

# CS 3850—Object-Oriented Programming

California State University, Stanislaus

R. L. Zarling Winter, 2006

## Laboratory 6: Thursday-Monday, January 19-23

### lab06a.cpp: dealing poker hands

Poker is a popular card game which comes in many variations. In this lab we will implement C++ code to simulate some basic operations central to five-card poker. As the name implies, each player is dealt five cards, and the winner is the player whose hand ranks highest in the poker ranking system. This system first classifies the hand into one of nine levels, listed below from highest to lowest:

| Name            | Description   | Example        |
|-----------------|---|----------------|
| Straight Flush  | All cards of same suit & in consecutive ranks; Ace may be highest or lowest | 8H 9H TH JH QH |
| Four of a Kind  | Four of the cards are of equal rank   | 2D 7C 7S 7H 7D |
| Full House      | Three cards are of one equal rank, and two more of another equal rank       | KH KS 7C 7H 7D |
| Flush           | All five cards are of the same suit   | 3D 5D 9D TD KD |
| Straight        | The five cards are in consecutive ranks; Ace may be highest or lowest       | AH 2S 3S 4C 5H |
| Three of a Kind | Three cards are of equal rank   | 5D AD JS JC JD |
| Two Pair        | The hand contains two sets of two cards of equal rank                       | QH 3C 3S 8C 8H |
| Pair            | Two of the cards are of equal rank  | 4C 5S AS TC TH |
| High Card       | None of the above apply   | 2H 4H 9S JH QH |

If two players' hands classify at the same level, the ranks of the cards come into play: first, the most frequently repeated ranks (e.g. the three in a Full House), followed by ones that are repeated less frequently. So 3H 3S 9C 9S 9H beats JH JS 8H 8D 8C, and 4C 8C 9S JS JH beats 3C 8S 9H JC JD.

Build a model for a poker game based on lab05c.cpp. Revised *decktype* to be a *list* instead of a *vector*, to make it easier to access the top and bottom. This is almost as simple as replacing “vector” with “list,” but to shuffle you will have to copy the deck to a vector and back, and you will have to do something about `print()`. Declare a new class `handtype: public decktype`, so a “hand” is structurally identical to a “deck.” The main difference is that hands start out empty instead of size 52, so the *handtype* constructor should be a default constructor (i.e. parameterless). There is a small problem here: since *handtype* is a specialization of *decktype*, the *decktype* constructor is automatically called before the *handtype* constructor begins executing. Where does the *decktype* size value come from? You must code the *decktype* constructor call explicitly in the initializer list for the *handtype* constructor, complete with parameter value:

```
handtype ( void ): decktype(0) { }
```

A fundamental operation in many card games is the ability to move cards from one deck or hand to another. Examples of this would include “dealing” a card from the deck to a player’s hand, or collecting all of the cards from a player’s hand back into the deck when the game is over. The polymorphism we have built into the deck/hand relationship works well for us here. Define an operation `void take ( decktype &d, int howmany = 1 )` into the *decktype* class (so it applies both to *decktype* and *handtype*, by inheritance). This operation removes the first *howmany* cards from the front of *d* and pushes them to the back of the object doing the taking. Define a separate *decktype* operation *takeall* for convenience, to empty *d* into the object doing the taking. This is a good time to include `<stdexcept>` and check before taking each card into *take*:

```
if ( d.empty() ) throw ( out_of_range ( "Deck is empty" ) );
```

Define an operation `void arrange ( void )` to sort the cards in a hand by rank (and within equal ranks, by suit). You will need to define operator `<` for *cardtype*, of course. This operation will usually be applied to *handtype* objects, but it is just barely possible we may someday want to arrange an entire *decktype* back to its original order, so make *arrange* a *decktype* operator. Use the *list* member function, *sort()*.

Throw away the old *main* from lab05c.cpp, and write one that instantiates a deck and “deals” two hands of five cards each. Print the deck before the deal; print each of the two hands both unsorted and sorted; print the deck after the deal; collect all of the cards from the hands back into the deck and print the deck again.

Print a listing and a sample run.

