p3-euchre

EECS 280 Project 3: Euchre

Project Due Friday, February 28 2020, 8pm

Euchre (pronounced "YOO-kur") is a trick-taking card game popular in Michigan. It is most commonly played by four people in two partnerships with a deck of 24 cards. Partnerships accumulate points for winning tricks, and the game continues until one side reaches the maximum number of points.

In this project, you will write a simulator for a game of Euchre. You will gain experience with abstract data types, object-oriented programming, and polymorphism. While building the simulator, you will use classes, `std::array`, `std::vector`, and C++ style strings.

This project will be autograded for correctness, comprehensiveness of your test cases, and programming style. See the style checking tutorial for the criteria and how to check your style automatically on CAEN.

You may work alone or with a partner. Please see the syllabus for partnership rules. You may not share any part of your solution outside of your partnership. This includes both code and test cases.

Authors

The original project was written by Andrew DeOrio, Fall 2013. The project was modified to use C++ style object oriented programming and the specification updated by the Fall 2015 staff.

Table of Contents

Project Roadmap

This is a big picture view of what you'll need to do to complete this project. Most of the pieces listed here also have a corresponding section later on in the spec that goes into more detail.

1. **Learn the rules for EECS 280 Euchre**

   We understand that not all students are familiar with Euchre, so a complete description of the rules for "EECS 280 Euchre" is included in this specification.

   Our step-by-step explanation of a game of "EECS 280 Euchre" can be found in a YouTube video and a Powerpoint.

   Before getting started on this project, consider playing a game of Euchre with three friends or online. It will make the spec easier to understand, and it's a Michigan tradition!

2. **Download the Starter Code**

   Use the tutorial from project 1 to get your visual debugger set up. Use this `wget` link `https://eecs280staff.github.io/p3-euchre/starter-files.tar.gz`.

   Before setting up your visual debugger, you'll need to rename each `.cpp.starter` file to a `.cpp` file.

   ```
   $ mv Card.cpp.starter Card.cpp
   ```

   You'll also need to create these new files and add function stubs.

   ```
   $ touch Pack.cpp
   $ touch Player.cpp
   $ touch euchre.cpp
   ```

These are the executables you'll use in this project:

- `Card_public_test.exe`
- `Card_tests.exe`
- `Pack_public_test.exe`
- `Pack_tests.exe`
- `Player_public_test.exe`
- `Player_tests.exe`
- `euchre.exe`

If you're working in a partnership, set up version control for a team.

3. **Familiarize Yourself with the Code Structure**

   The code structure is object-oriented, with classes representing entities in the Euchre world.

4. **Test and Implement the Basic Euchre ADTs**

   You are provided interfaces for basic Euchre ADTs. Test and implement those functions.

5. **Test and Implement the Euchre Game**

   Write and test a `main()` function with a command-line interface that plays a game of Euchre.

6. **Submit to the Autograder**

   - `Card.cpp`
   - `Card_tests.cpp`
   - `Pack.cpp`
   - `Player.cpp`
   - `Player_tests.cpp`
   - `euchre.cpp`

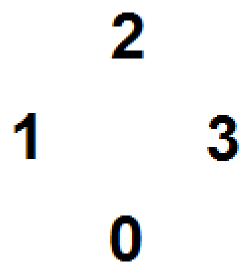
   You do not have to submit `Pack_tests.cpp` to the autograder.

EECS 280 Euchre Rules

There are many variants of Euchre. Our particular version is based on a variety commonly played in Michigan with a few changes to make it feasible as a coding project.

Players

There are four players numbered 0-3. If the players sat around the table, it would look like this:



There are two teams: players 0 and 2 are partners, as are 1 and 3. Each player has left and right neighbors. For example, 1 is to the left of 0, and 3 is to the right of 0. That means 1 is 0's left neighbor, and 3 is 0's right neighbor.

The Cards

Euchre uses a deck of 24 playing cards, each of which has two properties: a rank and a suit. The ranks are 9, 10, Jack, Queen, King, and Ace, and the suits are Spades, Hearts, Clubs, and Diamonds. Each card is unique — there are no duplicates. Throughout this document, we sometimes refer to ranks or suits using only the first letter of their name. Farther below, we describe how to determine the ordering of the cards.

Playing the Game

At a high level, a game of Euchre involves several rounds, which are called hands. Each hand consists of the following phases.

Each hand:

1. Setup
   i. Shuffle
   ii. Deal
2. Making Trump
   i. Round One
   ii. Round Two
3. Trick Taking
4. Scoring

We describe each in more detail below.

