Laboratory 5: Wednesday, January 18

Given a shuffled deck of fifty-two playing cards, I propose to deal them one at a time, counting the values “2, 3, 4, …, King, Ace” as I do. To get through the whole deck, I repeat this count four times. Would you bet me that I will get through the deck without the rank I call out turning up as I call it? What if I give you two dollars every time we make it through the deck, and you give me one if we don’t? I think we better let the computer figure the odds first.

lab05a.cpp: modeling a deck of cards

We will build objects to represent suits, ranks, cards, and decks of cards.

suittype and ranktype: These two are similar. Represent them each as an enum, a type which takes symbolic names as its values. For instance,

```cpp
enum suittype { Diamonds, Hearts, Spades, Clubs };
```

Build ranktype similarly, with values like R3 for a 3, and Ten, Jack, Queen, King and Ace at the high end. Then create overloaded operators so we can print and pre-increment objects of these types. Since enums are not classes, we cannot use member functions and do not need friend functions. The pre-increment operator has the signature

```cpp
const suittype & operator ++ ( suittype & s )
```

(Post-increment would be the same, but with a “dummy” second parameter of type int, which is never used but simply differentiates pre-increment from post-increment.) As you write this function, you will make use of the fact that C++ will automatically convert from an enum to an int as necessary (e.g. to do addition). It does not, however, automatically convert back from int to enum; you must do an explicit conversion, using a syntax like

```cpp
s = suittype ( s + 1 )
```

Make the incrementation circular, so incrementing from Clubs, for instance, takes you back to Diamonds.

Inside the stream insertion operator use a local array like

```cpp
const static char PrintRep[] = { 'D', 'H', 'S', 'C' };
```

The static means that, even though this is a local variable, it is not re-created every time the function is invoked; there is just one instance created at compile time, and its lifetime is the lifetime of the program. Index this array with the suittype you are trying to print, and the corresponding char will be printed. Design the stream inserters so any card can be represented by a two character sequence; e.g. 7H. Use T, J, Q, K, A for the largest ranks.

cardtype: Design this as a class, with a suittype and a ranktype member variable. Create a constructor to initialize it given any ranktype and suittype values. Create a pre-increment operator that will first increment the rank, and when it reaches Ace, also increments the suit. Make an ostream inserter that will print the name of the card as a two-character sequence. Define operator ==, and include <utility> to get operator != for free. (You won't get, e.g., operator >= though, unless you define operator <. I didn't need to do that; maybe you won't either.) Note that you do not need to define relational operators for suittype or ranktype; this is because C++ does automatic conversions to int from those types.

decktype: It will be useful to think of decktype as a container class, supporting random access iterators. So you could create such a class from scratch, defining lots of overloaded operators and defining an iterator with lots more overloaded operators. We will take the simpler—and much more sensible—approach: derive decktype from vector<cardtype>! This allows you to inherit everything that vector can do for free. The signature syntax is

```cpp
class decktype: public vector<cardtype>
```

Add a constructor with an int parameter to tell how many cards should be in the deck (normally 52, of course). The constructor should fill the deck, using push_back() with that many cards, in increasing order from 2D to AC. Note that the syntax here is simply push_back(card); not xxx.push_back(card). That's because all of the methods of vector are also methods of decktype, by inheritance.

main: To test all this, instantiate decktype and dump it to an ostream_iterator bound to cout and separating with ' ' using the copy algorithm. You should get a long list of 52 cards, starting 2D 3D 4D ...

lab05b.cpp: shuffling and better printing

To clean up the printout, move the copy algorithm call, along with the declaration of ostream_iterator into a
decktype method ostream & print ( ostream & os = cout ) const. The = cout part makes cout a default parameter, so if you call the print() method with no argument you get cout. Put the copy call in a for loop controlled by a const_iterator (you don’t need to qualify it as decktype::const_iterator, because print is a decktype member function) which increments by 13 each time through, and prints 13 cards per iteration. A const_iterator cannot be dereferenced as an l-value, and you need one here to reference const data. Test print: you’ll get 4 lines of output, each with 13 cards. But they are still in new-deck order.

Add a shuffle method using the random_shuffle algorithm to decktype, and invoke it in main before the printout. Do two sample runs. Notice any problems?

Yes, the “random” output is always the same! This is because “random” really means pseudo-random. It depends upon a seed value: started from the same seed, it will always produce the same sequence of output. To take control of the seed, we need to build our own random_shuffle generator. Include header files <ctime>, to get time-of-day readings which can be used to set the seed, and <cstdlib> to get the standard random number generator long rand(void). The latter returns a long between 0 and RAND_MAX inclusive. This header also defines a routine void srand(long) to set the seed.

Create a class randgen with an operator() that returns an integer less than its integer parameter by multiplying by the parameter by a fraction between 0 and 1. The cast to double guards against integer overflow in case RAND_MAX is the same as INT_MAX (defined in <climits>), and also does the whole computation in double arithmetic to avoid integer truncation. The constructor calls srand with the current time of day as an argument.

Create an instantiation of this generator named getrandom as a static member variable of decktype, and use it as the third parameter of random_shuffle. We use a static instantiation, because we may wish to shuffle the cards multiple times, possibly for different instantiations of decktype, but we only want to seed the random number generator once, at the beginning of the program. But a non-const static member variable has a strange-seeming technical requirement: the declaration inside the class declaration does not get instantiated when the class as a whole gets instantiated (because there is only supposed to be one instantiation of a static member, regardless of how many class instances). So add an external definition:

    randgen decktype::getrandom;

Test your program again a couple of times. Print a listing, and a sample run.

lab05c.cpp: testing for cards in rank position

 Instantiate a second deck in main, which will be kept in its original order. Define a function object rankequal that returns true if the ranks of two cards are equal, and use it with the adapter not2 and the mismatch STL algorithm to see if any card in your shuffled deck is in its original rank position. Remember that you operator() in the function object must be const to be used with an adapter. Set up a for loop to shuffle and print deck, check for cards in position, and print the result (either “No Match” or “Match at position xx”) repeatedly say, five times. Visually examine the output for errors.

Then take out the printing, and just keep a counter of how often a match is found. Run it a few thousand times, and print the percentage that made it all the way through the deck. Print a listing, and a sample run.

lab05d.cpp: testing for cards in their original position

While we’re at it, let’s see how the results change if we insist that a card be in its original position—both rank and suit—before we call it a match. Use not_equal_to<cardtype> as the function object in mismatch (you did define == for cardtype, didn’t you?), and run it again. Print a listing and sample run.

(This version of the “game” is isomorphic to a more playable one using two shuffled decks: two players turn up cards one by one—how likely is it that at some turn they will turn up identically the same card?)