## lab06b.cpp: evaluating poker hands

| Hand           | eval::classif | eval::ranklist |
|----------------|---------------|----------------|
| 7H 7S 7C TS 3D | eval::Trips   | R7 Ten R3      |
| 3H 3D KS KC 7D | eval::Pair2   | King R3 R7     |

Evaluation of a poker hand involves both classification and ordering by the ranks of cards, so we need a class `eval` to represent evaluations. Inside this class define a public enum `plevel` {`Nix`, `Pair`, `Pair2`, `Trips`, `Straight`, `Flush`, `Full`, `Fours`, `SFlush`}, and private variables representing a classification and a list of *ranktypes* (most significant first). Create three constructors: a default constructor and ones with a *classification level* and optionally a *ranktype* variable. Define operator `<`, operator `==`, and operator `<<` for *eval*. Note that *list* implements lexicographic operator `<` and `==` which you can make use of. Define void operator `() ( const ranktype r )` to add another rank to the end of the ranklist. This makes *eval* a function object; derive it from `unary_function`. Compile and debug.

Finally, add a method `eval evaluate ( void )` to *handtype*. It will need access to the *ranktype* and *suittype* fields of *cardtype*, so add access functions `ranktype rank(void) const` and `suittype suit(void) const` to *cardtype*; this is safer and clearer than making *evaluate* a friend of *cardtype*. For safety, begin the code of *evaluate* by checking that `size()` is not zero; if it is, return `eval ( eval::Nix, R2 )`, or some other impossible evaluation. That way, we can be sure when we try to look at a card, there actually is at least one there. This is an example of *defensive programming*. It will be useful to use random access iterators when evaluating the hand, so create a local `vector<cardtype> h(begin(), end())`, a copy of the current hand, and `sort` it; you will need to use `::sort`, the STL algorithm, because if you try to use `sort`, you will get the list member function. Create local `bool` variables `isstraight` and `isflush` to check for straights and flushes. Check for flushes by using the `find_if` algorithm to see if any cards have a suit different than the first one, using a function object `suitequal` with `not1(bind2nd...`. Check for straights using `adjacent_find` with a function object `adjacentrank` which tells if two *cardtype* ranks are adjacent (first argument one less than the second). Since ace can be low in a straight, you have to check if `h.front().rank()` is 2 and `h.back().rank()` is Ace; if it is, run `adjacent_find` only on the first four cards of `h`. In *evaluate*, if either `isstraight` or `isflush` or both is true, return a properly set *eval* object, complete with a list of ranks (five for a flush; only the highest one is enough for straight or straight flush). For flushes, it would be nice to use `for_each` to scan `h.rbegin()` to `h.rend()` (why a reverse iterator?), applying `eval::operator()` for each card. But the `base` type of `h` is *cardtype*, and `eval::operator()` requires a *ranktype*. So go back to the declaration of *cardtype*, and add a member function `operator ranktype() { return _r; }`. This is a type cast operator; it allows the compiler to silently cast from *cardtype* to *ranktype* any time the context requires it. These kinds of automatic casts should be used sparingly, because each one is a nail in the coffin of strong typing. (To make it only be applied if you write an explicit cast, you would add the keyword `explicit` at the beginning of the above declaration—we need the implicit cast for this lab, however). Be sure to instantiate an *eval* object with the correct *plevel* (for a flush, `eval::Flush`) as the third *for\_each* argument, and return the copy of it you get as the *for\_each* return value.

If the hand is not a straight flush, we must look for repeated ranks. Instantiate a “repeat-rank” list

```
list< pair<int, ranktype> > rrlist;
```

which will hold information about the repetition counts of all the various ranks in the hand. The `int` (repetition) field comes first in the pair, so in a sort, repeat-ranks with highest repetition counts will sort to the end of the list. Using a for loop ranging over `h`, our sorted-by-rank hand, create a new repeat-rank each time the rank changes, and increment the repetition count if the rank is the same as the previous card. Accumulate all such repeat-ranks in `rrlist`, and when you are done, sort it using the list member function `sort()`.

With this done, the rest of the evaluation is pretty easy. The last entry in the list `rrlist.back()`, describes the most-repeated rank, so if `rrlist.back().first` is four, for instance, the hand is four-of-a-kind. Use a C++ *switch* statement based on this to set a local variable to the appropriate `eval::plevel` classification. Some of the *switch* cases will have to further differentiate between classifications; for instance, the code for three will have to check if the hand is a full house or only three-of-a-kind, and the code for one will have to check for straight or flush. Create an *eval* structure, and add to its rank list all of the ranks in your repeat-rank list, in reverse order. I think a simple *for* loop is best for this; *for\_each* would require a new function object.

Add code to *main* to evaluate hands and print and compare evaluations, and do extensive testing. Print a listing and some sample runs.