# CS 162 Assignment #2 Go Fish Game

**Design Document** due: Sunday, 7/12/2020, 11:59 p.m. (Canvas)

**Assignment** due: Sunday, 7/19/2020, 11:59 p.m. (TEACH)

**Goals:**

- Practice good software engineering design principles:
    - Design your solution before writing the program
    - Develop test cases before writing the program
- Practice Object-Oriented Programming using C++ classes
- Implement "Has-a" relationship using class composition
- Practice file separation and create Makefile
- Use functions to modularize code to increase readability and reduce repeated code

---

**Problem Statement:**

For this assignment, you will write a program that allows one player to play a game of Go Fish against the computer. Go Fish is a game that uses a standard deck of 52 cards. Specifically, each card has a rank and a suit. There are 13 ranks: the numbers 2 through 10, Jack (usually represented with the letter J), Queen (Q), King (K), and Ace (A). There are 4 suits: clubs, diamonds, hearts, and spades. In a 52-card deck, there is one card of each rank for each suit.

A game of Go Fish between two players proceeds as follows:

- The deck of cards is shuffled, randomizing the order of the cards.
- Each player is dealt 7 cards.
- The remaining cards are placed face-down (i.e. with their rank and suit hidden) on the table to form the "stock".
- One of the players (player A) begins the game by asking the other player (player B) for all of his/her cards of a specific rank (e.g. "Please give me all of your 7's"). To ask for a given rank, a player must have at least one card of that rank in his/her hand.

- If player B has cards of the requested rank, he/she must give ***all* of** his/her cards of that rank to player A, and player A gets to take another turn.
- If player B does not have any cards of the requested rank, he/she says "Go fish", and player A must select one card from the stock. If that card has the rank player A originally requested, then player A gets to take another turn. Otherwise, it becomes player B's turn.
- If at any point a player runs out of cards, then, when it is that player's turn to play, they may draw a card from the stock and ask for cards of that rank. If a player runs out of other cards when it is the other player's turn to ask for a rank, the other player may continue to ask for a rank and draw from the stock until they draw a rank other than the one they asked for.
- If a player collects all four cards of the same rank, this is called a "book", and the player lays down his/her book on the table.
- The game continues with the players alternating turns until all of the books are laid down. At the end of the game, the player with the most books wins.

Here is a nice little online version of Go Fish that you can play to get a feel for how the game works if you've never played it before: https://cardgames.io/gofish/.

---

**Required Classes:**

To write your Go Fish game, you should implement the following classes, including the specified members and functions. You may also add more members and functions, as needed.

```
class Card {
private:
    int rank;  // Should be in the range 0-12.
    int suit;  // Should be in the range 0-3.
  public:
  // must have constructors, destructor, accessors, and mutators
};
```

In the `Card` class above, rank and suit are represented with int values, but you must also have some way to map those values to representations players will be familiar with (e.g. a string representation of the suit or rank).

```
class Deck {
  private:
     Card cards[52];
     int n_cards;  // Number of cards remaining in the deck.
  public:
 // must have constructors, destructor, accessors, and mutators
};
```

The `Deck` class is the source of all of the cards. Cards will initially start in a `Deck` object and then be transferred to players' hands. An important member function that should be implemented for the `Deck` class is one to remove a card and return it so it can be placed in a player's hand.

```
class Hand {
  private:
    Card* cards;
    int n_cards;  // Number of cards in the hand.
  public:
  // must have constructors, destructor, accessors, and mutators
};
```

The `Hand` class will hold the cards in one player's hand. The number of cards a player holds may change, so the size of the array of Card objects in a hand may also need to change. Cards may be added to a player's hand and removed from a player's hand, so the Hand class will need functions to do both of those things. Other useful functions might check for a book and remove a book from the player's hand.

```
class Player {
  private:
    Hand hand;
    int* books;  // Array with ranks for which the player has books.
    int n_books;
  public:
  // must have constructors, destructor, accessors, and mutators
};
```

The `Player` class represents a single player. Each Player will have a Hand object representing its hand and an array keeping track of the books the player has laid down. Depending on how you implement things, the Player class may need functions to add and remove cards from their hand or to check their hand for books. Another useful function the Player class might have is one to figure out what rank they want to ask for. Note that the Player class must represent both the human player and the computer player. You should write your class functions accordingly and add any extra data members needed to accomplish this.

```
class Game {
  private:
    Deck cards;
    Player players[2];
  public:
  // must have constructors, destructor, accessors, and mutators
}
```

The `Game` class represents the state of an entire game. It contains objects representing the deck of cards and both players. It would be useful to have functions in the Game class that check whether the game is over and that execute a player's turn.

---

**Implementation Requirements:**

- This list is not exhaustive. Your program must satisfy all the requirements given in this document
- Set up a deck of 52 cards as described above.
- Shuffle the deck (i.e. randomize the order of cards).
- Deal the cards to the two players.
- Play a game of Go Fish as described above, with the following gameplay features:
    - Print the current state of the game (e.g. the cards held by the human player and the books laid down by both the human player and the computer) after each turn (keeping the computer player's cards hidden).
    - On the human player's turn, prompt the user for a rank to ask for. When they enter a rank, either give all of the computer player's cards of that rank to the human (if the computer player has cards of the requested rank) or execute a "go fish", drawing a card from the stock into the human player's hand. Make sure to alert the user what the result of their turn is.
    - On the computer player's turn, the computer should randomly select a rank from its hand for which to ask the other player, and the appropriate actions should be taken (i.e. cards are taken from the human player's hand, or the computer draws a card from the stock). The user should be alerted about what the computer player asked for and what the result of its turn was.

