

Assignment #5: Advanced C++ and Design Patterns

Due Date 1: Wednesday, July 22, 2020, 5:00 pm

Due Date 2: Wednesday, July 29, 2020, 5:00 pm

Online Quiz: Friday, July 31, 2020, 5:00 pm

Topics that must have been completed before starting Due Date 1:

1. Advanced C++: Unique and shared pointers (smart pointers)
2. Advanced C++: RAII: Resource Acquisition Is Initialization
3. Advanced C++: Exception Safety

Topics that must have been completed before starting Due Date 2:

1. Advanced C++: Template Functions
2. Advanced C++: Function Objects and Lambdas
3. Advanced C++: STL Algorithms

Learning objectives:

- Advanced C++: Smart pointers and RAII
- Advanced C++: Exception Safety
- Advanced C++: STL Templates

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- Questions 1, 2a, 3a are due on Due Date 1; questions 2b and 3b are due on Due Date 2. You must submit the online quiz on Learn by the Quiz date.
- You must use the C++ I/O streaming and memory management facilities on this assignment. Marmoset will be programmed to **reject** submissions that use C-style I/O or memory management.
- You may include the headers `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, `<utility>`, `<stdexcept>`, `<vector>`, `<map>`, `<memory>`, and `<algorithm>`.
- You are **not allowed** to use `delete`. You must manage memory with smart pointers only.
- For question 2, you are additionally allowed to include `<cmath>`, for the `pow` function.
- Each question on this assignment asks you to write a C++ program, and the programs you write on this assignment each span multiple files. For this reason, we **strongly** recommend that you develop your solution for each question in a separate directory. Just remember that, for each question, you should be *in* that directory when you create your ZIP file, so that it does not contain any extra directory structure.
- You are required to submit a `Makefile` along with every code submission (except question 1). Marmoset will use this `Makefile` to build your submission.

Question 1

We have provided the code for a small banking application in the folder `codeForStudents`, which keeps track of accounts and balances and can transfer balances between accounts. Please check the files `bank_app.{h,cc}`, `bank.{h,cc}`, and `account.{h,cc}` to understand how it works.

The file `main.cc` provides a simple test and `expected.out` provides the expected output from `main`. However, the current implementation has an issue that prevents it from generating the correct output: methods `Bank::transferBalance`

and `Bank::transferToNewAccount` are not exception safe. Therefore, if an exception occurs, the data is left at an inconsistent state. Additionally, the implementation currently does not delete any of the heap-allocated memory.

Your task is to make two changes to this program:

1. Use smart pointers and/or RAII principles to fix the memory leaks. For this assignment, you are not allowed to use `delete` to free memory. If you implement everything correctly with smart pointers, your program won't have any `delete` statement but won't leak any memory.
2. Fix the methods `Bank::transferBalance` and `Bank::transferToNewAccount` to be exception safe (strong guarantee): if any exception happens, the program's state after the method execution ends should be as if the method had never been called. For this assignment, you are not allowed to try and avoid the exceptions. You must change the methods so that the exceptions are still raised, but the program state is consistent after that.

Implementation Notes:

- Your solution should only modify the files `bank_app.{h,cc}` and `bank.{h,cc}`. None of the other files need to be modified.
- `main.cc` is only for testing the solution. For this question, you don't need to create a test suite, but feel free to create one for your own use if you wish. You don't need to submit `main.cc`. If you do, it will be ignored and we will test your solution with our own `main` program. If you wish, feel free to modify `main.cc` or create alternative test programs for your own use.

Submission:

1. **Due on Due Date 1:** Make the required changes in the implementation. Submit your `bank_app.{h,cc}` and `bank.{h,cc}` files.
Note: we will test your implementation with our version of `account.{h,cc}` and `*_exception.h` (identical to the ones we provided to you) and our versions of `main.cc` (which can be different than the example provided to you). This question will also be hand-marked to ensure that you followed the requirements in your implementation.
2. **Due on Due Date 2:** Nothing.

Question 2

In this question, you will write a simple flight planning library. Note that *no code is provided* as a baseline: You must create all headers and implementations. You must use smart pointers and RAII principles throughout your code. You should never directly use `new` or `delete`.

The purpose of this library is to suggest flight plans, given a list of airports and flights between them. A flight plan is a list of flights connecting two airports. Each airport is a node in a graph, and each flight connects two airports. Each flight has a duration in hours, a price in dollars, and a name (e.g. Dominion Air Flight 123). For the purposes of this library, we aren't concerned with flight schedules (i.e., all layovers are possible), and it is assumed that all flights have an equivalent return flight (that is, if there's a flight from airport A to airport B, then there is an equivalent flight, taking the same amount of time, at the same price, with the same name, from B to A). There may be multiple flights between the same two airports, but **no two flights from or to any airport may have the same name**.

