Weekly Programming Quiz 7: NEUchre

CS 5002
Released: November 3, 2017
Due: November 17, 2017

Abstract

In this quiz, you are going to implement some key components to a card game. The card game is NEUchre (pronounced "N-E-U-ker"), which is a 2-player version of a game called Euchre (pronounced "U-ker"). There are two parts to this project, and many options for a challenge. Part 1 requires you to implement some data structures for the game. Part 2 requires you to implement some logic to support the game play.

Once you've implemented these two parts, you can play a simple version of the game. Solving some of the Challenges at the end of this assignment will build on the game and make it more fun!

Basic rules of NEUchre

NEUchre is a 2-player game. The player with the most points after 5 rounds wins. A player wins a round by "collecting the most tricks". A round consists of each player being dealt 5 cards, choosing a trump (a suit designated as the highest/most powerful for this round), and then each player takes turns playing cards for a "trick". A player wins a trick by playing the highest value card. There are a couple of rules that must be followed for playing a card: the first player can play any card, but the second player must "follow suit": if they have a card of the same suit, they must play it (but they can choose which card they want to play). Further, a card of the trump suit is higher than all other cards in the deck. For example, if Spades is trump, a 9 of Spades is higher than an Ace of Diamonds. Within a suit, including trump, the face value of the card is highest, with Aces being the highest value card.

In this version of the game, Player 1 (the computer) ALWAYS goes first and leads the first card.

The deck used for NEUchre is a subset of the traditional 52-card deck. It includes the Ace, King, Queen, Jack, 10 & 9 of all four suits (Spades, Diamonds, Hearts, and Clubs).

We've provided support code to play the game; you need to fill in some very specific pieces of code.

Files Provided

This time, instead of just providing a test file, we're providing a run file to allow you to actually play the game. The test file will run the tests, providing feedback about your implementation of each function, and the play target will play the game.

There is also a source file with some helper methods that you don't need to implement yourself (**`quiz7_helpers.c`**).

To play the game:

```
make play
./play
```

To run the tests:

```
make test
./test
```

Details

You'll see that in `quiz7.h`, we've begun to define some data structures for you. `Cards` have `Suits`, `Names` and `Values`. `Suits` have both a `Name` and a `Color`. A `Deck` is a Stack of `Cards`, and a `Hand` is a doubly-linked-list of list nodes called `CardNodes`.

There are two basic data structures you need to implement: the `Deck` and the `Hand`. In addition, we have already defined `Cards` for you.

Here is a summary of how the three structures are intended to work in regards to memory:

For each round of the game, you create 2 new `Hands`, deal `Cards` from the `Deck` to the `Hands`. As Players play `Cards`, the cards go back to the `Deck` (removed from the `Hand`; the `CardNode` is freed). At the end of the round, the `Hands` go away (are freed), but the `Cards` are now in the `Deck` and can be re-used for the next round.

The function `returnHandToDeck()` is a helper that takes all the `Cards` out of the `Hand` and returns them to the `Deck`.

At the end of the game, destroy the `Deck` by removing all the `Cards` from the `Deck`, free them, and free the `Deck`.

1 Part 1: Implementing the Game Data Structures

1.1 Implementing the Deck

The basic struct for `Deck` is already there, but you need to implement the `Deck` functions yourself.

Implement the `Deck` as a **`Stack`** based on an array. You are asked to implement the following stack functions that will form the stack:

- **`Deck* createDeck(Deck*)`**
  Creates the deck, initializing any fields necessary. Returns a pointer to the deck, which has been allocated on the heap.
- **`Deck* pushCardToDeck(Card*, Deck*)`**
  Adds a card to the top of the deck. Returns a pointer to the deck.
- **`Card* peekAtTopCard(Deck*)`**
  Shows the top card, but does not remove it from the stack. Returns a pointer to the top card.
- **`Card* popCardFromDeck(Deck*)`**
  Removes the top card from the deck and returns it. Returns a pointer to the top card in the deck.
- **`int isDeckEmpty(Deck*)`**
  Determines if the deck is empty. Returns 0 if the Deck has any cards; 1 otherwise.
- **`void destroyDeck(Deck*)`**
  Removes all the cards from the deck and frees() them, preventing a memory leak

1.2 Implementing the Hand

The Hand has the following characteristics:

1. It can be ordered in various ways
2. We can easily add new cards and remove cards from the hand

For this quiz, implement the Hand as a doubly-linked list.

- **`Hand* createHand();`**
  Creates a Hand struct and initializes any necessary fields. Returns a pointer to the new Hand, which has been allocated on the heap.
- **`void addCardToHand(Card *card, Hand *hand);`**
  Adds a card to the hand.
- **`Card* removeCardFromHand(Card *card, Hand *hand);`**
  Removes a card from the hand. Return a pointer to the card that's been removed from the hand. Consider if you need to remove the CardNode from the heap.
- **`int isHandEmpty(Hand*);`**
  Determines if there are any cards in the hand. Return 0 if the hand is empty; 1 otherwise.
- **`void destroyHand(Hand*);`**
  Destroys the hand, freeing any memory allocated for it.

