EECS 280 Project 4: Euchre

Due Friday, 20 November 2015, 11:55 pm

Euchre 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::vector, and C++ style strings.

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.

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.

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!

Download the starter code

Download the starter code from Ctools.

Familiarize yourself with the code structure.

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

Test and Implement the Basic Euchre ADTs.

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

Test and Implement the Euchre Game

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

Submit

Submit the following files to the autograder.

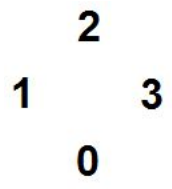
```
Card_tests.cpp
Card.cpp
Pack_tests.cpp
Pack.cpp
Player_tests.cpp
Player.cpp
euchre.cpp
```

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. Further 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
    a. Shuffle
    b. Deal
2. Making Trump
    a. Round one
    b. 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 we use for shuffling is a variant of the “one handed shuffle”. Cut the deck at 17 cards (that is, split the deck into two halves: cards 0-16 and 17-23) and swap the two parts. Do this 3 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 2. 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. nail down that 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.

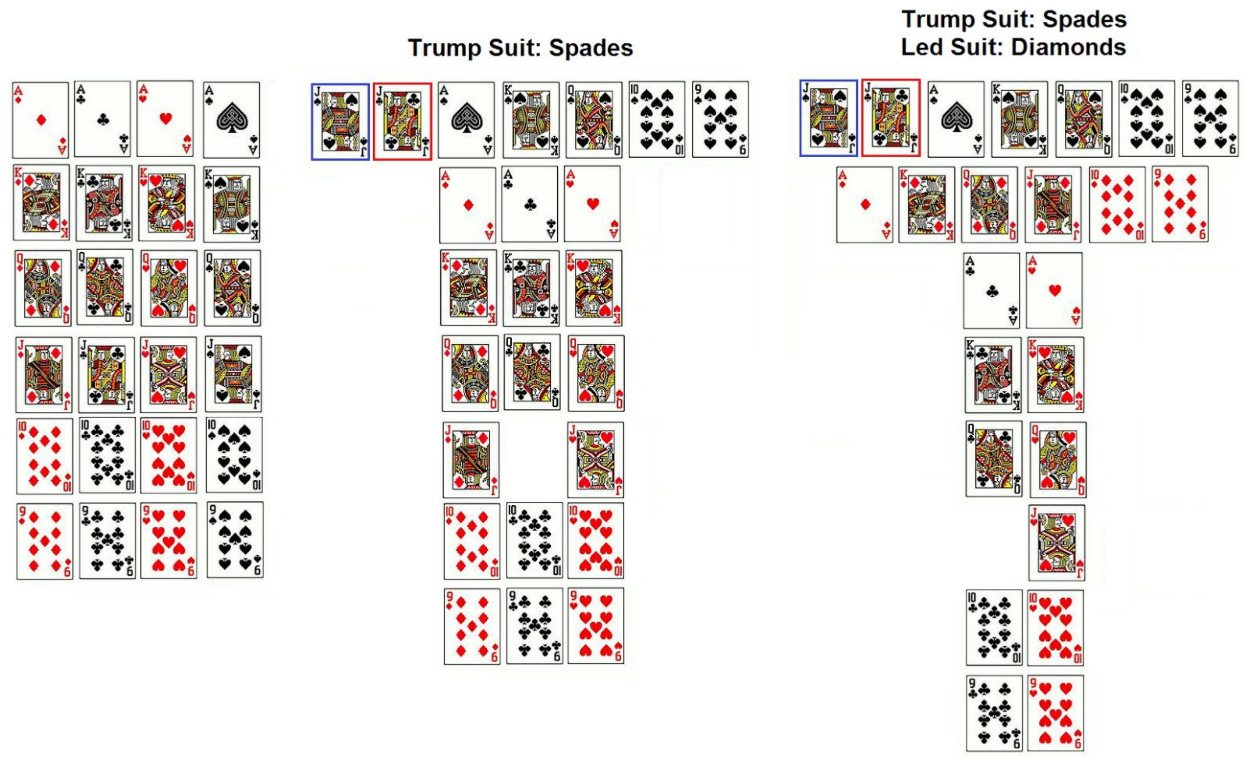
In the simplest case, cards are ordered by rank (A > K > Q > J > 10 > 9), with ties broken by suit (D > C > H > S).