In a file `flight.h`, you should expose a class `FlightPlanner`, class `FlightPlan`, class `Flight`, and class `FlightNameException`. Although you must maintain a graph of airports to implement this problem, the API of the graph itself and airports is not specified. You may implement them in any way you please. The API of `FlightPlanner`, `FlightPlan`, `Flight`, and `FlightNameException` will be defined, but in each case, you will likely need additional, internal methods for your own implementation. `FlightPlanner` should have the following API:

- The constructor should initialize an empty flight planner, i.e., with no airports or flights.
- There should be no explicit destructor, but all memory used by the flight planner should nonetheless be freed correctly when the flight planner and all other smart pointers to its internal resources are destroyed.

- The method `void addAirport(const std::string &name)` adds an airport to the graph. If an airport already exists with the given name, a `FlightNameException` is thrown.
- The method
`void addFlight(const std::string &from, const std::string &to, int duration,
int price, const std::string &name)`
adds a flight to the graph, between airports `from` and `to`, with the given duration, price, and name. If a flight already exists at either of these two airports with the given name, or if one or the other named airport does not exist, a `FlightNameException` is thrown. Since flights always have return flights, this method should additionally add an equivalent flight from `to` to `from`.
- The method
`std::vector<std::shared_ptr<FlightPlan>> planTrip(const std::string &from,
const std::string &to)`
should return every possible flight plan from the airport named by `from`, to the airport named by `to`. `FlightPlan` itself is described below. No flight plan should loop, i.e., no plan should visit an airport twice. The vector should be sorted by pain (see below), from least to most pain. If the `from` or `to` airport does not exist, a `FlightNameException` is thrown. If no flight plans are possible between the two given airports, an empty vector should be returned.

`FlightPlan` is a particular flight plan. A flight plan is a list (implemented as a vector) of steps, where each step is a flight. If the `FlightPlan` describes a trip from airport A to airport B, then the “from” airport of the first step should be A, the “to” airport of the last step should be B, and the “to” airport of each non-final step should be the “from” airport of the following step. That is, the steps form a chain of flights. The API of `FlightPlan` is:

- You may design the constructor(s) in any way.
- There should be no explicit destructor.
- The method `std::vector<std::shared_ptr<Flight>> getSteps()` should return the vector of flights in this flight plan.
- The method `int getDuration()` should return the total duration of this plan, assuming a 1 hour layover between each pair of flights. That is, the duration is the sum of the durations of each step, plus the number of steps, minus one.
- The method `int getPrice()` should return the total price for this trip, which is simply the sum of the price of each step. (In real flight planning, of course, this is not how price is calculated.)
- The method `int getPain()` should return the “pain” for this trip. As the experience of commercial flight gets exponentially more awful the longer it lasts, we define pain as $1.2^d * p$, rounded down, where d is the total duration as returned by `getDuration`, and p is the total price as returned by `getPrice`. For instance, a plan with a duration of 2 hours costing \$800 has pain 1,152, while a plan with a duration of 12 hours costing \$200 has pain 1,783.

`Flight` is a particular flight, with the following API:

- You may design the constructor(s) in any way.
- There should be no explicit destructor.
- The method `std::string getFrom()` returns the airport this flight leaves from.
- The method `std::string getTo()` returns the airport this flight goes to.
- The method `int getDuration()` returns the duration of this flight.
- The method `int getPrice()` returns the price of this flight.
- The method `std::string getName()` returns the name of this flight.

Finally, `FlightNameException` has no specific API, but must exist. You may want to add a message method to it for your own debugging purposes, and create `FlightNameExceptions` with messages.

The recommended algorithm for finding paths through the graph is simply to recurse over every possible flight, keeping track of which airports you've already traversed so you can avoid loops.

Many, many components of this problem are made simpler by available algorithms in the STL! Your grade will depend on not just a correct implementation, but good use of the STL. Further, you must not use *any* explicit destructors for the entire implementation, but must, of course, still properly clean up memory.

A sample test harness is provided in `codeForStudents` directory, in `flight_planner.cc`. A compiled version is also provided, named `flight_planner`, to test against. Compile `flight_planner.cc` along with your own code to create your own `flight_planner`, which is a simple command-line application which creates a single `FlightPlanner` and handles simple commands. Because of the simplicity of the command-line interface, it supports only airport and flight names with no spaces, but your library must support any characters in names. The commands are:

Command	Explanation
<code>a name</code>	Adds an airport with the given name.
<code>f from to duration price name</code>	Adds a flight from and to the given airports, with the given duration, price, and name.
<code>p from to</code>	Plans a trip from the given airport, to the given airport, and displays all possible flight plans.
<code>m</code>	Enables or disables a memory testing mode, in which <code>p</code> deletes the flight planner before outputting flight plans. This allows you to test that nothing is incorrectly deleted, but <code>valgrind</code> is recommended to verify that nothing is leaked.
<code>q</code>	Quits.

- a) **Due on Due Date 1:** Design a test suite for this program, using the `main` function provided in the test harness. Submit a file called `a5q2a.zip` that contains the test suite you designed, called `suiteq2.txt`, and all of the `.in` and `.out` files.

- b) **Due on Due Date 2:** Write the library in C++. Save your solution in a file named `a5q2b.zip`. It must contain a `Makefile` that creates an executable named `a5q2` when the command `make` is given, and must include the `flight_planner.cc` test harness, unmodified.

