Lab 08 Stacks

Exercise 1: Creating and using stacks with arrays

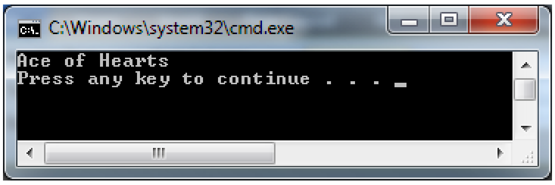
a) Create the following object class using C++ code

Card
-deck:string -suit:string

+Card()
+Card(d:string,s:string)
+getCard():string

Overload the `ostream` operators such that when a `Card` type object is passed to `cout`, the card's details are displayed e.g. `cout<<myCard` is shown below

```
int main(){
        Card myCard("Ace","Hearts");
        cout << myCard << endl;
}
```



```
Ace of Hearts
Press any key to continue . . .
```

b) Add the following `ArrayStack` ADT class to your code (either in the same file or as separate files with necessary `#include` headers)

```
template <typename ItemType>
class ArrayStack{
public:
        ItemType* items; // Stack array
        int top,arraysize;  // Index to top of stack
public:
        ArrayStack(int sz);
        bool isEmpty() const;
        bool push(ItemType newEntry);
        bool pop();
        int getSize();
        ItemType peek() const;
};

template<typename ItemType>
ArrayStack<ItemType>::ArrayStack(int sz) : top(-1),arraysize(sz){
        items=new ItemType[arraysize];
}

template<typename ItemType>
bool ArrayStack<ItemType>::isEmpty() const {
        return (top < 0);
}

template<typename ItemType>
bool ArrayStack<ItemType>::push(ItemType newEntry){
        bool result = false;
        if (top < arraysize){
                // Stack has room for another item
              top++;
              items[top] = newEntry;
              result = true;
       }  // end if
return result;
}

template<class ItemType>
bool ArrayStack<ItemType>::pop() {
       bool result = false;
       if (!isEmpty()) {
              result = true;
              top--;
       }  // end if
       return result;
}

template<typename ItemType>
int ArrayStack<ItemType>::getSize(){
    return top;
}

template<class ItemType>
ItemType ArrayStack<ItemType>::peek() const{
       if (!isEmpty()) { // no exception handling used, so check validity elsewhere
              return items[top];
       }  // end if
}
```

Using the `ArrayStack` class above, learn to create a “stack” of cards and test all the functions of the class in your driver program.

c) Implement *inheritance* with the `ArrayStack` ADT class above and create a subclass called `DeckOfCards`.

- The `DeckOfCards` subclass should have a proper default constructor that creates an `ArrayStack` of 52 cards (13 face values for 4 different suits) everytime an instance of the subclass is created.
- Using `random_shuffle` from the `algorithm` library, create a shuffle method that shuffles the cards. `random_shuffle` needs three arguments – array starting point, array ending point and a seed value
- Create a `getCard` method that returns the top most card in the stack (i.e. `peek` then `pop`)

Test the `DeckOfCards` subclass in your driver program

d) Create a simple draw card game by copying the following driver program code and completing the sections required

```
int main(){
       deckofcards newdeck;
       // call a shuffle of the deck
       int decision,numofplayers;
       bool cont=true;
       cout << "Enter number of players: ";
       cin >> numofplayers;
if (!cin.fail()){
              while(cont){
                     cout<<"[1] Draw cards\n[2] End Game\nEnter a number: ";
                     cin>>decision;
                     if (!cin.fail()){
                            switch (decision){
                            case 1:
```

```
                    // add code to draw correct number of cards
                case 2:
                    // add code to end game
                default:
                    cout<<"Unknown option entered\n\n";
                    cin.clear();
                    cin.ignore(numeric_limits<streamsize>::max(), '\n');
                    break;
                }
            } else{
                cout <<"Enter a number only\n";
                cin.clear();
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
            }
        }
    }
    else
        cout<<"Ending game.\n";
}
```

Exercise 2: Creating and using stacks with link-lists

Repeat exercise 1 but this time replace the `ArrayStack` ADT class with the `LinkedStack` ADT class instead. There should be minimal necessary changes to the main/driver program or the `Card` classes and you can reuse the `node` classes from the previous tutorial.

Note: The final game program does not evaluate which player wins – how would you change the classes to make the program (either Array/Linked Stack version) be able to check which card drawn is the winning card?

Extra Exercises

Question 1: Postfix calculator

Some calculators require you to enter postfix expressions. Recall that an operator in a postfix expression applies to the two operands that immediately precede it. Thus, the calculator must be able to retrieve the operands entered most recently. The ADT stack provides this capability. In fact, each time you enter an operand, the calculator pushes it onto a stack. When you enter an operator, the calculator applies it to the top two operands on the stack, pops the operands from the stack, and pushes the result of the operation onto the stack. The final result will be on the top of the stack.

Use the pseudocode algorithm given below to evaluate postfix expressions. Use only the operators +, –, *, and /. Assume that the postfix expressions are syntactically correct.

```
for (each character ch in the string) {
  if (ch is an operand)
    Push value that operand ch represents onto stack
  else { // ch is an operator named op
    // evaluate and push the result
    operand2 = top of stack
    Pop the stack
```

```
            operand1 = top of stack
            Pop the stack
            result = operand1 op operand2
            Push result onto stack
        }
    }
```

Task:
Implement the calculator that evaluates a postfix expression by filling out the empty method. Make sure your method can process any number of digits from an integer.

Sample Run 1:

```
Enter a postfix expression to be evaluated: 2 10 100 / * 1000 +
2 10 100 / * 1000 + = 1000.2
```

Sample Run 2:

```
Enter a postfix expression to be evaluated: 2 3 4 + *
2 3 4 + * = 14
```

Sample Run 3:

```
Enter a postfix expression to be evaluated: 2 9 2 / * 8 + 7 - 4 /
2 9 2 / * 8 + 7 - 4 / = 2.5
```