Assignment 3 Kaylene Lim
CS 362 July 21, 2019

1. Unit Testing

unittest1.c

Output:

```
--- UNIT TEST 1: refactBaron ---

>>> Testing for ability to run function... program runs

>>> Testing for increased buys... FAILED... buys decreased
instead

>>> Testing for coin increase... FAILED... coins did not
increase

>>> Testing for coin increase by 4... FAILED... coin did not
increase by 4

--- END OF TESTING for refactBaron ---
```

Description: Unit tests the function *refactBaron*. Tests the functions ability to be called and run by asserting it running when an integer 0 is returned at the end of the function. Then tests if the number of buys are increased once a baron card is played. This was one of the bugs I had introduced in assignment 2. This was tested by setting a value for numBuys and seeing if the value increased after the function ran. Next, the number of coins needed to be increased. This testing was similar to the numBuys in which a value was set and checked if it was greater than the initialized value after the function ran. Finally I needed to test if the coins increased by 4 if it increased at all. This test is not really useful if the test for increasing coins failed but it is to ensure that the increment of coin increasing is correct. If I had more time I would have liked to test for whether or not the correct number of cards were discarded and if they were correct estate cards as these are trickier tests to write but would aid in ensuring the baron function is functioning as it should be.

unittest2.c

Output:

```
--- UNIT TEST 2: refactMinion ---
```

```
>>> Testing if action points are increased... PASSED... action
points increase

>>> Testing if coin increases... PASSED... action points
increase

Segmentation fault (core dumped)
```

Description: The minion function is extremely buggy and it was difficult to write a unit test that did not segfault. Currently the unit test does completely seg fault, but due to time constraints and lack of checking in on GitHub regularly I did not have the chance to find the cause and fix for the fault. The output above was the most recent *working* output before the whole thing seg faulted. The unit test looked if the action points of the player were increased by comparing the final action point value to before and after the function ran. Same technique for the coins. I also wrote a test for making sure the next player is the net player since that was one of the bugs I introduced in assignment 2. This was particularly difficult to test as the currentPlayer variable was local to the function. The test looked at the whoseTurn variable and compared it to the one after the function ran, if they were the same the test failed. Another test I wrote but could not get to work was testing if the handsize of players that needed to redraw. It would have checked the handsize of all the players before and after the function, if the players that redrew had a handsize other than greater than 4 the test would have failed.

unittest3.c

Output:
```
--- UNIT TEST 3: refactAmbassador ---

>>> Testing if correct return value for choice 2... PASSED...
correct return value
PASSED... correct return value

>>> Testing if correct return value when choice 1 = handPos...
PASSED... correct return value

>>> Testing if card was discarded from hand... FAILED... did
not discard cards

--- END OF TESTING for refactAmbassador ---
```

Description: This unit test examined the ambassador function. First it tested how the function handled choice value inputs. The first one tested that if a choice2 value was between 0 and 2 it would return a -1. I tested the boundary cases for this one where choice2 equaled 0 and 2. Next I tested if the correct return statement would come when choice1 had the same value as handPos. This was done by seeing the return when the two values were the same. With more time, I would have tested more edge cases for this test. Next I tested if cards were discarded by comparing the number of cards in hand before and after the function was called. If the value was less than cards were correctly discarded otherwise the test failed.

unittest4.c

Output:
```
--- UNIT TEST 4: refactTribute ---

>>> Testing action points... PASSED... action points are
increased

>>> Testing coin increased... PASSED... coins are increased

Segmentation fault (core dumped)
```

Description: This test takes a look at the tribute function. There were many tests that I could think of for this including verifying the current player, testing which cards are revealed, testing which cards are discarded, and if the deck count is correct. Unfortunately I couldn’t get any of the tests to work and they constantly seg faulted. This may be due to a bug that I introduced in the function which altered the way a function was called. The two tests that did work at one point were testing the number of action points and the number of coins. These were both done by comparing the before and after values of the action and coin, respectively.

unittest5.c

Output:

```
--- UNIT TEST 5: refactMine ---

Segmentation fault (core dumped)
```

Description:

Kaylene Lim
July 21, 2019

2. Bugs

   The unit testing for the refactored baron code provided feedback on where buys were faulting when indicating that numBuys were not increasing as they should have been:

```
state->numBuys--;//Increase buys by 1!
```

   The root cause of this bug was a typo in setting the increment into a decrement. This was an easy catch since it failed a straightforward performance test when it behaved opposite of what it needed to.

   Another bug was the increase of coin which also meant that the coins were not increasing by 4.

```
if (state->hand[currentPlayer][p] == estate) {//Found an estate card!
             // add bug here, remove addition of +4 coins when baron card is
             state->discard[currentPlayer][state->discardCount[currentPlayer]]=
                state->hand[currentPlayer][p];
             state->discardCount[currentPlayer]++;
```

   This bug was caused by a missing line in the code where the coin increase never happened.

3. Code Coverage

   Code coverage was difficult to do and many of my tests did not exceed the 30-40% range which is definitely not enough to be deemed a thorough test suite. This can be attributed to a lack of varied tests and test cases and a lack of more tedious tests that involve going into different functions outside of the ones given. WIth more time I would definitely work to increase the coverage to at least 80% or within that range before deeming the test suite ready.