Assigned: Wednesday September 3, 2014, **Due: Friday, January 23, 2015, 11:59pm.**

Building a List with a Dynamic Array (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 C++ *dynamic arrays*, and in the next assignment, we will re-implement the list using *linked lists*.

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_dynamic_array`** 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_dynamic_array`** needs to know what a **`Card`** is, and the **`Hand`** class needs to know how to use our **`List_dynamic_array`**. But, the **`List_dynamic_array`** 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_dynamic_array`** header file, and also with the **`List_dynamic_array`** 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 dynamic arrays. Recall that a dynamic array is a standard C++ pointer to a given array in memory, and "grows" as more memory is needed. We use quotes around *grow*, because in order to increase the size of an array in C++, you need to create a completely new

array with a larger size, copy the old data to the new array, and clean up after yourself by deleting the old array and setting the list variable pointer to the new array.

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_dynamic_array`**, 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_dynamic_array` class.**

The `List_dynamic_array` Class

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

```
#include "card.h" // the definition of a Card

#define INITIAL_CAPACITY 10 // the initial size of our list.

class List_dynamic_array
{
public:
        List_dynamic_array(); // constructor
        List_dynamic_array(const List_dynamic_array& source); // copy constructor

        // operator= overload
        List_dynamic_array operator =(const List_dynamic_array& source);

        ~List_dynamic_array(); // destructor

        void print_list(); // prints the list in human-readable form
        void print_list_int(); // prints the list as integers based
    // on rank and suit
        bool is_empty() { return cards_held==0; }
        void make_empty() { cards_held = 0; } // makes the list empty

        void insert_at_head(Card c); // inserts at the beginning of the list
        void insert_at_tail(Card c); // inserts at the end of the list
        void insert_at_index(Card c, int index); // inserts at an index
        void replace_at_index(Card c, int index); // replaces the card
```

```
    // at an index
        Card card_at(int index); // returns the card at the index

        bool has_card(Card c); // returns true if the card is in the list

        bool remove(Card c); // removes the card
        Card remove_from_head(); // removes the beginning card
        Card remove_from_tail(); // removes the last card
        Card remove_from_index(int index); // removes the card at index

        int cards_in_hand() { return cards_held; }

private:
        void expand(); // expand the list when necessary
        Card *cards; // the array of cards
        int cards_held; // how many cards are currently in the hand
        int hand_capacity; // the capacity of the array that holds the cards
};

#endif // List_dynamic_array_h
```

The comments provide enough information to get you started on writing the code for the class. We have provided some other hints in the **`List_dynamic_array.cpp`** file, as well. We will also be going over some implementation details in class lectures. You should search for the comment "TODO" inside the file to find out what functions you need to write (there are twelve).

3. General Tips and Instructions

Test, test, test! In class, we will be discussing how to test your functions, and you should write a test for virtually every function. In other words, you need to set up a test for each function to see if it is working. For example, if you want to test the **`void insert_at_tail(Card c);`** function, you could write the following function in your **`main.cpp`** file (before the **`main()`** function):

```
void test_insert_at_tail() {
        List_dynamic_array test_list;
        Card card1('A','D'); // ace of diamonds
        Card card2('5','S'); // five of spades
        Card card3('8','C'); // eight of clubs

        test_list.print_list(); // should be blank

        test_list.insert_at_tail(card1); // insert a card
        test_list.print_list(); // should print the Ace of Diamonds (AD)

        test_list.insert_at_tail(card2); // insert another card
        test_list.print_list(); // should print AD,5S

        test_list.insert_at_tail(card3); // insert another card
        test_list.print_list(); // should print AD,5S,8C

}
```

In your main() function, you could then write the following:

```
int main(int argc, char **argv)
{
        // test functions
        test_insert_at_tail();
        exit(0);
}
```

If the output is as you expected, then your **`insert_at_tail()`** function works.

When we grade your assignments, we will run your functions through a battery of tests that will check all of the so-called "corner cases" — in other words, we will check to ensure that your functions are thoroughly correct for all possible cases.

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 hw1
```

change into that directory

```
cd hw1
```

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

```
cp /comp/15/public_html/assignments/hw1/files/* .
```

Method 2: pull from GitHub:

At the command prompt, enter

"`git clone https://github.com/Tufts-COMP15/2014f_HW1.git hw1`"

change into the directory that git created:

"`cd hw1`"

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

1. See online video https://www.youtube.com/watch?v=DzjnocBNRwc
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 (see here: http://xquartz.macosforge.org/landing/), or (Windows) MobaXterm (see here:http://mobaxterm.mobatek.net).
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 and C 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->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”
General->Workspace
Save automatically before build.

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., hw1_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/hw1)
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_dynamic_array.h to see the header file and List_dynamic_array.cpp to see the code. “TODO:” shows you where you need to write 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 “**`./hw1`**” and press return to run the new program.

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