EECS 280 Project 3: Euchre

Due Wednesday, 6 November 2013, 11:55 pm
Mid-project checkpoint due Wednesday, 16 October 2013, 11:55pm

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 composable data structures, object-oriented programming techniques and decomposing a task using functions. While building the simulator, you will use pointers, arrays, structures, strings and file IO.

Euchre Rules

There are many variants of Euchre. The rules for this project are derived from Goren's Hoyle Encyclopedia of Games [Goren 1961], with the addition of several variations common in Michigan play.

Number of players

Four, in two partnerships.

The pack

A pack of 24 cards, made by discarding from a full pack all cards below the nines. In other words, ranks nine through ace of all four suits.

Order of cards

Cards are ordered according to which suit is *trump*, a suit whose cards are elevated above their normal rank during play. The highest card is jack of the trump suit, and is called the *right bower*. The second highest card is the jack of the other suit of the same color as the trump, and is called the *left bower*. The left bower is also considered a trump 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.

To simplify debugging and ensure that every correct solution will get the same output, we've added an additional rule here. In addition to being ordered by rank, cards are also ordered by suit: diamonds first, then clubs, hearts, spades.

The complete ordering of all the cards is: J (right bower, high), J (left bower), A, K, Q, 10, 9 of the trump suit. The aces of the remaining three suits follow in suit order, the remaining kings in suit order, …, and finally the remaining nines in suit order.

Drawing for partnerships

We've simplified the rules a bit here. Partnerships will be fixed, where four players (0-3) are

partnered 0 & 2, and 1 & 3.

Shuffle and cut

Again, we've simplified the rules here. The dealer shuffles and cuts the deck before each hand. The algorithm we use for shuffling is a variant of the "one handed shuffle". Cut the deck at 17 cards (that is, cards 0-16 and 17-23) and swap the two parts. Do this 3 times. Later, we'll discuss an option that disables shuffling for easier testing and debugging. If shuffling is disabled, then reset the pack back to the beginning.

The deal

In our game, the first dealer is the first player. Each hand, the deal moves one player to the left. Each player receives five cards, dealt in batches of 3-2. The first person to receive a card is left of the dealer. That is, deal 3-2-3-2 cards then 2-3-2-3 cards, for a total of 5-5-5-5 cards. After all hands are dealt, the next card of the pack is turned up to propose the trump suit.

Making trump

The turned up card in front of the dealer proposes the trump suit, but its acceptance depends on up to two rounds of *making.* Eldest hand (player at left of the dealer, e.g., player 3 is left of player 2, and player 0 is left of player 3) speaks first. He may either *pass* or *order up*. *Order up* accepts the turn-up suit. If the trump suit is ordered up during round one, the dealer adds the up card to his hand and discards one card to bring his hand back to five cards, total. Hint: be careful not to "go off the end of an array" here! If a player passes, the next player to the left speaks.

If all players pass during the first round, there is a second round of declaring, again beginning with the eldest hand. This time, each player in turn may either pass or order up by naming a suit other than the rejected suit.

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.

Side note for pro Euchre players: we have omitted "going alone" in this project, a simplification.

The play

The first card, called the lead, is played by the eldest hand, regardless of who is the maker. The leader (person playing the lead card) to any trick may lead any card. Play moves to the left around the table, with each hand playing one card. Each other hand must follow the led suit, if able, and otherwise may play any card. A trick is won by the highest trump, or, if it contains no trump, by the highest card of the suit led. At the end of a trick, played cards are discarded. The winner of the trick (or round) leads the next.

Object of play

The objective is to win a majority of tricks.

Scoring

Only the side that wins the majority of tricks scores. The side that made trump wins 1 point for 3 or 4 tricks, and 2 points for taking all 5, which is called *march.* The other side wins 2 points for taking 3, 4 or 5 tricks, which is called *euchered.*

The first side to reach 10 points wins. In this project, the number of points to win will be configurable.

