HW: Crazy 8s

Overview

Objectives

1. Practice with classes (constructors, getters, setters, and encapsulation).
2. Working with dynamic memory and pointers.
3. Using vectors.
4. Additional practice with file I/O.
5. Additional practice with exceptions.

Submission

Submit the following files to the autograder:

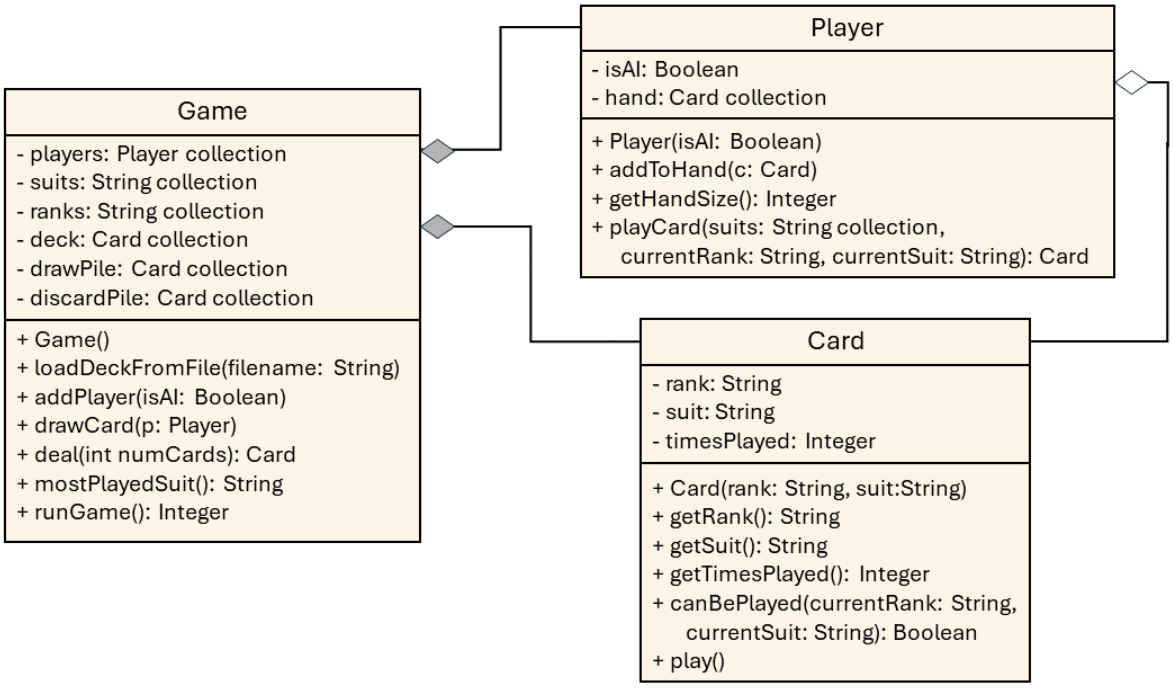
- `Card.cpp`
- `Player.cpp`
- `Game.cpp`
- `main.cpp`

Introduction

Crazy 8s is a simple classic card game. In this homework, you will create a C++ version of the game, which you can play against friends, or against a (very basic) artificial intelligence. Along the way, you'll get to practice your skills at handling dynamic memory and object-oriented programming.

Getting Started

- Get the starter code
    - `Game.h/.cpp`: the class definition for the Crazy 8s game; Keeps track of all players and cards.
    - `Player.h/.cpp`: the class definition for the players of the game, who may be humans or AIs.
    - `Card.h/.cpp`: the class definition for the cards used to play the game.
    - `main.cpp`: this file is useful for testing your code locally, and contains a menu for setting up a game.
- Note the data files are in the starter code
- Review the code.
    - Read header and source files.
    - Compile and run the initial state of the starter code and submit to Gradescope to get an idea of the test cases.
    - Review the organization of the classes. The following UML class diagram on the next page shows relationships among the `Game`, `Player`, and `Card` classes used for this program.



