Blackjack!
EECS 280 – Fall 2013
Due: Tuesday 26 November 2013, 11:55pm

Introduction

This project will give you experience implementing abstract data types, using interfaces (abstract base classes), and using interface/implementation inheritance.

Blackjack (Simplified)

Blackjack, also sometimes called 21, is a relatively simple game played with a standard deck of 52 playing cards. There are two principals, a dealer and a player. The player starts with a bankroll, and the game progresses in rounds called hands.

At the start of each hand, the player decides how much to wager on this hand. It can be any amount between some minimum allowable wager and the player's total bankroll, inclusive.

After the wager, the dealer deals a total of four cards: the first face-up to the player, the second face-up to himself, the third face-up to the player, the fourth face-down to himself.

The player then examines his/her cards, forming a total. Each card numbered 2-10 is worth its rank value (the number on the card); each “face” card (jack, king, queen) is worth 10. An ace is worth either 1 or 11--whichever is more advantageous to the player. If the total includes an ace counted as 11, the total is called "soft", otherwise it is called "hard".

Play progresses first with the player, then the dealer. The player's goal is to build a hand that totals as close to 21 as possible without going over. If the total is over 21, this is called a "bust", and a player who busts loses the hand without forcing the dealer to play. As long as the player believes another card will help, the player "hits" by asking the dealer for another card. Each of these additional cards is dealt face-up. This process ends either when the player decides to "stand" (ask for no more cards) or the player busts. Note that a player can stand with two cards; one need not hit at all in a hand.

If the player is dealt an ace plus any ten or face card, the player's hand is called a "natural 21", and the player's wager is paid off with 3 to 2 odds, without examining the dealer's cards. In other words, if the player had wagered 10, the player would win 15 if dealt a natural 21.

If the player neither busts nor is dealt a natural 21, play then progresses to the dealer. The dealer **must** hit until he either reaches a total greater than or equal to 17 (hard or soft), or busts. If the dealer busts, the player wins. Otherwise, the two totals are compared. If the dealer's total is higher, the player's bankroll decreases by the amount of his/her wager. If the player's total is higher, her bankroll increases by the amount of her wager. If the totals are equal, the bankroll is unchanged; this is called a "push". The only case where the hands are equal that is not a push is when the player and dealer are each dealt natural 21s. In that case, the player is still paid 3:2.

Note that this is a very simplified form of the game: we do not split pairs, allow double-down bets, or take insurance. Likewise, a natural 21 for the dealer does not end the hand preemptively.

Programming Assignment

You will provide write implementations of several abstractions for this project: a card, a deck of cards, a blackjack hand, a blackjack player, and a game driver. All files referenced in this specification are located at:

/afs/umich.edu/class/eecs280/proj4

You may copy them to your private directory space, but may not modify them in any way. This will help ensure that your submitted project compiles correctly. For this project, the penalty for code that does not compile will be severe, regardless of the reason.

The Card ADT

Your first task is to implement a Card ADT, which is specified in `Card.h`. Put your implementations in `Card.cpp`.

The Deck ADT

Your next task is to implement a Deck ADT representing a deck of cards, which is specified in `Deck.h`. Put your implementations in `Deck.cpp`.

The Hand Interface

Your next task is to implement a blackjack Hand ADT representing a hand of cards, which is specified in `Hand.h`. Put your implementations in `Hand.cpp`.

The Player Interface

Your next task is to implement three different blackjack players. The Player ADT is specified in `Player.h` You will implement three different derived classes from this interface, coding them in `Player.cpp`.

The first derived class is the Simple player, who plays a simplified version of basic strategy for blackjack. The simple player always places the minimum allowable wager, and decides to hit or stand based on the following rules and whether or not the player has a “hard count” or “soft count”:

The first set of rules apply if the player has a "hard count" (i.e. his/her best total counts an Ace (if any) for 1, not 11).

- If the player's hand totals 11 or less, he always hits.
- If the player's hand totals 12, he stands if the dealer shows 4, 5, or 6; otherwise he hits.

- If the player's hand totals between 13 and 16 inclusive, he stands if the dealer shows a 2 through a 6 inclusive; otherwise he hits.
- If the player's hand totals 17 or greater, he always stands.

The second set of rules applies if the player has a "soft count" (i.e. his/her best total includes one Ace worth 11). Note that a hand would never count two Aces as 11 each – that's a bust of 22.

