lab07a.cpp: using the exception mechanism

If you turned over all the cards in a shuffled deck one by one, counting each card until a spade turned up, on the average what would be the longest such sequence you would get for each run through the deck? (Make a guess; see if you’re right…)

Begin with your code from lab06b.cpp, and replace main with a version that is structured as follows:

```cpp
// Declarations and initializations
...
deck.shuffle();
while ( true ) // deal the entire deck
{
    try
    {
        nonspades = 0;
        upcard.take ( deck ); // throws exception when deck empty
        while ( upcard.front().suit() != Spades )
        {
            ++nonspades;
            discards.take ( upcard );
            upcard.take ( deck );
        }
        discards.take ( upcard );
        if ( nonspades > largestcount ) largestcount = nonspades;
    }
    catch ( out_of_range & ex )
    {
        break; // exit while loop when deck is empty
    }
}
if ( nonspades > largestcount ) largestcount = nonspades;
deck.takeall ( discards ); // restore deck
assert ( deck.size() == 52 );
cout << "The largest run of non-spades had length " << largestcount;
```

The try…catch code uses the exception you built into decktype::take to warn you when the deck is empty. This may not be the best way to control looping for this problem (you could just check deck.size()), but it is a least a good demonstration of exception handling!

When you have this working, put it in a loop and average the results of a few thousand times through the deck to get a reliable average. Print a listing, and a sample run.

lab07b.cpp: using poker objects

In a game with seven players, how good would your poker hand have to be to give you, on the average, a fifty percent chance of being the winner?

To answer this question, first define a typedef vector<handtype> table and an object table hands(7) which represents all of the players in the game. Set up for loop to play a set number (start small, for now…) of games. For each game, shuffle and deal cards, one at a time, to each of the players

```cpp
for ( table::iterator p = hands.begin(); p != hands.end(); ++p ) ...
```
Use `max_element` to find the winning hand. You may need to add some code to the various card objects to make this happen; try to anticipate what you will need as you write the `max_element` call.

Create a `vector<eval>` which will hold the evaluations of winning hands from all the games. Create it with an initial size of zero, so you can `push_back()` elements to the end, but immediately after the instantiation use the `reserve()` member function to pre-allocate the space for all the trials you will do (which you should define as a constant `unsigned long` at the beginning of `main`). This avoids the inefficiency and inconvenience (because iterators can become invalidated) of periodically allocating storage as the vector grows. Add code to record the best evaluation for each game to the end of this vector. Use the `nth_element` algorithm to find the median (the element that would be in the middle, if the list were sorted) and print it.

Test it until you are sure it works right, then crank the number of trials up to a few thousand. Print a listing, and a sample run.

**lab07c.cpp: summarizing sorted data**

An ASCII data file `lab07c.dat` consists of lines of data, each having an alphanumeric part number and an integer quantity. The file is sorted lexicographically by part number, but there may be more than one line for any given part number. Write a program that will check that the data is indeed sorted and then summarize the information with a total and a list of quantities for each part number. For instance,

<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>GA345 21</td>
<td>Part Number GA345: total 46</td>
</tr>
<tr>
<td>GA345 7</td>
<td>21 7 18</td>
</tr>
<tr>
<td>GA345 18</td>
<td></td>
</tr>
<tr>
<td>GA411 211</td>
<td>Part Number GA411: total 370</td>
</tr>
<tr>
<td>GA411 159</td>
<td>211 159</td>
</tr>
<tr>
<td>GC2 9</td>
<td></td>
</tr>
<tr>
<td>GC2 33</td>
<td>Part Number GC2: total 42</td>
</tr>
<tr>
<td></td>
<td>9 33</td>
</tr>
</tbody>
</table>

Design, name and use an object type to hold an input record, and read the input into a list using an `istream_iterator`. Write a function object that will check for records where the partnumber of one record is greater than the partnumber of another, and check the order of records in your list using `adjacent_find`. (Note: I was not able to get g++ 3.3 to do this correctly using the adapter `greater<invrec>`; I had to actually write the function object.) Then loop through the list using `equal_range` to find each sublist for a single part number. Use `accumulate` (header file `<numeric>`) to total the sublist, and an `ostream_iterator` to print the list of quantities.

This exercise requires overloaded operators of the type you have worked with in previous labs. To use `accumulate`, you will need to define an automatic type conversion from your record type to `int` (which will also come in handy in connection with the `ostream_iterator`).