CS246—Assignment 3 (Fall 2013)

R. Ahmed B. Lushman M. Prosser

Due Date 1: Friday, October 18, 5pm
Due Date 2: Friday, November 1, 5pm

Questions 1a, 2a, 3a, and 4a are due on Due Date 1; the remainder of the assignment is due on Due Date 2.

Note: You must use the C++ I/O streaming and memory management facilities on this assignment. Moreover, the only standard headers you may `#include` are `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, and `<cstdlib>`. Marmoset will be programmed to **reject** submissions that violate these restrictions.
Note: 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 your zip file does not contain any extra directory structure.
Note: Questions on this assignment will be hand-marked for style, and to ensure that your solutions employ the programming techniques mandated by each question.

1. In this exercise, you will write a C++ class (implemented as a struct) to control a simple robotic drone exploring some terrain. Your drone will start at coordinates (0,0), facing north. Use the following structure definition for coordinates:

   ```
   struct Position {
     int ew, ns;
   };
   ```

   The east-west direction shall be the first component of a position, and the north-south direction shall be the second. Your `Drone` class must be properly initialized via a constructor, and must provide the following methods:

   - `void forward();` – move one unit forward
   - `void backward();` – move one unit backward
   - `void left();` – turn 90 degrees to the left, while remaining in the same location
   - `void right();` – turn 90 degrees to the right, while remaining in the same location
   - `Position current();` – returns the current position
   - `int totalDistance()` – total units of distance travelled by the drone
   - `int manhattanDistance()` – Manhattan distance between current position and origin (Manhattan distance is absolute north-south displacement plus absolute east-west displacement).
   - `bool repeated()` – true if the current position is one that the drone has previously visited

For simplicity, you may assume that the drone will never visit more than 50 positions before running out of fuel or otherwise breaking down.

A test harness will be provided, by which you may interact with your drone for testing purposes.

For this question, you are allowed to include the header `<cmath>` and use the functions declared therein.

Due on Due Date 1: Test suite (`suiteq1.txt`)

Due on Due Date 2: Solution (`drone.h, drone.cc`)

2. Consider the following object definition for an "improved"[1] string type:

```
struct iString {
  char * chars;
  unsigned int length;
  unsigned int capacity;
  iString();
  iString(const char *a);
  iString(const iString &a);
  ~iString();

  iString &operator=(const iString &other);
};
```

You are to implement the undefined constructors and destructors for the `iString` type. Further, you are to overload the input, output, addition, and multiplication operators according to the following examples:

```
iString s1(); // Create an empty string (length is zero, chars is null)
iString s2("foobar"); // Create a string initialized with the word "foobar"
iString s3(s2); // Call the copy constructor, initialize s3 to have the contents of s2
iString s4();

cin >> s1 >> s4; // Read in whitespace-delimited strings from stdin
cout << s1 << " " << s2 << " " << s3 << " " << s4 << endl; // Print iStrings to stdout

s1 = s1 + s4; // Concatenate s1 and s4
s2 = s2 + "baz"; // Concatenate s2 and the word "baz"
s3 = 3 * s3; // Multiply an iString by a scalar (duplicate the string 3 times)
             // Equivalently:  s3 = s3 * 3;  or  s3 = s3 + s3 + s3;
```

Implementation notes

- The declaration of the `iString` type can be found in `istring.h`. For your submission you should add all requisite declarations to `istring.h` and all routine and member definitions to `istring.cc`.
- You are not allowed use the C++ `string` type to solve this question. However, you may include the header `<cstring>` and use the functions declared therein.
- Becoming familar with `cin.peek()` and the `isspace` function located in the `<locale>` library may aid you in solving this question.

[1] For some definition of improved. Namely, overloading multiplication.

- The provided driver (`a3q2.cc`) can be compiled with your solution to test (and then debug) your code. This test harness requires you to follow a protocol when using it (read the source code carefully). This is because the test harness is written with the expectation of always being used correctly. Accordingly, the test harness is not robust. The job of the test harness is to provide you with a simple framework to test your solutions without having to constantly write new mainline programs.

Deliverables

(a) **Due on Due Date 1**: Design a test suite for this program (call the suite file `suiteq2.txt` and zip the suite into `a3q2a.zip`)

(b) **Due on Due Date 2**: Implement this `iString` type in C++ (include the provided driver and all .h and .cc files that make up your program into the zip file, `a3q2b.zip`).

3. In this question, you will write a program to track your family tree. A family tree begins with a topmost ancestor, and indicates that ancestor's children, and then their children, and so on. In this particular style of family tree, which is oriented towards decendants rather than ancestors, each non-root member is shown as a descendant of one parent (the other parent, who (possibly) married into the family, is not mentioned).

   When you run your program, it will listen on stdin for input. The following commands are recognized:

   - `! name` Adds `name` as the topmost ancestor of the tree; if the tree already has a topmost ancestor, this new topmost ancestor is added as the parent of the current topmost. If the tree is empty, this command creates a new tree that contains only `name`.
   - `+ name1 name2` Adds `name2` to the tree as a child of `name1`. If `name1` is not found in the tree, print out `Failed` (and a newline) to stdout.
   - `? name` Answers whether `name` occurs in the tree. If `name` is found, output the sequence of names (separated by commas) from the topmost ancestor to `name`, inclusive (and a newline); otherwise, output `Not found` (and a newline).
   - `* name` Oops! It turns out that `name` was not part of this family after all. Remove `name` and all of its descendants from the tree. If `name` is not in the tree, this command has no effect.
   - `p` Prints the tree. Your print routine should perform a preorder traversal of the tree, following the format of the model presented below. Children should be presented and traversed in the order in which they were added to the tree.
   - `include filename` Reads the file `filename` and executes the commands contained therein.