- If the player's hand totals 17 or less, he always hits.
- If the player's hand totals 18, he stands if the dealer shows a 2, 7, or 8, otherwise he hits.
- If the player's hand totals 19 or greater, he always stands.

The Simple player does nothing for expose and shuffled events.

The second derived class is the Counting player. This player counts cards in addition to playing the basic strategy. The intuition behind card counting is that when the deck has more face cards (worth 10 each) than low-numbered cards, the deck is favorable to the player. The converse is also true.

The Counting player keeps a running "count" of the cards he's seen from the deck. Each time he sees (via the `expose()` method) a 10, Jack, Queen, King, or Ace, he subtracts one from the count. Each time he sees a 2, 3, 4, 5, or 6, he adds one to the count. When he sees that the deck is `shuffled()`, the count is reset to zero. Whenever the count is +2 or greater, the Counting player bets double the minimum, otherwise he bets the minimum. If the count is +2 or greater, but the player does not have double the minimum in his bankroll, then he should bet his entire bankroll. The Counting player should not re-implement methods of the Simple player unnecessarily.

The final derived class you are to implement is the Competitor. The Competitor can play any strategy you choose. The Competitor cannot play the same strategy as the Simple or Counting players---there must be some difference, however minor. The quality of the Competitor's play will not count toward your grade, however, you should try to get it to perform better than the Counting player.

You must also declare a static global instance of each of the three Players in player.cpp . Finally, in player.cpp, you should implement the following factory function that returns a pointer to one of these three global instances.

```
Player * player_factory(const char * s);
// REQUIRES: s is a C-string: "simple" "counting" or "competitor"
// EFFECTS: returns a pointer to the specified Player
```

We've structured the Player as an Abstract Base Class in `Player.h` so that you have complete design freedom for the Competitor and its state.

The Driver program

Finally, you will implement a driver program that simulates our version of blackjack in `blackjack.cpp` . The driver program takes three command line arguments:

```
blackjack <bankroll> <hands> [simple|counting|competitor]
```

The first argument is an integer denoting the player's starting bankroll. The second argument is the maximum number of hands to play in the simulation. The final argument is one of the three strings "simple", "counting", or "competitor", denoting which of the three players to use in the simulation.

The driver first shuffles the deck. To shuffle the deck, you choose seven cuts between 13 and 39 inclusive at random, shuffling the deck with each of these cuts. We have supplied a header, `rand.h`, and an implementation, `rand.cpp`, that together define a function that provides these random cuts. Each time the deck is shuffled, first announce it:

```
cout << "Shuffling the deck\n";
```

And announce each of the seven cut points. Note, this output must be generated by code in `blackjack.cpp`, not in `Deck::shuffle()`.

```
cout << "cut at " << cut << endl;
```

then be sure to tell the player that a shuffle has occurred via `shuffled()`.

Then, while the player's bankroll is greater than or equal to minimum bet of 5 and there are hands left to be played:

- Announce the hand:

  ```
  cout << "Hand " << thishand << " bankroll " << bankroll << endl;
  ```

- If there are fewer than 20 cards left in the deck, reshuffle the deck as described above.

- Ask the player for a wager and announce it:

  cout << "Player bets " << wager << endl;

- Deal four cards: one face-up to the player, one face-up to the dealer, one face-up to the player, and one face-down to the dealer. Announce the face-up cards using `cout`. For example:

  ```
  Player dealt Ace of Spades
  Dealer dealt Two of Hearts
  ```

  Use the output operator for this, and be sure to `expose()` any face-up cards to the player.

- If the player is dealt a natural 21, immediately pay the player 3/2 of his/her bet. Note that, since we are working with integers, you'll have to be a bit careful with the 3/2 payout. For example, a wager

of 5 would pay 7 if a natural 21 is dealt: (3*5)/2 is 7 in integer arithmetic. In this case, announce the win:

```
cout << "Player dealt natural 21\n";
```

- If the player is not dealt a natural 21, have the player play his/her hand. Draw cards until the player either stands or busts. Announce and expose each card dealt as above.

- Announce the player's total

  ```
  cout << "Player's total is " << p_count << endl;
  ```

  and if the player busts, say so

  ```
  cout << "Player busts\n";
  ```

  deducting the wager from the bankroll and moving on to the next hand.

- If the player hasn't busted, announce and expose the dealer's hole card. For example:

  ```
  Dealer's hole card is Ace of Spades
  ```

  (Note: the hole card is NOT exposed if either the player busts or is dealt a natural 21.)

