12/16

```
    1: //Will Foley & Ryan Fu
    2: //LCStack class and functions
    3:
    4: #ifndef LCSTACK_H
    5: #define LCSTACK_H
    6:
    7: template <typename T>
    8: class LCStack{
    9: private:
   10:         struct Node
   11:         {
   12:                 T data_;
   13:                 Node* next_;
   14:         };
   15:         Node* pHead_;       /*the first node in stack*/
   16:         int size_;
   17: public:
   18:         //constructor
   19:         LCStack(void)
   20:         {
   21:                 //empty stack
   22:                 pHead_=NULL;
   23:                 size_=0;
   24:         }
   25:
   26:         //copy constructor
   27:         LCStack(const LCStack<T>& rhs)
   28:         {
   29:                 if (rhs.pHead_ == NULL)         //if the stack you want copy is empty
   30:                         pHead_=NULL;
   31:                 else
   32:                 {
   33:                         Node* rbegin = rhs.pHead_;
   34:                         pHead_ = new Node;                      //new node for the begi
nning
   35:                         Node* begin = pHead_;
   36:                         while (rbegin != NULL)          //while you havent reached the
end of the rhs stack
   37:                         {
   38:                                 begin->data_ = rbegin->data_;   //copy data
   39:                                 begin->next_ = new Node;                //new node for
the next one
   40:                                 begin = begin->next_;                   //iterating
   41:                                 rbegin = rbegin->next_;
   42:                         }
   43:                         size_ = rhs.size_;
   44:                         begin->next_ = NULL;
   45:                 }
   46:         }
   47:
   48:         //assignment operator
   49:         LCStack& operator =(const LCStack& rhs)
   50:         {
   51:                 if (rhs.pHead_ == NULL)
   52:                         pHead_=NULL;
   53:                 else
   54:                 {
   55:                         while(!empty()) //deleting all items in this stack
   56:                         {
   57:                                 pop();
   58:                         }
   59:
   60:                         pHead_ = new Node;                      //new node for the begi
```

Passes Tests

I get weird errors when I run this with valgrind. You may want to take a look at that.

```
nning
   61:                          Node* begin = pHead_;
   62:                          Node* rhs_node = rhs.pHead_;
   63:
   64:                          while (rhs_node != NULL)        //while you haven't reached the
 end
   65:                          {
   66:                                  begin->data_ = rhs_node->data_;         //copy data
   67:                                  begin->next_ = new Node;                        //new n
ode for then next one
   68:                                  begin = begin->next_;                           //itera
ting
   69:                                  rhs_node = rhs_node->next_;
   70:                          }
   71:
   72:                          size_ = rhs.size_;
   73:                          begin->next_ = NULL;
   74:                  }
   75:                  return *this;
   76:          }
   77:
   78:          //destructor
   79:          virtual ~LCStack(void)
   80:          {
   81:                  while(pHead_!=NULL)
   82:                  {
   83:                          Node* tmpNode= pHead_;
   84:                          pHead_=pHead_->next_;
   85:                          delete tmpNode;
   86:
   87:                  }
   88:
   89:          }
   90:
   91:          //push front
   92:          void push(const T& t)
   93:          {
   94:                  Node* pNode = new Node; //new node to hold data
   95:                  pNode->data_ = t;               //put data in new node
   96:                  pNode->next_=pHead_;    //have new node point to beginning
   97:                  pHead_=pNode;                   //let new node be the beginning
   98:                  size_++;
   99:          }
  100:
  101:          //pop front
  102:          T pop()
  103:          {
  104:                  T tmp = pHead_->data_;  //save the data
  105:                  Node* tmpNode= pHead_;  //save the node to be popped
  106:                  pHead_=pHead_->next_;   //move phead to the next one
  107:                  size_--;
  108:                  delete tmpNode;
  109:                  return tmp;
