

# NET2006: Object-Oriented Programming

## Lab 6: *Card Wars*

### Overview

As you implement a game of *Card Wars*, you will build on your understanding of classes and of arrays. To simplify your work, develop the solution in several phases:

- Create and test the **Card** class component independently.
- Create and test the **Stack** class component. The **Stack** class will be a stack of **Card** objects, so there are some minor modifications from the **Stack** class in Lab 5<sup>1</sup>. This will be tested before trying to implement your *Card Wars* algorithm.
- When you are confident that you have a working **Card** class and a working **Stack** class, you are ready to implement the algorithm.

### The Details

#### Card Wars: Game Rules

- Uses a traditional 52 card deck.
- Aces are low.
- Shuffled cards are evenly distributed between players (26 each for two players).
- Each player presents a card from the top of their stack of cards.
- The player with the higher face value card wins both cards and places them in a winnings stack.
- If the two presented cards have identical face values, then neither player wins; it is deemed a tie. The tied cards are placed in a temporary stack. Whomever wins the next round, captures all the tied cards in addition to the pair from the current round.
- Game play concludes when either player exhausts his or her stack of cards.<sup>2</sup>

#### Card Wars: Strategy for Development

Here are some suggestions to achieve the successfully working program. In general, the steps involve building the required individual components; testing these components to see if they work; assembling components into more complex groupings; testing these, and so on.

The following list shows a sequence of development that I would probably follow. You don't need to do the work in this order, but you may find it useful.

- Create and test a *Card* class.

<sup>1</sup> The **Stack** class in Lab 5 was a stack of *int* values.

<sup>2</sup> With the rules of this card game, you are guaranteed that both players will exhaust their cards at the same time. In many other card games, players may exhaust their cards at different rates. Think about controlling the main driving loop for game play, by watching for either player's stack to be empty (rather than watching for 26 iterations of the loop, which is how this particular card game will proceed).

- Create a simple **Card** class with the two primary data members: **nFaceValue** and **sSuit**<sup>3</sup>.
- Create two essential functions:
  - **Set()**<sup>4</sup>: Accepts two arguments to give each **Card** object its initial values: face value and suit.
  - **Display()**: Outputs the values stored in the data members.
- Write a simple function **main()** which tests the basic capabilities of the **Card** class.
  - Can you create multiple individual **Card** objects and use the **Set()** and **Display()** functions?
- Create a simple **Stack** class to allow the storage of **Card** objects.
  - To build the **Stack** class, start with the working **Stack** class from Lab 5. It created a stack of *int* values. Modify the code to work with your **Card** data type instead of the *int* data type.
  - Migrating from a stack of *int* values to a stack of **Card** objects can be as simple as changing the data type that's being stored (that is, changing *int* to **Card**). Of course, there are several spots in the **Stack** class where this change must be made.
- Write a simple function **main()** which tests the basic capabilities of the new **Stack** class.
  - Create several **Card** objects in **main()**.
  - Call **Set()** and **Display()** to prove that those **Card** objects exist and have valid values.
  - Create a **Stack** object.
  - **Push()** the **Card** objects onto the **Stack**?<sup>5</sup>
  - **Pop()** **Card** objects from the **Stack**. You should try capturing the values in the popped **Card** object to a temporary **Card** variable (object) created in **main()**.
  - Call **Display()** to see if the popped **Card** objects behaved as expected.

At this point, you have built enough of the components to consider the issue of game play with *Card Wars*.

#### Card Wars: The Game Algorithm: Overview

*Card Wars* involves two computer-based players. Each receives 26 cards (half the deck). During each round of play each player reveals the top-most card from his or her stack of cards. The player with the larger card

<sup>3</sup> You can use either an *int* to manage the suit or an *enum* type; it's your choice.

<sup>4</sup> Classes also commonly implement the **Get()** function, but we don't really need it here.

<sup>5</sup> Use the debugger to prove that the **Push()** operation is in fact working.

(defined by its face value only — not its suit) wins both cards and places these cards in his or her pile of winnings.

During a round of play, if each player presents cards with identical face values, then the battle is considered a tie. The two tied cards are placed in a separate pile of tied cards which will be won the next time a player wins a battle.

The game ends when either player exhausts his or her set of cards. In this simplified card game, both players will exhaust their cards simultaneously.

### Game Algorithm: Create Stacks

Create 6 stacks of cards for the following purposes.

- Full Deck of original 52 cards.
- Player 1's initial hand of cards.
- Player 2's initial hand of cards.
- Player 1's winnings as game play proceeds.
- Player 2's winnings as game play proceeds.
- A stack of tied cards.

### Game Algorithm: Create Full Deck

You have 52 cards to create and push onto your full deck stack. That suggests looping. In the looping process, you will create a temporary **Card** object, and give that **Card** object its initial value. The initial value of the first object could be Ace of Spades. The value of subsequent **Card** objects will increase sequentially. How you initialize it, depends on how you plan to store the values in your object<sup>6</sup>. That new temporary **Card** object will be pushed onto the full deck.

You can implement the looping process as one large loop that cycles 52 times, or as two nested loops. With two nested loops, the outer loop will cycle through the 4 suits; the inner loop will cycle 13 times creating 13 **Card** objects (Ace through King) of the current suit. (The outer loop would then move to the next suit.) In total, the nested looping approach will also create 52 cards (4 x 13).