- Look at the dependencies in classes. Best to implement in approximately the following order:
    1) Card
    2) Player
    3) Game
    4) main()
- Allowed includes:
    - `<iostream>`
    - `<fstream>`
    - `<sstream>`
    - `<vector>`
    - `<string>`
    - `<cctype>`
    - `<stdexcept>`
    - `"Card.h"`
    - `"Player.h"`
    - `"Game.h"`
- Refer to the appendix as needed.

Recommendations

- **Read the entire document first** as class and function relationships can get confusing if you aren't careful.
- As you develop your code, you only need to implement the methods/functions you need at the moment. You don't have to get everything done at once (It is not a good strategy to follow anyways).

- You can utilize `main.cpp` to test the functions that you have implemented so far. Alternatively, you can create a separate tester file with its own main function.
- Plan and **think before you code** (write test cases first).
- **Test locally and start early**.

```
$ g++ -std=c++17 -Wall -Wextra -pedantic-errors -Weffc++
-fsanitize=undefined,address *.cpp
$ ./a.out
```

Sample Execution

```
Choose a file to load the deck from:
microDeck.txt
Enter number of players:
hi
Please enter a positive number
0
Please enter a positive number
2
Is player 0 an AI? (y/n)
q
Please enter y or n
n
Is player 1 an AI? (y/n)
y
How many cards should each player start with?
none
Please enter a positive number
5
The initial discard is 7 Hearts
Player 0's turn!
Your hand contains: 8 Clubs, 7 Spades, 4 Spades, 9 Hearts, 5 Hearts
The next card played must be a 7 or Hearts
What would you like to play? (enter "draw card" to draw a card)
8 Hearts
```

That's not a card you have. Try again.

4 Spades

You can't play that card. Try again.

7 Spades

Player 0 plays 7 Spades.

Player 1's turn!

Player 1 plays 7 Clubs.

Player 0's turn!

Your hand contains: 8 Clubs, 4 Spades, 9 Hearts, 5 Hearts

The next card played must be a 7 or Clubs

What would you like to play? (enter "draw card" to draw a card)

8 Clubs

What suit would you like to declare?

Squares

That's not a suit in this deck. Try again.

Spades

Player 0 plays 8 Clubs and changes the suit to Spades.

Player 1's turn!

Player 1 draws a card.

Player 0's turn!

Your hand contains: 4 Spades, 9 Hearts, 5 Hearts

The next card played must be a 8 or Spades

What would you like to play? (enter "draw card" to draw a card)

4 Spades

Player 0 plays 4 Spades.

Player 1's turn!

Player 1 plays 8 Hearts and changes the suit to Hearts.

Player 0's turn!

Your hand contains: 9 Hearts, 5 Hearts

The next card played must be a 8 or Hearts

What would you like to play? (enter "draw card" to draw a card)

9 Hearts

Player 0 plays 9 Hearts.

Player 1's turn!

```
Player 1 plays 9 Clubs.
Player 0's turn!
Your hand contains: 5 Hearts
The next card played must be a 9 or Clubs
What would you like to play? (enter "draw card" to draw a card)
draw card
Draw pile, empty, flipping the discard pile.
Player 0 draws a card.
Player 1's turn!
Player 1 plays 3 Clubs.
Player 0's turn!
Your hand contains: 5 Hearts, 7 Hearts
The next card played must be a 3 or Clubs
What would you like to play? (enter "draw card" to draw a card)
draw card
Player 0 draws a card.
Player 1's turn!
Player 1 plays 10 Clubs.
Player 0's turn!
Your hand contains: 5 Hearts, 7 Hearts, 7 Spades
The next card played must be a 10 or Clubs
What would you like to play? (enter "draw card" to draw a card)
draw card
Player 0 draws a card.
Player 1's turn!
Player 1 plays 2 Clubs.
Player 1 wins!
The most played suit was Clubs
```

Requirements

- You will be implementing the three classes as well as the functions in `main.cpp`.
- Constructors should **use** member initialization lists.
- Exceptions should have meaningful descriptions.
- The program must compile without warnings or errors.