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.



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

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 will order up if 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. 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, counting the upcard, for a final total of five cards.

During round two, a Simple Player will order up the suit with the same color as the upcard if they have one or more face cards of that suit in their hand. 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.

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. For simplicity, you may assume that all user input is correctly formatted and has correct values. You may also assume that 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. 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.

If a Human Player is the dealer and someone orders up during the first round of making, 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:

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

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
Put any extra helper functions in the `.cpp` files and declare them `static`	Modify `.h` files
Use these libraries: `<iostream>`, `<fstream>`, `<cstdlib>`, `<cassert>`, `<cstring>`, `<vector>`, `<string>`	Use other libraries
`#include` a library to use its functions	Assume that the compiler will find the library for you (some do, some don't)
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
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`.

Starter Code

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

`Card.h`	Procedural abstraction representing operations on a playing card.
`Card.cpp.starter`	Starter code for `Card.cpp`, which will contain function implementations for the prototypes in `Card.h`. Copy this file to `Card.cpp` to get started.
`Card_tests.cpp`	Unit tests for functions in `Card.h`. Your Card unit tests will be graded on how well they expose bugs.
`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`	Unit tests for functions in `Pack.h`. You are required to write at least one additional test per function.
`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`	Unit tests for functions in `Player.h`. Your unit tests for Simple Player will be graded on how well they expose bugs.
`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.

Implement and Test the Basic Euchre ADTs

Test and Code Card

Implement the functions in `Card.cpp` whose prototypes are declared in `Card.h`. Some static variable definitions are given in `Card.cpp.starter`. Remember, only modify `Card.cpp`, not `Card.h`!

Compile the Card unit tests by typing **`make Card_tests`**, which runs the command

```
g++ -pedantic -Wall -Werror -O1 Card_tests.cpp Card.cpp -o Card_tests
```

Run the unit tests by typing **`./Card_tests`** at the command line.

Test and Code Pack

Write your functions in `Pack.cpp`. Some tests are provided in `Pack_tests.cpp`.

Compile the Pack unit tests by typing **`make Pack_tests`**, which will run the command

```
g++ -pedantic -Wall -Werror -O1 Pack_tests.cpp Pack.cpp Card.cpp -o
Pack_tests
```

Run the unit tests by typing **`./Pack_tests`** at the command line.

Test and Code Player

Write your functions in `Player.cpp`. Hint: the comparison functions in `Card.h` will be very helpful! Like previous units, use `Player_tests.cpp` to test the functions in `Player.h`. Write more unit tests to do more thorough testing.

Compile the Player unit test by typing **`make Player_tests`**, which will run the command

```
g++ -pedantic -Wall -Werror -O1 Player_tests.cpp Player.cpp Card.cpp
-o Player_tests
```

Run the unit test by typing **`./Player_tests`** at the command line.

Note: to debug your code in gdb, replace `-O1` with `-g` in your Makefile or compilation command.

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);
      exit(1);
}
```

Hints for Unit Testing

Do not, under any circumstances, split the work and have one partner write the code and the other partner write the tests. This is a good way to get VERY FEW POINTS™ on the project.

Test-Driven Development

Protip (again): Write tests for the functions FIRST! (e.g. write tests for Card_less(), and then implement Card_less()). It sounds like a pain, but you gain two important things by coding this way:

1. You avoid being under the illusion that your code works when it's actually full of bugs.
2. When you make changes to your code, you can re-run *all* your test cases and immediately know if you broke something (yes, you will break things).

Submitting Unit Tests for Card, Pack, and Player

For each of the public functions in Card.h, Pack.h, and Player.h that you implement in the corresponding .cpp files, you must write and submit test cases. Add your Card test cases to Card_tests.cpp, add your Pack tests to Pack_tests.cpp, and add your Simple Player tests to Player_tests.cpp. As with previous projects, your tests must use **`assert()`** or return nonzero to force the program to exit with nonzero status when a test fails, and each test must be implemented as its own function.

We will be grading your **Card** and **Simple Player** tests based on how effectively they expose bugs. In order to do this, we will run your test cases against a correct implementation of Card.cpp or Player.cpp and against a number of intentionally buggy implementations of Card.cpp and Player.cpp. To receive full credit, your test cases must return 0 from main when linked with a correct solution and must return nonzero (which an `assert()` will accomplish) when linked with an incorrect solution.

Do NOT submit test cases for the Human Player.

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];
     players[i] = 0;
}
```

Your program should finish with `return 0` at the end of main.