The full deck is much like a deck of cards that you purchase from a store. It is organized by suits and is in sorted order. Clearly, you need to shuffle the deck. Your textbook shows how to accomplish this operation in about 5 lines of code on page 288. I would suggest adding this shuffling capability as a member function of the **Stack** class.

**Testing:** You might want to set a breakpoint on the call to **Shuffle()** the stack of cards in the full deck. You can inspect the cards before shuffling and verify that they are in the sequential order established during the creation process. After allowing **Shuffle()** to execute, you should see that the card sequence has been randomized.

<sup>6</sup> For example, your **Card** object might store the face value as an **int** ranging from 1 to 13, representing the *Ace* through *King*. The suit might also be stored as an **int** ranging from 0 to 3, representing: *heart*, *diamond*, *club*, *spade*. You could also choose to use an **enum** type for suit instead of **int**, the choice is yours.

### Game Algorithm: Deal the Cards

You will repeatedly take a card from the full deck and give it to Player 1; take another card from the full deck and give it to Player 2.

This means you **Pop()** a card off the full deck, and **Push()** it onto player 1's initial hand of cards. Then, **Pop()** a card off the full deck, and **Push()** it onto player 2's initial hand of cards.

This process continues in a loop while the full deck is not yet empty.

**Testing:** You might want to set a breakpoint just before the dealing loop. View the internals of the full deck, Player 1's hand, and Player 2's hand. In the beginning, the **nTop** variable in the full deck will have a value of 51 — indicating that there are 52 cards. The stacks for the two players' hands will both start out with **nTop** having -1 — indicating that the two hands start out empty.

As loop execution continues, you should see **nTop** decrease in the full deck, and increase in the two players' hands.

You should also see card values appearing in the array of **Card** objects in each of the two players' hands.

### Game Algorithm: Battle the Cards

Another loop will continue game play. On each iteration of the loop, two **Card** objects will be presented: using **Pop()** you'll get one card from player 1's initial hand; using **Pop()** again, you'll get a card from player 2's initial hand. The face value of the two cards is compared.

In the event of a tie, use **Push()** to place the two tied cards onto the stack of tied cards. Execution proceeds back to the top of the loop.

In the case where one player has a winning card (which is, in fact, the more probable case), that player receives both **Card** objects. This is done by using **Push()** to place both **Card** objects onto that player's stack of winnings. If there are any cards already on in the tied stack, the winning player will receive those as well. This is accomplished by using **Pop()** to retrieve a card from the tied stack, and **Push()** to move the card to the players' stack of winnings. Since there could be many tied cards in the tied stack, the **Pop()-Push()** process will be placed in a loop. This transfer loop (that is transferring **Card** objects from the tied stack to a winnings stack) will continue while the tied stack is not empty.

This battle loop continues while both player 1 and player 2's original hand is not empty.

### Game Algorithm: Determining the Winner

Whomever has more **Card** objects in their stack of winnings, wins the game. I would suggest creating the following function in your **Stack** class:

```
int GetNumCards() { return nTop + 1; }
```

In the case of an empty stack, **nTop** has the value -1, so this function would return 0. If a player won all the cards, then **nTop** would be 51, and this function would return 52.

You can use this function to find out the score for each player.

## Test Plan

You will include a test plan outlining your approach to testing. Good program design suggests that you craft your test plan **before** you begin your coding. A test plan helps you organize the sequence of steps you'll take when developing your program solution.

The test plan is probably most useful when organized as a chart. It might have the following headings:

Code	Pre-Conditions	Test Method	Expected Result	Actual Result

Notice that the first four columns can be completed **before** you begin any coding.

## Submission

- Cover page:
  - Course Name / Number (including Lab Section)
  - Lab Number
  - You and your partner's names
  - Date of Submission
- Problem Statement (Assumptions)
- Test Plans
- Algorithms
- Memory Maps using Visio
- C++ source code

## Submission Process

- Staple all paper components together with the evaluation sheet.

## NET2006: Object-Oriented Programming Lab 5: Card Wars

Your final project has been evaluated using the following criteria: This lab is to be completed with your partner. You will receive the same mark (unless there is evidence that one of the partners did not make a real contribution). Look to Lab 1b for coding standards (sample code, variable naming conventions, etc.)

- Statement of Assumptions
- Test Plan
- Algorithm
- Memory Map (Use **Visio**)
- Program and function headers (remember, *main()* is a function), including history.
- Comment each significant line of code. Comment each related section of your program.
- Variable naming as defined in handout.
- Indentation follows standard in handout.
- (2 marks) Program performs runs correctly from source code (Including meaningful user prompts).

**Subtotal /10**

- Penalties (up to 3 marks deducted): Penalty for redundant code, unused code, unnecessary extra steps. Penalty for poor organization and binding of program and documentation. (Individual Adjustment: Minimal or no participation. This will adjust the grade of an individual within the group. The deduction could up to the full marks available for the lab.)
- Bonus marks (up to 3 marks added): Clever, clear, useful enhancements. (Individual Adjustment: Clear evidence of supportively explaining C++ concepts to partner.)

**Total /10**