CS246—Assignment 3 (Fall 2014)

Due Date 1: Friday, October 17, 11:55pm
Due Date 2: Friday, October 31, 11:55pm

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 simulate a simple hand-held calculator. A calculator has a limited amount of memory, suitable for holding the following values:

   - the current contents of the display;
   - the current computed result (with which the displayed value will eventually be combined);
   - the contents of memory (to emulate the M+/MR/MC functionality on calculators);
   - the most recently pressed operator key (a character: +, -, *, or /);
   - whether or not the most recent operation produced an error.

   You will use the provided `calc.h` file as the basis for your class. **You are not allowed to change `calc.h`.**

   Note on M+/MR/MC: Simple hand-held calculators often give you one int of memory to store a value for later use. The memory initially contains the value 0. Pressing M+ adds the displayed value to the contents of memory, and stores the result in memory (at which point an M customarily appears on the screen). Pressing MR copies the contents of memory to the display. Pressing MC clears the contents of the memory (sets it to 0).

   You are to implement the following methods:

   - Constructor and copy constructor
   - `void digit(int digit)` — adds `digit` (which must be between 0 and 9, inclusive) to the end of the display value.
   - `void op(char operator)` — sets the operator field. If the operator field previously contained an operator, it combines display and result with that operator, and stores the result. If not, display is copied to result. In either case, display is cleared (reset to 0).

- `void equals()` — combines result and display using operator, and stores the result in result and display. Clears the operator field.
- `void memPlus(), void memClear(), void memRecall()` — as above
- `bool isError()` — answers true if the error flag is set (because of division by 0).
- `void allClear()` — resets all memory to 0, clears the operator, and resets the error flag.

You are also to implement the output operator, which works as follows:

- if memory contains 0, just print out the value of display;
- if memory is non-zero print out

  value-of-display `M:` *value-of-memory*

- if the error flag is set, append `E` (a space and an E) to the above.

We have provided the file `calcMain.cc` which you can use to test your calculator class.

A sample session follows:

```
456-123+456=
789
1+1=
7892
AC
1+1=
2
M+
=
2 M: 2
/0
=
0 M: 2 E
AC
=
0
```

Note the second calculation: because the display already contains 789, the 1+1 actually appends the 1 to 789, and requests 7891+1 (7892). This was done for reasons of simplicity.

Note carefully: We have set up a special Marmoset project to help you to clarify the input and output requirements. The project is called a3q1InputTest, and it expects a file called `a3q1.in`. **It is not worth any marks,** but if you submit your input in `a3q1.in`, the result of the public test will tell you the corresponding correct output.

(a) **Due on Due Date 1:** Design a test suite for this program (call the suite file `suiteq1.txt` and zip the suite into `a3q1a.zip`).

(b) **Due on Due Date 2:** calc.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, divide and mod 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"
s4 = "babababa";
s1 = "bab";
cout << s4 / s1 <<endl;   //Will print number of times s1 appears in s4
                    //(not counting overlaps) Prints 2 for this example
s3 = s4 % s1;  // Removes the occurrences of s1 in s4
               // s3 will contain "aa"
```

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 familiar with `cin.peek()` and the `isspace` function located in the `<locale>` library may aid you in solving this question.
- In the case of the / operator, only non overlapping instances of the RHS are counted.
- In the case of the % operator, only non overlapping instances of the RHS are removed.
- The provided driver (`a3q2.cc`) can be compiled with your solution to test (and then debug) your code. Please keep in mind that the purpose of the test harness is to provide a convenient means of verifying that code you are asked to write is working correctly. Therefore, although some effort has been expended to make the harness reasonably

[1] For some definition of improved.

robust, we do not guarantee that it is perfect, as that is not the point. The test harness should function correctly if you use it as intended; it may fail horribly if you abuse it. But the point of your testing is to verify *your* code, rather than the harness, and so test cases that attempt to find flaws in the harness are not required in your test suites.

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 the progress of a tournament. A game in the tournament is played between two currently undefeated players and always results in one player being declared the winner. A new player may join the tournament at any time, unless the number of undefeated players in the tournament has already reached 20. A player is declared the winner of the tournament when that player is the only undefeated player remaining.

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

   - `! name` — Adds `name` as a player in the tournament.
   - `* name1 name2 name` — A game is played between `name1` and `name2` and the winner of the game is `name` (The winner's name is either name1 or name2)
   - `- name` — Player `name` cheated in the last game he won. `name` is kicked out and the most recent losing opponent is declared the winner.
   - `?` — Prints the names of all currently undefeated players **in the order** they joined the tournament.
   - `? name` — Prints a detailed history of all games played by the undefeated player, `name` (always prints the winner's name first, see example)
   - `include filename` Reads the file `filename` and executes the commands contained therein

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

   Implementation Notes:

   - You must use the `Node` class provided to you in `node.h`. You may add function headers to this file.
   - When a new player enters the tournament create a new Node object for this player and store it in a collection of undefeated players
   - When a game is played, a new Node is created which indicates the winner of the game and the two competitors that took part in the game. In essence you are building a forest of trees. A winner can be declared when this forest has one tree remaining.
   - 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.

- You may assume that player names do not contain spaces.

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