  110:          }
  111:
  112:          //look at top
  113:          T top()
  114:          {
  115:                  return pHead_->data_;
  116:          }
  117:
  118:          bool empty()
  119:          {
```

```
  120:                return size_ == 0;
  121:        }
  122:
  123:        int size()
  124:        {
  125:                return size_;
  126:        }
  127: };
  128: #endif
  129:
```

```
 1: //Will Foley & Ryan Fu
 2: //LCQueue class and functions
 3:
 4: #ifndef LCQUEUE_H
 5: #define LCQUEUE_H
 6:
 7: template <typename T>
 8: class LCQueue{
 9:
10: public:
11:         //default constructor
12:         LCQueue()
13:         {
14:                 data_ = new T [INITIAL_CAPACITY];
15:                 capacity_ = INITIAL_CAPACITY;
16:                 size_ = 0;
17:                 last_ = -1;
18:                 first_ = 0;
19:         }
20:
21:         //constructor for given capacity
22:         LCQueue(int capacity)
23:         {
24:                 data_ = new T [capacity];
25:                 capacity_ = capacity;
26:                 size_ = 0;
27:                 last_ = first_ = capacity/2;
28:         }
29:
30:         //copy constructor
31:         LCQueue(const LCQueue& old)
32:         {
33:                 data_ = new T [old.capacity_];
34:                 capacity_ = old.capacity_;
35:                 size_ = old.size_;
36:                 first_ = old.first_;
37:                 last_ = old.last_;
38:                 if (old.size_ > 0)
39:                 {
40:                         for (int i = 0; i < capacity_; ++i)     //copy over data
41:                         {
42:                                 data_[i] = old.data_[i];
43:                         }
44:                 }
45:         }
46:
47:         //assignment operator
48:         LCQueue& operator =(const LCQueue& old)
49:         {
50:                 delete [] data_;
51:                 data_ = new T [old.capacity_];
52:                 capacity_ = old.capacity_;
53:                 size_ = old.size_;
54:                 first_ = old.first_;
55:                 last_ = old.last_;
56:
57:                 if (old.size_ > 0)
58:                 {
59:                         for (int i = 0; i < capacity_; ++i)     //copy over data
60:                         {
61:                                 data_[i] = old.data_[i];
62:                         }
63:                 }
```

```
   64:
   65:                 return *this;
   66:         }
   67:
   68:         //destructor
   69:         virtual ~LCQueue()
   70:         {
   71:                 delete [] data_;
   72:         }
   73:
   74:         //push back
   75:         void push(const T& item)
   76:         {
   77:                 if (size_ == capacity_)
   78:                 {
   79:                         doubleCapacity();
   80:                 }
   81:
   82:                 last_ = (last_ + 1) % capacity_;        //update last
   83:                 data_[last_] = item;    //put the item into last
   84:                 ++size_;
   85:         }
   86:
   87:         //pop first
   88:         T pop()
   89:         {
   90:                 int first_index = first_;
   91:                 --size_;
   92:                 first_ = (first_ + 1) % capacity_;      //update first
   93:                 return data_[first_index];
   94:         }
   95:
   96:         //look at front
   97:         T& front()
   98:         {
   99:                 return data_[first_];
  100:         }
  101:
  102:         //double capacity
  103:         void doubleCapacity()
  104:         {
  105:                 T* newitems = new T [capacity_*2];
  106:                 for (int i = first_; i < capacity_; ++i)        //copy items from first
 to capacity
  107:                 {
  108:                         newitems[i] = data_[i];
  109:                 }
  110:                 for (int i = 0; i < first_; ++i)        //copy items from beginning to
first
  111:                 {
  112:                         newitems[i+capacity_] = data_[i];
  113:                 }
  114:                 last_ = first_ + size_ -1;
  115:                 delete [] data_;
  116:                 data_ = newitems;
  117:                 capacity_*=2;
  118:         }
  119:
  120:
  121:         bool empty()
  122:         {
  123:                 return size_ == 0;
  124:         }
```

```
125:         int size()
126:         {
127:                 return size_;
128:         }
129:
130: private:
131:         int size_;
132:         int capacity_;
133:         T* data_;
