Straights

Overview

This report describes how the game program “straights” is compiled and the underlying structure of the program.

Design

The program learned from the Model-View-Control design pattern.

The Model-View-Control (simplified MVC) has a central model class which stores all information of the program. The model has two major manipulations which are view and control. View means displaying the information of the program and control means manipulate the contained statistics.

This program “straights” has a major class which is named subject. This subject class is behaving like the model class in MVC. The subject class has a few display and move functions which have the same ability as view and control.

Since the game is a card game and each card its unique rank and suit, I represent each card using an integer. 0-12 represent A club to K club. 13-25 represent A diamond to K diamond, 26-38 represent A Heart to K Heart and 39-51 represent A Spade to K Spade. The vectors of cards are substituted with vectors of integers.

This program “straights” has 8 source files and related header files.

1. pub_func.h and pub_func.cc

These files contain no structures but provide 4 public functions. Since all deck cards are represented with an integer, 3 functions, string getRank(int), string getSuit(int), string getName(int), are set to access the rank, suit and name of the card from an integer which is named index, the other function, int getCard(string) is set to obtain the index of a card from its name of the string type. If the name is invalid which does not represent any card in deck, the function throws an exception InvalidInput(1) (explained next).

2. exceptions.cc and exceptions.h

These files provide three exception classes. One is named InvalidInput, one is named Quit and one is named Restart.

Class InvalidInput has only one field – int type, and is initialized with InvalidInput(int t)

type equals 1 means the input fails to refer to a card. type is other numbers means the input command is invalid.

When the read input does not meet requirement, a InvalidInput will be thrown and the system will stay unchanged and wait for the next command.

Class Quit has no fields and no member functions. This class will only be thrown when the command is quit. Catching this exception will terminate the program.

Class Restart also has no fields and no member functions. This class will only be thrown when the command is restart and members agree to restart. Catching this exception will reshuffle the deck and deal cards.

3. pile.cc and pile.h

These files provide one class named Pile. Pile has two fields – string suit_name and vector<int> cards.

Each Pile represents a row of cards with a same suit. The suit_name is the complete name of suit of this row of cards like Club, Diamond. The vector, cards, is always ordered from low rank to high rank (same as from low index to high index).

Pile has 3 public functions in addition to its constructor and destructor. One can add a card (index) to the field cards and place the cards in increasing order. One can display the information in a specific format. The other can clear the vector cards.

4. player.cc, player.h, computer.cc, computer.h, human.cc, human.h

These files provide one parent class Player and its 2 child classes, Computer and Human.

Player has 4 fields—int id, int score, vector<int> hands, vector<int> discards. Hands refers to the cards in the player's hand and discards refer to the cards the player has discarded.

Player can be constructed with either an integer id or a pointer of another Player. When constructed using integer, id is set with the integer, score is set as 0 and hands, discards are empty. If constructed using another Player, it copies all information of the other Player.

Player has a public function which sets the hands.

When the player plays, one function play(int c, shared_ptr<Pile>p) plays the card c. This function adds the card to the Pile and removes the card c from the player's hands.

If the player discards a card, a function discard(int c) moves the card c from the player's hands to discards. The function name discard is overloaded. Player has another function discard() which automatically choose a card from hands discard the card. This function is for computer to play.

Player has a function that returns a vector of int which represents the indexes of playable cards. This function has a parameter, vector<int> accepts. Accepts have the indexes of cards that the system accepts. This function compares the cards in the accepts vector and cards in hands and return a vector of int in which the card exists in both sides.

Player has a function addscore(). This function sums the ranks of cards in the discards and add the sum to the score of the Player and returns the sum.

Player has one pure virtual function, bool virtual is_computer(). Class Computer overrides and returns true while class Human overrides and returns false. Computer and Human do not have any other fields and member functions except the fields and member functions of the Player.

Class Human has only one constructor Human(int id):Player{id}{} while class Computer has two constructors--Computer(int id):Player{id}{} and Computer(shared_ptr<Player> pl):Player{pl}{}. This is set to change a Human player to Computer player.

5. subject.cc subject.h

Class subject behaves as model contains all information of the game system. It has 6 fields

- vector<int> cards is reserved with size 52. This field refers to the deck of cards.
- vector<shared_ptr<Player>> players has size 4 since the number of players is set as 4.
- vector<shared_ptr<Pile>> piles is reserved with size 4 since there are 4 kinds of suits.
- vector<int> acceptable refers to cards that the system can accept as a played card.
- int current indicates the id of the current player.
- int played indicates the number of cards that has been played.

