**Submitted By Eric Miller 08/30/2014**

**Final Project Testing and Report.**

**Setting up the Program:**

1. The general program will be created as I outlined in my design document, with a minor change being the use of a deque container.
2. I will also have winners carry over their strength (and poison status) from any previous battles.
3. I had initially wanted to use a deque container for the fighter line-up, however upon getting input from the professor I will use a regular queue instead since the added features of a deque type queue (such as iterators) aren’t really needed.
4. The main combat function from the prior assignment will remain (mostly) untouched outside of changing the win/loss/victory member variables of the winner and loser at the end of combat.
5. To make the code easier to read and improve on encapsulation, I will include all battle related functions (battle menu and the actual battle function itself) in a separate header file called “**Battle.h**”. Implementation will also be coded for this header’s functions in a separate file.
6. For implementing my ranking system of wins/losses, the base samurai class will be updated in the following manner:

|  |
| --- |
| class samurai : public character//base class for all samurai  **public:**  samurai();  virtual int calculate\_defense();  virtual int calculate\_attack()  virtual void Damage(int attack, int psn);  void show\_profile();  int show\_initiative();  void change\_initative()  **void add\_win()**  **void add\_loss()**  **void add\_victory()**  **show\_(wins/loss/victory) //accessors**  **protected:**  string ryuha;  bool poison;  int initiative  **int wins;**  **int losses;**  **int victory;** |

1. Initially I will disregard rankings however and just focus on having the users select their teams, then battle until one team runs out of samurai.
2. I will adapt my previous code where I had the user select two characters, and bump up the selection amount to three. This time the selections will be stored in a queue container of type **<samurai\*>** as opposed to an array.
3. To cut down on code, I will have the selection process be its own function that accepts a queue of type **<samurai\*>** as its argument. Both teams will have their team queue container inserted into this function.
4. Depending on the victor of a battle, I will have the following code recycle a character to the back of their queue or be added to their graveyard stack:

*if (team1->show\_victory() == 1) //if player 1's team member won*

*{*

*cout<< Team 2’s fighter <NAME> has been added to the team 2 graveyard;*

*cout<<Team 1’s Fighter <NAME> has gone to the back of their roster.\n";*

*player\_2\_grave.push (player\_2\_team.front());*

*pointer destroy = player\_2\_team.front();*

*delete destroy;*

*player\_2\_team.pop();*

*team1->add\_victory(0); //reset victory flag*

*player\_1\_team.push(team1.front());*

*player\_1\_team.pop();*

*}*

*Else*

*{Reverse happens}*

1. At the end of battle, iterators will be used to loop through the team containers (queue and stack) and delete the dynamic samurai objects at each node. Then the containers themselves will be deleted.
2. I will run an initial test to make sure that data is being stored in containers properly and that the battles are working.

Testing: **Test A: For this test I will see if variables are being properly stored in there containers and sorted properly depending on the results of each battle. Rankings will not be calculated.**

|  |  |  |
| --- | --- | --- |
| Testing Conditions | Expected Output | Actual Output |
| I will select random characters to make up two teams of 3, which will then battle each other.  The size of the team queues and graveyard stacks will be output after each combat round to make sure that nodes are being sorted properly depending on who won.  I will be watching for late binding on node variables so that individual samurai characters will be utilized when called into a function, winners are cycled to the back of their queues, and losers are added to their graveyard stack.  Test 1-5: Run combat between 2 teams, taking note of the results and flow of characters in the team containers. | Desired characters will be put in the proper order into a queue. They then will be called out one at a time into the battle function and utilize late binding for their special abilities.  The loosing character goes into a stack while the winner goes to the back of their queue.  Characters carry over any damage or poison from prior battles.  Battle ends when one player’s queue size becomes zero. | Test 1: Everything went as expected until the very end. Special abilities were used, the combatants were sorted into the proper containers after a battle, etc. However when attempting to DELETE the dynamic variable in the stack containers, the program crashed. FIXED (see notes)  Tests 2-5: as expected. |