134:         int last_;
135:         int first_;
136:         // default capacity, if none is specified
137:         const static int INITIAL_CAPACITY = 2;
138: };
139: #endif
140:
```

```
 1: //Will Foley & Ryan Fu
 2: //KeyStream class
 3:
 4: #ifndef KEYSTREAM_H
 5: #define KEYSTREAM_H
 6:
 7: #include <iostream>
 8: #include <algorithm>
 9: #include <stdlib.h>
10: #include <vector>
11: #include "CryptCard.h"
12: #include "LCQueue.h"
13: #include "LCStack.h"
14:
15: using namespace std;
16:
17: class KeyStream
18: {
19:
20: public:
21:
22:   KeyStream(int seed);
23:   virtual ~KeyStream(void)
24:   {
25:         delete [] suit_;
26:   }
27:
28:   unsigned int generateNextKey(); //generates stream of integers
29:   unsigned int firstKey();              //access function to return the first key
30:
31:   static CryptCard::Suit suits[];
32: private:
33:   //shuffled deck of cards
34:   vector<CryptCard> shuffled_deck_;
35:
36:   //going to be an array of LCQueues to represent the suit decks
37:   LCQueue<CryptCard>* suit_;
38:
39:   //index locations of the suit decks
40:   int clubs;
41:   int spades;
42:   int diamonds;
43:   int hearts;
44:
45:   //stack for the algorithm
46:   LCStack<CryptCard> cards_;
47:
48:   //first key
49:   unsigned int first_key_;
50: };
51:
52: #endif
53:
54:
55:
```

```
    1: //Will Foley & Ryan Fu
    2: //functions for KeyStream
    3:
    4: #include "KeyStream.h"
    5:
    6: //constructor and initializing the suit decks
    7: KeyStream::KeyStream(int seed)
    8: {
    9:         clubs = 0;
   10:         spades = 1;
   11:         diamonds = 2;
   12:         hearts = 3;
   13:         int shuffle_count = 7;
   14:
   15:         srand(seed);    //prompting srand with the seed for the random shuffle
   16:
   17:         CryptCard::Rank r = CryptCard::ACE;
   18:
   19:         //creates a deck of cards to be snuffled
   20:         for (int suit = 0; suit < CryptCard::MAX_SUITS; ++suit) //we used an array of s
uits in the order we needed them to be
   21:         {
                                                //scroll to the bottom to look at the array
   22:                 r = CryptCard::ACE;
   23:                 for(int rank = 0; rank < CryptCard::MAX_RANKS; ++rank)
   24:                 {
   25:                         shuffled_deck_.push_back(CryptCard(suits[suit], r));
   26:                         CryptCard::incrementRank(r);    //increase the rank by one
   27:                 }
   28:         }
   29:
   30:         //shuffle the deck seven times
   31:         for (int i = 0; i < shuffle_count; ++i)
   32:                 random_shuffle(shuffled_deck_.begin(), shuffled_deck_.end());
   33:
   34:         suit_ = new LCQueue<CryptCard>[CryptCard::MAX_SUITS];   //intialize array of LC
Queues
   35:
   36:         //organizes the cards into 4 decks with there respected suits
   37:         for (size_t i = 0; i < shuffled_deck_.size(); ++i)
   38:         {
   39:                 if((shuffled_deck_[i]).getSuit() == CryptCard::CLUBS)
   40:                         suit_[clubs].push(shuffled_deck_[i]);
   41:                 if((shuffled_deck_[i]).getSuit() == CryptCard::SPADES)
   42:                         suit_[spades].push(shuffled_deck_[i]);
   43:                 if((shuffled_deck_[i]).getSuit() == CryptCard::DIAMONDS)
   44:                         suit_[diamonds].push(shuffled_deck_[i]);
   45:                 if((shuffled_deck_[i].getSuit()) == CryptCard::HEARTS)
   46:                         suit_[hearts].push(shuffled_deck_[i]);
   47:         }
   48:
   49:         first_key_ = -10;
   50:
   51:         //creating the first key
   52:         for(int i = 0; i < CryptCard::MAX_SUITS; ++i)
   53:         {
   54:                 first_key_+=(suit_[i].front()).convertToInt();
   55:         }
   56:
   57: }
   58:
   59: //access function for the first key
   60: unsigned int KeyStream::firstKey()
```

Make static constants

```
   61: {
   62:         return first_key_;
   63: }
   64:
   65: //generate keys for the encryption and decryption
   66: unsigned int KeyStream::generateNextKey()
   67: {
   68:         int key = -10;
   69:
   70:         cards_.push(suit_[clubs].pop()); //rotating the clubs deck
   71:         suit_[clubs].push(cards_.pop());
   72:
   73:         int index;
   74:         int i = 0;
   75:
   76:         if ((suit_[clubs].front()).getRank() == CryptCard::ACE) //if the first card in
the clubs deck is an ace then begin rotation
   77:         {
   78:                 while((suit_[i].front()).getRank() == CryptCard::ACE && i < CryptCard::