- If the player hasn't busted, play the dealer's hand. The dealer must hit until reaching seventeen or busting. Announce and expose each card as above.

- Announce the dealer's total

  ```
  cout << "Dealer's total is " << p_count << endl;
  ```

  and if the dealer busts, say so

  ```
  cout << "Dealer busts\n";
  ```

  Add the wager to the bankroll and move on to the next hand.

- If neither the dealer nor the player bust, compare the totals and announce the outcome. Add to the bankroll, subtract from it, or leave it unchanged as appropriate.

```
cout << "Dealer wins\n";
cout << "Player wins\n";
cout << "Push\n";
```

- Finally, when the player either has too little money to make a minimum wager **or** the allotted hands have been played, announce the outcome:

```
cout << "Player has " << bankroll << " after "
     << thishand-1 << " hands\n";
```

Implementation Rules

- You may `#include <iostream>`, `<iomanip>`, `<string>`, `<cstring>`, `<cstdlib>`, and `<cassert>`. No other system header files may be included, and you may not make any call to any function in any other library (even if your IDE allows you to call the function without including the appropriate header file). You may also include <cmath>, but only for use in your Competitor. Other `#include` files for your Competitor may be allowed at the discretion of the course staff, so just ask.

- Input and/or output should only be done where it is specified.

- You may not use the `goto` command.

- You may not have any global variables in the driver, but global constants are allowed. You may use global state in the class implementations, but it must be static and (except for the three players) const.

- There is no user input. You may assume that functions are called consistent with their advertised specifications. This means you need not perform error checking. However, when testing your code in concert, you may use the assert() macro to program defensively.

- You are strongly discouraged from using any dynamic memory facilities (new/delete and friends).

- Since each of your implementation (.cpp) files will be graded separately by compiling it with our solution code, do not write code that contains inter-file dependencies such that one of your files depends on another one of your files. For example, do not write an extern function in hand.cpp that you use in player.cpp.

Testing

For this project, you are required to write, describe, and submit individual, focused test cases for your Deck and Counting Player. For both of these ADTs, determine the behaviors required of the implementation. Then, for each of these behaviors:

- List the specific behavior that the implementation must exhibit in at most three sentences.

- Write a program that, when linked against the implementation of the ADT tests for the presence/absence of that behavior.

For the Deck and the Counting player, submit the following files:

`<impl>_test.overview`	the list of behaviors and test case filenames.
`<impl>_test01.cpp`	the first test case named in `<impl>_test.overview`
`<impl>_test02.cpp`	the second
...	
`<impl>_testNN.cpp`	the NNth

For example, if you identify three behaviors in Deck, and two in the Counting player, there would be seven files (**be sure to use the correct filenames or you will not receive full credit!**):

```
Deck_test.overview
Deck_test01.cpp
Deck_test02.cpp
Deck_test03.cpp
Player_test.overview
Player_test01.cpp
Player_test02.cpp
```

Test cases for this project are considered "acceptance tests". Your tests for your deck ADT (each of which includes at least a main() function) will be linked against a correct Card.cpp and a possibly incorrect Deck.cpp when they are compiled. Your tests for your player ADT (each of which includes at least a main() function) will be linked against a correct card.cpp, a correct hand.cpp, and a possibly incorrect player.cpp when they are compiled. All files must compile from within the same directory.

Your test cases must decide, based on the results from calls to deck/player methods, whether the deck/counting player ADT is correct or incorrect. If your case believes the deck/player to be correct, it should return 0 from main(). If your case believes the deck/player to be incorrect, it should return any value other than zero (the value -1 is commonly used to denote failure). assert() will return a non-zero value, so this is another acceptable way to test. We do not compare the output of your test cases against correct/incorrect implementations. Instead, we look at the return value of your program when it is run in Linux to see if you return the right value based upon whether your test finds an error in the implementation of the ADT that is being tested. However, you may find it helpful to add error messages to your output and you are free to make use of them as much as you like. Here is an example of code that tests a hypothetical "integer add" function (declared in addInts.h) with an "expected" test case:

```
// Tests the addInts function
```

```
#include "addInts.h"
int main()
{
   int x = 3;
   int y = 4;
   int answer = 7;

   int candidate = addInts(x, y);
   assert( candidate == answer );
}
```

We will write a collection of decks/players with different, specific bugs. You will be evaluated based on how many of our buggy versions your test cases identify. If your test case misidentifies a correct library as buggy, it will not be used to evaluate buggy libraries. So, for example, writing a test case that returns non-zero in all cases will generate no credit.

