1. Unit Testing

unittest1.c

Output:

```
=== UNIT TEST 1: refactBaron ===

>>> 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

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

=== END OF TESTING for refactBaron ===
```

Description:
Unit test 1 tests refactBaron. First, the test covers whether or not the number of buys increase or decrease when the baron card is played. This was a bug that I introduced in assignment 2 and this test expressed that the bug did in fact affect game play. Next I tested if the coin amount increases after baron is played since that was the other bug that I introduced in assignment 2. Next the test looks if the refactBaron function runs to completion. The game is initialized in the unit test and all the assert statements printed to the screen were customized assert code like the one below:

```
// Testing buys
printf("\n>>> Testing for increased buys... ");

refactBaron(choice1, teststate);
if(teststate.numBuys > 2){
   printf("PASSED... increased buys\n");
}else if(teststate.numBuys == 2){
   printf("FAILED... buys stayed the same\n");
}else if(teststate.numBuys < 2){
   printf("FAILED... buys decreased instead\n");
}
```

The results of unit test 1 indicated that there is an issue with the buys and coin incrementation. But, the unit test indicated that the function executes properly as there are no breaks or other returns before the end of the function.

unittest2.c

Output:

```
=== UNIT TEST 2: refactMinion ===

>>> Testing if coin increases... FAILED... coin amount did not increase

>>> Testing if action points are increased... FAILED... action points did not
    increase

>>> Testing if hand size is greater than 4... FAILED... drawing when hand size
    other than greater than 4

>>> Testing if function runs... PASSED... function runs

=== END OF TESTING for refactMinion ===
```

Description:

Unit test 2 takes a look at refactMinion and this test initializes the game the same as unit test 1. This minion code was extremely buggy so it was difficult trying to implement a lot of the tests that I wanted to on the code since the function inherently doesn't work correctly. With that being said, the tests that I chose to implement were to see if the coin and action values increase, if the hand size of players who draw are greater than 4 (a bug I introduced in assignment 2), and if the function runs. I chose these tests since they had a straightforward effect to a certain variable and also affected gameplay. The structure and the way the tests were asserted were the same as in unit test 1. The results of unit test 2 indicated that there was another error in coin and action incrementation. There is also an issue with the players' hand size when drawing cards. The results also indicated that the function executes completely despite these bugs mentioned earlier.

unittest3.c

Output:

```
=== UNIT TEST 3: refactAmbassador ===

>>> Testing if correct return value for choice 2... 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:
Unit test 3 examined the refactAmbassador function. The specific tests I made related heavily to what the overall function returned depending on a certain test case. I believe that I could have done a lot more to explore a larger pool of test cases but for the scope of this assignment, I just wanted to make sure that the code executed as expected and that there weren't any blatant errors that could be fixed easily. This unit test looked at the return function for when choice 2 had a value between 0 and 2, it also looked into the return value of when the value of choice 1 was equal to handPos, and if the card was discarded properly. The results of the test indicated that, with the limited test case that I used, the return values were correct and that the function would output the correct value. The results also suggested that there was an error in discarding when the card was used. This was a bug that I introduced which is a line that is missing the call to discard the card. To improve this unit test, further boundary cases and inputs need to be examined.

unittest4.c

Output:

```
=== UNIT TEST 4: refactTribute ===

>>> Testing action points... FAILED... action points are not increased

>>> Testing coin increased... FAILED... coins are not increased

>>> Testing if function runs properly... PASSED... function runs properly

=== END OF TESTING for refactTribute ===
```

Description:
Unit test 4 examined refactTribute. Again this unit test initialized the game the same way as all the other unit tests did. Similar to unit test 1 and 2, this unit test takes a look at how the action points and coins are affected by the function and if the entire function executes. The results of the test indicated that the action points and coins did not increase as expected, which may be due to a syntax error in the dominion source code, although it may be more convoluted than that since there are multiple ways for action points and coin values to change in the tribute function. I found a hard time covering this function since there were so many moving parts. I think if given the chance, I would like to increase the branch coverage for this function since this card does a lot. I also found it hard to approach all the different if statements in a straightforward way.

unittest5.c

Output:

```
=== UNIT TEST 5: refactMine ===

>>> Testing correct return when given choice2 value...
PASSED... correct return value


>>> Testing correct return value based on hand value.. PASSED... correct return
    value

>>> Testing if the function runs properly... FAILED... error in running
    function

=== END TESTING for refactMINE ===
```

Description:
This test looked at the mine function. All of the tests checked if the function will return the correct value under certain cases. Since most of the variables that are compared in the

if-statements are accessible to the unit test, many of these tests were done simply by setting the value of the gameState data and then seeing if the function did indeed return the value that was expected. Some bugs introduced in this function were difficult to find since they dealt with local variables or were just certain lines completely missing. The unit test showed that the function did not execute as expected and did not give a return value of 0 when the function ran to completion. This may be attributed to the first bug found which affected the return values.

cardtest1.c

Output:

```
=== CARD TEST 1: initializeGame ===

>>> checking if function runs properly... PASSED


>>> checking if numPlayers is initialized... PASSED


>>> checking if kingdomCards is initialized... PASSED


>>> checking if randomSeed is initialized... PASSED

=== END CARD TEST 1 ===
```

Description:
Card test 1 looked into the initializeGame function. First I wanted to make sure that the function executed, since if it did not, that would make other testing difficult. This was done by asserting the correct return value of the function which was pretty easy to do. Next I wanted to check that all the variables needed for this function to run was valid by checking the values of each variable. All those tests passed which indicated that this function ran as expected.

cardtest2.c

Output:

```
=== CARD TEST 2: shuffle ===

>>> checking if function runs properly.. PASSED

>>> checking player variable... PASSED

=== END CARD TEST 2 ===
```

Description:
This unit test examined the shuffle function. First, the test ensures that the function executes and runs properly. This was done by asserting that the return value of the function was correct. The result showed that the function returned a 0 which passed the test since the function ran as expected. Next I checked the variables needed to run the function by asserting that the variable had an existing value. This passed therefore the function was running as expected.

cardtest3.c

Output:

```
=== CARD TEST 3: endTurn ===

>>> checking that function runs properly... PASSED

>>> checking that hand count is reset... PASSED

=== END CARD TEST 3 ===
```

Description:
This unit test performs identically to card test 1 and card test 2 except it examines the endTurn function. Again I tested if the function executed properly by asserting the function's return value which the test concluded was correct. I also checked if the hand count was reset to zero using an assert function. The unit test result indicated that the hand count was reset which passed the test.

cardtest4.c

Output:

```
=== CARD TEST 4: getWinners ===

>>> checking that function runs properly... PASSED

>>> checking that players is initialized... PASSED

=== END CARD TEST 4 ===
```

Description:
Card test 4 runs almost exactly the same as card test 2 and examined the getWinners function. Again I ensured that the function executed properly which it did and I checked that the variables needed for the function had an existing value, which it did.

cardtest5.c

Output:

```
=== CARD TEST 5: drawCard ===

>>> checking if function runs properly... PASSED

=== END CARD TEST 5 ===
```

Description:
The drawCard function was fairly straightforward and I thought that asserting that the function executed would be enough to show that the function worked properly.

Overall

It was simpler and cleaner to test for return values or compared variables that were accessible globally. This allowed for a conclusive yet general result that would allow for straightforward code and straightforward feedback. Many of the things I chose to test were very specific in that it would either work or not work therefore I targeted aspects of the function/card that needed to behave a certain way. I believe that this provided a good start towards debugging and unit testing the functions but with more experience and time, I believe more nitty gritty components of gameplay could potentially be examined and more exhaustive testing can be performed.

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.

In the refactored minion code, it was found that the coin and the action points did not increase. The code snippet of where the coin value increases is:

```
   if (choice1)            //+2 coins
   {
       state->coins = state->coins + 2;
   }
```

In this snippet, it is expected that the coin value would increase since the choice1 value was set to 1 but the unit test for this failed indicating that this if statement is not executing. This can be fixed by changing the if statement to something that compares the choice1 value to. This also may be a bug that is due to a bug that isn't shown in the unit test but was introduced in assignment 2 where the next player and the current player are the same:

```
       int nextPlayer = currentPlayer;  // add bug here, nextPlayer does not actually
                                                go to next player
```

Next, the action points in this function also did not increase as expected. This is where the action points increase:

```
       //+1 action
       state->numActions++;
```

I believe this is also caused by the bug introduced in assignment 2 which changes the values of the wrong player.

Another bug found was that indicated that the hand size of players redrawing had a hand size greater than 4 which occurs here:

```
//other players discard hand and redraw if hand size > 4
for (i = 0; i < state->numPlayers; i++)
{
	if (i != currentPlayer)
	{
		if (state->handCount[i] > 5)  // add bug here, changed 4 to 5
		{
			//discard hand
			while (state->handCount[i] > 0)
			{
				discardCard(handPos, i, state, 0);
			}
				//draw 4
			for (j = 0; j < 4; j++)
			{
				drawCard(i, state);
			}
		}
	}
}
```

This was caused by changing the value so players that have a hand size greater than or equal to 5 would redraw, therefore players with a hand size of 4 would wrongly redraw.

Along with these bugs, there were a lot of other bugs which already existed in the minion function that were very difficult to find and fix and may have caused the bugs that I found to come about.

In the refactored ambassador function, there was a bug found which showed that the cards were not discarded properly. This is because the discardCard function was never called shown here:

```
//trash copies of cards returned to supply
	for (j = 0; j < choice2; j++)
	{
		for (i = 0; i < state->handCount[currentPlayer]; i++)
		{
			if (state->hand[currentPlayer][i] ==
state->hand[currentPlayer][choice1])
			{
				// add bug here, remove discardCard(i, currentPlayer,
state, 1); making the card not discarded
				break;
			}
		}
	}
```

In the refactored tribute function, again the coin and action point increase did not work since there the branches which would increase those values were never called, shown here:

```
for (i = 0; i <= 2; i++) {
		if (tributeRevealedCards[i] == copper || tributeRevealedCards[i]
== silver || tributeRevealedCards[i] == gold) {//Treasure cards
			state->coins += 2;
		}

		else if (tributeRevealedCards[i] == estate ||
tributeRevealedCards[i] == duchy || tributeRevealedCards[i] == province ||
tributeRevealedCards[i] == gardens || tributeRevealedCards[i] == great_hall)
{//Victory Card Found
			drawCard(currentPlayer, state);
			drawCard(currentPlayer, state);
		}
		else {//Action Card
			state->numActions = state->numActions + 2;
		}
	}
```

The other bugs found were not as straightforward where they could be traced back to syntax errors. With the existing bugs being in the source code already, it was hard to tell whether or not the bugs were a symptom of a bigger issue on something that I did not test. I believe with more time, I would want to explore more test cases and work on increasing my coverage through the functions. I also want to figure out how to better test less straightforward aspects of each function.

3. Code Coverage

Statement Coverage - a whitebox testing technique that looks at whether or not all the written statements are executed when the code is executed. This technique shows how much of the code is actually observed to be working when looking at it line by line. During my testing, the line coverage I obtained was about 20% of 577 of the dominion source code per unit test. This is a very low coverage but since each unit test only cover one function, it may actually be a pretty good coverage of the overall source code when used all together. This still may be increased by going through more test cases per function which will increase the statements that will be executed.

Branch Coverage - a testing method which examines how many decisions points are executed which shows that all the different paths of code can be reached. These 'branches' are made when there are decisions made like if-statements, function calls, or other conditional statements. In the coverage report, it shows that for each function, about 20% of the branches in dominion source code was made. I would say that since each unit test examines different branches, this shows pretty good coverage for a unit testing suite. Again, with more test cases provided, the coverage could improve.

Boundary Coverage - a testing method which examines the use of boundary cases during testing. The coverage report gcov provided did not provide a boundary coverage report, but as I stated in the unit test descriptions, boundary cases could really be improved. All of the test cases that I used during testing were mid cases to make it easy for me to predict. This does not properly cover end behaviors and edge cases which is really important when testing for bugs.

Overall I believe my coverage can be greatly improved with more exhaustive testing. If more cases are observed with more test cases used, coverage would definitely increase which would heavily improve my test suite.