MAX_SUITS)
   79:                 {
   80:                         if(i == clubs)  //seperate case for rotating the spades deck
   81:                         {
   82:                                 cards_.push(suit_[i+1].pop());
   83:                                 suit_[i+1].push(cards_.pop());
   84:                         }
   85:                         else
   86:                         {
   87:                                 index = i;
   88:                                 if (index == hearts)    //if the ace is in the hearts t
hen we will need to look at the clubs deck
   89:                                 {
   90:                                         index=-1;
   91:                                 }
   92:                                 if (index+1 != clubs)   //rotate the deck to the right
as if its not the clubs deck
   93:                                 {
   94:                                         cards_.push(suit_[index+1].pop());
   95:                                         suit_[index+1].push(cards_.pop());
   96:                                 }
   97:
   98:                                 //while the top card on the left does not = the one on
the right pop the cards on the left on to a stack
   99:                                 while((suit_[i-1].front()).getRank() != (suit_[index+1]
.front()).getRank())
  100:                                 {
  101:                                         cards_.push(suit_[i-1].pop());
  102:                                 }
  103:
  104:                                 //then push them back onto the deck
  105:                                 while(cards_.size() != 0)
  106:                                 {
  107:                                         suit_[i-1].push(cards_.pop());
  108:                                 }
  109:                         }
  110:                         ++i;
  111:                 }
  112:                         i = 0;
  113:         }
  114:
  115:         //calculates the key
  116:         for(int n = 0; n < CryptCard::MAX_SUITS; ++n)
  117:         {
```

- 4

Break into functions

- 4

```
  118:                 key+=(suit_[n].front()).convertToInt();
  119:         }
  120:         return key;
  121: }
  122:
  123: //suit array used when creating a deck of cards
  124: CryptCard::Suit KeyStream::suits[] = {CryptCard::HEARTS, CryptCard::CLUBS, CryptCard::D
IAMONDS, CryptCard::SPADES };
```

```
 1: //Will Foley & Ryan Fu
 2: //encrypt class
 3:
 4: #ifndef ENCRYPT_H
 5: #define ENCRYPT_H
 6:
 7: #include <fstream>
 8: #include <string>
 9: #include <stdio.h>
10: #include <stdlib.h>
11: #include "KeyStream.h"
12:
13: using namespace std;
14:
15: class Encrypt
16: {
17: public:
18:         const static int ASCII_RANGE = 128;
19:         Encrypt(int seed, char* file);
20:         void encipher();
21: private:
22:         char* file_;
23:         int seed_;
24:
25: };
26: #endif
```

```
    1: //Will Foley & Ryan Fu
    2: //Encrypt class functions
    3:
    4: #include "encrypt.h"
    5:
    6: //constructor
    7: Encrypt::Encrypt(int seed, char* file)
    8: {
    9:         seed_ = seed;
   10:         file_ = file;
   11: }
   12:
   13: //encipher a file
   14: void Encrypt::encipher()
   15: {
   16:         ifstream encr(file_);   //get the file
   17:
   18:         if (encr.is_open())
   19:         {
   20:                 unsigned int letter;
   21:                 KeyStream keys(seed_);  //initialize the keystream
   22:                 letter = encr.get();    //get the first letter
   23:                 if (letter < ASCII_RANGE) //if the letter is in the ascii range
   24:                 {
   25:                         letter += keys.firstKey();      //add the first key to it
   26:                         letter %= ASCII_RANGE;          //take the modulus
   27:                         cout << (unsigned char)letter;
   28:
   29:                         letter = encr.get();            //get the next letter
   30:                         while(encr.good() && letter < ASCII_RANGE)      //while you hav
ent reached the end of the file
   31:                         {
                                //or haven't run into a bad letter
   32:                                 letter += keys.generateNextKey();       //add the next
key to the letter
   33:                                 letter %= ASCII_RANGE;                          //take
the modulus
   34:                                 cout << (unsigned char)letter;          //print it