Card Class

We provide

- Card class definition (`Card.h`)
- Methods / Member functions (`Card.cpp`)
    - **`Card::play()`**
        - Increments the number of times the card has been played

You implement in Card.cpp

- Constructor
    - **`Card::Card(string rank, string suit)`**
        - Make sure to initialize ALL the card's attributes, not just rank and suit
        - Throw `std::invalid_argument` if `rank` or `suit` are empty or contain non-alphanumeric characters (i.e. anything other than letters and numbers)
- Getters
    - **`string Card::getRank()`**
    - **`string Card::getSuit()`**
    - **`int Card::getTimesPlayed()`**
- Methods / Member functions
    - **`bool Card::canBePlayed(string currentRank, string currentSuit)`**
        - Determine whether this card can be played right now.
            - currentRank and currentSuit are the rank and suit from the previous card (the one that was just played)
        - Normally a card can only be played if it matches the previous card in suit or in rank
        - 8s are wild, however, and can always be played

Player Class

We provide

- Player class definition (`Player.h`)

You implement in Player.cpp

- Constructor
    - **`Player::Player(bool isAI)`**
- Getters
    - **`size_t Player::getHandSize()`**
        - Note: size_t is a special numeric type guaranteed to be able to hold the size of any object. It functions basically the same as `unsigned int`.
    - **`string Player::getHandString()`**
        - Returns a string representation of the player's hand.
        - Cards should be listed from the front of the hand (oldest card) to the back (newest card)
        - The rank and suit of a card should be separated by a space
        - Consecutive cards should be separated by a comma and a space

        - For example, if the cards in the player's hand are the 8 of Hearts and the 3 of Diamonds, this method should return: "`8 Hearts, 3 Diamonds`"
- Methods / Member Functions
    - **`void Player::addToHand(Card* c)`**
        - Add the given card to this player's hand.
        - The hand should be ordered from least recently added to most recently added (i.e. new cards should go at the back of the hand)
    - **`Card* Player::playCard(vector<string> const& suits, string& currentRank, string& currentSuit)`**
        - Choose a card to play, remove it from the player's hand and return it.
        - Make sure to increment the card's `timesPlayed` and to update the current rank and suit
        - If the player chooses not to play a card (perhaps because they have no card they can play), return nullptr to indicate that they are drawing a card.
        - The process of choosing a card to play is different for AI players and for humans
            - The AI will always play the first card from its hand that it can legally play, or draw a card if it has no cards it can play
            - For human players, you must tell them the current rank and suit and ask them to choose a card to play by entering two strings.
                - Prompt the player to choose again if they choose a card they don't have or a card they have in hand cannot play (see the sample execution above)
            - Upon playing an 8, a player has the opportunity to change the current suit to any suit in the game
                - The AI will always choose to use the suit of the 8
                - Human players should be prompted to pick a suit
                    - Make sure to check whether the suit they choose is actually present in this deck

Game Class

We provide

- class definition (`Game.h`)
- Constructor (`Game.cpp`)
    - **`Game::Game()`**
- Destructor (`Game.cpp`)
    - **`Game::~Game()`**
    - If you haven't learned about destructors **yet**, that's okay, you don't need to understand this function right now