Euchre strategy

Making

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

During round one, a player will order up if he has two or more trump face cards in his 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 up card in this case. If the trump suit is ordered up during round one, the dealer receives it. He then discards the lowest card among his hand and the turn-up, for a final total of five cards.

During round two, a player will order up the suit with the same color as the up card if he has one or more face cards of that suit in his hand. For example, if the up card was a heart and he had the king of diamonds in his hand, he would order up diamonds. 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 up card.

Leading

When a player leads a trick, he plays the highest non-trump card in his hand. If he has only trump, he plays the highest trump card in his hand.

Playing

When playing a card, 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, but again, we will keep it simple.

If a player can follow suit, he plays the highest card that follows suit. Otherwise, he plays the lowest card in his hand.

Example

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

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

```
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 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!
```

Code structure

The code is structured as an *object oriented* program. The C++ struct mechanism is used to represent entities in the Euchre world, for example `Card`, `Pack`, and `Player`. These structures are defined in several header files, and they interact with each other through functions that form a *procedural abstraction*. Your task is to implement these functions 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.

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 Bob Cathy Drew
```

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
`Bob`	Name of player 1
`Cathy`	Name of player 2
`Drew`	Name of player 3

The simulator checks for the following errors:

- There are exactly 7 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`

If the simulator finds any of the previous errors, it should print the following message and quit by calling `exit(EXIT_FAILURE)`, which is from `<cstdlib>`. The `exit()` function terminates the program.

```
    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 (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.

C-strings and C++ strings

The vast majority of this project uses C-strings. Use C++ strings only in conjunction with C++ IO, for example, when reading from a file.

When using C-strings, you can assume a maximum array size of 1024. Hint: the only place you will need this is in `Player.cpp`, which already has `const int MAX_STR_LEN = 1024;` in `Player.h`. Also, don't forget to use `strcpy()` when copying C-strings!

Restrictions

In this project, it is our goal for you to gain practice with good C-style code, structs, and pointers. To encourage this style, we have a few restrictions, described in the following table.

Do's	**Don'ts**
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>`, `<cstdio>`, `<cstring>`, `<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 to read from files and C-strings everywhere else	Use C++ strings everywhere
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
C-style functions that pass a struct by pointer	Classes or member functions
Pass large structs by pointer	Pass large structs by value
Pass by const pointer when appropriate	"I don't think I'll modify it …"

Variables on the stack	Dynamic memory (`new`, `malloc()`, etc.)

How to complete this project

Play a (real) game of Euchre

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

Download the starter code

Download the starter code from `/afs/umich.edu/class/eecs280/proj3/` The following table describes each file.

`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_test.cpp.starter`	Unit tests for functions in `Card.h` . Copy this file to `Card_test.cpp` to get started.
`Pack.h`	Procedural abstraction representing operations on a pack of playing cards
`Pack_test.cpp.starter`	Unit tests for functions in `Pack.h`. You will need to add more. Copy this file to `Pack_test.cpp` to get started.
`Player.h`	Procedural abstraction representing operations on a euchre player
`Player_test.cpp.starter`	Unit tests for functions in `Player.h`. You will need to add more. Copy this file to `Player_test.cpp` to get started.
`Game.h`	Procedural abstraction representing operations on a game of euchre
`pack.in`	Input file containing a euchre deck
`Makefile.starter`	Used by the make command to compile the executable. Type `make` at the command line.

	Copy this file to `Makefile` to get started.

`euchre_test00.out.correct`
 	
`euchre_test01.out.correct`
	Correct output for system tests of main executable. The first line of each file contains the command used to generate it.

Code and test Card

Implement the functions in `Card.cpp` whose prototypes are declared in `Card.h`. Many of the function implementations are given in `Card.cpp.starter`. Remember, only modify `Card.cpp`, not `Card.h`! After writing each function, test it using unit tests in `Card_test.cpp`. Many tests are provided in `Card_test.cpp.starter`.