- Once the game is over, you should announce the winner and ask the user if they want to play again.
- Your program must also accept a command line argument which will be either “true” or “false”. If the parameter is set to false (e.g. `./go_fish false`) then the program will run as normal. If the command line value is specified as “true”, then your game must operate in debug mode.
- When your program operating in debug mode, the current state of the game will show a “cheat view” with the computer player’s cards face up as well as the books held by both the human player and the computer.
- You should not have any memory leaks in your program.
- Each class needs accessor functions, mutator functions, constructors, copy constructor, destructors, assignment operator overload, and use of const where appropriate.
- The program needs to be compiled with a makefile
- Files must be separated into a .h and a .cpp for each class. You should have one driver file.

---

## README.txt Description

A README is a text file that introduces and explains a project. Once you have implemented and fully tested the Go Fish Game with all functionalities described above, write a README.txt file that includes the following information:

1. Your name and ONID
2. **Description**: One paragraph advertising what your program does (for a user who knows nothing about this assignment, does not know C++, and is not going to read your code). Highlight any special features.
3. **Instructions**: Step-by-step instructions telling the user how to compile and run your program. If you expect a certain kind of input at specific steps, inform the user what the requirements are. Include examples to guide the user.
4. **Limitations**: Describe any known limitations for things the user might want or try to do but that program does not do/handle.
5. **Extra Credit**: If your program includes extra credit work, describe it here for the user.

Here is an example README.txt file for a hypothetical assignment (not this one):

http://classes.engr.oregonstate.edu/eecs/summer2020/cs162-001/assignments/README_example.txt

---

## Programming Style/Comments

In your implementation, make sure that you include a program header. Also ensure that you use proper indentation/spacing and include comments! Below is an example header to include. Make sure you review the style guidelines for this class , and begin trying to follow them, i.e. don’t align everything on the left or put everything on one line!

```
/*****************************************************
** Program: Go_Fish.cpp
** Author: Your Name
** Date: 07/10/2020
** Description:
** Input:
** Output:
*****************************************************/
```

---

**Extra Credit: Implement a Smarter Computer Component**

Instead of using a simple random number generator to select the rank from its hand, write your computer player implementation so that it has more intelligence. If you choose to attempt this extra credit, be sure to document your algorithm in the README.txt.
As you can imagine, there are many ways to implement a smarter computer opponent. The options are endless.

---

**Design Document – Due Sunday 7/12/2020, 11:59pm on Canvas**
**Refer to the Example Design Document – Example_Design_Doc.pdf**

**Understanding the Problem/Problem Analysis:**

- What are the user inputs, program outputs, etc.?
- What assumptions are you making?
- What are all the tasks and subtasks in this problem?

**Program Design:**

- What does the overall big picture of each function look like? (Flowchart or pseudocode)
    - What member variables do you need to create for each class?
    - What member functions do you need to create for each class?
    - How do classes interact with each other? (How do you implement a "has-a" relationship?)
    - What member functions are you going to have for each class? And what's their job?
    - How would you modularize the program?
- What kind of bad input are you going to handle?

Based on your answers above, list the **specific steps or provide a flowchart** of what is needed to create. Be very explicit!!!

**Program Testing:**
Create a test plan with the test cases (bad, good, and edge cases). What do you hope to be the expected results?

- What are the good, bad, and edge cases for ALL input in the program? Make sure to provide enough of each and for all different inputs you get from the user.

**Electronically submit your Design Doc (.pdf file!!!) by the design due date, on Canvas.**

---

**Program Code – Due Sunday, 7/19/2020, 11:59pm on TEACH**

**Additional Implementation Requirements:**

- Your user interface must provide clear instructions for the user and information about the data being presented
- Your program must catch all required errors and recover from them.
- You are not allowed to use libraries that are not introduced in class, more specifically, you may not use the <algorithm> library in your program. Any searching or sorting functionality must be implemented "manually" in your implementation.
- Your program should be properly decomposed into tasks and subtasks using functions.
  To help you with this, use the following:
    - Make each function do one thing and one thing only.
    - No more than 15 lines inside the curly braces of any function, including main(). Whitespace, variable declarations, single curly braces, vertical spacing, comments, and function headers do not count.

    - Functions over 15 lines need justification in comments.
    - Do not put multiple statements into one line.
- No global variables allowed (those declared outside of many or any other function, global constants are allowed).
- No goto function allowed
- No vectors allowed
- You must not have any memory leaks
- You program should not have any runtime error, e.g. segmentation fault
- Make sure you follow the style guidelines, have a program header and function headers with appropriate comments, and be consistent.

---

## Compile and Submit your assignment

When you compile your code, it is acceptable to use C++11 functionality in your program. In order to support this, change your `Makefile` to include the proper flag.
For example, consider the following approach (note the inclusion of **`-std=c++11`**):

```
g++ -std=c++11 <other flags and parameters>
```

In order to submit your homework assignment, you must create a **tarred archive** that contains your `.h, .cpp,` and `Makefile` files. This tar file will be submitted to TEACH . In order to create the tar file, use the following command:

```
tar -cvf assign2.tar <list of all .h and .cpp files> Makefile README.txt
```