CS246—Assignment 3 (Spring 2015)

Due Date 1: Wednesday, June 17, 4:55pm
Due Date 2: Monday, June 29, 4:55pm

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>`, `<cassert>` 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.

Note: Sample test cases have been provided for each question. These are available under the `a3/sample-tests` directory.

Note: Starter code, when applicable, is provided under the `a3/starter-code` directory.

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. In other words, once the drone has made 50 moves, `forward`, `backward`, `left` and `right` are considered invalid input and the program behaviour is undefined.

A test harness is provided (`a3q1.cc`), by which you may interact with your drone for testing purposes. Incorrect usage of the test harness should not be part of your test cases i.e. you do not need to write tests to try to break the test harness.

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

Due on Due Date 2: Solution (`drone.h, drone.cc (do not submit a3q1.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, assignment, 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 contents to be a copy of 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`.

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

- **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.
- 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 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++. Submit the iString.h and iString.cc files as a zip file, `a3q2b.zip` (do not submit a3q2.cc).

3. When you work with a modern Unix shell like Bash, you can take advantage of several shortcuts, one of which is autocompletion: if you type the first few characters of a command, and only one command starts with those characters, then pressing TAB will fill in the rest of the command. Similarly, after typing the command, if you press TAB after typing the first few characters, the shell will fill in the (unique) name of the file in the current directory that matches those characters. (This is similar to the notorious autocompletion facility that accompanies text messaging systems.)

   On the other hand, if there are several possible completions for the input characters, pressing TAB twice yields a list of possible completions; from there, you can type enough additional characters to make it clear which one you want, and then press TAB again to complete the rest.

   In this problem, you will write a program that performs autocompletion. It will maintain a dictionary of available words, and use them to autocomplete prefixes that you provide.

   Your program will accept the following commands on stdin:

   - `+ word` — adds `word` to the dictionary
   - `- word` — removes `word` from the dictionary
   - `? word` — prints a list of possible autocompletions for `word`, in alphabetical order separate by a single space
   - `$` — prints the total number of nodes in the dictionary data structure, which is described below
   - `include filename` Reads the file `filename` and executes the commands contained therein.

   For simplicity, you may assume that words consist only of lowercase letters, with no numbers or other symbols.

   An appropriate data structure to use for this problem is a *trie*, which we describe as follows:

   A trie is an N-ary tree data-structure with a branching factor equal to the number of different characters in the associated alphabet. For simplicity, assume the dictionary only consists of lowercase letters, resulting in a branching factor of 26.

Nodes in a trie can have a child for each letter of the alphabet. Thus, a word can be identified by its location in the tree. The word "dog" is indicated by the node obtained by following the 'd' child of the root, the 'o' child of that node, and the 'g' child of that node. If the 'g' child marks the end of a word (i.e., if "dog" is a word in the trie), then the 'g' child's `isWord` field will be true. If "dog" is not a word in the trie, but "dogs" is a word in the trie, then the 'g' child's `isWord` field will be false, but that node will have an 's' child, whose `isWord` field is true.

For example, the trie in Figure 1 contains the words: "do", "doe" , "dog", "dub", "punt", and "we". Each complete word in the trie diagram is marked with an asterisk.

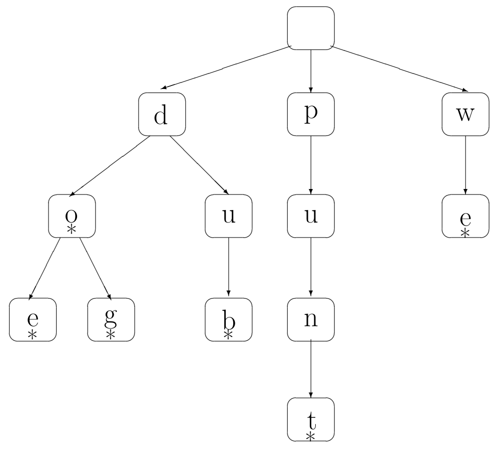

Figure 1: Trie Example

Input commands (and components of commands) are separated by an arbitrary (non-zero) amount of whitespace. The program terminates when it encounters an EOF signal on stdin.

Notes:

- Use the provided file `trie.h` as a starting point for building your trie functions, in support of your solution.
- 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.
- Removing a word from a trie is simply a matter of marking the appropriate `isWord` field as false.
- However, to save space, if a `? prefix` command is issued, and no autocompletions are found, then delete all nodes below `prefix` in the trie (if any). For example, if "punt" were removed from the sample trie, a query `? pu` would return no matches. You would then remove the two nodes below "pu" (namely n and t) from the trie.
- You must deallocate all dynamically allocated memory by the end of the program; hand-markers will be checking for this.
- Your trie must define a constructor for initialization and a destructor for proper deallocation of nodes.

(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 Tic-Tac-Toe (`http://en.wikipedia.org/wiki/Tic-Tac-Toe`), which involves two players, X and O (uppercase alphabet not the value zero). Players take turns claiming squares on a 3x3 grid, until one player has claimed a straight line of three squares (that player wins the game) or until the grid is full (and no one wins). Your program will play several rounds of this game and report the winner. A sample interaction follows (your input is in **bold**):

   game stdin stdin
   ```
   X's move
   ```
   NW
   ```
   O's move
   ```
   C
   ```
   X's move
   ```
   SW
   ```
   O's move
   ```
   N
   ```
   X's move
   ```
   W
   ```
   X wins
   Score is
   X 1
   O 0
   ```
   quit

   In between games, two commands are recognized:

   - `game sX sO` Starts a game. `sX` denotes the name of the file from which X's moves will be taken. Specifying the string `stdin` instead of a filename indicates that the moves will come from `cin`, i.e., X's moves will be interactive. Similarly for O.
   - `quit` Ends the program

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

   ```
   NW N NE
   W  C E
   SW S SE
   ```

   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 `movesX.txt` contains the following:

   ```
   NW SE NE C E N S W SW
   ```

   Suppose that `movesO.txt` contains the following:

   ```
   C SE SW E N NW NE S W
   ```

   Then the interaction would be as follows:

```
game  movesX.txt movesO.txt
X's move
(plays NW)
O's move
(plays C)
X's move
(plays SE)
O's move
(plays SW)
X's move
(plays NE)
O's move
(plays E)
X's move
(plays N)
X wins
Score is
X 1
O 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 NW)` 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. You may also assume that if a player's moves are taken from a file, then that file exists and is readable.

Within a game, play alternates between X and O. X plays first in odd-numbered games (starting from 1), and O 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 `X wins` or `O wins`. The score is printed after every game.

The quit command is considered valid input only when no game is in progress.

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 `X's move` and `O'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 `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.

Note: 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.