Setup

Shuffle

The dealer shuffles the deck at the beginning of each hand. The algorithm you will implement for shuffling is a variant of a riffle shuffle called an "in shuffle" (https://en.wikipedia.org/wiki/In_shuffle). Cut the deck exactly in half and then interleave the two halves, starting with the second half. Thus, the card originally at position 12 goes to position 0, the one originally at position 0 goes to position 1, the one originally at position 13 goes to position 2, and so on. Do this in-shuffle process 7 times.

You will also implement an option to run the game with shuffling disabled - when this option is chosen, just reset the pack any time shuffling would be called for. This may make for easier testing and debugging.

Deal

In each hand, one player is designated as the dealer (if humans were playing the game, the one who passes out the cards). In our game, player 0 deals during the first hand. Each subsequent hand, the role of dealer moves one player to the left.

Each player receives five cards, dealt in alternating batches of 3 and

1. That is, deal 3-2-3-2 cards then 2-3-2-3 cards, for a total of 5 cards each. The player to the left of the dealer receives the first batch, and dealing continues to the left until 8 batches have been dealt.

Four cards remain in the deck after the deal. The next card in the pack is called the upcard (it is turned face up, while the other cards are all face down). It plays a special role in the next phase. The three remaining cards are not used for the current hand.

Making Trump

During this phase, the trump suit is determined by whichever player chooses to order up.

Round One

The suit of the upcard is used to propose a trump suit whose cards become more valuable during the upcoming hand. Players are given the opportunity to order up (i.e. select the suit of the upcard to be the trump suit) or pass, starting with the player to the dealer’s left (also known as the eldest hand) and progressing once around the circle to the left. If any player orders up, the upcard’s suit becomes trump and the dealer is given the option to replace one of their cards with the upcard.

Round Two

If all players pass during the first round, there is a second round of making, again beginning with the eldest hand. The upcard’s suit is rejected and cannot be ordered up. Instead, the players may order up any suit other than the upcard’s suit. The dealer does not have the opportunity to pick up the upcard during round two.

If making reaches the dealer during the second round, a variant called screw the dealer is invoked: the dealer must order up a suit other than the rejected suit.

(Note for pro Euchre players: for simplicity, we have omitted “going alone” in this version.)

Trick Taking

Once the trump has been determined, five tricks are played. For each trick, players take turns laying down cards, and whoever played the highest card takes the trick.

During each trick, the player who plays first is called the leader. For the first trick, the eldest hand leads.

At the beginning of each trick, the leader leads a card, which affects which cards other players are allowed to play, as well as the value of each card played (see below). Each other player must follow suit (play a card with the same suit as the led card) if they are able, and otherwise may play any card (it is removed from their hand). Play moves to the left around the table, with each player playing one card.

A trick is won by the player who played the highest valued card (see below to determine comparative values). The winner of the trick leads the next one.

Scoring

The team that takes the majority of tricks receives points for that hand. If the winning team had also ordered up, they get 1 point for taking 3 or 4 tricks, and 2 points for taking all 5, which is called a march. Otherwise, they receive 2 points for taking 3, 4 or 5 tricks, which is called euchred.

Traditionally, the first side to reach 10 points wins the game. In this project, the number of points needed to win is specified when the program is run.

Value of cards

In order to determine which of two cards is better, you must pay attention to the context in which they are being compared. There are three separate contexts, which depend on whether or not a trump or led suit is present.

In the simplest case, cards are ordered by rank (A > K > Q > J > 10 > 9), with ties broken by suit (D > C > H > S).