Protip: Good `main()` functions are VERY short! Make helper functions do the work!
Protip: Write a `Game` struct to more easily pass game data between your helper functions (by reference)!

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

```
g++ -pedantic -Wall -Werror -O1 euchre.cpp Player.cpp Pack.cpp
Card.cpp -o euchre
```

Run the program by providing command line arguments, for example:

```
./euchre pack.in shuffle 10 Alice Simple Bob Simple Cathy Simple Drew
Simple
```

Running the program

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 pack.in shuffle 10 Alice Simple Bob Simple Cathy Simple Drew Simple
```

Each of the arguments are:

`./euchre`	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
`Alice`	Name of player 0
`Simple`	Type of player 0
`Bob`	Name of player 1
`Simple`	Type of player 1
`Cathy`	Name of player 2
`Simple`	Type of player 2
`Drew`	Name of player 3

`Simple` Type of player 3

The simulator checks for the following errors:

- There are exactly 11 arguments, in addition to 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 calling `exit(EXIT_FAILURE)`, which is from `<cstdlib>`. The `exit()` function terminates the program, and passing it `EXIT_FAILURE` indicates an error.

```
cout << "Usage: euchre PACK_FILENAME [shuffle|noshuffle]
POINTS_TO_WIN NAME1 NAME2 NAME3 NAME4" << endl;
```

Reading the Pack

The Euchre simulator reads a pack from a file (implemented in `Pack.cpp`). 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 call `exit(EXIT_FAILURE)`.

```
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.

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:

```
g++ -pedantic -Wall -Werror -O1 euchre.cpp Game.cpp Player.cpp
Pack.cpp Card.cpp -o euchre
./euchre pack.in noshuffle 1 Alice Simple Bob Simple Cathy Simple
Drew Simple > euchre_test00.out
```

```
sdiff euchre_test00.out.correct euchre_test00.out
```

Acknowledgements

The original project was written by Andrew DeOrio, Fall 2013. Thanks to Vidal Borromeo and Eyad Makki for their valuable feedback.

Appendix A: Example With Simple Players

The output for `./euchre pack.in noshuffle 1 Alice Simple Bob Simple Cathy Simple Drew 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 pack.in noshuffle 1 Alice Simple Bob Simple Cathy Simple Drew Simple
```

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

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

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

```
Bob passes
Cathy passes
Drew passes
Alice passes
Bob 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 Bob
King of Spades played by Cathy
Ace of Spades played by Drew
Nine of Diamonds played by Alice
Drew 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, Alice was specified on the command line before Cathy, so she goes first.

```
Alice and Cathy 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.

```
Alice and Cathy have 2 points
Bob and Drew have 0 points
```

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

```
Alice and Cathy win!
```

Appendix B: Example With Human Players

The output for `./euchre pack.in noshuffle 1 Alice Human Bob Human Cathy Human Drew 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 pack.in noshuffle 1 Alice Human Bob Human Cathy Human Drew Human
```

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

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

Print the hand of each player during the making procedure, followed by a prompt for their making decision.

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

Print the decision of each player during the making procedure.

```
Bob passes
…
Cathy orders up Diamonds
```

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. Print an extra newline when making is done.

```
Human player Alice’s hand: [0] Jack of Hearts
...
Human player Alice’s hand: [4] Ten of Diamonds
Discard upcard: [-1]
Human player Alice, please select a card to discard:
```

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

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

Then print the card played or lead.

```
Nine of Spades led by Bob
```

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

```
Drew 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, Alice was specified on the command line before Cathy, so she goes first.

```
Alice and Cathy 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.

```
Alice and Cathy have 1 points
Bob and Drew have 0 points
```

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

```
Alice and Cathy 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
}
```