# Approach

I am testing the buyCard function through random unit tests. Currently, on hundred random tests are done. In the beginning, a random sequence of the kingdom cards is recorded. The two players take turns buying the next card in the sequence until the test case is complete. Every turn produces a string like the following: “3-1: $=4, B(13 = 4)=0, $=0, OKAY BI-------!0, 0 BD-------!10, 10 BNI-------!0, 0 BND-------!10, 10”. The structure is as follows:

* A-B:
  + A is turn number.
  + B is Player number.
* $=C,
  + C is how much money the player had at the beginning of the turn.
* B(D = E)=F,
  + D is the enum value of the card type chosen to buy.
  + E if the cost of the card.
  + F is the result of the buy action, typically 0 for success and -1 for failure.
* $=G,
  + G is how much money the player had at the end of the turn.
* OKAY
  + This is “OKAY” when the action taken is correct, else “BAD-------!” is printed.
* BI-------!H, I
  + Indicates that the player total card count did not increment correctly. H and I are the values of the card count before and after. (Bad Increment)
* BD-------!J, K
  + Indicates that the supply card count did not decrement correctly. J and K are the values of the supply count before and after. (Bad Decrement)
* BNI-------!L, M
  + Indicates that the player total card count did not stay the same. L and M are the values of the card count before and after. (Bad Non-Increment)
* BND-------!N, O
  + Indicates that the supply card count did not stay the same. N and O are the values of the supply count before and after. (Bad Non-Decrement)

Segfaults and crashes are not handled. The proper decrementation of the money and whether or not a card can be bought is well tested. However, the decrementation or incrementation of card count still needs to be verified if the test is correct.

# Code

#include "dominion.h"

#include <stdio.h>

#include <assert.h>

#include <time.h>

#define TESTS 100

int main **()**

**{**

srand**(**time**(NULL));**

int result **=** 0**;**

int before **=** 0**;**

int cost **=** 0**;**

int after **=** 0**;**

int bsupply **=** 0**;**

int asupply **=** 0**;**

int btot **=** 0**;**

int atot **=** 0**;**

int x **=** 0**;**

struct gameState G**;**

struct gameState G2**;**

int k**[**10**]** **=** **{**adventurer**,** council\_room**,** feast**,** gardens**,** mine**,** remodel**,** smithy**,** village**,** baron**,** great\_hall**};**

int buytry**[**TESTS**];**

**for(**x **=** 0**;** x **<** TESTS**;** x**++)**

**{**

buytry**[**x**]** **=** k**[**rand**()%**9**];**

**}**

printf**(**"BEGIN\n\r"**);**

**if((**result **=** initializeGame**(**2**,** k**,** 5**,** **&**G**))** **==** 0**)**

**{**

printf**(**"initializeGame OKAY\n"**);**

**}**

**else**

**{**

printf**(**"initializeGame BAD-------!%d\n\n"**,**result**);**

**}**

memcpy **(&**G2**,** **&**G**,** **sizeof(**struct gameState**));**

**for(**x **=** 0**;** x **<** TESTS **&&** **!(**isGameOver**(&**G**));** x**++)**

**{**

result **=** whoseTurn**(&**G**);**

printf**(**"%d-%d:\t"**,**x**,**result**);**

**switch(**result**)**

**{**

**case** 0**:**

btot **=** fullDeckCount**(**result**,** buytry**[**x**],** **&**G**);**

bsupply **=** supplyCount**(**buytry**[**x**],** **&**G**);**

printf**(**"$=%d, \t"**,** before **=** G**.**coins**);**

printf**(**"B(%d = %d)=%d, \t"**,** buytry**[**x**],** cost **=** getCost**(**buytry**[**x**]),** result **=** buyCard**(**buytry**[**x**],** **&**G**));**

