CS 162 - Lect 1 - Rec 16
January 27, 2018
Nick Olson

Assignment 2, Go Fish Game

Understanding the Problem:

The number one assumption is that I should prioritize abstraction and encapsulation in this lab. The classes I'll need are Game, Player, Deck, Hand, and Card. The hand will have a lot of methods that are useful for the running of the game, and so it would be really easy to give the Player class a method that just returns its Hand instance for easier access, but that breaks the principles of OOP. I'll probably need functions that just returns the result of one of their fields methods, but that's okay. I'm going to assume that I don't need to do anything fancy with how the user inputs like with the mouse or by interpreting string input, so I'll just use the integer indices of the cards in a hand. I'm assuming I should also print the CPU's moves in a similar way to the players, but with the identifying information hidden, so that they can tell it is acting fairly. I should probably slow down the flow of the game somewhat so they don't have to go back to read a full terminal height of steps that occurred when their turn can't happen and so the CPU wouldn't seem to instantly take all their cards. I will need to implement methods in each class to display its contents (like displaying a hand as either hidden or exposed), etc. The "size" of various elements are variable throughout playtime (like the deck, hand, and books), so I have a couple of options. I could instantiate new instances of these lists every time something must grow or shrink, I could use vectors, or I could use fixed lists that are created as the maximum size they could be. I'll assume that any of these approaches are valid, even if creating new instances seems more in line with our memory focus, so I'll see how I can do it by manipulating elements in fixed-size lists instead. Using this method, instead of remaking the game object and starting as if a fresh execution had taken place if the player wants to try again, I can just reshuffle and reset these integer sizes back to their initial values.

Project Structure:

The main method in gofish.cpp creates an instance of the game class, which creates two players which each create a hand. The game class then fills these players hands with cards from the deck. The main method would then call a function of the game class that represents a turn from each player. The turn method returns 0 if the game is not over, and 1 if it is. For a card, rank of 0-9 are Ace through Ten and 10, 11, and 12 represent Jack, Queen, and King respectively. For suit, the order Clubs, Diamonds, Hearts, then Spades equates to the order 0, 1, 2, then 3. In order to imitate drawing cards from a deck (but not actually removing cards from a vector) I'll treat the top card as the furthest right card of the number of cards 'still in the deck.' This way, when drawing a card, I'll just return the card at num_cards - 1 and reduce num_cards by 1, such that the methods no longer reference that card. I'll use a few constant lists of strings in the classes that need them two hold the string representations of the card suits and ranks at each integer value's index. They'll be short_ranks[] which would be {"A", "2", …, "Q", "K"} for each rank; short_suits[] which would hold {"C", "D", "H", "S"} for each suit in alpha order; long_ranks[] which would hold the longer representations like {"Ace", "Two", … "Queen", "King"} and long_suits[] which does the same for the suits. These last two would be used for reading the results of drawing and stuff to the player. *All methods have a precondition that the object has been properly constructed and all fields are initialized unless otherwise noted.*

Classes:

```
 class Card {
   uses <string>

   private:
     int rank;  // Should be in the range 0-12 (A, 2-10, J, Q, K).
     int suit;  // Should be in the range 0-3 (Clubs, Diamonds, Hearts, Spades).
     std::string name;  // Will be the short string rep ("A") followed by the short suit rep ("C")

   public:
     Constructor takes the rank and suit, returning nothing, and sets those, also setting the name.
             Post: all fields are initialized

     'get_rank' method takes nothing and returns the value of rank.

     'get_suit' method takes nothing and returns the value of suit.

     'get_name' method takes nothing and returns the value of name.

     'get_long_rank' returns the string representation of the rank like 'Two' for rank = 1.
             { return long_ranks [ rank ] }

     'get_full_name' returns the string representation of the card like 'Two of Hearts' for the 2 of hearts.
             { return ( long_ranks[ this.rank ] ) concat ( long_suits[ this.suit] ) }
 };


 class Deck {
   uses <random>

   private:
     Card cards[52];
     int n_cards;  // Number of cards remaining in the deck.

   public:
     Constructor takes nothing, creates 52 cards for each value of each suit, in order, adding them to cards.
             Post: all fields initialized
             {
                     For each suit
                             For each rank
                                     Cards [ 13 * suit + rank ] set to new Card with rank and suit.
             }
```

```
  'shuffle' method takes nothing, and randomizes the order of the cards in the cards array.
      Pre: Deck is full of cards.
      Post: Deck is randomized.
      {
          For integers i to n_cards
              Generate random integer from i to index of last card
              Swap card at i and random int
      }

  'draw' method takes nothing, decrements n_cards, and returns the top card (one at n_cards).
      Pre: n_cards > 0
      Post: n_cards is one smaller, the card that was at n_cards -1 before is no longer considered.
      {
          Set n_cards to n_cards - 1
          Return card now at n_cards
      }

  'reset' method takes nothing, resets n_cards to 52 (initial value), and reshuffles the deck.
      Pre: game is restarting due to user choice
      Post: deck is shuffled and ready for a new round
      {
          n_cards = 52
          shuffle()
      }

};


class Hand {
 Uses <iostream>
 Uses <random>

 private:
  Card* cards;
  int n_cards;  // Number of cards in the hand.
  'swap' method that takes the indices of two cards in the hand, and swaps them.
      Post: the cards at the two passed indices are swapped
      {
          Temporary card slot is filled with card at first index passed
          The spot at the first index in the hand is filled with the card at the second index
          The spot at the second index in the hand is filled with card in the temporary slot
      }
```