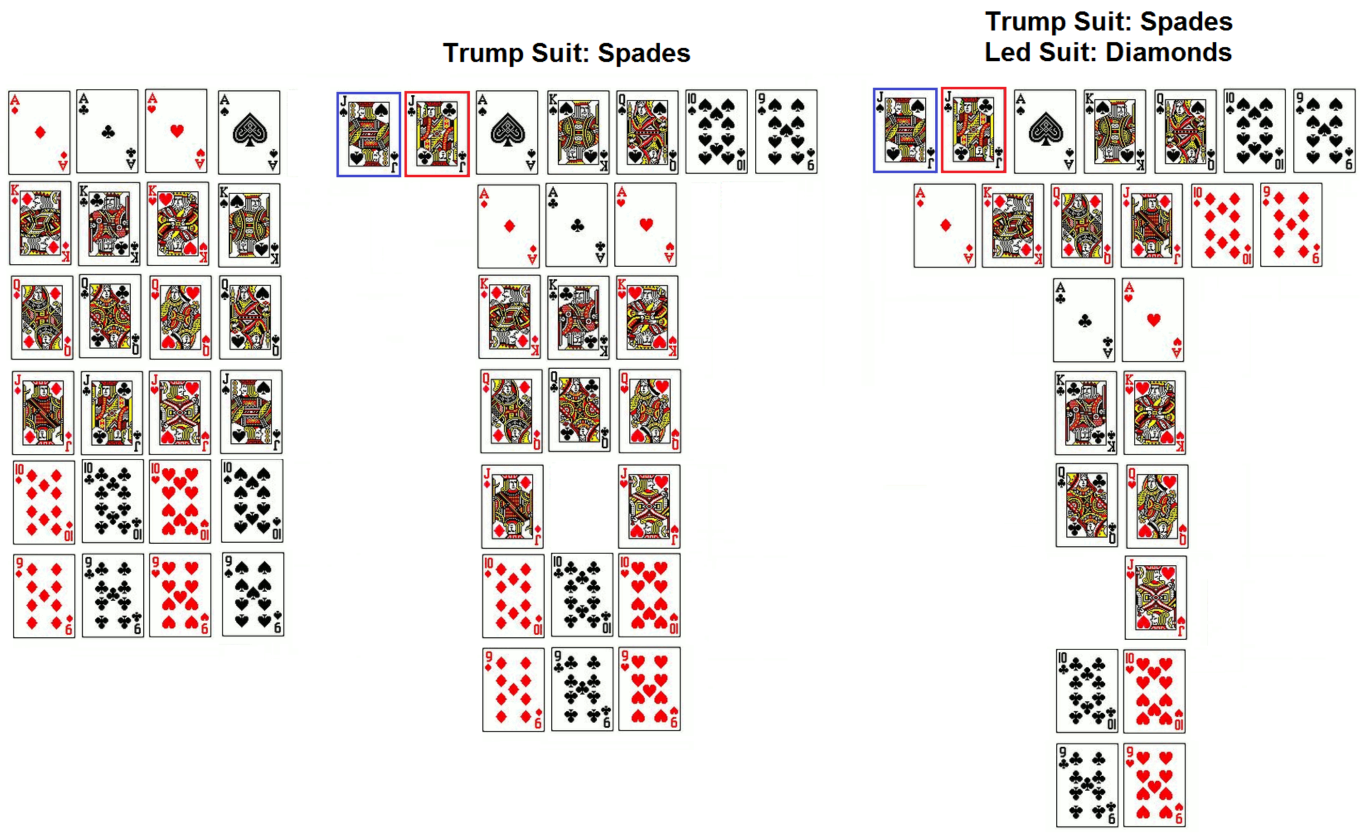
If a trump suit is present, all trump cards are more valuable than non-trump cards. That means a 9 of the trump suit will beat an Ace of a non-trump suit. Additionally, two special cards called bowers take on different values than normal.

- Right Bower: The Jack of the trump suit. This is the most valuable card in the game.

- Left Bower: The Jack of the "same color" suit as trump is **considered to be a trump** (regardless of the suit printed on the card) and is the second most valuable card.

For example, if Diamonds is trump, the Jack of Hearts is also considered a Diamond, not a Heart. The suit of the left bower is called next, while the two suits of the opposite color are called cross suits.

If a led suit is present as well as a trump suit, the ordering is the same except that all cards of the led suit are considered more valuable than all non-trump-suit, non-led-suit cards. Note that it is possible for the trump suit and led suit to be the same.



The above shows card orderings in the possible contexts. Cards in higher rows are greater than those in lower rows. Within rows, cards farther to the left are greater. Note the right bower (blue outline) and left bower (red outline).

Euchre Simple Player strategy

Here we describe the strategy used by a Simple Player. A Simple Player will always behave in this way, but these are **not** rules of the game (i.e. a Human Player need not follow this strategy).

Much of the strategy for our Simple Player can be implemented using the comparison functions provided by the Card interface.

Making

In making trump, a Simple Player considers the upcard, which player dealt, and whether it is the first or second round of making trump. A more comprehensive strategy would consider the other players' responses, but we will keep it simple.

During round one, a Simple Player considers ordering up the suit of the upcard, which would make that suit trump. They will order up if that would mean they have two or more trump face cards in their hand. Trump face cards are the right and left bowers, and Q, K, A of the trump suit, which is the suit proposed by the upcard. (A Simple Player does not consider whether they are the dealer and could gain an additional trump by picking up the upcard.)

