CS162 Assignment 4

Shoshana Abrass
abrasss@onid.oregonstate.edu
Mar 1, 2015

Discovering requirements

The requirements were given in the assignment, except for the choices and reflections documented below.

Design

Tournament classes, containers, and functions

```
Class Player
{
        List<Character*> Lineup;
        Stack<Character*>  DeadPile;

        RetireCharacter(Character *)
}
```

new functions:

- Enter the number of characters each player will have
- PlayerOne,PlayerTwo: Choose your lineup

Changes to the Character class

- Add a Character::Recover() function
  The recover function restores a percentage of strength to the character, for example (trunc)(.3 * Strength). Different values are used to give weaker characters a slight advantage during the tournament (otherwise the results are too skewed to be interesting
- Add Character::Score variable; getScore() and setScore() functions

Changes to the Tournament, main, or Combat functions

- Add Calculate Score for the winning player
  Keep this as simple as possible, I'm not a baseball nut.
  10 point for the win, plus (LoserStrengh-WinStrength) points of the loser is stronger
- Insert a Character::Recover() line

Reflection

Using a list for this assignment turned out to be a royal pain in the rear, because lists have no direct addressing mechanism. The only way to address members of a list is through an iterator,

which is a pointer to an element of list data. Of course it was a great exercise for learning more about iterators.

Since the list contained Character *s, the iterator was a pointer to a pointer, which caused some obfuscated dereferencing (see, for example, the Player::Print function). It was important to keep in mind the distinction between the iterator and the data:

list::erase(iterator)
list::remove(Character*)

It turns out that elements need to be removed with erase() in order to keep the iterator working correctly.

Printing out the winners at the end of the tournament also proved more challenging than I expected; it was awkward to search both player's Lineups and Graveyards to get all the character pointers. I ended up creating a std::set<Character *> in main; the benefit of using set is that it doesn't allow duplicate entries. I simply inserted all the Fighters into the set as they went into combat, which supports the interesting future possibility of scoring only those Fighters who have been in a bout.

From the set, at the end of the tournament, I created a list that used insert() to sort the fighters in order of their score.

I changed my scoring a little as I went through testing; my main goal was to reduce the possibility of a tie and to make the scores closer to each other (rather than having all the barbarians end up with 0, for example). Two kinds of scoring take place:

1. At the end of a Combat bout, bonuses may be awarded for certain conditions. For example, the loser gets a bonus if they held on for more than 11 rounds; the winner gets a bonus if they didn't make the first attack.
2. After the Combat bout the winning fighter gets 3 points for the win, and if the winner had less strength than the loser they get a bonus for that.

Tie-breaking: the assignment said we could leave this to random chance, so I did. The winner of the tie is determined by their placement in the std::set **`allFighters`**. Since set ordering is based on pointer addresses this is, within reason, random.

Testing

There are two types of automated test included with the assignment.

First, I've included the code for testing the Combat() function with various Character subclasses. Type

```
./stats_test
```

to run these tests (after running **`make`**).

Second, I wrote a new test function that runs a tournament without any user input, which can be started with the command

```
./test
```

This creates rosters for both players from pre-defined arrays. The starting rosters are:

Player 1	Gobl	Barb	Blue	Rept	Shad	Gobl	Barb	Blue	Rept	Shad	Blue	Barb
Player 2	Gobl	Barb	Blue	Rept	Shad	Barb	Blue	Rept	Shad	Gobl	Shad	Shad

The first five bouts insure that each type of character is paired with its own type once; the next seven bouts represent some interesting pairings. Subsequent pairings are impossible to predict since it depends on the winners of the first round of combat. Running this test several times yields a wide variety of test conditions.

Unit testing was done during development with print statements. These are disabled in the turned-in code, however they're flag-controlled and may be turned on by changing the value of DEBUG, VERBOSE and VVERBOSE in the Flags.h file.