   35:                                 letter = encr.get();                            //and g
et the next one
   36:                         }
   37:                 }
   38:                 if (!encr.eof())        //if you did not reached the end of the file
   39:                 {
   40:                         if (letter >= ASCII_RANGE)      //if you found a letter outside
 the range
   41:                                 cout << "Error character found that it is outside ASCII
_RANGE" << endl;
   42:                 }
   43:         }
   44:         encr.close();
   45:
   46: }
```

cerr

```
    1: //Will Foley & Ryan Fu
    2: //Launches program to encipher a file
    3:
    4: #include <iostream>
    5: #include <string>
    6: #include <stdio.h>
    7: #include <stdlib.h>
    8: #include "encrypt.h"
    9:
   10: using namespace std;
   11:
   12: int main(int argc, char* argV[])
   13: {
   14:         int seed = 1;
   15:         int txt_file = 2;
   16:
   17:         if (argc == 3)          //you need the name of the program, a seed and a file t
o encrypt
   18:         {
   19:                 Encrypt e(atoi(argV[seed]), argV[txt_file]);
   20:                 e.encipher();
   21:         }
   22:         else
   23:                 cout << "You need to have a seed followed by a file you want to encrypt
." << endl;
   24:
   25:
   26:         return 0;
   27: }
```

```
 1: //Will Foley & Ryan Fu
 2: //Decrypt class
 3:
 4: #ifndef DECRYPT_H
 5: #define DECRYPT_H
 6:
 7: #include <iostream>
 8: #include <fstream>
 9: #include <sstream>
10: #include <string>
11: #include "KeyStream.h"
12:
13: using namespace std;
14:
15: class Decrypt{
16: private:
17:         int seed_;
18:         char* fileName_;
19:         const static int ASCII_RANGE=128;
20: public:
21:         Decrypt(int seed, char* fileName);
22:         void decipher ();
23: };
24: #endif
```

```
    1: //Will Foley & Ryan Fu
    2: //decrypt functions
    3:
    4: #include "decrypt.h"
    5:
    6: //constructor
    7: Decrypt::Decrypt(int seed, char* fileName)
    8: {
    9:         seed_=seed;
   10:         fileName_=fileName;
   11: }
   12:
   13: //decipher a file
   14: void Decrypt::decipher()
   15: {
   16:         ifstream file;
   17:         file.open (fileName_);  //open a file
   18:         KeyStream keys(seed_);  //initialize keystream
   19:
   20:         unsigned int letter;
   21:         unsigned int currentKey;
   22:
   23:         if (!file.is_open())
   24:         {
   25:                 cout<<"ERROR: File was not opened!"<<endl;
   26:         }
   27:         else
   28:         {
   29:                 letter = file.get();            //getting the first letter
   30:                 if (letter < ASCII_RANGE)       //if the letter is less than the ASCII_
RANGE
   31:                 {
   32:                         currentKey=keys.firstKey();     //get the first key
   33:                         while(letter < currentKey)      //while the letter is less than
 the ascii range add the range to it
   34:                         {
   35:                                 letter+=ASCII_RANGE;
   36:                         }
   37:                         letter-=currentKey;                     //subtract the key
   38:                         letter%=ASCII_RANGE;            //take the modulus to make sure
 it stay in the ascii range
   39:                         cout<<(unsigned char)letter;
   40:                         letter = file.get();            //getting next letter
   41:                         while (file.good() && letter < ASCII_RANGE)     //while you hav
ent reached the end or run ino a letter outside the range
   42:                         {
   43:                                 currentKey=keys.generateNextKey();      //getting next
key
   44:                                 while(letter <currentKey)
   45:                                 {
   46:                                         letter+=ASCII_RANGE;    //add the range while t
he letter is less than the key
   47:                                 }
   48:                                 letter-=currentKey;                     //subtract the
key
   49:                                 letter%=ASCII_RANGE;            //take the modulus to m
ake sure it stays in the proper range
   50:                                 cout<<(unsigned char)letter;    //print it
   51:                                 letter = file.get();            //get the next letter
   52:                         }
   53:                 }
   54:                 if(!file.eof())         //if you haven't reached the end
