Assigned: Monday September 15, 2014, **Due: Friday, February 6, 2015, 11:59pm.**

Building a List with a Linked List (to model a deck of cards)

1. The List Abstract Data Type (ADT)

A "List" is an example of an Abstract Data Type (ADT): it defines the interface that is used to store data, but *not* the underlying data structure that the list is implemented in. In this assignment, we will build a list using *linked lists* (as opposed to the dynamic array from the last assignment). The linked list implementation we use will be singly linked.

An ADT is abstract in the sense that it defines the operations that need to be implemented, and it is up to the programmer to decide *how* to implement those operations. This layer of abstraction gives the programmer the flexibility to make decisions that are transparent to the end user of the ADT, although there may be trade-offs, such as in performance (how fast the operations happen), and size (how much memory the implementation takes up).

Our list will be used in the implementation of a program that models a deck of playing cards. There is a **`Card`** class, which defines a playing card, and which will be used by our **`List_linked_list`** class to store individual playing cards (see the implementation for details). For our purposes, we will model traditional playing cards from a 52-card deck, with standard suits and ranks. There will also be a **`Hand`** class, which will use our list to produce a deck or a hand of cards. Our **`List_linked_list`** needs to know what a **`Card`** is, and the **`Hand`** class needs to know how to use our **`List_linked_list`**. But, the **`List_linked_list`** does not need to know anything about a **`Hand`**.

2. Implementation Specifics

We have provided you the **`Card`** and **`Hand`** implementations in full. We have also provided you with the **`List_linked_list`** header file, and also with the **`List_linked_list`** implementation file, with empty function definitions. You will have to create the code for the functions in order to implement the list.

You must implement the list using a singly-linked list.

The `Card` Class

A card has a *suit* and a *rank* defined by an **`enum`** as follows:

```
enum Suit {CLUB,DIAMOND,HEART,SPADE};
enum Rank {TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN,JACK,QUEEN,KING,ACE};
```

A **`Card`** is created by one of two constructors:

```
Card();
Card(char r, char s);
```

The first constructor creates a default card with a *rank* of ACE and a *suit* of CLUB. The second constructor creates a card with the *rank* and *suit* defined in the parameters of the constructor, e.g.,

```
Card c = new Card('5','H');
```

Cards can be printed to the screen using the **`print_card()`** function.

The rank and suit can be set or read using setters and getters:

```
Suit get_suit() { return suit; }
void set_suit(Suit s) { suit = s; }

Rank get_rank() { return rank; }
void set_rank(Rank r) { rank = r; }
```

A card can be copied to another card using the "=" operator:

```
Card card2 = card1;
```

Two cards can be compared using the following function, which returns *true* if the cards are the same, and *false* if they are different:

```
bool same_card(Card c);
```

Cards can also be converted into an integer value using the **`card_int()`** function. This is useful for comparing two cards so that a hand may be ordered. Clubs are considered lower than Diamonds, then Hearts, and finally Spades. The lowest ranking card is a TWO, and the highest is an ACE.

The good news: your list implementation will only have to be concerned with copying cards, and printing cards (and we already wrote the print function for you!). In other words, most of the above is for reference and not even necessary for this project.

The `Hand` Class

A **`Hand`** holds all of its cards in a **`List_linked_list`**, called *hand*. A hand can be printed using the **`print_hand()`** or **`print_hand_int()`** functions. A 52-card deck can be created in two different ways, using either the **`create_deck()`** function, or by reading a deck in from standard input (i.e., the terminal, or a file piped in) using the **`read_deck()`** function.

Hands can be shuffled (re-ordered randomly) with the **`shuffle()`** function.

The number of cards in a hand can be determined using the **`cards_in_hand()`** function.

A card can be added to a hand with the **`add_card(Card c)`** function. A card can be removed from a hand with the **`remove_card(Card c)`** function, if the card exists in the hand. An error will be generated if the card is not already in the hand.

Cards can be moved between two hands ("dealt") with the **`deal_card_from_top(Hand &h)`** and **`deal_card_from_bottom(Hand &h)`** functions. These functions also return the card to the calling function, as well. An error is generated if there are no cards in the dealing hand.

The good news: We have already implemented all of these functions for you! (We will use them to test the functionality of your list). In future projects, we may have you add more functions to the **`Hand`** class. **It might be a good idea to become familiar with this class in order to do your own testing on your `List_linked_list` class.**