2 Part 2: Implementing Game Helper Functions

This part of the quiz is filling out some of the helper functions that make the game run.

Many of these functions (and the ones you implemented for Part 1) are helper functions you can be using. For example, you can use popCardFromDeck() to get rid of cards from a `Deck` before you destroy it (if needed). However, don't assume that these are the only functions you can write for this assignment. You are always free to write your own helper functions.

For example, we give you signatures for creating new Decks and new Hands. But, we did not document building a constructor for creating a newCardNode (used internally in the list that we use for keeping Hands). This is potentially an option for helpful functions you could build yourself.

1. **`int isLegalMove(Hand *hand, Card *leadCard, Card *playedCard);`**
   Given a card that is lead, a player's hand, and the card the player wants to play, is it legal? If the player has a card of the same suit as the `leadCard`, they must play a card of the same suit. If the player does not have a card of the same suit, they can play any card. For this, it might be helpful to define a helper function that determines if a hand has any cards of a given suit.
2. **`int whoWon(Card *leadCard, Card *followedCard, Suit trump);`**
   Given two cards that are played in a hand, which one wins? If the suits are the same, the higher card value wins. If the suits are not the same, player 1 wins, unless player 2 played trump. Return 1 if the person who led won, 0 if the person who followed won.
3. **`populateDeck();`**
   Create all the cards and pushes them into the Deck.
4. **`shuffleDeck(Deck*);`**
   Takes all the cards in the deck, rearrange the order, and push the cards back into the Deck.
5. **`deal(Deck*, Hand*, Hand*);`**
   Takes a shuffled deck and deals the cards one by one to the each hand.

3 Part 3: Challenge

If you've implemented the above, consider the following improvements to the game. Note: currently, tests haven't been written to test the challenge functions. These will be added later, or you can write your own.

Submit these changes with your final quiz submission. Partial implementation of any of these will be reviewed. Progress will be considered for your final grade. Not attempting any of these will have no impact on your quiz grade or final grade.

3.1 Challenge 1: Shuffle the computer's hand so it is unpredictable Difficulty: Medium

Implement **`void shuffleHand(Hand*)`** in `quiz7.c`:
Right now, the computer (Player 1) is only playing the first card in the hand. Implement **`shuffleHand()`**, which will random a hand of cards. The computer will then play a random card.

Reflection question: You implemented **`shuffle()`** on the Deck, which is a Stack. How is this similar or different from how you can shuffle a Hand, which is a LinkedList?

3.2 Challenge 2: Sort a hand in terms of value Difficulty: Medium

Implement **`void sortHand(Hand*, Suit)`** in `quiz7.c`:
A nice feature would be to print out the hand for the user (Player 2) in descending order of value. Given a Hand and a suit for trump, sort the hand so that it is in decreasing value, with cards of the trump suit first, then all other suits.

3.3 Challenge 3: Replace Deck using a LinkedList Difficulty: Medium

In this quiz, you implemented both a linked list (Hand) and a stack (Deck). Create a new deck.c file (call it `deck_2.c`), and re-implement Deck. This time, instead of using an array to back the Deck, use a linked list. You can re-use the CardNode used in the Hand implementation. To test and use your implementation of deck_2.c, modify the Makefile by replacing deck.c with deck_2.c. Note, some of the tests for Deck will fail because they are checking your array implementation.

3.4 Challenge 4: Be smarter about calling trump Difficulty: Hard

Right now, trump is decided randomly. In original Euchre, the process for calling trump requires the dealer and the other player to take turns having the opportunity to call trump in a structured way.

The process is like this:

1. After the deal, show the top card in the deck.
2. The player that is not the dealer looks at that card and decides if they want that suit for trump.
   (a) If the player wants the trump, they"order it up".
   (b) The dealer adds that top card to their hand.
   (c) The dealer discards one card.

(d) Play begins.

3. If the player does not want the top card for trump, the dealer gets to decide. If the deal wants that card, the process is the same as above (dealer picks up the card, and discards another card).
4. If the deal does not want the top card suit for trump, the top card is turned over.
5. The player gets to "call trump" or choose a suit.
6. If the player declines to call trump, the dealer must call trump.

For this challenge, modify the **`getTrump(Card*)`** function in quiz7_run.c to use this process. Look at the other functions for examples of how to get input from the user.

3.5 Challenge 5: Give the computer player strategy Difficulty: Hard

`getBestMove()`:
Given a hand of cards, the card that was just led, and the current trump, pick the best card in the hand to play. Note: there might be some strategy in this one!