CSE3150

Homework Assignment: The War Card Game

Dr. Justin Furuness

Deadline

October 30th. NO RESUBMISSIONS AFTER THIS POINT.

Overview

In this assignment, you will design and implement an object-oriented C++ program that simulates the card game **War**. This project will reinforce your understanding of:

- Abstract classes and inheritance
- Virtual functions and polymorphism
- Resource management with `std::unique_ptr`
- STL containers (particularly `std::deque`)
- Operator overloading (`operator<<`, `operator[]`, comparison operators)
- Exception handling and file I/O
- Proper project structure with headers and source files

You will organize your project into two directories:

```
include/   all .h files
src/       all .cpp files
```

Your compiled program must produce an executable named `war_game`.
This will start to get us ready for our course project as we tackle larger codebases.

Directory and File Structure

Your final directory should look like this:

```
 include/
   Card.h
   PlayingCard.h
   FaceCard.h
   JokerCard.h
   Deck.h
   FileReader.h
   FileWriter.h
   WarGame.h

src/
   Card.cpp
   PlayingCard.cpp
   FaceCard.cpp
   JokerCard.cpp
   Deck.cpp
   FileReader.cpp
   FileWriter.cpp
   WarGame.cpp
   main.cpp
```

1 Step 1 Create the Abstract Base Class `Card`

Create `Card.h` and `Card.cpp`.

- The class `Card` should be abstract and define the common interface for all cards.
- It must declare at least one **pure virtual function**.
- Required members:
  - `int value() const` returns the card's numeric value.
  - `void print(std::ostream& os) const` prints the card's representation.
  - Virtual comparison operators: `operator<` and `operator==`, comparing by numeric value.
  - A `virtual` destructor.

- Add a `friend operator<<` that calls `print()` to support polymorphic streaming. We can't make friend functions virtual since they are functions and not methods, which is why we have friend call print.

This class defines the shared behavior that all cards must implement.

2 Step 2 Implement Concrete Card Types

Each subclass should be implemented in its own header and source file pair.

2.1 `PlayingCard`

- Inherit from `Card`.
- Store the card's suit (a string) and rank (an int) as protected members. We want to use protected rather than private here so that other subclasses can still easily access these.
- have a function called rankToString that is a switch statement that takes in an int rank and returns ”A”, ”J”, ”Q”, or ”K”, and by default just convert the rank to a string using std::to_string.
- Playing card should have a custom constructor that takes in the suit string and rank
- The value method should be overridden (and use the overriden keyword) from the parent class. It should just return the rank of the card.
- The print method should be overriden as well. Keep in mind the print method does not modify the object, so make it const and make sure all implementations of print are labelled as const. When printing it should stream rankToString(rank_) ¡¡ ” of ” ¡¡ suit_.

2.2 `FaceCard`

- Inherit from `PlayingCard`.
- Add a custom constructor
- Override `print()`. This time when streaming, have a different switch statement for the ranks which uses the name ”Jack”, ”Queen”, ”King” instead of ”J”, ”Q”, ”K”. Note this is also why we needed the superclass to have the suit and rank as protected members rather than private, so that we can now access them here. If these attributes were private FaceCard could not access them.

- Jack = 11, Queen = 12, King = 13.

2.3 `JokerCard`

- Derive directly from `Card`.
- The joker has no child classes, add the proper keyword to enforce this.
- For the joker custom constructor, have it take only the string for the color and set this member color.
- For the custom constructor, put the keyword explicit in front of it. Explicit will protect us from accidental conversions. Without it, someone could do Joker j = "red" and that will actually get picked up as a string and converted to a JokerCard, unless we add explicit here to tell the compiler JokerCard can only be explicitly constructed using the JokerCard("red") format.
- Override the value and return 14.

3 Step 3 Implement the `Deck` Class

Create `Deck.h` and `Deck.cpp`.

- Use `std::deque<std::unique_ptr<Card>>` internally. This has a better runtime for adding and removing from the bottom and top of the deck, and in C++ we can access the middle of a deque with O(1) runtime.
  - We want to use unique pointers to cards here because we can't just store a vector or deque of Card, since Card is not concrete (and is abstract). We can however store pointers to objects that are of type Card, that works perfectly fine.
- Set the default constructor and the two move constructors to be the defaults.
- Delete the compiler generated copy constructors. Copying a deck should not be allowed since it is using unique pointers.
- Implement a method called size which should return a size_t type and also be const since it doesn't modify the object, and it should just return the card members size.
- Implement a function empty that returns a bool and is const that just returns if the cards are empty.
- Implement functions begin and end, simply returning the deque for cards begin and end. Use auto for the return type.

- Implement begin and end a second time, but this time label the functions as const. These are needed for when deck is const and you iterate over it. This should return the cards_.cbegin method and cards_.cend() method. This is a new concept but it's fairly simple. Any time the deck is const, the const version of the iterator gets called by the compiler, otherwise the non const version gets called by the compiler. If the deck is const and we did not provide a const begin method but tried to call begin on a const deck, the compiler would error.
- Implement a method draw that returns a unique pointer to a card. If the cards deque is empty return a nullptr. Get the front() of the cards dqueue, pop the front, and then return that stored value for front. Feel free to use auto here!
- Implement addToBottom, which should take in a unique ptr for Card and add this to the bottom using push_back to our deque (remember you can't pass around unique pointers, you must move them with std::move).
- I will provide the code for split that takes in no parameters and returns std::pair¡Deck, Deck¿. This is a very common pattern in C++ for returning multiple values from a function. In this function, we use adapters to make move iterators (which iterates over a container of values and moves them, like we need for unique pointers), as well as adapters like back inserter (which we've seen already) to move these values into our two new decks.

```
std::pair<Deck, Deck> split() {
    Deck a, b;
    size_t half = cards_.size() / 2;

    std::move(std::make_move_iterator(cards_.begin()),
              std::make_move_iterator(cards_.begin() + half),
              std::back_inserter(a.cards_));

    std::move(std::make_move_iterator(cards_.begin() + half),
              std::make_move_iterator(cards_.end()),
              std::back_inserter(b.cards_));

    cards_.clear();
    return {std::move(a), std::move(b)};
}
```

- Lastly, we want to override the << operator. Here simply return a comma separated list with a space after each comma of the cards in the deck, using the cards built in stream operator. When getting these cards, build a simple range loop over deck, not the decks cards. Range loops will call the iterators that we have implemented for the deck class.