The `List_linked_list` Class

You are responsible for writing most of the functions in this class. The following is the header definition of the **`List_linked_list`** class:

```
#include <iostream> // for NULL
#include "card.h"

struct Card_Node {
      Card card;
      Card_Node *next;
};

class List_linked_list
{
public:
      List_linked_list(); // constructor
      ~List_linked_list();
      // copy constructor
      List_linked_list(const List_linked_list& source);
      // operator= overload
      List_linked_list operator =(const List_linked_list& source);

      void print_list();
      void print_list_int();
      bool is_empty() { return head==NULL; }

      // students must write the following functions in the .cpp file
      int cards_in_hand(); // returns the number of cards in the hand
      void make_empty(); // empties the list; head should equal NULL

      // inserts a card at the head
      // Should allow insert into an empty list
      void insert_at_head(Card c);

      // inserts a card at the tail (final node)
      // should allow insert into an empty list
      void insert_at_tail(Card c);
```

```
        // inserts a card at an index such that
        // all cards following the index will be moved farther
        // down the list by one.
        // The index can be one after the tail of the list
        // I.e., you can insert at index 0 into an empty list,
        // and you can insert at index 4 in a list with 4 nodes.
        // In the first case, the node inserted would become the head
        // In the second case, the node inserted would be inserted
        // after the current tail.
        void insert_at_index(Card c, int index);

        // replaces the card at index
        // A card at index must already exist
        void replace_at_index(Card c, int index);

        // returns the card at index.
        // allowed to crash if index is not in the list
        Card card_at(int index);

        // returns true if the card is in the list
        // returns false if the card is not in the list
        bool has_card(Card c);

        // removes the card from the list
        // Returns true if the card was removed
        // Returns false if the card was not in the list
        bool remove(Card c);

        // Removes the card from the head, and assigns the head
        // to head->next
        // Returns the card that was removed
        // Allowed to fail if list is empty
        Card remove_from_head();

        // Removes the card from the tail
        // Returns the card that was removed
        // Allowed to fail if the list is empty
        Card remove_from_tail();

        // Removes the card from index
        // Returns the card that was removed
        // Allowed to fail if index is beyond the end of the list
        Card remove_from_index(int index);

private:
        Card_Node *head; // the head of the list
};

#endif // List_linked_list_h
```

We have provided much less for this assignment than for the first assignment — there are no longer any hints for implementation, nor have we provided most of the function declarations. You must get used to creating those on your own. We have, however, been more clear about the functions in the header file — use those comments to ensure you are meeting the specification! Use the TAs for help if you get stuck. You should also use the previous assignment as a guide. Clever students might in fact do a lot of copying and pasting to get the function declarations into their **`List_linked_list.cpp`** file…

3. General Tips and Instructions

Testing is even more important for this assignment than for the dynamic array assignment. There are numerous corner cases when programming linked lists that you have to consider. We will do our best to test all possible corner cases when grading the assignment.

4. Low Level Details

Getting the files

There are two ways to get the files for this assignment. The first is by copying the original files from the class folder. The second is to use "git" to pull the files from the GitHub cloud server.

Method 1: copy files from the class folder

First ssh to the homework server and, and make a directory called orderedListAssignment

```
ssh -X your_cs_username@homework.cs.tufts.edu
mkdir hw2
```

change into that directory

```
cd hw2
```

At the command prompt, enter (don't forget the period):

```
cp /comp/15/public_html/assignments/hw2/files/* .
```

Method 2: pull from GitHub:

At the command prompt, enter

"`git clone https://github.com/Tufts-COMP15/2015s_HW2.git hw2`"

change into the directory that git created:

"`cd hw2`"

Using Eclipse (assuming you have followed the steps above to get the files):

1. See online video https://www.youtube.com/watch?v=BbnLZ2oqrD0
2. If in the lab, simply open Eclipse, and then start following from step 6 below.
3. If at home, ensure that you have installed (Mac) XQuartz, or (Windows) Cygwin (see here: https://www.youtube.com/watch?v=BbnLZ2oqrD0) or (Win) putty and XMing (http://sourceforge.net/projects/xming/).
4. ssh to the homework server using the -X argument:
   ssh -X your_cs_username@homework.cs.tufts.edu
5. start eclipse by typing "eclipse" (no quotes)

6. Use the following steps to set up your project (you'll get used to the steps quickly, steps A & B only need to be done once at the beginning of the course):
   A. Default location for your workspace is fine
   B. Click on "Workbench", then click "Window->Open Perspective->Other", and choose C++
   (note: if C++ is not listed, close Eclipse and re-open by typing Eclipse)
   C. Window->Preferences
      General->Workspace
         Save automatically before build

General->Editors->Text Editors
Displayed tab width (8) (recommended)
Show print margin: 80 col
Show line numbers
C/C++ -> Code Style
Select K&R [built-in] and then “Edit”, then rename to “K&R 8-tab”
Change Tab size to 8 (recommended)
Click “Apply” then “Ok”

D. File->New C++ Project
Fill in project name (no spaces!) IMPORTANT: the name CANNOT be exactly the same as your folder name. I suggest to simply put name_project (with the _project) for all project names (e.g., hw2_project)
Use default location should be checked.
Select Empty Project->Linux GCC
Click Next
Click "Advanced Settings"
1. On the left, under C/C++ General->Paths and Symbols (left hand side)
(you may have to click on the triangle next to C/C++ General)
Select "[Debug]" at the top for Configuration
Source Location Tab
Click "Link Folder"
Check "Link to folder in the file system"
Browse to find your folder.
(should be something like h/your_username/hw2)
Click OK
Click Apply

2. C++ Build->Settings (may have to click on the triangle)
For Configuration (at the top), select [All configurations]
GCC C++ Compiler: Command should be "clang++" (no quotes)
GCC C++ Linker: Command: clang++
Click "Apply"
Click OK

Click Finish

E. Click on the white triangle next to your project name.
The folder you linked to should be listed. Click on its triangle.
The files in that folder should be listed.

F. Test the build by clicking on the hammer in the icon bar.
You should see some compiling messages in the Console window at the bottom. (Things like, "Building file...; clang++, etc.)

G. The program will compile, but will have many warnings. You need to write the functions — double-click on List_linked_list.h to see the header file and List_linked_list.cpp to see the code.

Compiling and running:

1. At the command prompt, enter “**`make`**” and press enter or return to compile.
2. At the command prompt, enter “**`./hw2`**” and press return to run the new program.

Providing:

At the command prompt, enter “**`make provide`**” and press enter or return.