During round two, a Simple Player considers ordering up the suit with the same color as the upcard, which would make that suit trump. They will order up if that would mean they have one or more trump face cards in their hand (the right and left bowers, and Q, K, A of the order-up suit). For example, if the upcard is a Heart and the player has the King of Diamonds in their hand, they will order up Diamonds. The Simple Player will not order up any other suit. If making reaches the dealer during the second round, we invoke screw the dealer, where the dealer is forced to order up. In the case of screw the dealer, the dealer will always order up the suit with the same color as the upcard.

Adding the Upcard and Discarding a Card

If the trump suit is ordered up during round one, the dealer picks up the upcard. The dealer then discards the lowest card in their hand, even if this is the upcard, for a final total of five cards. (Note that at this point, the trump suit is the suit of the upcard.)

Leading Tricks

When a Simple Player leads a trick, they play the highest non-trump card in their hand. If they have only trump cards, they play the highest trump card in their hand.

Playing Tricks

When playing a card, Simple Players use a simple strategy that considers only the suit that was led. A more complex strategy would also consider the cards on the table.

If a Simple Player can follow suit, they play the highest card that follows suit. Otherwise, they play the lowest card in their hand.

Code Structure

The code is structured as an object-oriented program. The C++ class mechanism is used to represent entities in the Euchre world, for example `Card`, `Pack`, and `Player`. The interfaces for these classes are defined in several header files, each representing a different Abstract Data Type. Your task is to provide the corresponding implementations in `.cpp` files, adding any additional helper functions to the `.cpp` files. Finally, you will write a `main()` function with a command line interface to the game.

Polymorphic Players

A game of Euchre would be pretty boring if every player was limited to one strategy. However, we don't want to clutter the main driver with code that implements different strategies. Instead, we'll use the abstract base class `Player` to define an interface of players' abilities and then implement a few derived classes that use different player strategies. That way our driver can run the game without knowing about the individual Players' strategies.

You will be implementing two types of Player:

- "Simple": A computer-controlled player that uses the basic strategy described earlier.
- "Human": A human-controlled player that reads instructions from `cin`.

Human Player Protocol

The Human Player reads input from the human user. You may assume all user input is correctly formatted and has correct values. You may also assume the user will follow the rules of the game and not try to cheat. See Appendix B for exact output for a game with a human player.

Making Trump

When making trump reaches a Human Player, first print the Player's hand. For consistency with the autograder test cases, print the Cards in ascending order, as defined by the `<` operator in `Card.h`. Then, prompt the user for their decision to pass or order up. The user will then enter one of the following: "Spades", "Hearts", "Clubs", "Diamonds", or "pass" to either order up the specified suit or pass. This procedure is the same for both rounds of making trump.

Adding the Upcard and Discarding

If a Human Player is the dealer and someone orders up during the first round of making, the Human Player will pick up the upcard and discard a card of their choice. Print the Player's hand and an option to discard the upcard. Then, prompt the user to select a card to discard. The user will then enter the number corresponding to the card they want to discard (or -1 if they want to discard the upcard).

Playing and Leading Tricks

When it is the Human Player's turn to lead or play a trick, first print the Player's hand. For consistency with the autograder test cases, print the cards in ascending order, as defined by the `<` operator in `Card.h`. Then, prompt the user to select a card. The user will then enter the number corresponding to the card they want to play.

HINT: here's how to use the STL to sort a vector `hand`:

```
std::sort(hand.begin(), hand.end());
```

You will need to `#include <algorithm>` in order to use `std::sort()`.

Requirements and Restrictions

It is our goal for you to gain practice with good C++ code, classes, and polymorphism.

DO	DO NOT
Modify `.cpp` files	Modify `.h` files
Write helper functions in `Card.cpp` and `Pack.cpp` as non-member functions in the `.cpp` files and declare them `static`	Modify `.h` files
Use these libraries: `<iostream>`, `<fstream>`, `<cstdlib>`, `<cassert>`, `<cstring>`, `<string>`, `<array>`, `<vector>`, `<sstream>` Use the `<algorithm>` library, but only for the `sort()` function.	Use other libraries. Use `<algorithm>` library for anything other than the `sort()` function.
`#include` a library to use its functions	Assume that the compiler will find the library for you (some do, some don't)
	`#include` an unnecessary `.h` file, such as `#including Pack.h` in `Card.cpp` (This introduces undesirable dependencies, e.g. that the `Card` ADT requires the `Pack` ADT to exist and be properly implemented)
Use C++ strings	Use C-strings other than when checking `argv`
Send all output to standard out (AKA stdout) by using `cout`	Send any output to standard error (AKA stderr) by using `cerr`
`const` global variables	Global or static variables
Pass large structs or classes by reference	Pass large structs or classes by value

DO	DO NOT
Pass by `const` reference when appropriate	"I don't think I'll modify it …"
Variables on the stack	Dynamic memory (`new` , `malloc()` , etc.) outside of the `Player_factory` .