Public:

Constructor which takes nothing, sets n_cards to 0, creates cards as dynamic array of length 52 (the max possible size), and returns nothing.

Post: hand is initialized and ready to have cards added, all default cards in hand currently are null cards with name “Null Card”

Destructor that deallocates cards by delete[]-ing the array.

‘get_n_cards’, which takes nothing and returns the value of n_cards.

‘get_rank_of’, which takes an index int and returns the rank of the card in the hand at that index.

‘display,’ which takes a bool of whether the cards are ‘hidden’ (the CPU’s) and prints out the contents of the hand as a single line to cout, returning nothing. If the hand is not hidden, each card in the hand should be represented by “[RS]” or “[RRS]” where R is the cards rank and S is its suit. It should also print out the index in the hand of each card if it is the players (not hidden).

```
Pre: hand is initialized
{
        If cards are hidden
                Print [    ] n_cards times
        Else
                For each card in n_cards
                        Print “[ “ + card.get_name() + “ ]      ”
                End line
                For each card in n_cards
                        Print “( “ + current_index + “ )      “
                End line
}
```

‘add,’ which takes a Card object, adds it to the array ‘cards’ and returns nothing.

```
Post: n_cards is one greater and the spot n_cards pointed to before execution holds the new card
{
        Cards at n_cards = passed_card
        n_cards = n_cards + 1
}
```

‘steal,’ which takes in a card rank (int), checks if the hand contains a card of that value, returns the first one it finds, swaps that card with the last card in the range of n_cards, and decrements n_cards.

```
Post: if the hand contains a card of that rank, the *first instance* is taken away from the hand and returned, if none are found, a default Card (name = “Null Card” is returned).
{
        For each card before n_cards
```

```
                If the card’s rank == the passed rank
                    n_cards = n_cards - 1 (now points to last card in hand)
                    Swap the current card with the card at n_cards (no longer considered)
                    Return that card (now at n_cards)
            Return default Card
    }
```

‘is_held,’ that checks ‘cards’ for a card with a passed int rank, returns true or false.

```
    {
        For each card before n_cards
            If the card’s rank == the passed rank
                Return true
        Return false
    }
```

‘random’ which returns the int rank of a random card in the hand (for generating valid CPU moves).

```
    {
        Generate random int from 0 to n_cards - 1 (range of cards held)
        Return the card at that random int
    }
```

‘check_books,’ which checks the hand for any books. If one is found, swaps those cards to the end, subtracting 1 from n_cards each time, returns either the rank of the first book found or -1. Only swaps the first found, so subsequent executions are required.

Post: if a book was found, the cards making it up are outside the range of held cards

```
    {
        Create book_rank = -1 that holds rank of book found
        Create card_counts that is list of ints (start as 0) of how many of each rank are found
        For each card before n_cards
            card_counts at rank of card is incremented
            If card_counts at that rank == 4 (a book was found)
                book_rank = the cards rank
                For each card in the hand
                    If the card has rank == book_rank (a part of that book)
                        Decrement n_cards
                        Swap that card and the card at n_cards
        Return book_rank
    }
```

‘reset’ which takes nothing and resets the hand to initial state by setting n_cards to 0

Pre: game is being reset for a new round

Post: n_cards is 0, meaning hand is “empty” again

};

class Player {

Uses <iostream>

private:

Hand hand;

int* books; // Array with ranks for which the player has books.

int n_books;

bool is_cpu; // Whether this player object is a CPU (changes how things are displayed)

public:

Constructor that takes whether the player is a CPU (defaults to false) creates a Hand for field ‘hand’, creates a new ‘books’ array of length 13 (maximum number of books for a 52 card deck), sets n_books to 0, sets is_cpu to parameter value).