4 Step 4 File Input and Output

To make this lab a little bit easier, here are the two implementations for the FileWriter and FileReader. Now however you should be able to build these just fine! Go ahead and type these out so that you get some practice.

```
// ============================================================
// FileReader: loads a deck from a CSV
// Format: Suit,Rank   or   Joker,Color
// ============================================================
class FileReader {
public:
    static Deck readDeckFromCSV(const std::string& path) {
        Deck deck;
        std::ifstream file(path);
        if (!file.is_open())
            throw std::runtime_error("Failed to open input deck: " + path);

        try {
            std::string line;
            while (std::getline(file, line)) {
                if (line.empty()) continue;

                size_t commaPos = line.find(',');
                if (commaPos == std::string::npos)
                    throw std::runtime_error("Malformed CSV input");

                std::string suit = line.substr(0, commaPos);
                std::string value = line.substr(commaPos + 1);

                if (suit.empty() || value.empty())
                    throw std::runtime_error("Malformed CSV input");

                if (suit == "Joker") {
                    deck.addCard(std::make_unique<JokerCard>(value));
                } else {
                    int rank = std::stoi(value);
                    if (rank < 1 || rank > 13)
                        throw std::runtime_error("Malformed CSV input");

                    if (rank >= 11)
                        deck.addCard(std::make_unique<FaceCard>(suit, rank));
                    else
                        deck.addCard(std::make_unique<PlayingCard>(suit, rank));
                }
            }
        } catch (...) {
            throw std::runtime_error("Malformed CSV input");
        }

        if (deck.size() == 0)
            throw std::runtime_error("Empty or invalid CSV deck");

        return deck;
    }
};

// ==============================================================
// FileWriter: writes each round to a CSV file
// ==============================================================
class FileWriter {
    std::ofstream file_;
public:
    explicit FileWriter(const std::string& path) {
        file_.open(path, std::ios::out | std::ios::trunc);
        if (!file_.is_open())
            throw std::runtime_error("Failed to open output CSV: " + path);
        file_ << "Round,PlayerA_Count,PlayerB_Count,PlayerA_Cards,PlayerB_Cards\n";
    }

    void writeRound(int round, const Deck& a, const Deck& b) {
        file_ << round << "," << a.size() << "," << b.size() << ",\""
              << a << "\",\"" << b << "\"\n";
    }
```

```
};
```

5 Step 5 Implement the `WarGame` Class

Create `WarGame.h` and `WarGame.cpp`.

- The class manages two player decks, an integer for the round, and a FileWriter object.
- In the custom constructor, take a deck and an outputPath for the output file. Take the input deck and call the split function to split it between two players like so `auto [a, b] = deck.split()`. Then move a and b into the two player decks.
- Implement a function play. Play should print ”Starting War”. While either of the two decks are not empty, the game should print the round with Round ¡num¿ and add a newline, and then it should play a round. The writer should then call the writeRound function, and the round should be incremented. When the while loop exits, it should print Game Over, and print Player A or B wonts with x cards!” ro ”It’s a tie!”, along with a newline.
- Have a private function playRound. In playRound, each player draws a card. If either card drawn evaluates to False return (a deck is empty and the game is over). Otherwise, print what each player plays, such as ”Player A plays: (card here)” and a newline. Check which card is greater than the other, and if the card from player Bs deck is less than or equal to the card from player As deck, then A wins the round (add As card, and then Bs card, to the bottom of As deck). If not, B wins the round (Add Bs card, then As card, to the bottom of the deck). NOTE: follow the order of adding cards to the bottom of the deck exactly. (in the real card game there’s special logic for ties, but to make this assignment a bit shorter we omit that logic).

6 Step 6 The `main()` Function

Create `main.cpp` in the `src/` directory.

- Accept two command-line arguments:
  - Input CSV path
  - Output CSV path

- If three arguments aren't passed in (check this using argc for argument count) stream to std::cerr `"Usage: ./war_game <input_csv> <output_csv>\n"`
- Read the deck from the input file using `FileReader`.
- Construct a `WarGame` with the deck and output path.
- Call `play()` to run the simulation.
- Catch any exceptions and print meaningful error messages and return 1, printing any errors to std::cerr with code that looks like this (otherwise return 0):

```
            } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
        return 1;
    }
```

7 Step 7 Rules and Behavior

NOTE: this step should already be implemented in your code, just double check!

- Aces are **low** (value = 1).
- Jokers are **highest** (value = 14).
- Cards are compared by numeric value only.
- The deck is not shuffled; use the order from the CSV file.
- The game writes each round to the output CSV.

8 Step 8 Submission

Run the tests provided locally. Submit following the steps outlined in cse3150_week_6_lab. For the name of the repo and PR, use cse3150_final_hw. Also, I've updated PRPals permissions so that I can add myself as a collaborator to more easily view the repo (since it is private), you may get prompted to update this.