**Notes: Clearing out dynamic variables did not work with the stack container (although a queue was no problem). Because (unlike the queue container) the in/out order isn’t important for the graveyard, I have decided to use a ‘vector<samurai\*>’ type container for the graveyards instead. This seemed to fix the problem, although it does deviate from my original plan.**

**Otherwise it seems to be working mostly fine. I will now need to look into calculating scores for the winners.**

1. For calculating an overall winner, I will first have the program put any remaining queue nodes into the graveyard (now a vector). Then I will use an iterator to tally up the wins/losses for all the characters contained therein.
2. As a review: Wins = wins-losses. Ties are settled by chance (50/50).
3. First I shall contain the winners and their names in 3 sets of int/string variables.
4. Each team member is then cycled through to place them in a ranking position.
5. This is first done for team 1, then for team 2. Any new rankings cause the previous holder to go down a ranking.
6. It will go something like this (example for 1st place):

*while (iterator != player\_1\_dead.end())*

*{*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*First place*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*pointer adjust = iterator;*

*if (adjust->show\_wins() > player\_1\_rank1)*

*{*

*player\_1\_rank3 = player\_1\_rank2; //the prior #2 fighter is bumped down to #3*

*player\_1\_rank3\_name = player\_1\_rank2\_name;*

*player\_1\_rank2 = player\_1\_rank1; //the prior #1 fighter is bumped down to #2*

*player\_1\_rank2\_name = player\_1\_rank1\_name;*

*player\_1\_rank1 = adjust->show\_wins();*

*player\_1\_rank1\_name = "Player 1 : " +adjust->show\_name();*

*}*

*else if (adjust->show\_wins() == player\_1\_rank1)*

*{*

*random = rand()%1;*

*if (random ==1)*

*{*

*player\_1\_rank1 = adjust->show\_wins();*

*player\_1\_rank1\_name = "Player 1 : " +adjust->show\_name();*

*player\_1\_rank3 = player\_1\_rank2; //the prior #2 fighter is bumped down to #3*

*player\_1\_rank3\_name = player\_1\_rank2\_name;*

*player\_1\_rank2 = player\_1\_rank1; //the prior #1 fighter is bumped down to #2*

*player\_1\_rank2\_name = player\_1\_rank1\_name;*

*}*

*Else*

*{*

*player\_1\_rank2 = adjust->show\_wins();*

*player\_1\_rank2\_name = "Player 1 : " +adjust->show\_name();*

*player\_1\_rank3 = player\_1\_rank2; //the prior #2 fighter is bumped down to #3*

*player\_1\_rank3\_name = player\_1\_rank2\_name;*

*}*

1. At the end the top three rankings should be set.
2. Each team’s ranking points are calculated, then a winner is declared.

Testing: **Test B: For this test I will see if the ranking structure is working. I will artificially set the strength of the Hattori hanzo character to ‘0’ so I can control losses. I will also set Shusaku to strength ‘1000’ so that I can artificially control wins.**

|  |  |  |
| --- | --- | --- |
| Testing Conditions | Expected Output | Actual Output |
| As outlined above I will have Hanzo be a guaranteed loss, and Shusaku be a guaranteed win. This lets me set up the following scenario:  Test 1:  Team 1:  Hanzo (0 wins)  Hanzo (0 wins)  Yagyu (1 win, 1 loss, 0 net wins)  Team 2:  Shusaku (2 wins)  Shusaku(1 win)  Hanzo (0 wins)  Tests 2-6: Random teams. I will write down on note paper the results and see if the rankings match. | Characters have a net win count of wins-losses.  The top three characters by net win count then are output to the screen.  If two characters have the same win count, there is a 50/50 draw to see which one will come out on top.  The team with the most characters in the top 3 rankings win.  Tests 2-6: I will keep tally of rankings throughout the match, then compare my tally with the program results. | Test 1: The results don’t seem to match at all and are quite jumbled. I am thinking that I am making this too complicated. I will redesign the ranking part of the program.  TESTS 2-6 CANCELLED. |