printf**(**"$=%d, \t"**,** after **=** G**.**coins**);**

atot **=** fullDeckCount**(**result**,** buytry**[**x**],** **&**G**);**

asupply **=** supplyCount**(**buytry**[**x**],** **&**G**);**

**if(**result **==** 0**)**

**{**

**if(**before **!=** **(**after**+**cost**))**

**{**

printf**(**"BAD-------! "**);**

**}**

**else**

**{**

printf**(**"OKAY "**);**

**}**

**if(**btot **!=** **(**atot**-**1**))**

**{**

printf**(**"BI-------!%d, %d "**,**btot**,** atot**);**

**}**

**if(**bsupply **!=** **(**asupply**+**1**))**

**{**

printf**(**"BD-------!%d, %d "**,** bsupply**,** asupply**);**

**}**

**}**

**else**

**{**

**if(**before **!=** after **&&** before **<** cost**)**

**{**

printf**(**"BAD-------! "**);**

**}**

**else**

**{**

printf**(**"OKAY "**);**

**}**

**if(**btot **!=** atot**)**

**{**

printf**(**"BNI-------!%d, %d "**,**btot**,** atot**);**

**}**

**if(**bsupply **!=** asupply**)**

**{**

printf**(**"BND-------!%d, %d "**,** bsupply**,** asupply**);**

**}**

**}**

endTurn**(&**G**);**

**break;**

**case** 1**:**

btot **=** fullDeckCount**(**result**,** buytry**[**x**],** **&**G**);**

bsupply **=** supplyCount**(**buytry**[**x**],** **&**G**);**

printf**(**"$=%d, \t"**,** before **=** G**.**coins**);**

printf**(**"B(%d = %d)=%d, \t"**,** buytry**[**x**],** cost **=** getCost**(**buytry**[**x**]),** result **=** buyCard**(**buytry**[**x**],** **&**G**));**

printf**(**"$=%d, \t"**,** after **=** G**.**coins**);**

atot **=** fullDeckCount**(**result**,** buytry**[**x**],** **&**G**);**

asupply **=** supplyCount**(**buytry**[**x**],** **&**G**);**

**if(**result **==** 0**)**

**{**

**if(**before **!=** **(**after**+**cost**))**

**{**

printf**(**"BAD-------! "**);**

**}**

**else**

**{**

printf**(**"OKAY "**);**

**}**

**if(**btot **!=** **(**atot**-**1**))**

**{**

printf**(**"BI-------!%d, %d "**,**btot**,** atot**);**

**}**

**if(**bsupply **!=** **(**asupply**+**1**))**

**{**

printf**(**"BD-------!%d, %d "**,** bsupply**,** asupply**);**

**}**

**}**

**else**

**{**

**if(**before **!=** after **&&** before **<** cost**)**

**{**

printf**(**"BAD-------! "**);**

**}**

**else**

**{**

printf**(**"OKAY "**);**

**}**

**if(**btot **!=** atot**)**

**{**

printf**(**"BNI-------!%d, %d "**,**btot**,** atot**);**

**}**

**if(**bsupply **!=** asupply**)**

**{**

printf**(**"BND-------!%d, %d "**,** bsupply**,** asupply**);**

**}**

**}**

endTurn**(&**G**);**

**break;**

**default:**

printf**(**"whoseTurn BAD-------!\n"**);**

**}**

printf**(**"\n"**);**

**}**

printf**(**"END\n\r"**);**

**return** 0**;**

**}**

# Results

The buyCard function was ran exactly the same times as the number of tests. This is to be expected. However, according to Gcov, the process only succeded 16% of the time, not testing success well.

## Randb

My implementation of buy card handles the money well. However, most times it does not decrement the supply (BD), increment the players card count (BI), or does affect the deck when it should not (BNI).

## Olsojeff

This code handle’s the money correctly, However, a few BI and BNI errors came up (~25%).

## Parkan

This code handle’s the money correctly, However, a few BI and BNI errors came up (~25%).

## Westbyb

This buyCard always returns 0. Every test throws a BD error, or BI and BD. Money was handled vaguely correctly if the card could be bought. However failure still returns 0, same as success. The supply count is never decremented and the deck count is never incremented. The money count can also go negative.

# Future

The randomness of the testing for now is sound. However, it is not the most intelligent testing. Later, the tests will incorporate more controlled game states and automatic test repeating to make sure that the generated test cases are repeatable and will produce the same output. Early testing of this indicated that they may not be. One more test is to see what happens when a card type not in the game is bought. The switch case also seems unnecessary and is an obstacle in expanding to larger-player games. It may also be beneficial to be able to declare number of players and length of test in the test program arguments.