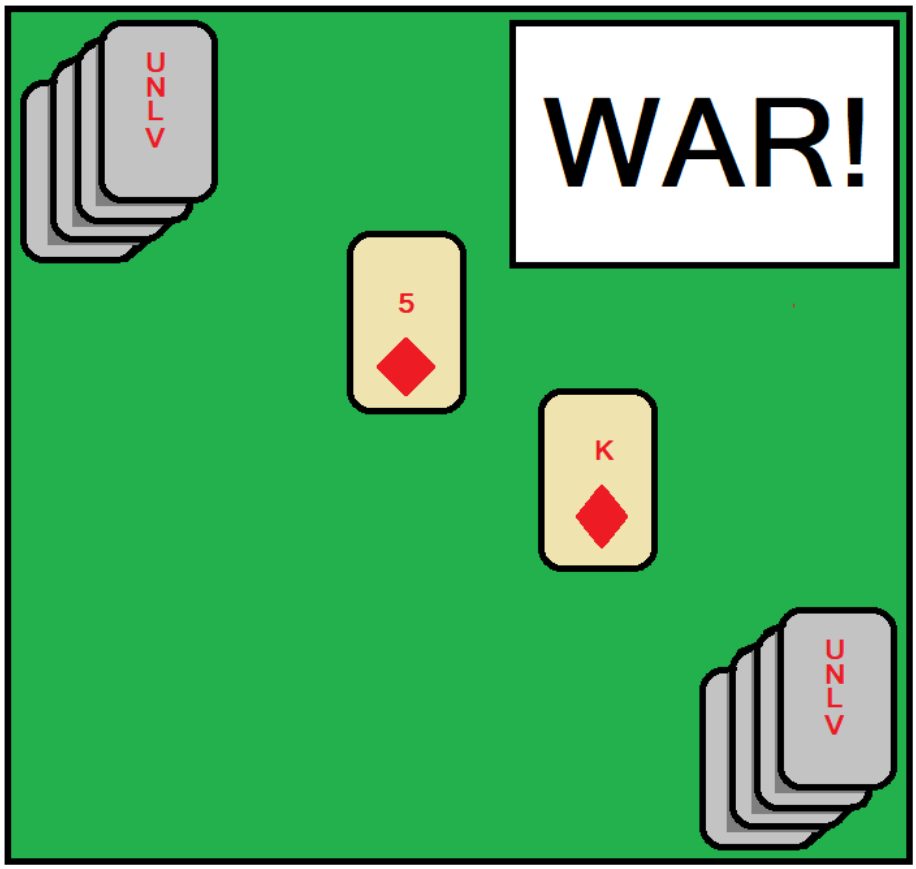
CS 202 - Assignment 9 - Linked Lists

War Assignment - UNLV

April 2021



1. This Assignment will compile as skeleton code.
2. There will be a YouTube Video availible to comprehend this assignment at James Piotrowski's YouTube channel.

   https://www.youtube.com/channel/UCa7-7o4wiZi5-m2FoEbm6UQ

1 Introduction

Today we are coding up the card game **War**. The rules are simple.

1. At the start of the game, 2 players are dealt an equal amount of cards. In each hand, players will have the same amount of each weight of cards. This means each player will have 2 Aces, 2 Kings, 2 Queens, etc. The hands are shuffled before step 2.
2. Each player turns one card over and puts it in a pile:
   (a) If Player 1's card has a higher weight, then Player 1 gets to claim the pile and put the cards into a "Foot" (as opposed to the player's "Hand"). The foot is a pile of cards belonging to a player that are not in the players hand.
   (b) If Player 2 has a higher card, then Player 2 gets to claim the pile for their foot.
   (c) If each card is the same weight (i.e. 9 vs. 9), then the players will each add 3 cards to the pile and restart step 2. This is called a **Duel**.
3. If at any point a player has no more cards in thier hand, they can shuffle their foot and add those cards to thier hand.

The game of war can potentially go on forever if players keep alternating wins. In this program, the game will end in one of 3 ways:

1. One player runs completely out of cards by losing all of the cards to the other player.
2. One player deals all their cards into the pile during a duel. In this case, the pile will not be claimed.
3. The players will get bored after 1000 turns.

Sample outputs are included for you. Please reference the different cases to check and see if the game is working.

Decks are represented in the C++ code and in the computer as linked lists. Your job is to code any portion of this code interacting with the linked lists.

2 The Files and Their Classes

There are 6 Files:

1. Card.h & Card.cpp (Your Code Here)
2. Player.h & Player.cpp
3. Main.cpp
4. makefile

2.1 Card.h and Card.cpp

The Card.h contains both the Card and Deck classes. We will start with the Card Class.