1. I’ve decided to redo the ranking.
2. The first change is that I will just count wins and ignore factoring in losses.
3. The second will be to create a special struct object to hold character data (name/win count).
4. I will have an array of this struct type to hold data for all of the nodes.
5. Then (after clearing the dynamic nodes out) I will sort this new array by win count.
6. The setup will look similar to the following:

*struct ranking //holds name/win data of the fighters*

*{ string player\_rank\_name;*

*int player\_rank\_wins; };*

*ranking finale[6]; //array for holding ranking structs, 1 for each fighter*

*zanshin = player\_1\_dead.begin();*

*for (int i = 0; i < 3; i++) //inputs team 1 name/win data into an array to be sorted*

*{*

*adjust = \*zanshin;*

*finale[i].player\_rank\_name = "Player 1" + adjust->show\_name();*

*finale[i].player\_rank\_wins = adjust->show\_wins();*

*zanshin++;*

*}*

*zanshin = player\_2\_dead.begin();//inputs team 2 name/win data into an array to be sorted*

*for (int i = 3; i < 6; i++)*

*{*

*adjust = \*zanshin;*

*finale[i].player\_rank\_name = "Player 2" + adjust->show\_name();*

*finale[i].player\_rank\_wins = adjust->show\_wins();*

*zanshin++;*

*}*

1. This sets up an array that is easy to sort, and I no longer have to worry about keeping track of memory leaks.
2. With this structure I will retest the ranking system.

Testing: **Test C: Redo of test B with the new ranking system in place.**

|  |  |  |
| --- | --- | --- |
| Testing Conditions | Expected Output | Actual Output |
| Test 1:  Team 1:  Hanzo (0 wins)  Hanzo (0 wins)  Yagyu (1 win)  Team 2:  Shusaku (2 wins)  Shusaku(1 win)  Hanzo (0 wins)  Tests 2-6: Random teams. I will write down on note paper the results and see if the rankings match. | Characters are sorted by win count.  The top three characters by win count are then output to the screen.  If two characters have the same win count, there is a 50/50 draw to see which one will come out on top.  The team with the most characters in the top 3 rankings win.  Test 1:  Final Rankings:  Team2 Shusaku,  Tie between Team 1 Yagyu and Team 2 Shusaku  Tests 2-6: I will write down a tally of rankings throughout the match, then compare my tally with the program results to make sure that they are being calculated properly. | Test 1: As expected.  Tests 2-6: As expected. |

1. Now both the fighting list system and ranking system seem to be up and running.
2. With this I shall conclude this project.

**Reflections**:

This project took up quite a bit of time with the ranking system. In the end the solution was to make everything simpler- I was thinking too hard about it. I also wonder if I made a mistake with the data structures that I used when making the planning document. A stack with first in/last out data flow could potentially have been implemented, but a vector (which is what I ended up using after trying the stack) did the job just as well.

In the end I should have worked on planning just a little bit more before coding. While planning does take time, the time it potentially end up saving from having to redo code really adds up.

For the end program itself, all in all it is fine. Teams and rankings work like they are supposed to and the final project goals are achieved. If I had a bit more time though here is what I would have liked to have added:

-**A better ranking system**. I don’t really care for the one I ended up with as there are too many ties. A solution might be either to increase the number of combatants (more matches could mean more wins), or solve ties by ‘sudden death’ matches.

**-More differentiation between characters and better balancing**. We were warned in this assignment not to get too carried away so I stuck with the main task at hand, but it would have been nice to play around with the character stats more as well as adding in more abilities.

-**More compact code.** In particular, I noticed that the battle function basically repeats itself twice with the attacking/defending roles reversed. Perhaps I should have had a separate function that the combatants are plugged into based on turn order.

Regardless, that is for another time and place. Thank you for taking the time to grade this program (and all of my program up until now), and please have a good break this September.