   55:                 {
```

```
   56:                         if (letter >= ASCII_RANGE)      //then you must of found a bad
charecter
   57:                                 cout << "Error character found that it is outside ASCII
_RANGE" << endl;
   58:                 }
   59:         }
   60:         file.close();
   61: }
```

```
    1: //Will Foley & Ryan
    2: //Launches decipher program
    3:
    4: #include <iostream>
    5: #include <string>
    6: #include <stdio.h>
    7: #include <stdlib.h>
    8: #include "decrypt.h"
    9:
   10: using namespace std;
   11:
   12: int main(int argc, char* argV[])
   13: {
   14:         int seed = 1;
   15:         int txt_file = 2;
   16:
   17:         if (argc == 3)  //you need the name of the program, a seed and a file to decryp
t
   18:         {
   19:                 Decrypt d(atoi(argV[seed]), argV[txt_file]);
   20:                 d.decipher();
   21:         }
   22:         else
   23:                 cout << "You need to have a seed followed by a file you want to decrypt
." << endl;
   24:
   25:         return 0;
   26: }
```

```
    1: //Will Foley & Ryan
    2: //CryptCard Class
    3:
    4:
    5: #ifndef CRYPT_CARD_H
    6: #define CRYPT_CARD_H
    7:
    8: #include <ostream>
    9:
   10: class CryptCard
   11: {
   12: public:
   13:   enum Suit{HEARTS = 0, CLUBS, DIAMONDS, SPADES, NO_SUIT};
   14:   enum Rank{ACE = 0, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN,
 KING, NO_RANK};
   15:   const static int MAX_RANKS = 13;
   16:   const static int MAX_SUITS = 4;
   17:   // Number of cards in a deck
   18:   const static int CARDS_PER_DECK = 52;
   19:
   20:   static const int SUIT_OFFSET = 1;
   21:   static const int RANK_OFFSET = 1;
   22:
   23: private:
   24:   Suit suit_;
   25:   Rank rank_;
   26:
   27:   static std::string *rank_symbols_;
   28:   static char *suit_symbols_;
   29:
   30:   static std::string * initRankSymbols();
   31:   static char * initSuitSymbols();
   32:
   33: public:
   34:   CryptCard();
   35:   CryptCard(Suit suit, Rank rank);
   36:   virtual ~CryptCard(void);
   37:
   38:   Suit getSuit() const;
   39:   Rank getRank() const;
   40:
   41:   // convert card to an integer by multiplying rank by suit
   42:   int convertToInt();
   43:
   44:   //Increment the rank using the standard formula, e.g., A->2, 2->3, 3->4, ..., Q->K.
   45:   static void incrementRank(Rank& rank);
   46:
   47:   friend std::ostream& operator << (std::ostream& os, const CryptCard& card);
   48: };
   49:
   50: #endif //CRYPT_CARD_H
```

```
    1: //Will Foley & Ryan
    2: //CryptCard functions
    3:
    4: #include "CryptCard.h"
    5: #include <string>
    6:
    7: using namespace std;
    8:
    9: CryptCard::CryptCard() : suit_(NO_SUIT), rank_(NO_RANK){}
   10:
   11: CryptCard::CryptCard(Suit suit, Rank rank) : suit_(suit), rank_(rank){}
   12:
   13:
   14: CryptCard::~CryptCard(void){
   15:   // nothing yet
   16: }
   17:
   18: CryptCard::Rank CryptCard::getRank() const{
   19:   return rank_;
   20: }
   21:
   22: CryptCard::Suit CryptCard::getSuit() const{
   23:   return suit_;
   24: }
   25:
   26: void CryptCard::incrementRank(Rank& rank){
   27:   //rank = (Rank)(rank + 1);
   28:   int higher_rank = static_cast<int>(rank) + 1;
   29:   rank = static_cast<Rank>(higher_rank);
   30: }
   31:
   32: int CryptCard::convertToInt()
   33: {
   34:   int i_suit = static_cast<int>(suit_) + SUIT_OFFSET;
   35:   int i_rank = static_cast<int>(rank_) + RANK_OFFSET;
   36:   return (i_suit * i_rank);
   37:
   38: }
   39:
   40: ostream& operator << (ostream& os, const CryptCard& card){
   41:   os << CryptCard::rank_symbols_[card.rank_] << CryptCard::suit_symbols_[card.suit_];
   42:   return os;
   43: }
   44:
   45: char * CryptCard::initSuitSymbols(){
   46:   char * dynTmp = new char[MAX_SUITS];
   47:   dynTmp[HEARTS] = 'H';
   48:   dynTmp[DIAMONDS] = 'D';
   49:   dynTmp[CLUBS] = 'C';
   50:   dynTmp[SPADES] = 'S';
   51:   return dynTmp;
   52: }
   53:
   54: string * CryptCard::initRankSymbols(){
   55:   const char * tmpRankSymbols[] = {"A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "
J", "Q", "K"};
   56:   string * dynTmp = new string[MAX_RANKS];
   57:   for(int i = 0; i < MAX_RANKS; i++){
   58:     dynTmp[i] = string(tmpRankSymbols[i]);
   59:   }
   60:
   61:   return dynTmp;
   62: }
```

```
63:
64: char * CryptCard::suit_symbols_ = CryptCard::initSuitSymbols();
65: string * CryptCard::rank_symbols_ = CryptCard::initRankSymbols();
```