5 + 2*(-7i) + (3i)*5 + (3i)*(-7i)
                    = 10 - 14i + 15i - 21i^2
                    = 10 + i + 21   // remember that i^2 = -1
                    = 31 + i
```

Note that this process can be simplified. If `A = a + bi` and `B = c + di` then `A * B` equals

```
(a + bi) * (c + di) = ac + (ad)i + (bc)i + (bd)i^2
                    = ac + (ad+bc)i - bd  // remember i^2 = -1
                    = (ac-bd) + (ad+bc)i
```

The final formula above computes the product of a complex number more simply.

Unary Operations: A unary operation `op` takes only a single operand and is of the form `op A`. In postfix notation, a unary operation is of the form `A op`.

1. **Negation:** This operation converts a complex number of the form `a + bi` to `-a - bi`.

```
-(5 - 7i) = -5 + 7i
```

2. **Conjugation:** The conjugate of a complex number `A` (here denoted `cA`) simply changes the sign of the imaginary part only. Thus, if `A = a + bi`, then `cA = a - bi`.

```
c(5 - 7i) = 5 + 7i
```

For more resources complex numbers and on the operations discussed above, see the following:

- Khan Academy, Complex Number Operations: A concise review of addition, subtraction, and multiplication.
- The Complex Conjugate: Discusses the definition of conjugate for complex numbers along with its most important mathematical property.
- Symbolab provides a calculator for expressions involving complex numbers. You can compare your implementation to the results of this calculator to make sure it is correct.

Implementing the Postfix Calculator:

In a file called `complex_postfix.cpp`, write a postfix notation calculator for complex numbers. When run, your program should expect to read in the postfix notation expression in a particular format (described below). It should then store and evaluate it using the `pair` and `stack` data structures in the STL.

Requirements:

- The purpose of this exercise is to become more familiar with the STL. Therefore, you must implement your calculator using `pair` and `stack` to store the data. Add the following lines at the top of your program:

```
#include <utility>      // std::pair lives here
#include <stack>        // std::stack lives here
using namespace std;    // can now refer to std::pair as pair
```

to gain access to the `pair` data structure, which resides in the `std` namespace just like `cin` and `cout`. For more information on using `pair`, see cplusplus.com and geeksforgeeks.org. For specific information on using STL stacks, see this page.

- You are required to implement all complex number operations yourself, and **you may not use any external libraries for this purpose**.

Input Specification:

Each line of input will take one of the following forms:

- `V r1 r2`: A complex number operand, where `r1` and `r2` are both integers that will fit into the type *long long*. `r1` represents the real part and `r2` represents the imaginary part of the complex number.
- `B op`: Binary operator, where `op` is one of '+' (addition), '-' (subtraction), or '*' (multiplication).
- `U op`: Unary operator, where `op` is one of '-' (negation) or 'c' (conjugation).
- `S`: Represents end of input (*i.e.*, `STOP`), indicating that there are no further instructions.

You are guaranteed that all input will be valid. This means that:

- You will not see letters other than `V`, `B`, `U` or `S`. All numerical input following each letter will be of the expected form.
- No computation (including the results of intermediate operations) will result in arithmetic overflow.
- No input expression will cause stack overflow or underflow, meaning that at the end of computation there will always be exactly one value left on the stack. You will never be asked to perform an operation when there are insufficient operands on the stack.

Output Specification:

Output the result of evaluating the postfix expression. That is, output $r1$ and $r2$ separated by a single space, where $r1$ is the real part and $r2$ is the imaginary part of the result, followed by a newline.

Sample Input 1

```
V 3 3
V 5 -7
B *
S
```

Sample Output 1

```
36 -6
```

Explanation: $(3 + 3i) * (5 - 7i) = 36 - 6i$.

Sample Input 2

```
V 2 3
V 5 -7
B *
V 4 -3
U c
B +
S
```

Sample Output 2

```
35 4
```

Explanation: This is the expression

$$\begin{aligned}(2 + 3i) * (5 - 7i) + c(4 - 3i) &= (31 + i) + c(4 - 3i)\\ &= (31 + i) + (4 + 3i)\\ &= (35 + 4i)\end{aligned}$$

Hint: A good approach to solving this problem may be to organize your calculator into a C++ class, which contains the required data structures and implements each operation as a separate member method.

Note that you should use the type *long long* to store the real and imaginary parts of each complex number inside the `pair` data structure. Since the resulting type is lengthy, you are welcome to (but not required to) use a typedef if you want to type fewer characters each time you use it.

Custom Makefile Instructions:

Consult the updated Code Submission Guidelines to see what is expected from a `Makefile` in general. In this weekly exercise, you must include the following targets:

- The main target `all` which generates both executables (`student` and `calc`). This should be the topmost target, so it can be built simply by typing `make`.
- The target `student` which links `student_hash.o` and generates the `student` executable.
- The target `calc` which links `complex_postfix.o` and generates the `calc` executable.
- The targets `student_hash.o` and `complex_postfix.o` which compile the relevant objects.
- The target `clean`, which removes all objects and executables.

Make sure all dependencies are properly listed for each target.

Optional Challenges:

This will not be graded but it is perfectly fine to include any part of this in your final submission, in case you want more practice.

- Implement complex number division in your postfix complex number calculator and handle division by zero errors (e.g., by throwing an exception).
- Add error handling to your postfix complex number calculator to handle cases of stack overflow and stack underflow (ie, too few or too many values on the stack). Ideally, you should do this by implementing your own custom exception class. A simple example can be found here.

Submission Details:

Compress all the following files in a compressed archive called `stl_concepts.tar.gz` or `stl_concepts.zip`. Submit only this `.tar.gz` or `.zip`.

- `student_hash.cpp` - Containing your implementation of the first task.
- `complex_postfix.cpp` - containing your implementation of the second task.
- `Makefile` - Generates the `student` and `calc` executables.
- Any additional .cpp or .h files required to solve the problem. Note that including additional files is acceptable as long as your `Makefile` functions as specified and running instructions are clearly provided.
- `README`

Make sure to follow the Code Submission Guidelines! Submit a clean compressed archive (ie, no object files or executables should be included in your submission).