Post: all fields are initialized

Destructor that deallocates the ‘books’ array using delete[] books.

Post: all dynamically allocated memory in existence of the Player is freed

‘get_n_books’ returns the value of n_books, takes nothing.

‘add_card’ takes a Card object and adds it to the players hand.

Post: the card has been added to the hand using the hand’s add method

‘display_hand’ which takes nothing, and calls the hands display method, passing the val of is_cpu.

Post: the hand has been displayed using the Hand display() method.

‘get_hand_size’ which takes nothing and returns the number of cards in the hand by calling get_n_cards() on the player’s hand.

‘get_rank_of_card’ which takes an int index and returns the rank of the card at that index in the player’s hand using the Hand get_rank_of method. (Necessary for getting valid user input).

‘steal’ which takes a rank int, and returns the result of hand.steal(int rank) with that argument.

Post: If the Player’s Hand had a card of that rank, it is ‘removed’ from the Hand.

‘is_held’ which takes a rank int and returns the result of hand.is_held(int rank) with that argument.

Post: If the Player’s Hand had a card of that rank, true is returned.

‘random_card’ which returns the result of calling their Hand’s random method.

‘display_books’ prints the rank of each of the books on a single line using cout.

```
        Post: The Player’s books are printed as in diagram at bottom of design section. (one line)
        {
                Print “Books: “
                For each book before n_books
                        Print “[ “ + (short_ranks  /*(list of rank string)*/  at book’s rank) + “]  ”
                End line.
        }
```

‘update_books’ which checks hand for books, adding values to ‘books’ array, incrementing n_books. Because the Hand class’s check_books function only returns the first book it finds, we call this until the result check_books == -1, which is the value for none found.

```
        Post: If the player had any books in their hand, they have been removed from the hand, and the
        Player’s books array has been filled with the ranks of these books.
        {
                Set int variable book_rank to result of the hand’s check_books()
                While (book_rank is not -1)  /*(Does not execute if no books were found)*/
                        Set books list at n_books to book_rank (rank of found book)
                        n_books += 1 (books now includes this new book)
                        Set book_rank to result of fresh hand.check_books() call.
        }
```

‘reset’ which takes nothing, sets n_books to 0, and calls the hand’s reset function to reset the game for a new round, as the books list now appears empty to the program.

```
 };

class Game {
 Uses <iostream>
 Uses <iomanip>

 private:
  Deck cards;
  Player players[2];
  int hand_size;

    Public:
```

Constructor that takes starting hand_size (default of 7), creates new Deck, creates new array of players, setting all the fields in the process. Sets players[0] to a human and players[1] to a CPU.

Post: all fields are initialized, player 0 is human, 1 is CPU.

Destructor that deallocates (delete[]’s) the players array.

Post: all memory allocated in game existence is freed.

‘setup’ that shuffles the deck and draws hand_size cards for each player.

```
    Pre: game is starting or has been reset, this is the first action taken on game
    Post: game is ready to be played, each player has the starting number of cards, deck is shuffled.
    {
        Call the Decks shuffle() function
        For each Player
            For each int from 0 until hand_size - 1
                Call that Player’s add_card function, passing the Deck’s draw result
    }
```

‘player_turn’ that prints the table as is, asks for a user’s choice of card to request, and checks the request. If it is good, update books and let them ask again, else they draw, return nothing. This function handles each turn’s logic and progression. “Restart turn” means recursively return result of a fresh player_turn() call. “Turn ends” means return void.

```
    Post: the player has played out their turn, however many moves that takes
    {
        display_table()
        If player is out of cards
            If there are cards in the deck
                Player draws a card, restart turn
            Else (deck is empty)
                Player’s turn ends.
        Else (player has cards in hand)
            Call get_valid_request to handle user input of card rank they want to ask for
            If CPU has the requested card
                While CPU still has at least of that card ( using cpu.is_held() )
                    Steal card from CPU (cpu.steal(rank)) and give to player

                Print result of turn (how many cards stolen)
                Restart player turn
            Else (CPU does not have requested card)
                Player draws card (deck.draw() and player.add_card())
                Print result of draw (name of card (card.get_full_name()))
                If drawn card is rank they requested initially
                    Restart player turn
                Else (they drew card other than that they requested)
                    End player turn
    }
```