See the coding practices checklist for more specific guidlines on writing your code.

Starter Code

The following table describes each file included in the starter code.

File(s)	Description
`Card.h`	Procedural abstraction representing operations on a playing card.
`Card.cpp.starter`	Starter code for the `Card` module. Rename to `Card.cpp` when you start.
`Card_tests.cpp`	Add your `Card` unit tests to this file.
`Card_public_test.cpp`	A "does my code compile" test case for `Card.cpp` .
`Pack.h`	Procedural abstraction representing operations on a pack of playing cards.
`Pack_tests.cpp`	Add your `Pack` unit tests to this file.
`Pack_public_test.cpp`	A "does my code compile" test case for `Pack.cpp` .
`Player.h`	Procedural abstraction representing operations on a euchre player.
`Player_tests.cpp`	Add your `Player` unit tests to this file.
`Player_public_test.cpp`	A "does my code compile" test case for `Player.cpp` .
`pack.in`	Input file containing a Euchre deck.
`Makefile`	Used by the `make` command to compile the executable.
`euchre_test00.out.correct` `euchre_test01.out.correct` `euchre_test50.out.correct`	Correct output for system tests of main executable. The first line of each file contains the command used to generate it.
`euchre_test50.in`	File containing input for `euchre_test50` .

File(s)	Description
`unit_test_framework.h`	The unit test framework you must use to write your test cases.

Implement and Test the Basic Euchre ADTs

Test and Code Card

Write implementations in `Card.cpp` for the functions declared in `Card.h`. Using the unit test framework, write tests for `Card` in `Card_tests.cpp`. You can use the provided Makefile to compile and run the `Card` unit tests:

```
$ make Card_tests.exe
$ ./Card_tests.exe
```

Test and Code Pack

Write implementations in `Pack.cpp` for the functions declared in `Pack.h`. Using the unit test framework, write tests for `Pack` in `Pack_tests.cpp`. You can use the provided Makefile to compile and run the `Pack` unit tests:

```
$ make Pack_tests.exe
$ ./Pack_tests.exe
```

While you should write your own tests for `Pack` to ensure that your implementation is correct, you do not have to submit your tests to the autograder.

Test and Code Player

Write implementations in `Player.cpp` for the functions declared in `Player.h`. Using the unit test framework, write tests for `Player` in `Player_tests.cpp`.

The player tests should test the simple player, but not the human player.

You can use the provided Makefile to compile and run the `Player` unit tests:

```
$ make Player_tests.exe
$ ./Player_tests.exe
```

Writing the `Player_factory`

Since the specific types of Players are hidden inside `Player.cpp`, we need to write a factory function that returns a pointer to a `Player` with the correct dynamic type. We also need the pointed-to objects to stick around after the factory function finishes, so we'll create the players using dynamically allocated memory. The prototype for `Player_factory` can be found in `Player.h`, and the implementation will go in `Player.cpp`.

```
Player * Player_factory(const std::string &name,
                        const std::string &strategy) {
  // We need to check the value of strategy and return
  // the corresponding player type.
  if (strategy == "Simple") {
    // The "new" keyword dynamically allocates an object.
    return new SimplePlayer(name);
  }
  // Repeat for each other type of Player
  ...
  // Invalid strategy if we get here
  assert(false);
  return nullptr;
}
```

Writing Unit Tests for Card and Player

You must write and submit tests for the `Card` and `Player` classes. Your test cases MUST use the unit test framework, otherwise the autograder will not be able to evaluate them. Since unit tests should be small and run quickly, you are limited to **50** `TEST()` items per file and your whole test suite must finish running in less than 5 seconds. Please bear in mind that you DO NOT need 50 unit tests to catch all the bugs. Writing targeted test cases and avoiding redundant tests can help catch more bugs in fewer tests.

How We Grade Your Tests

We will autograde your `Card` and `Player` unit tests by running them against a number of implementations of those modules. If a test of yours fails for one of those implementations, that is considered a report of a bug in that implementation.