Question 3

Some early handheld calculators, as well as several programming languages, use a system for describing arithmetic expressions called *reverse Polish notation*. Reverse Polish notation is easiest thought of as describing a mathematical expression in a sequence of steps that use a stack. For instance, you might push 5 to the stack, push 3 to the stack, then perform the “add” step, which pops the top two elements from the stack and adds them together, then pushes the result to the stack. In reverse Polish notation, this sequence of actions is written as “5 3 +”. All steps are either pushing a number onto the stack, or popping two numbers to perform an operation, pushing the result. These sequences can be arbitrarily long, and because of the stack-like action of pushing and popping. A valid reverse Polish notation expression always results in exactly one element on the stack.

For instance, the mathematical expression “(5+2) / (4-1) + 7” can be rewritten as “5 2 + 4 1 - / 7 +”. Since the + and - each pop their respective two numbers and push one, the / operates over the results of the + and -. Here are the steps in evaluating that expression, one by one, assuming / is int division:

Current operation	Stack after operation
5	5
2	5 2
+	7
4	7 4
1	7 4 1
-	7 3
/	2
7	2 7
+	9

You will write a reverse Polish notation calculator, which additionally reformats reverse Polish notation expressions to standard mathematical expressions, also called *infix notation*. For instance, the sequence “5 3 +” from above is written as “5+3” in infix notation, and evaluates to 8. Your calculator must support the standard four arithmetic operators: +, -, *, and /, operating over ints. Please note that *no base code at all* is provided. You must write the entire solution from scratch.

Your program will read lines in reverse Polish notation from standard input, and output lines containing both the infix expression and its solution. Correctly placing parentheses in infix expression is complex, so simply parenthesize *every* operator with its operands. For instance, if the user inputs 5 2 + 4 1 - / 7 +, the program should output (((5+2) / (4-1)) + 7) = 9. Note that in our reverse Polish notation, all operations must be separated by spaces, but our infix notation will never use spaces.

A demonstration of an interaction with the program (lines in reverse Polish notation are input, and lines with infix notation and solution are program output):



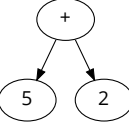
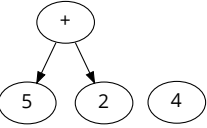
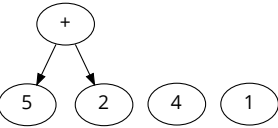
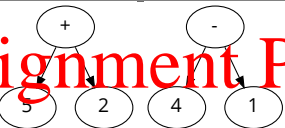
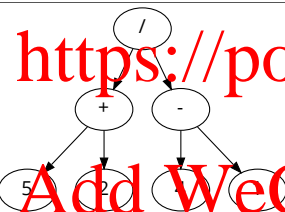
```

5 2 + 4 1 - / 7 +
(( (5+2) / (4-1) ) + 7) = 9
5 3 +
(5+3) = 8
1 2 3 4 5 6 7 + - * / + -
(1 - (2 + (3 / (4 * (5 - (6 + 7) ) ) ) ) ) = -1
1 2 + 3 - 4 * 5 / 6 + 7 -
((( (( (1+2) - 3) * 4) / 5) + 6) - 7) = -1
49 8 * 1000 * 3 100 * 5 12 * 7 + - 3 * + 8 *
((( (49*8)*1000) + (((3*100) - ((5*12)+7)) * 3)) * 8) = 3141592

```

To do this, you will have to create a tree from the reverse Polish notation input, then traverse the tree. **You are expected to use the visitor pattern to traverse the tree**, as this will make it easy to write one visitor that creates the infix notation expression, and another that gets the solution. Note that it is possible to solve reverse Polish notation expressions with no data structures, but not to rewrite them in infix notation, so you will need to create a tree.

To build a tree from a reverse Polish notation expression, simply follow the same stack operations as described above, but pushing or popping trees instead of pushing or popping values. For an operation, rather than actually performing the operation, make a new tree with the two trees you popped as its children. Here are the steps of converting “5 2 + 4 1 - /” to a tree, visually:

Current operation	Stack after operation
5	
2	
+	
4	
1	
-	
/	

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

You will likely want a subclass of your tree class for numbers, and subclasses for each of the valid operations. To solve, the visitor of an operation should call the visitor of each child, and return the result of the operation as performed on the returns from the visitor calls to each child. To create an infix notation expression, visitors should instead return strings, and concatenate them as appropriate.

You will need to check that the input uses the stack correctly (i.e., it never attempts to use an operation with only one element on the stack, and ends with exactly one element on the stack), but exactly what to do in this case is not specified.

- Due on Due Date 1:** Design a test suite for this program. Submit a file called `a5q3a.zip` that contains the test suite you designed, called `suiteq3.txt`, and all of the `.in` and `.out` files.
- Due on Due Date 2:** Write the program in C++. Save your solution in a file named `a5q3b.zip`. It must contain a `Makefile` that creates an executable named `a5q3` when the command `make` is given.