You implement in Game.cpp

- Methods / Member Functions
    - **`void Game::loadDeckFromFile(string filename)`**
        - If file could not be opened, throw `std::runtime_error`
        - Refer to the Data Files section, which specifies the format of `Card` information included in the file.
            - Use the rank and suit information to initialize `ranks` and `suits`
            - Create `Cards` as you read the lines from the file and use them to populate `deck` and `drawPile`
        - If the file content does not match the specified format, throw `std::runtime_error`. This includes:
            - A line that is too short or too long (i.e. doesn't have both rank and suit, or has extra information)
            - A card with a rank or suit that is not in the list at the beginning of the file
            - If the card constructor throws an `std::invalid_argument` you should catch it here and throw a `std::runtime_error` instead.
        - When creating a new `Card` object, remember that we are working with dynamic memory (Use the `'new'` keyword to allocate memory on the heap)
        - Cards read from the file should be added to both `deck` and `drawPile`
            - `deck` is a permanent index of all cards and will not change over the course of the game
            - `drawPile` keeps track of the cards that have not been drawn yet, and will shrink as cards are drawn during the game.
            - `deck` should list the cards in the same order that they appear in the file (i.e. the first card in the input file should be at position 0)
            - `drawPile` should list the cards in the opposite order (i.e. the *last* card in the input file should be at position 0)
            - Cards will be drawn from the end of `drawPile` (recall that removing from the end of an array/vector is faster than removing from the beginning)
    - **`void Game::addPlayer(bool isAI)`**
        - Create a new player and add them to the end of players
        - When creating a new `Player` object, remember that we are working with dynamic memory (Use the `'new'` keyword to allocate memory on the heap)
    - **`void Game::drawCard(Player* p)`**
        - Move the top card of the draw pile (the *last* card in `drawPile`) into the player's hand.
        - If the draw pile is empty, you will need to replenish it before you draw the player their card:
            - If the discard pile has at least two cards then print "`Draw pile, empty, flipping the discard pile.`" and then form a new draw pile by leaving just the top card in the discard pile (as a reminder of what was played last) and flipping the rest of the discard pile (the second card from the top of the discard pile will become the bottom card of the draw pile, and the bottom card of the discard pile will end up as the top card of the draw pile)

        - In a typical game we would shuffle the discard pile instead, but this non-random approach makes testing and debugging easier
    - If the draw pile is empty and the discard pile contains only one card, there is nothing to draw, so print nothing and throw an `std::runtime_error`
- **`Card* Game::deal(int numCards)`**
    - First form a starting discard pile by discarding the top (last) card of the draw pile
    - Then, deal cards from the top of the deck to the players' hands one at a time (use the `drawCard` method)
        - You should *not* catch any exceptions thrown by `drawCard`
        - That is, if you are unable to draw enough cards for all the players, `deal` should allow the exception thrown by `drawCard` to continue being thrown.
    - You should give each player their first card (starting with player 0) before giving any player their second card.
        - That is, you should repeatedly give each player one card, in turn order, until each player has received `numCards` many cards.
    - Return the initially discarded card
- **`string Game::mostPlayedSuit()`**
    - Figure out how many times each suit has been played by adding up the timesPlayed values for all cards of that suit
    - Return the suit that has been played the most times
    - In case of a tie, return any of the tied suits
    - Remember that `deck` contains all cards in the game, whether they are in the draw pile, the discard pile, or a player's hand.
- **`int Game::runGame()`**
    - Run the Crazy 8s game and return the number (index in `players`) of the winning player
    - You may assume that the game has been set up using `loadDeckFromFile`, `addPlayer`, and `deal`
    - Refer to the appendix for details on the rules of our version of Crazy 8s
    - Refer to the sample execution for examples of the messages you should print
    - At the start of each turn you should announce "Player <Player #>'s turn!"
    - You should determine which card the player is playing, if any
    - If the player is playing a card:
        - You should announce the rank and suit of this card
        - If the card is an 8, you should announce the new suit that the player has chosen
        - You should add the played card to the top (end) of the discard pile
    - If the player is drawing a card
        - You should announce this
        - You should draw a card and add it to their hand

        - If there are no more cards to draw, print "Player <Player #> cannot draw a card." and return -1 to indicate a draw (tied game).
            - You will need to catch the exception thrown by `drawCard` to detect this
    - Whenever any player has `0` cards remaining in hand, end the game and return the winner.

main.cpp

Use the Sample Execution in the Overview above to fill in any gaps when implementing the following helper functions.