The number of test cases you write, how they are ordered, and whether or not they are split into different files is entirely up to you. We will run all of the submitted test cases against a buggy implementation (which has one bug). If any of your test cases return non-zero, you will get the points for that bug. We repeat this for several buggy implementations. So, to us, it doesn't matter how many test cases you have, how they're ordered, or how many files they're in.

There will be no "surprise bugs". All of the bugs that we expect you to find are normal bugs that would occur if someone just misread the spec (or made a common coding mistake like off-by-one). We didn't create implementations that will erase the hard-drive if you try to shuffle by 42.

Fair Warning:
While we don't care how many test files you split your test cases into, it is to your advantage to split them into more rather than less files. If you have a single file with all of your tests, and that file catches all of the bugs, you will get just as many points as someone who caught all of the bugs with 100 test files. *Here's the difference:* any test file with a test case that incorrectly says a good implementation is buggy is thrown out. So if you have only one test file with 100 test cases inside it, and ONE of the tests returns non-zero for a good implementation, the whole file is thrown out, and you will get no points for testing. Thus, you should split each test case into its own file, so that only the bad test files will be thrown out, and any bugs caught by your remaining tests can still get points. Finally, note that if a test file identifies a good implementation as buggy, *not only* will your test file not be used to evaluate buggy libraries, but the test file will cause you to *lose points*. This is done to discourage you from submitting a bunch of random tests and hoping that the some of them will get "lucky" by finding bugs.

Building the project

This project has several components: when complete, there are six source files. To build the program, type the following all on one line into the terminal in Linux (making sure you've copied all header files):

```
g++ -pedantic -Wall -Werror –O1 -o blackjack blackjack.cpp Deck.cpp Hand.cpp Player.cpp rand.cpp Card.cpp
```

If you want to see only if a single file compiles properly, you can compile it individually using the "-c" flag. For example, to check your deck:

```
g++ –pedantic –Wall –Werror –O1 –c Deck.cpp
```

Note the –O1 compiler flag in the compilation strings (that's a capital letter 'O', not the number zero, followed by the number one). This flag turns on a small amount of compiler optimization. Not only will it make your code a bit faster, but it will also give you warnings if you try to use a variable uninitialized. **However**, this flag will make it difficult to debug certain parts of your program when using a debugger. So, if you are fixing your program using a debugger (i.e. by using the –g flag), you must omit the –O1 flag when compiling your code and only include it once you think that you have fixed all the errors in your program. In other words, you should be using either the –O1 flag or the –g flag for debugging, **but not both**.
Programs that do not compile properly will be severely penalized in this project, so be sure to check them in the CAEN environment.

We have supplied one simple set of output produced by a correct deck, hand, simple player, and driver. It is called blackjack_test00.out.correct. To test your entire project, type the following into the Linux terminal once your program has been compiled:

```
./blackjack 100 3 simple > blackjack.out
diff blackjack.out blackjack.out.correct
```

If the `diff` program reports any differences at all, you have a bug.

Handing in and grading

Use the submit280 program to submit the following files for project 4:

`Card.cpp`	your Card implementation
`Deck.cpp`	your Deck implementation
`Player.cpp`	your three Players
`blackjack.cpp`	your simulation driver
`Hand.cpp`	your Hand implementation
`Deck_test.overview`	overview of your deck test cases
`Deck_test01.cpp`	first deck test case
`Deck_test02.cpp`	second deck test case
`...`	
`Deck_testNN.cpp`	NNth deck test case
`Player_test.overview`	overview of your player test cases
`Player_test01.cpp`	first player test case
`Player_test02.cpp`	second player test case
`...`	
`Player_testNN.cpp`	NNth player test case

Submit all your files with a single submit280 command. Do not try to submit them individually. For example:

```
submit280 3 Card.cpp Pack.cpp Player.cpp Game.cpp euchre.cpp
```

There are three components to your grade in this project:

- Correctness of deck, player, driver, and hand
- Testing of deck and player
- The style of your deck, player, driver, and hand implementations

Collaboration

Please note that the testcase portion of this project is included under the usual course policy against sharing any code, as described in the syllabus. Your project, including the testcases, must be entirely your own work.

Appendix A: Overloading the Output Operator

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) {} // default 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
```