The subject is the game system. The system has both view and control functions and all start, move, end functions.

void begin(int id) displays start information and void initialize(string inp) constructs either a Computer player or a Human player due to the inp string or throw InvalidInput(2). void shuffle(bool have_seed, unsigned seed) shuffles the deck with the seed if the have_seed is true. void start() clears the hands and discards of players, clears the cards of piles, reset the played to 0 and run shuffle(..) to shuffle the deck.

Subject has two major functions to react to each Player's actions. void move() runs after checking the current player is a Computer. This imitates the action of the computer like play and discard. void move(istream * inp) accepts a pointer of an input stream and react according to the input command.

move(istream * inp) initialises a string command and store the first input word in command and check if the command meets any allowed command of the interface. If the command is not acceptable, throw a InvalidInput and recollect the input.

main.cc

The main.cc file has a helper function, bool allnum(char* argu) which checks if the char array is a number, and a main function int main(int argc, char* argv[]).

The main function initialises an unsigned seed. Then it checks the value of argc to see if a seed is given. If the argc is greater than 2, print an error information and terminate the program. If the allnum(argv[1]) returns false, terminate the program. Otherwise set seed to the number. If argc is 1, randomly generate a seed with respect to the system time.

Initialize a shared Subject pointer and display the first board to accept input about game difficulty. Then initialize the players with input until four players are all initialized. Then shuffle the deck with the seed and deal cards to each player and set the player with 7-spade to the first player as the current player.

Now the subject (the game system) moves. Each move is either a player's play or a player's discard. I constructed a while(true) loop. Each time the loop runs, the subject judges the current player is a Human or Computer and moves once. If the current player is a Human, accept the input stream and run s->move(&cin). This function provides expected behavior to the command. If any errors happen like invalid command or invalid card name, displays an error message and restart the loop without changing any states.

If the current player is a Computer, run s->move(). This allows Computer player to automatically choose a card to either play or discard.

In the end of each loop, Subject s checks if the round of game ends by checking the value of played. If the played equals 52 which means all cards have been played or discarded, this round of game ends and moves to check if the game ends by checking if anyone's score is greater than 80. If there is, end the game and display the winner and break the loop. Otherwise shuffle the deck and start a new round.

Difference to Plan

In my plan design, I represented each card as a structure, class Card{ int rank, int index, string suit}. While after compilation, an error of std::bad_alloc occur due to a lack of available memory. Then I replaced all Card structure with its index and implemented three public functions to obtain rank, suit, name of a card of a specific index. This shrinks memory needed for all structures. Since the previous Deck was merely a vector of cards and provides only one useful function which gets the index of a card with the name of a card, I deleted the Deck and moved it into Subject and name the field deck (vector<int>). When constructing a Subject, I initialize the deck and fill it from 0 to 51 in order. This allows me to shuffle and check the order of the deck. Since Deck is deleted, I provide another public function that returns the index of a card name(0 of AC).

Regarding changing a Human to Computer, I wanted to use cast but failed because the parent class is virtual. I then implemented copy constructor to change a Human to Computer. Since the pointers are smart pointers, there is no worry of deleting the previous Human pointer.

Other changes from the original design are some minor changes like helper functions and other functions with special abilities. The game model is identical to my original design.

Resilience to Change

The abilities of all functions are unique. The parameters of all functions are specified. These classes have separated their jobs into various special functions.

For example, if I want to change how the rows of suits display, I would only need to change Pile::display() function and does not need to modify Subject::display() function because Subject::display() calls Pile::display() to display the rows and both function do not change anything in the structures.

Or, if I need to add or remove or edit acceptable command, it is enough to modify Subject::move(istream * inp) because this function compares the input and respond to the command. If command autoplay needs to be added, I could add a judgement condition that (command == “autoplay”) and use the existing functions to complete the requirement.

Or, I want to allow the board to be displayed for all kinds of players, I could add s->display() in front of s->move().

Or, I want to change the acceptable starting cards, I only need to change the Subject::start() regarding how vector acceptable is initialized.

Overall, if any function is changed, if only its ability and parameters are not changed, none of other functions need to be changed.

Answers to Questions

Question 1: What sort of class design or design pattern should you use to structure your game classes so that changing the interface or changing the game rules would have as little impact on the code as possible? Explain how your classes fit this framework.

Answer: I used MVC to structure my game classes. MVC is structed with 3 parts, model, view and control. My class Subject is the model since it has all information the game needs. The various display functions of Subject are view. The Subject::move(istream * inp) is control.

Display functions allow players to view their hands and the table. Subject::move(istream *inp) allows player to enter input and make the system to react to acceptable commands.

Question 2: If you want to allow computer players, in addition to human players, how might you structure your classes? Consider that different types of computer players might also have differing play strategies, and that strategies might change as the game progresses i.e. dynamically during the play of the game. How would that affect your structure?