‘cpu_turn’ that follows the exact same logic as player_turn, but doesn’t reveal card information to the player, still walking through each action that applies to player. All printed string pronouns adjusted.

```
    Post: the CPU has taken their turn, however many moves it takes
    { Refer to player_turn, swap all references of CPU and Player, etc. }
```

‘display_table’ that takes nothing and prints out the current status of the table. Specifically prints the CPU’s hand (hidden) and their books (not hidden), the number of cards left in the deck, and the player’s hand and books (both not hidden). Under the player’s hand are the indices for each card. Refer to diagram at bottom of plan section for example.

```
Post: the table has been printed to cout.
{
        Print “CPU’s:”
        End line
        cpu.display_hand()
        cpu.display_books()
        Two blank lines
        Print (centered) “[ “ + cards.get_n_cards() + “ ]”
        One blank line
        Print “Your”
        End line
        player.display_books()
        player.display_hand()
}
```

‘get_valid_request’ which takes the number of the player requesting (defaults to 0), and prompts that user for cin input, accepting only valid inputs within the range of indices of cards in their hand, returning the rank of the card the user chooses from their hand for their request.

```
Pre: it is the user’s turn, and they are choosing a card to ask the cpu for
Post: the user has chosen a valid card from their hand to request more of
{
        Prompt user for choice, get input from cin
        If input is not an int
                Print error and ask again
        Else if input is an int but is not in range (from 0 to their hand’s (n_cards - 1))
                Print error and ask again
        Return valid user input
}
```

‘check_game_over’ that takes nothing, checks if the game is over by checking if all the books have been played, congratulating winner on cout and returning true if so.

```
Post: if all books have been placed down, the winner has been acknowledged and true is returned.
{
        Set variable total_books to 0
        For each player
                Add their get_n_books() to total_books
        If total_books equals 13 or more (the max books playable)
                If player.get_n_books() > cpu.get_n_books()
```

```
                        Congratulate player
                    Else
                        Congratulate cpu
                    Return true
                Return false
            }

  'update_all_books' which calls update_books() for each player (the user and the cpu)
            Post: all players' books have been updated and played
            {
                For each player
                    player.update_books()
            }

  'reset' that calls each player's reset() function and the deck's reset() function
            Post: the game is reset and ready for a new round to be played
            {
                For each player
                    player.reset()
                cards.reset()
            }
};

go_fish.cpp
        Main method logic:
            Print flavor text
            Create new Game
            Create boolean keep_going as true
            game.setup()
            while keep_going is true
                game.player_turn()
                game.cpu_turn()
                If game.check_game_over() is true
                    Get user input (0 or 1) as to whether they want to play again
                    If input is 1
                        game.reset()
                    Else
                        keep_going = false
            Return 0
```

Table Layout Example:

```
#=~
CPU's:
    Cards:  [    ] [    ] [    ] [    ] [    ]
    Books:  [ 7 ] [ Q ] [ 5 ]


                [[[ 30 ]]]

Your:
    Cards:  [ AD ] [ 6S ] [ 9D ] [ AS ] [ 10H ]
    Books:  [ J ] [ 3 ]
#=~
```

Testing Plan:

Test Cases				
Variable	Type	Value	Expected Result	Actual Result
whether player wants to play again	edge	<empty>	Ask for integer re-input.	TBD
	bad	"yep"	Ask for integer re-input.	TBD
		5	Ask for re-input in range.	TBD
		0.5	Ask for integer re-input.	TBD
	good	0	Exit program.	TBD
		1	Reset game, continue.	TBD
player_turn card request input (assuming player has 7 cards in their hand)	edge	<empty>	Ask for integer re-input.	TBD
	bad	"yep"	Ask for integer re-input.	TBD
		9	Ask for re-input in range.	TBD
		0.5	Ask for integer re-input.	TBD
	good	0	Return rank of card at index 0.	TBD
		5	Return rank of card at index 5.	TBD
		7	Return rank of card at index 7.	TBD