```
! alice
! bob
* alice bob alice
! eve
?
alice
eve
* eve alice alice
? alice
alice vs eve
alice vs bob
?
alice
- alice
?
eve
```

(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. For this problem, the classes you write must be implemented using the `class` keyword. In this problem you will write a C++ program to administer the game of SOS (`http://en.wikipedia.org/wiki/SOS_(game)`), which involves two players, A and B. Players take turns marking either an 'S' or an 'O' on an 4 by 4 grid, until all squares are filled and no spaces remain unfilled. A player scores a point by completing a line of the characters, "SOS." If a player successfully makes an "SOS" on their turn, they take an additional turn immediately. The player who has scored highest at the end of a match is the winner of the round. If scores are tied, this ends in a draw. Your program will play several rounds of this game and report the winner. A sample interactions folows (your input is in italics):

```
game stdin stdin
A's move
1 0 S
B's move
2 3 S
A's move
3 0 S
B's move
2 0 O
B's move
1 3 S
A's move
0 3 S
B's move
0 0 S
A's move
3 3 S
```

```
B's move
2 2 S
A's move
2 1 S
B's move
1 2 O
B's move
0 2 S
B's move
0 1 S
B's move
1 1 S
B's move
3 1 S
A's move
3 2 O
B wins with 5 points
Score is
A 0
B 1
quit
```

In between games, two commands are recognized:

- `game sA sB` Starts a game. `sA` denotes the name of the file from which A's moves will be taken. `stdin` indicates that the moves will come from `cin`, i.e., A's moves will be interactive. Similarly for B.
- `quit` Ends the program

Within a game, players take turns claiming squares. The 16 squares are arranged as follows:

```
(0,0) (0,1) (0,2) (0,3)
(1,0) (1,1) (1,2) (1,3)
(2,0) (2,1) (2,2) (2,3)
(3,0) (3,1) (3,2) (3,3)
```

When a player's moves come from stdin, it is considered invalid input to claim a square that has already been taken. On the other hand, when the moves come from a file, it is hard to know in advance what squares will have been taken. Thus, when the moves come from a file, a player's move is defined to be the next square in the file that is not already claimed. Note that it would be redundant for a square to occur more than once in the file, and therefore we consider any file that contains a square more than once to constitute invalid input.

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

```
0 0 S 0 1 O 0 2 S 0 3 O
1 0 O 1 1 O 1 2 S 1 3 S
2 0 S 2 1 O 2 2 S 2 3 O
3 0 S 3 1 S 3 2 O 3 3 S
```

Suppose that `movesB.txt` contains the following:

```
0 0 O 0 1 S 0 2 O 0 3 S
```

```
1 0 S 1 1 S 1 2 O 1 3 O
2 0 O 2 1 S 2 2 O 2 3 S
3 0 O 3 1 O 3 2 S 3 3 O
```

Then the interaction would be as follows:

```
game  movesA.txt movesB.txt
A's move
(plays 0 0 S)
B's move
(plays 0 1 S)
A's move
(plays 0 2 S)
B's move
(plays 0 3 S)
A's move
(plays 1 0 O)
B's move
(plays 1 1 S)
A's move
(plays 1 2 S)
B's move
(plays 1 3 O)
A's move
(plays 2 0 S)
A's move
(plays 2 1 O)
B's move
(plays 2 2 O)
A's move
(plays 2 3 O)
B's move
(plays 3 0 O)
A's move
(plays 3 1 S)
A's move
(plays 3 2 O)
B's move
(plays 3 3 O)
A wins with 2 points
Score is
A 1
B 0
quit
```

Note, in particular, that when the moves come from a file, your program prints to stdout the move that was made (e.g., `(plays 0 0 S)` above).

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. A plays first in odd-numbered games (starting from 1), and B plays first in even-numbered games.

A win is worth one point. A loss is worth no points. If a game is drawn (no one wins), then

no points are awarded for that game. In the case of a draw, print `Draw` to stdout, instead of `A wins with X points` or `B wins with X points`.

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

- `ScoreBoard` which tracks the number of games won by each player and the current state of the board. This class must be responsible for all output to the screen, except `B's move` and `A'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. If necessary, a player object may query the board by calling a method `ScoreBoard::isOccupied` to find out whether a given position is taken. This method would only be called if the player object's input comes from a file, for the purpose of determining the next unoccupied position in the file.

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 main program will call `Scoreboard::madeSOS()` which will return true if the last move made by the current Player successfully made an SOS. If it returns true then the same Player takes an additional turn.
- 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.

Debugging Hint: You might find it useful to print the configuration of the board after each move for debugging. In class we briefly discussed how such debug information can be embedded in your code and activated using arguments to the preprocessor. For more on this refer to the file `1149/lectures/c++/6-preprocess/debug.cc`. This code, when compiled using `g++ -DDEBUG debug.cc` will print out additional debug information.

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.