Compile the Card unit test by typing `make Card_test`, which runs the command
`g++ -pedantic -Wall -Werror -O1 Card_test.cpp Card.cpp -o Card_test`

Run the unit test by typing `./Card_test` at the command line. The test will return 0 and print a message if it passes. It will exit non-zero with an error message if a test fails.

Code and test Pack

Write your functions in `Pack.cpp` and test them in `Pack_test.cpp`. Some tests are provided in `Pack_test.cpp.starter`, but you will need to add more.

Compile the Pack unit test by typing `make Pack_test`, which will run the command
`g++ -pedantic -Wall -Werror -O1 Pack_test.cpp Pack.cpp Card.cpp -o Pack_test`

Run the unit test by typing `./Pack_test` at the command line. The test will return 0 and print a message if it passes. It will exit non-zero with an error message if a test fails. In `Pack_test.cpp`, Use `assert()` to check for failures, or alternatively, use `if()` statements and call `exit(EXIT_FAILURE)` or `return EXIT_FAILURE` if a test fails.

Code and test Player

Write your functions in `Player.cpp` and test them in `Player_test.cpp`. Like previous tests, `Player_test.cpp` should contain units tests for the functions in `Player.h`.

Compile the Player unit test by typing `make Player_test`, which will run the command
`g++ -pedantic -Wall -Werror -O1 Player_test.cpp Player.cpp Pack.cpp Card.cpp -o Player_test`

Run the unit test by typing `./Player_test` at the command line. Like previous unit tests, the test will return 0 and print a message if it passes.

Code main

The main function coded in `euchre.cpp` will be very simple. It should read the command line arguments, check them for errors, and then print them. Next, call `Game_init()` followed by `Game_play()`. Finish with `return EXIT_SUCCESS` at the end of main.

In order for main to compile, it will need `Game.cpp` to compile. For `Game.cpp` to compile, all the functions from `Game.h` need to be in `Game.cpp`. Add *function stubs* to `Game.cpp`, which do nothing but return a constant value. This will allow you to compile the whole project, even if you aren't finished yet. Here is an example of a function stub, from `Game.cpp`:

```
void Game_play(Game *game_ptr) {
  return;
}
```

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

```
g++ -pedantic -Wall -Werror -O1 euchre.cpp Game.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 Bob Cathy Drew
```

Code 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. Remember, only add functions to `Game.cpp` and declare them static. Here is an example of a possible helper function from `Game.cpp`:

```
//REQUIRES: game_ptr is initialized
//MODIFIES: game_ptr
//EFFECTS: If the shuffle option is selected shuffle the pack.
//  Otherwise, reset the pack.
static void Game_shuffle(Game *game_ptr);
```

Test Game using main

Use `euchre.cpp` to perform system tests on your Game. Run a game from the command line and check its output using `diff`. We have provided several example tests and 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 Bob Cathy Drew > euchre_test00.out
diff -q euchre_test00.out euchre_test00.out.correct
```

Submit

This project has two submissions. The first is a mid-project checkpoint, where several components are due. The second is the final deadline for the completed project.

For the mid-project checkpoint, submit the following files using `submit280 3a .` The `Card`, `Pack` and `Player` components should pass the published unit tests.

```
Card.cpp Pack.cpp Player.cpp
```

For the final submission, submit the following files using `submit280 3 .` For this part of your grade, we will not use any code turned in at the checkpoint.