   The program terminates when it encounters an EOF signal on stdin.

   Notes:

   - You may assume that a name cannot occur in the tree more than once.
   - You may assume that names do not contain spaces.
   - Use the provided `family.h` file to build your tree data structure. You may add function headers to this file.
   - You must provide appropriate constructors for initializing tree nodes and make use of those constructors when creating new tree nodes.
   - You must use destructors to deallocate tree nodes when needed.
   - You must put the functions that operate on the tree in a separate `.cc` file from your main function. Your submission will be handmarked to ensure that you follow proper procedures for building separately-compiled modules.

- You must deallocate all dynamically allocated memory by the end of the program; hand-markers will be checking for this.

Here is a sample session (user input is given in italics):

```
! alice
! bob
? bob
bob
+ bob eve
+ eve fred
? eve
bob,eve
? alice
bob,alice
p
bob is parent of alice,eve
alice has no children
eve is parent of fred
fred has no children
```

(a) **Due on Due Date 1:** Design a test suite for this program (call the suite file `suiteq3.txt` and zip the suite into `a3q3a.zip`).

(b) **Due on Due Date 2:** Implement this program in C++ (put your mainline program in the file `a3q3.cc`, and include any other `.h` and `.cc` files that make up your program in your zip file, `a3q3b.zip`).

4. Consider the following game, involving two players A and B. Two integers n and k are chosen, both strictly greater than 0, with $k \leq n$. A global counter is initialized to the value of n, and players take turns choosing a number j, such that $1 \leq j \leq k$, and subtracting j from the global counter. The player who reduces the counter to 0 (or lower) wins. In this problem, you will write a C++ program to play several rounds of this game and report the winner. A sample interaction follows (your input is in italics):

```
game 25 7 stdin stdin
Total is 25, max deduction is 7
Player A's move
8
Total is 18
Player B's move
4
Total is 14
Player A's move
6
Total is 8
Player B's move
3
Total is 5
Player A's move
5
Total is 0
Player A wins
Score is
```

```
A 1
B 0
quit
```

As you can see from the first move above, if a move is larger than the maximum deduction, the maximum deduction is applied. Similarly, if the move is less than 1, a deduction of 1 is applied.

In between games, two commands are recognized:

- `game n k sA sB` Starts a game with the given values of `n` and `k`. `sA` denotes the source from which player A's moves will be taken. `stdin` indicates that the moves will come from `cin`, i.e., player A's moves will be interactive. Similarly for player B.
- `quit` Ends the program

A sample interaction where both players' moves come from files is presented below. Suppose that `movesA.txt` contains the following:

```
8 6 5 8
```

Suppose that `movesB.txt` contains the following:

```
4 3 1
```

Then the interaction would be as follows:

```
game 25 7 movesA.txt movesB.txt
Total is 25, max deduction is 7
Player A's move
Total is 18
Player B's move
Total is 14
Player A's move
Total is 8
Player B's move
Total is 5
Player A's move
Total is 0
Player A wins
Score is
A 1
B 0
quit
```

You may assume that if a player's moves are taken from a file, then the file will contain enough moves to complete the game.

Within a game, play alternates between A and B. Player A plays first in odd-numbered games (starting from 1), and player B plays first in even-numbered games.

To structure this game, you must include at least the following classes:

- `ScoreBoard` which tracks the number of games won by each player, the current value of the counter, and the current deduction limit. This class must be responsible for all output to the screen, except `Player A's move` and `Player B's move` (these will be printed by code in the `Player` class, described below).

- `Player` which encapsulates a game player, and keeps track of the source from which a player is receiving input.

You must use the **singleton** pattern to ensure that there is only one scoreboard in the game.

In addition, you must generalize the singleton pattern to ensure that only two `Player` objects can be constructed.

Each `Player` object must possess a pointer to this scoreboard, and these pointers will be intialized in the way prescribed by the singleton pattern. Each player object must be responsible for registering its move with the scoreboard by calling a `ScoreBoard::makeMove` method. This method should take parameters indicating which player is calling the method, and the amount to be deducted from the total.

Your main program will be responsible for keeping track of which player's turn it is. It will call a method in `ScoreBoard` to start a game, whenever the user enters a `game` command. In addition, it will alternately call a method on the two player objects that will cause the player to get the next move from its input source and then pass that move on to the scoreboard.

The chain of method calls is therefore roughly the following:

- main program, in response to a `game` command from the user, calls a method `ScoreBoard::startGame` to initiate a game.
- main program calls a method for each of the two player objects, to pass to them their respective input streams
- main program alternately calls `Player::makeMove` for player objects A and B. This method should take no parameters.
- The `Player::makeMove` method will be responsible for fetching the next move and calling the `ScoreBoard::makeMove` method to register the move with the scoreboard.

Your solution must use `const` declarations for variables, members, and parameters whenever possible.

Your solution must not leak memory.

You may assume that all input is valid.

Due on Due Date 1: Design a test suite for this program (call the suite file `suiteq4.txt` and zip the suite into `a3q4a.zip`).

Due on Due Date 2: Full implementation in C++. Your mainline code should be in file `a3q4b.cc`, and your entire submission should be zipped into `a3q4b.zip` and submitted. Your zip file should contain, at minimum, the files `a3q4b.cc`, `scoreboard.h`, `scoreboard.cc`, `player.h`, and `player.cc`. If you choose to write additional classes, they must each reside in their own `.h` and `.cc` files as well.