We grade your tests by the following procedure:

1. We compile and run your test cases with a **correct** solution. Test cases that pass are considered **valid**. Tests that fail (i.e. falsely report a bug in the solution) are invalid. The autograder gives you feedback about which test cases are valid/invalid. Since unit tests should be small and run quickly, your whole test suite must finish running in **less than 5 seconds**.
2. We have a set of intentionally **incorrect** implementations that contain bugs. You get points for each of these "buggy" implementations that your **valid** tests can catch.

3. How do you catch the bugs? We compile and run all of your **valid** test cases against each buggy implementation. If any of these test cases fail (i.e. report a bug), we consider that you have caught the bug and you earn the points for that bug.

Test and Implement the Euchre Game

This part will require the most planning. Before you begin, think about which helper functions you would like to add and what they should do. For example, functions that shuffle, deal and make trump are a good starting point. Your code in `euchre.cpp` should read the command line arguments, check them for errors, and then print them. Next, it should run the game simulation. After the game, it will need to delete the `Player` objects created by the `Player_factory` :

```
for (int i = 0; i < int(players.size()); ++i) {
  delete players[i];
}
```

Your program should finish by returning 0 from `main` (either with `return 0;` or just exiting `main` normally), unless an error is encountered as described below.

Protip: Good `main()` functions are VERY short! Make helper functions do the work!

Design and Implement a Game ADT

The Euchre simulator itself is a substantial program, as it must coordinate all the actions in the game. In order to manage the complexity, you will need to design and build a `Game` ADT. The game data, such as the players, pack of cards, and score, should be stored as member variables of the `Game` class. Each task in the game should have its own corresponding member function in the class. You should avoid writing a function that is too long, or that tries to accomplish more than one task without delegating to helper functions.

Compiling and Running the Program

Compile the main Euchre executable by typing `make euchre.exe` , which will run the command

```
$ g++ -Wall -Werror -pedantic --std=c++11 --g euchre.cpp Player.cpp Pack.cpp Card.cpp -o
```

The Euchre simulator takes several command line arguments to determine what kind of simulation to run. The following command will run a traditional game of Euchre:

```
$ ./euchre.exe pack.in shuffle 10 Edsger Simple Fran Simple Gabriel Simple Herb Simple
```

Each of the arguments are:

Argument	Purpose
`./euchre.exe`	Name of the executable
`pack.in`	Filename of the pack
`shuffle`	Shuffle the deck, or use `noshuffle` to turn off shuffling
`10`	Points to win the game
`Edsger`	Name of player 0
`Simple`	Type of player 0
`Fran`	Name of player 1
`Simple`	Type of player 1
`Gabriel`	Name of player 2
`Simple`	Type of player 2
`Herb`	Name of player 3
`Simple`	Type of player 3

The simulator checks for the following errors:

- There are exactly 12 arguments, including the executable name itself.
- Points to win the game is between 1 and 100, inclusive.
- The shuffle argument is either `shuffle` or `noshuffle` .
- The types of each of the players are either `Simple` or `Human` .

If the simulator finds any of the above errors, it should print the following message (and no other output) and quit by returning a non-zero value from `main` . **Do not use the `exit` library function, as this fails to clean up local objects.**

```
cout << "Usage: euchre.exe PACK_FILENAME [shuffle|noshuffle] "
     << "POINTS_TO_WIN NAME1 TYPE1 NAME2 TYPE2 NAME3 TYPE3 "
     << "NAME4 TYPE4" << endl;
```

Reading the Pack

The Euchre simulator reads a pack from a file. We have provided one pack, with the cards in "new pack" order. For example:

```
Nine of Spades
Ten of Spades
Jack of Spades
...
Queen of Diamonds
King of Diamonds
Ace of Diamonds
```

First, open the file and check for success. If the file open operation fails, use the following code to print an error message, and then quit by returning a non-zero value from `main`.

```
cout << "Error opening " << pack_filename << endl;
```

After the Pack file is open, you may assume that there are exactly 24 unique and correctly formatted cards. In other words, you don't have to worry about checking the contents of the file for errors.

Printing Output

Output that is specific to the Human Player should be printed in the appropriate place in the Human Player class. All other output that is common to both Simple and Human Players should be printed by the Euchre simulator itself. The Simple Player should not directly print any output.