```
Card.cpp Pack.cpp Player.cpp Game.cpp euchre.cpp
```

Acknowledgements

Original project written by Andrew DeOrio, fall 2013. The text for the basic rules of Euchre was adapted from Goren's Hoyle Encyclopedia of Games [Goren, 1961].

Appendix A: 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 card in Euchre.

Left Bower: The Jack from the other suit of the same color as the *Trump* card, considered as the second highest 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 of trump.

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

Eldest: Player to the left of the dealer.

Turn-up: The turned up card in front of the dealer that proposes the *trump* suit.

Order up: Accepts the *Turn-up* 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 B: Makefile tutorial

Make is a tool that helps you compile a program from the program's source files. Using make, you can avoid typing long and tedious compiler commands. For example, one way to compile this project is to type the following command at the terminal:

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

Another way to compile this project is to use `make`, which reads the provided `Makefile` and finds the right command. Simple type `make` at the command line.

Let's look into the project 3 Makefile. Makefiles contain rules, which tell Make how to execute a series of commands in order to create a target file from source files. It also specifies a list of dependencies of the target file. This list should include all files which are used as inputs to the commands in the rule. A command is preceded by a tab.

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

Card_test : Card_test.cpp Card.cpp Card.h
	g++ -pedantic -Wall -Werror -O1 Card_test.cpp Card.cpp -o Card_test

test : Card_test euchre
	./Card_test
	./euchre pack.in noshuffle 1 Alice Bob Cathy Drew > euchre_test00.out
	diff -q euchre_test00.out euchre_test00.out.correct
	./euchre pack.in shuffle 10 Alice Bob Cathy Drew > euchre_test01.out
	diff -q euchre_test01.out euchre_test01.out.correct

clean :
	rm -vf *.out euchre Card_test
```

The provided Makefile contains targets to build the `euchre` executable and a `Card_test` executable. By default make will build the first target. For a different target, specify it after the make command. For example, typing `make Card_test` will build the `Card_test` executable. `make test` will run several tests and check the output using `diff`. Finally, `make clean` will remove the executables all files ending in "`.out`".

Appendix C: Debugging pro-tips

First, don't forget to change your compiler flags before debugging. Remove `-O1` and add `-g`.

Use your debugger like a pro

Our euchre simulator has command line arguments and input files. First make sure your input file is in the same directory as the executable. Next , tell the debugger to provide command line arguments to the executable. You only need to give the arguments the first time you call run. After that, it will remember them. For example, in gdb:

```
$ gdb euchre
(gdb) run pack.in noshuffle 1 Alice Bob Cathy Drew
   ...
(gdb) run
   ...
```

Dereference pointers inside gdb to inspect them:

```
$ gdb euchre
(gdb) break Player_play_Card
(gdb) run pack.in noshuffle 1 Alice Bob Cathy Drew
(gdb) print player_ptr
$1 = (Player *) 0x7fffffffd350
(gdb) print *player_ptr
$3 = {
  name = ...
  hand = ...
  hand_size = 5
}
```

Call functions inside gdb:

```
$ gdb euchre
(gdb) break Player_play_Card
(gdb) run pack.in noshuffle 1 Alice Bob Cathy Drew
(gdb) call Player_print(player_ptr)
Cathy
  Queen of Spades
  King of Spades
  Nine of Clubs
  Ten of Clubs
  Jack of Clubs
```

Assert like a pro

Use `assert()` to detect problems before they happen. `assert(STATEMENT)` will crash the program with a debugging message if `STATEMENT` is false. For example `assert(0)` will fail

every time.

Checking REQUIRES clauses

For example you could use an assertion to avoid accidentally trying to ask a player with an empty hand to play a card:

```
Card Player_play_Card(Player *player_ptr, Suit trump,
                      Suit led_suit) {
  assert(player_ptr->hand_size > 0);
  //...
}
```

Checking for NULL pointers

An assert() is a great way to check for a NULL pointer. This technique is most useful if you make the habit of always initializing pointers to NULL.

```
void Player_print(const Player *player_ptr) {
  assert(player_ptr); // will crash if player_ptr is NULL
  cout << player_ptr->name << "\n";
  //...
}
```

Disabling assert

To disable the assert() statements in a file, simply add `#define NDEBUG` to each .cpp that uses assert().