Construct a parent class Player which has all needed functions and fields for a player (regardless of computer player or human player). Give Player a pure virtual function bool is_computer(). Derive a child class Computer of Player and a child class Human of Player. Both Computer and Human can use all member functions of Player and access all fields of Player. Computer overrides is_computer() and returns true while Human overrides is_computer() and returns false.

To let computer players to have play strategies, I can implement the functions of choosing a card to play. I can write a function which takes a vector of int (the integers in the vector are all allowed cards) as a parameter and return the most suitable card index according to play or discard. The computer can use these functions to find the best cards to either play or discard. Meanwhile, the structures of all classes do not need to change because these functions can exist as private helper functions.

Question 3: If a human player wanted to stop playing, but the other players wished to continue, it would be reasonable to replace them with a computer player. How might you structure your classes to allow an easy transfer of the information associated with the human player to the computer player?

I implemented a copy constructor of Player which takes a pointer of Player as parameter (Player(shared_ptr<Player>pl):id{pl->get_id()},score{pl->get_score()},hands{pl->get_hands},discards{pl->get_discards()}{}) and copies all fields of the other Player. I then give Computer a copy constructor which consumes a pointer of Player – Computer(shared_ptr<Player>pl): Player{pl}{}. Since the subject has only a vector of Players, I take out the pointer of the Human player and copies its statistics to a new Computer shared pointer and replace the Human pointer in the vector with the copied Computer pointer. Since the Human pointer is a smart pointer, this structure will automatically destruct with no occurrence.

Cast is not allowed in this situation because Player has a pure virtual function which means Player is a pure virtual class. Casting over virtual classes is unpredictable.

Extra Credit Features

1. The entire project has no leaks and all pointers are replaced with smart pointers. There is no delete statement. No array is used and vectors are used instead.

2. At the beginning of the game, the player is allowed to choose the difficulty of the game. If the player enters easy, the computer players will always play the first card in legal plays and discard the first card in hands. If the player enters hard, the computer players will be smarter and choose the cards with the highest rank in legal plays to play and choose the cards with the lowest rank in hands to discard.

3. Have extra commands. “change player(id)”, this can change player(id) from Computer to Human. “restart”, this restarts a new round of game but does not change the existing scores. Meanwhile, if there exist other players, other players will be asked to agree or not. If only over half (half will not be counted) players agree, the round can be restarted. All players can restart only once. If a player requires to restart 3 times or more, ai will play his cards once.

Final Questions

Question 1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

While writing large programs, a predesigned uml is significantly important. Determining design pattern should be the first step. Should I keep decorating the streaming class or control the model to carry out the objective. Then, applying the pattern, how many structures are needed? What are the structures representing and what roles are they playing? What relationships do they have to each other? Should there be child classes of for some special requirements? What structure is the subject that will be the system and how will the structure meet the needs?

Thinking of these questions and make the uml of all structures. The fields and member functions of the structures should be decided next. The member functions should have their specific and unique abilities. Their applied object should be determined. While thinking of this, should decide the functions of the subject and the main function. How would the main function loop to adapt the change in the subject? What functions the subject need to accept the commands and react to the changing states. What functions are needed to realize all required abilities? From the realization process, fill the member functions and fields of all structures. Will the functions change the fields or return a suitable need or displays the information? What parameters are needed? Should the parameters be reference or pointers or rvalues?

When all of these questions are answered, the uml of structures should have been completed. Now implement the main function. Then follow the main function, fulfill the details of each subject and other structures. All functions should have the expected behaviour like in the uml. If there are questions about improving the efficiencies or adding functions, can add notes to the uml. After the main function and all files are implemented, try compiling the program and start debugging. If the program is successfully compiled and the program has all required behaviours, save a copy of all files. Or, if the codes can’t compile after a bunch of changes, could check the notes and try to change the structures. After the success of compilation and running, can think of improving and adding features.

Overall, while writing a large program, the most important thing is the design. Everything I do should follow the design. Though changes and new ideas often occur while writing the codes. I should record all useful ideas and try to apply the ideas after the original design is completed or I noticed the original design is not realizable. I can add or delete some functions but I must figure out the roles of all functions and make sure all functions' objectives are unique and necessary.

Question 2. What would you have done differently if you had the chance to start over?

If I have a chance to start over, I would follow the MVC design pattern to design the classes. I would name the current Subject Model and make a View class containing Model. Then design the correct Subject with attach and detach and Observer.

In this way, I can enforce my understanding of design pattern especially the association between observer and subject which would contribute to future work.

Conclusion

Straights is a typical application of MVC. Most games are built in this model, but other patterns can be used to modify the model like decorator.

This project is a practice of my skills of C++ learned in CS246. This project should represent a speculum of my study.