MemberAccess	Type of Member	const	static	type	Name	Parameters
private	variable	No	No	char	Suit	N/A
private	variable	No	No	char	Value	N/A
private	variable	No	No	short	Weight	N/A
public	Constructor	N/A	N/A	N/A	Card()	None
public	Constructor	N/A	N/A	N/A	Card()	char, char
public	Operator	Yes	No	bool	operator<()	Card
public	Operator	Yes	No	bool	operator>()	Card
public	Function	Yes	No	string	GetCard()	None
public	Function	Yes	No	int	GetWeight()	None

1. The Default Card constructor will initialize both the Suit and Value of the card to be the NULL character '\0'. The weight will be initialized to any negative value (-999 perhaps).
2. The Parameter constructor will assign the suit parameter to Suit and the value parameter to Value. The weight will need to be initialized as well. The weight is needed to determine which card has a higher weight. See the following weights below:
   (a) A = 14
   (b) K = 13
   (c) Q = 12
   (d) J = 11
   (e) T = 10
   (f) 2-9 = 2-9
3. string GetCard() will return a string representing the card. The string looks like : " |[Suit Value]| ". Examples:
   (a) |[SA]| - Ace of Spades
   (b) |[CT]| - Ten of Clubs
   (c) |[H4]| - 4 of Hearts

4. Operator Overload < is used in sorting the cards at the end of the game. The Operator first considers the ascii value corresponding to the suit of the card. So, a sorted list of suits will contain the order of either (S,H,D,C) or (C,D,H,S). If the suits are the same, then the operator will consider the weight. You can check any of the sample outputs to view the sorted list of cards at the end of the game. Examples:

   (a) C2 is less than S2

   (b) DA is less than H2

   (c) ST is less than SA

5. Operator Overload > is the same thing as above but reverse. With some intuition, you can probably use one line of code for this operator (given you did the <).

6. Lastly, you have the GetWeight() function. This just returns the cards numeric weight. You can see this function is called in main().

Now the Deck Class. Most of the work is done in here.

MemberAccess	Type of Member	const	static	type	Name	Parameters
private	variable	No	No	Node*	Head	N/A
private	variable	No	No	int	SizeOfDeck	N/A
public	Constructor	N/A	N/A	N/A	Deck()	None
public	Constructor	N/A	N/A	N/A	Deck()	const Deck &copy
public	Destructor	N/A	N/A	N/A	~Deck()	N/A
public	Operator	No	No	void	operator=()	const Deck &copy
private	Operator	No	No	Node*	operator[]()	int

MemberAccess	Type of Member	const	static	type	Name	Parameters
public	Function	No	No	void	AddToTopOfDeck()	Card
public	Function	No	No	Card	RemoveTopCard()	None
public	Function	No	No	void	ShuffleDeck()	None
public	Function	No	No	void	ClearDeck()	None
public	Function	No	No	void	SortDeck()	None
public	Function	No	No	void	SwapCards()	Node*, Node*
public	Function	Yes	No	int	GetSizeOfDeck()	None
public	Function	Yes	No	bool	isEmpty()	None
public	Function	Yes	No	int	GetRandomCardIndex()	None
public	Function	No	No	void	PrintDeck()	None

Notice the **Node Struct** within the class. The Node contains a card and a single pointer to the next node. This is for the nodes in the linked list.

The member descriptions are listed in a specific order. I recommend completing them in this order (*because things like the operator[] can be used in the other functions!*):

1. The Default Deck Constructor should initialize Head to NULL and SizeOfDeck to 0.

2. ClearDeck() will deallocate the entire linked list and set SizeOfDeck to 0. Remember, if Head is already NULL, there is nothing to deallocate.

3. The destructor needs to deallocate the whole linked list. You can just call ClearDeck(), unless you want to code more.

4. Now the assignment operator. Consider the expression (A = B) where A and B are both decks. For the following explanation, I will refer to A as the Left-Hand-Side (LHS) Deck, which is also the deck belonging to *this*. I will refer to B as the Right-Hand-Side (RHS) Deck, which is also the parameter deck passed to the operator.

   First, the memory of the linked list on the LHS deck needs to be deallocated. Perhaps use ClearDeck() once again. Next, you should copy the size of the RHS deck. Now, you must go through the RHS deck's linked list. For each node in the RHS linked list, you will need to allocate a node in the LHS linked list. Once a node is allocated and linked, copy the Card within RHS node into the LHS node. Remember that the Head of

the LHS will need to be pointing to the first node of the list. Also, if the RHS deck is empty (i.e. the Head == NULL or SizeOfDeck == 0), there is no copying to be done. Lastly, ensure the last node of the LHS list is pointing to NULL or nullptr.

5. Now the Copy Constructor. This one should be easily done with all the work you did for the Assignment Operator. The copy constructor's job is to initialize a brand new deck to be an exact copy of the parameter deck.