Hints for System Testing

Use `euchre.cpp` to perform system tests on your game. Run a game from the command line and check its output using `sdiff`. We have provided several example tests, but you will need to add more. Use a regression test to rerun and check the output of all tests when you fix a bug or modify your code. We have provided the beginning of a regression test in the Makefile, which you can run by typing `make test`.

To run a simple system test manually, compile, run the program and redirect the output to a file. Then, use diff to compare the output to the correct output:

```
$ make euchre.exe
$ ./euchre.exe pack.in noshuffle 1 Adi Simple Barbara Simple Chi-Chih Simple Dabbala Simpl
$ sdiff euchre_test00.out.correct euchre_test00.out
```

To run a test with Human Players, you can redirect a file to standard input. The following redirects both input and output and compares the output to the correct output:

```
$ ./euchre.exe pack.in noshuffle 3 Ivan Human Judea Human Kunle Human Liskov Human < euchr
$ sdiff euchre_test50.out euchre_test50.out.correct
```

Appendix A: Example With Simple Players

The output for `./euchre.exe pack.in noshuffle 1 Adi Simple Barbara Simple Chi-Chih Simple Dabbala Simple` is saved in `euchre_test00.out.correct`. This section explains the output, line by line. Make sure that your simulator produces only output called for by this document.

First, print the executable and all arguments on the first line. Print a single space at the end, which makes it easier to print an array.

```
./euchre.exe pack.in noshuffle 1 Adi Simple Barbara Simple Chi-Chih Simple Dabbala Simple
```

At the beginning of each hand, announce the hand, starting at zero, followed by the dealer and the upcard.

```
Hand 0
Adi deals
Jack of Diamonds turned up
```

Print the decision of each player during the making procedure. Print an extra newline when making, adding, and discarding is complete.

```
Barbara passes
Chi-Chih passes
Dabbala passes
Adi passes
Barbara orders up Hearts
```

Each of the five tricks is announced, including the lead, cards played and the player that took the trick. Print an extra newline at the end of each trick.

```
Jack of Spades led by Barbara
King of Spades played by Chi-Chih
Ace of Spades played by Dabbala
Nine of Diamonds played by Adi
Dabbala takes the trick
```

At the end of the hand, print the winners of the hand. When printing the names of a partnership, print the player with the lower index first. For example, Adi was specified on the command line before Chi-Chih, so he goes first.

```
Adi and Chi-Chih win the hand
```

If a march occurs, print `march!` followed by a newline. If euchre occurs, print `euchred!` followed by a newline. If neither occurs, print nothing.

```
euchred!
```

Print the score, followed by an extra newline.

```
Adi and Chi-Chih have 2 points
Barbara and Dabbala have 0 points
```

When the game is over, print the winners of the game.

```
Adi and Chi-Chih win!
```

Appendix B: Example With Human Players

The output for `./euchre.exe pack.in noshuffle 3 Ivan Human Judea Human Kunle Human Liskov Human` is saved in `euchre_test50.out.correct`. The input is saved in `euchre_test50.in`. This section explains the output, line by line. Make sure that your simulator produces only output called for by this document.

First, print the executable and all arguments on the first line. Print a single space at the end, which makes it easier to print an array.

```
./euchre.exe pack.in noshuffle 3 Ivan Human Judea Human Kunle Human Liskov Human
```

At the beginning of each hand, announce the hand, starting at zero, followed by the dealer and the upcard.

```
Hand 0
Ivan deals
Jack of Diamonds turned up
```

Print the hand of each player during the making procedure, followed by a prompt for their making decision. End the prompt with a newline immediately after the colon.

```
Human player Judea's hand: [0] Nine of Spades
Human player Judea's hand: [1] Ten of Spades
Human player Judea's hand: [2] Jack of Spades
Human player Judea's hand: [3] King of Hearts
Human player Judea's hand: [4] Ace of Hearts
Human player Judea, please enter a suit, or "pass":
```

Print the decision of each player during the making procedure.

```
Judea passes
...
Judea orders up Hearts
```

Print the dealer's hand if a player orders up during the first round, as well as an option to discard the upcard. Prompt the dealer to select a card to discard, ending the prompt with a newline immediately after the colon. Print an extra newline when making, adding, and discarding is done.

```
Human player Ivan's hand: [0] Nine of Diamonds
...
Human player Ivan's hand: [4] Ace of Clubs
Discard upcard: [-1]
Human player Ivan, please select a card to discard:
```

For each trick, print the Human Player's hand and prompt them to select a card.

```
Human player Judea's hand: [0] Nine of Spades
...
Human player Judea's hand: [4] Ace of Hearts
Human player Judea, please select a card:
```