- **`void setupPlayers(Game& g, int numPlayers)`**
    - Determine whether each player is a human or AI by asking the user questions of the form `"Is player 0 an AI? (y/n)"`
        - If the user provides an invalid response, ask them to `"Please enter y or n"` and wait for a new input
    - As you figure out whether the players are AIs or humans, add them to the game `g`
- **`void setupGame(Game& g)`**
    - Ask the user `"How many cards should each player start with?"`
        - If the user enters anything other than a positive number, prompt them to try again by printing `"Please enter a positive number"`
    - Deal the chosen number of cards to each player in the game
        - You should **not** catch any exceptions thrown by `g.deal()`
    - Print the initial discard (e.g. `"The initial discard is 3 Spades"`)
- The other functions in main.cpp are provided for you, you will not need to modify them.

Appendix

Deck File Format

- Line 1: Suits: A list of suits in the deck separated by spaces
- Line 2: Ranks: A list of ranks in the game separated by spaces. One of these ranks should be “8”
- Lines 3+: These lines contain the individual cards, one per line. Each line consists of a rank and a suit, separated by a space

See the provided data files for examples

Rules of Crazy 8s

There are many different variations of Crazy 8s. To keep the amount of work for the assignment reasonable, we have chosen a basic version of the game.

The game is played with any deck using ranks and suits. Initially all of the cards in the deck are placed in the draw pile. To make testing and debugging easier, the pile is not shuffled. At the start of the game, the top card of the draw pile is discarded to determine the initial rank and suit. Each player is dealt the same number of cards (this number is specified by the user). The objective of the game is to achieve an empty hand by playing all of your cards.

Players take turns, going around in a circle. On your turn, you may play a card (moving it from your hand to the discard pile) if it matches the current rank or suit. If you do this, the new rank and suit become the rank and suit of the card you played. You can play at most 1 card per turn. If you cannot play a card, or do not wish to do so, you must instead draw a new card from the top of the draw pile and add it to your hand. After drawing a card, your turn ends, even if the newly drawn card is one you would have been able to play. There is no limit on the number of cards that you can have in hand.

As the name of the game suggests, cards of rank 8 are an exception to the usual rules. You may play an 8 on your turn regardless of the current rank and suit. Furthermore, when playing an 8, you may declare a new suit, which can be any suit in the game. For example, you could play the 8 of Spades and declare "Diamonds" as the suit, in which case the next card played would need to be a card of the Diamonds suit (or another 8).

If the draw pile ever runs out of cards, the next time a player needs to draw a card they first flip the discard pile over to form a new draw pile (leaving the top card as a reminder of the current rank and suit). In the unlikely event that it is not possible to form a new draw pile in this way (because all the cards except the top discard are in the players' hands), the game is declared a draw (a tie).

Useful functions and classes

You may use any of the functionality provided by C++ strings and C++ vectors in your solution. Most students may find the following operations useful:

- `vector`
  - `back()`: Returns the last element of the vector
  - `erase(pos)`: Remove the element of the vector at position `pos`
    - Note, this function takes an "iterator", not an `int`
    - You can obtain an iterator for position `i` in vector `v` using `v.begin()+i`
  - `pop_back()`: Removes the last element of the vector
  - `push_back(elem)`: Adds a new element elem at the end of the vector
  - `size()`: Returns the number of elements in the vector
  - More information about vectors is available in the Vector Class Overview section

`Vector` Class Overview

`std::vector` is a C++ standard class that provides the functionality of dynamically allocated arrays. You can learn more about C++ vectors in the documentation or in the Zybook readings.

A simple example of using `std::vector` to store a list of consecutive int values is below:

```
#include <iostream>
#include <vector>

int main() {
    // n is not fixed (providing the same capability as dynamic arrays)
    size_t n = 0;
    std::cout << "Enter value of n: ";
    std::cin >> n;

    std::vector<int> v;
    for (size_t i = 0; i < n; i++) {
```

```
        v.push_back(i); // add i to the last position in v
    }

    // let's print the elements in the vector
    // we can use a for loop again
    for (size_t i = 0; i < n; i++) {
        std::cout << v.at(i) << " " ;
    }
    std::cout << std::endl;
    return 0;
}
```