6. Next, the bracket operator. This operator will make life easy when it comes to sorting and printing the linked list. The job of the bracket is to return a pointer to a node within the linked list. Consider Deck A. A[0] would return the Head (1st node) of the linked list within A. A[43] would return the 44th node within the linked list. All you have to do is iterate through the linked list an index amount of times (*int index is the parameter*) and return the node at that position. Perhaps do some checking here to verify that the index is within bounds.

7. AddToTopOfDeck will add the parameter card to the top of the deck. This means allocate a new node, copy the parameter card into the node, and insert that node as the new head of the linked list. Don't forget to increase the size.

8. RemoveTopCard will return the card stored within the head node. It will also deallocate the head node and update the head node to be pointing at the next node in the linked list.

9. GetSizeOfDeck returns SizeOfDeck.

10. isEmpty returns true if SizeOfDeck is 0.

11. GetRandomCardIndex is done for you. It returns a random index for the linked list. This is used in ShuffleDeck.

12. PrintDeck goes through the list and prints the deck. This one is given to you. Make sure your bracket operator is working otherwise this function will not work.

13. SwapCards is next. You will be swapping the 2 nodes within the linked list. You must only change the **next pointers** within the nodes. If you decide to just swap the card within the node, you will not receive full credit, though the program should work normally.

    There is an easy way and a hard way to accomplish this correctly. Consider using the bracket operator, it may help you. **Please check with your instructor to see which they would prefer, and if there are Point Deductions/Extra Credit involved with the 2 methods.**

Method 1 (Easy Way):

(a) The SwapCards function will only swap 2 nodes that are adjacent (*right next to each other*). This method is significantly easier than method 2 because the left-most node is always the previous node to the right-most node. In order to swap adjacent nodes, you must identify which node is the left-most and which node is the right-most. You also need to identify the node previous to the left-most node so that you can update its pointer as well. Note that the left-most node may be the head node, and thus, the head node will need an update instead. You will always update 3 pointers:

i. left-previous next or head
ii. left next
iii. right next

Method 2 (Hard Way);

(a) The best SwapCard function would be a function that could swap any 2 nodes in the list, whether they are adjacent or not. If you do it this way, you will need to find the node previous to the left most node and the node previous to the right most node. There are quite a few cases you need to consider here:

i. Which node is left-most and which node is right-most?
ii. Is the left-most node the head?
iii. Are the 2 nodes adjacent?
iv. Are the 2 nodes the same node?

At most you will update 4 pointers, and at least 3 pointers if the nodes happen to be adjacent.

14. SortDeck will sort the deck using the operator overload in the Card class. Use whatever sort you would like, as long as it is not a library sort. The bracket operator will be very helpful here to access the nodes using for-loops. Also, if you did the easy SwapCards, you should probably use a sort that only swaps adjacent nodes such as bubble sort, or a sort that doesn't call SwapCards at all.

15. ShuffleDeck will shuffle the cards all randomly. For a set amount of iterations (I pick SizeOfDeck$*$3), swap 2 random cards in the deck. If you did the easy SwapCards, you should only call GetRandomCardIndex once and swap the node located at the index with either the node to its left, or the node to its right. If you implemented the hard SwapCards, you can call GetRandomCardIndex twice and swap those 2 nodes.

2.2 Player.h and Player.cpp

The player class has both a Hand and Foot, as well as functions to interact with the decks. They can deal cards and claim cards.

There is nothing for you to do in here. This class is relatively simple and can be easily understood by just reading the functions and comments.

2.3 Main.cpp

There is nothing you have to code in here. You should be familiar with how main() works however. main() is responsible for the entire game of war. In this function, players are dealt cards. The cards are shuffled and then the game begins. You can see pretty much all the functions that will be called from Player and Deck classes in here.

3 How do you know your code works?

I would recommend making a testing_main.cpp to test out your decks before you try using it with the Players Class and Main. In the YouTube video, I will demonstrate this.

4 Compiling and Submitting

Compile you code using **make**
Use **valgrind –leak-check=full ./a.out** to check for memory leaks.
Submit your Card.cpp file to Webcampus with a text file answering the following questions. Please keep your responses concise and clear.

5 Comprehension Questions

1. The SwapCards() function is supposed to only swap the next pointers and not the card within the node. This may be annoying to code, but can you explain an example of where swapping the next pointers will be significantly better than swapping the data within the node?

2. In the SortCards function description, I mention you can use a sorting method that doesn't call swap cards at all. Can you describe a very simple sorting algorithm using linked lists that does not require SwapCards? **Hint:** You will need the unsorted linked list and an empty linked list.

3. This card game implementation benefits heavily from using linked lists, as opposed to using dynamically allocated arrays using pointers. Why do you think that is?