Then print the card played or lead.

```
Nine of Spades led by Judea
```

At the end of each trick, print the player who took the trick as well as an extra newline.

```
Liskov takes the trick
```

At the end of the hand, print the winners of the hand. When printing the names of a partnership, print the player with the lower index first. For example, Ivan was specified on the command line before Kunle, so he goes first.

```
Ivan and Kunle win the hand
```

If a march occurs, print `march!` followed by a newline. If euchre occurs, print `euchred!` followed by a newline. If neither occurs, print nothing.

```
euchred!
```

Print the score, followed by an extra newline.

```
Ivan and Kunle have 2 points
Judea and Liskov have 0 points
```

When the game is over, print the winners of the game.

```
Ivan and Kunle win!
```

Appendix C: Euchre Glossary

Trump: A suit whose cards are elevated above their normal rank during play.

Right Bower: The Jack card of the Trump suit, which is considered the highest-valued card in Euchre.

Left Bower: The Jack from the other suit of the same color as the Trump suit, considered the second highest-valued card in Euchre. The Left Bower is also considered a Trump card.

Next: The suit of the same color as trump.

Cross Suits: The two suits of the opposite color as trump.

Making: The process in which a trump card is chosen, consists of two rounds.

Eldest: Player to the left of the dealer.

Upcard: The up-facing card in front of the dealer that proposes the trump suit.

Order Up: Accepts the Upcard suit.

Pass: Player rejects the suit and passes on the decision to the next player.

Screw the Dealer: When making* reaches the dealer on round two, the dealer must *order up a suit other than the rejected one.

Lead: The first card played by the eldest* hand, regardless of who *is the maker.

Leader: Person playing the lead* card in a trick, allowed to lead* *any card.

March: When the side that made trump* wins all 5 tricks.

Euchred: When the side that didn't make trump* wins 3, 4, or 5 *tricks.

Appendix D: Operator Overloading

In C++, we use the output operator to print built-in types. For example:

```
cout << "My favorite number is  " << 42 << endl;
```

We can also use this convenient mechanism for our own custom types. Consider a simple class called `Thing` that keeps track of an ID number:

```
class Thing {
  int id; //Things store their ID number
public:
  Thing(int id_in) : id(id_in) {} // constructor
  int get_id() const { return id; }
};
```

We can add a function that lets us print a `Thing` object using `cout`, or any other stream. This is called an overloaded output operator.

```
std::ostream& operator<< (std::ostream& os, const Thing& t) {
  os << "Thing # " << t.get_id(); //send output to "os"
  return os; //don't forget to return "os"
}
```

Now, we can print `Thing` objects just as conveniently as we can print strings and integers!

```
int main() {
  Thing t1(7);
  cout << t1 << endl; //use overloaded output operator
}
```

This produces the following output:

```
Thing # 7
```

Let's say that we also want to be able to check if two `Thing` objects are equal. For this, we'll overload the == operator:

```
bool operator==(const Thing& first, const Thing& second) {
  return first.get_id() == second.get_id();
}
```

Now, we can easily check two `Thing` objects for equality:

```
int main() {
  Thing thing_1(42);
  Thing thing_2(42);
  Thing thing_3(43);
  cout << thing_1 == thing_2 << endl; // true
  cout << thing_1 == thing_3 << endl; // false
}
```

Appendix E: Project 3 Coding Practices Checklist

The following are coding practices you should adhere to when implementing the project. Adhering to these guidelines will make your life easier and improve the staff's ability to help you in office hours. You **do not** have to submit this checklist.

General code quality:

- ☐ Helper functions used if and where appropriate. Helper functions designed to perform one meaningful task, not more
- ☐ Lines are not too long
- ☐ Descriptive variable and function names (i.e. `int radius` instead of `int x`)
- ☐ Effective, consistent, and readable line indentation
- ☐ Code is not too deeply nested in loops and conditionals
- ☐ Main function is reasonably short

Test case quality:

- ☐ Test cases are small and test one behavior each.

- ☐Test case names are descriptive, or test cases are commented with a short description of what they test.
- ☐Test cases are written using the unit testing framework.

Project-specific quality:

- ☐Euchre simulator uses a `Game` class, with game data stored as member variables and separate member functions for each task
- ☐Avoids redundant use of `this` keyword
- ☐Does not use explicit operator calls (i.e. uses `card1 == card2` instead of `operator==(card1, card2)`)
- ☐No prohibited libraries are used