BBM 203 Programming Assignment 1

Name: Anıl

Surname: Akkaya

Number: 21945781

Date: 19 November 2020

1) Problem Definition

In this programming assignment, goal was to simulate a game called Solitaire, by using input files deck and commands, and producing an output file containing data about commands and the board state after each command. Proper error handling was also required for the problem.

2) My Solution

I heavily used an object-oriented approach to the problem. I used classes for representing game components like board, cards, card slots, foundations, piles, stock, and waste. Since dynamic memory usage was restricted, I used arrays to represent internal gameplay data.

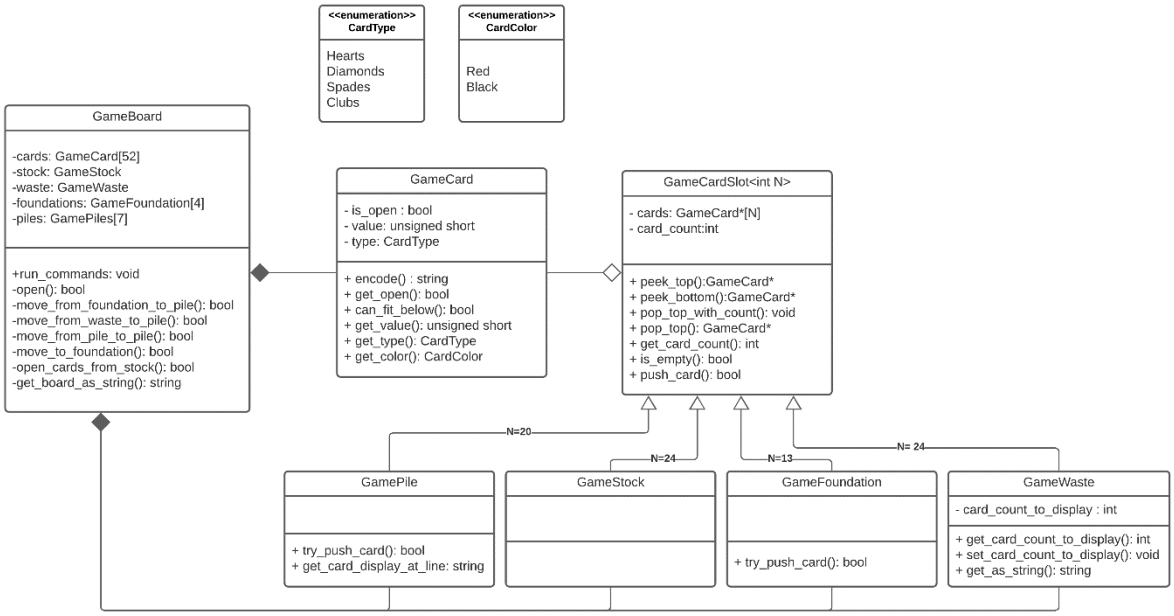
Key part of my design is the usage of pointers and class templates. All 52 cards that are on the board are stored as an array of **GameCard** objects inside the class **GameBoard**. You can think **GameBoard** class as a manager class, its task is to manage the game and relation between components.

Most important part of the problem was to represent which card was where. So, a card might be inside waste, pile, stock, or a foundation. In my opinion, best way to represent this relation was to use pointers to the cards and swap them to simulate a card move. As game goes on, card locations are changed on board, so instead of moving the whole card objects around, I have designed arrays that can hold pointers to the **GameCard** objects, and moved those pointers around these arrays to modify the state of the game.

The functionality of storing N pointers to the **GameCard** is supplied by the templated class **GameCardSlot<N>,** this class allows us to specify an integer as a template parameter N, which represents maximum amount of card pointers that this class can hold. By using this number N compile-time, an array that holds pointers by length N is created. Components **GamePile**, **GameStock**, **GameFoundation**, and **GameWaste** inherit from the **GameCardSlot** class, so they all have ability to store pointers to the **GameCard** objects, in different sizes. Main logic here is that there can only be 1 pointer that can point to a specific card, and depending on where that pointer is located, we can know its location.

For example, at the start of the game we have 24 **GameCard** pointers that are contained in **GameStock**, and 28 **GameCard** pointers that are contained on different **GamePile** instances. As the game goes on, these pointers change their location to other container classes to logically represent the game state. But the actual **GameCard** instances never change their location. This was a design choice made by myself, I could have also used a single variable on each card that represents where the card belongs, but that requires you to handle cases explicitly for each place: If card is in a pile, which pile is it? what index is it on that pile? what if card is not in a pile and in the waste? What is that index there? … By using my design, I get rid of all these complications and developed my software easily and neatly.

3) UML Class Diagram



GameBoard: Is the manager class that holds every data about the game, pointers to cards and cards themselves. Game runs from here.

GameCard: Represents a card object, has member values like type and value.

CardType and CardColor: These are simple enumerations that represent the type and color of a card.

GameCardSlot: Base (Interface) class that defines the ability to hold pointers to the cards, with a maximum number N. It has helper functions for operations like pushing and popping the cards.

GamePile: Represents a single pile instance. Holds maximum of 20 card pointers and has helper methods for pushing cards and getting display for output.

GameStock: Represents the stock component of the game, holds maximum 24 closed card pointers, that can be transferred to waste if command is taken. No additional functionality is needed, **GameCardSlot** interface is enough.

GameFoundation: Represents a single foundation instance, can hold up to maximum 13 card pointers. It has a helper method for pushing a card on top of it.

GameWaste: Represents the waste component that can hold up to 24 card pointers. It has specialized functionalities to deal with the display.

4) Usage of Arrays

Arrays are used often in my design process.

Here is a list of all classes that uses arrays:

- GameBoard, arrays are used for storing 52 **GameCard** instances, also for storing 7 **GamePile** instances and 4 **GameFoundation** instances.

- GameCardSlot<N>, this class creates an array of N **GameCard** pointers, and exposes an interface to push, pop, traverse through these cards. All implementation is done with non-dynamic arrays that have known size on compile time. This class also holds an integer data member called ***card_count***, that represents the amount of the cards stored in this slot. This class is one of the most important classes on my program, as it prevents lots of code repetition and provides proper interface to build other components upon this interface.

5) How To Compile & Run My Program

I have used C++11 features in my project, so it is a good idea to use a C++11 compiler:

```
g++ -std=c++11 *.cpp -o main
./main deck.txt commands.txt output.txt
```