МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ» ИНЖЕНЕРНО-ТЕХНОЛОГИЧЕСКАЯ АКАДЕМИЯ

Институт компьютерных технологий и информационной безопасности

Кафедра математического обеспечения и применения ЭВМ

Отчет по лабораторной работе №1

по курсу «Объектно ориентированное программирование»

«Классы и объекты в C++»

Выполнил:
студент гр. КТбо2-6
Калитин А.В.

Таганрог 2020

Оглавление

Вариант № 15

ТЕХНИЧЕСКОЕ ЗАДАНИЕ

Определить классы Карта (Card) и Колода_карт (Cardatch). Поля первого – масть (suit) и достоинство (rank). Методы класса возвращают масть и достоинство. Второй включает массив (32) объектов первого.

Методы класса:

− перемешивания колоды;

− сравнения 2-х карт по достоинству при условии, что масти одинаковы;

− создания 4-х мест и раздачи равного количества карт;

−моделирования упрощенного розыгрыша взятки: на стол выкладываются по одной карте от каждого из 4-х игроков; первая выложенная карта определяет масть; выигрывает карта, старшая по достоинству (картинки старше простых карт; козырной масти нет).

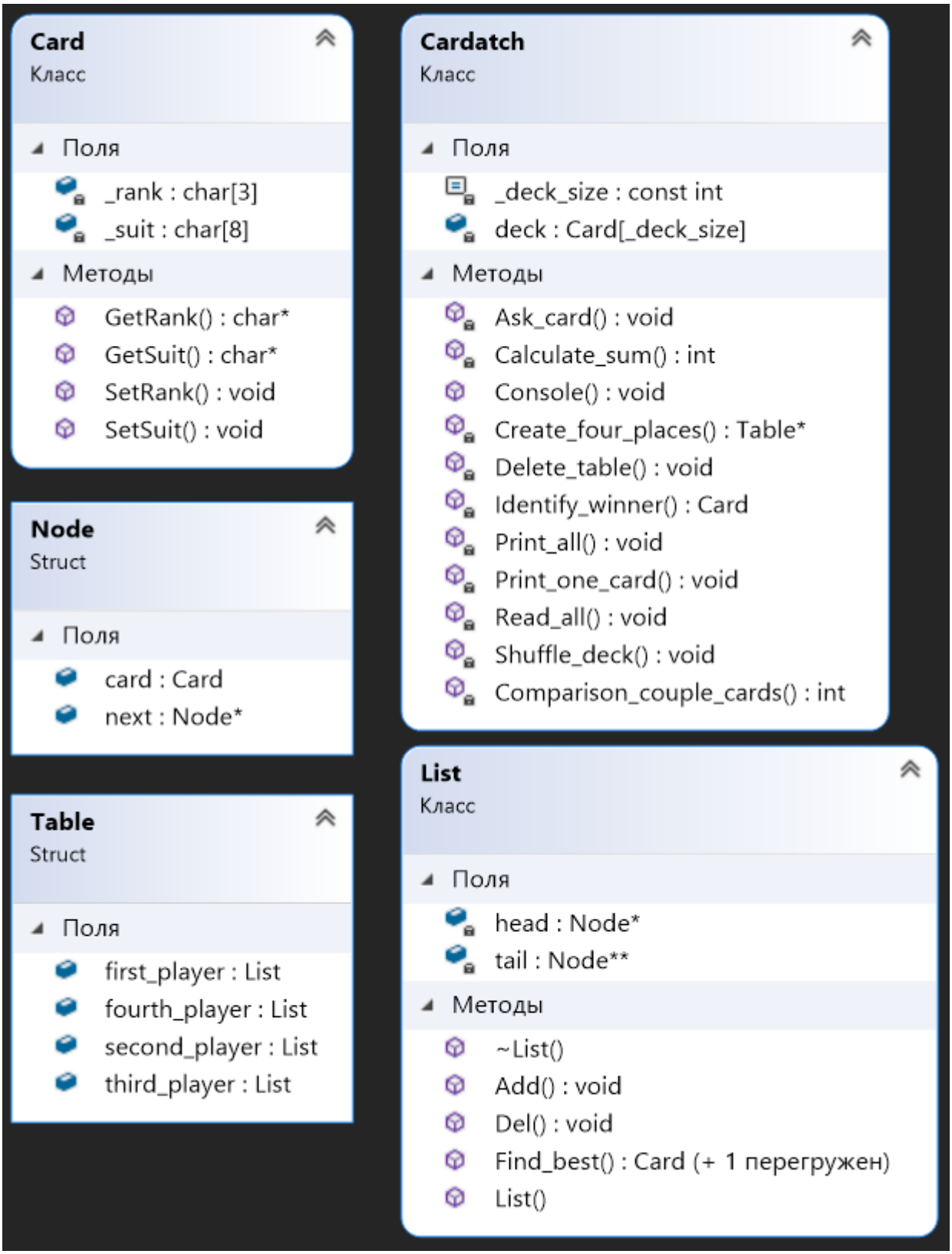
Список карт для инициализации программы хранить в файле.

ВЫПОЛНЕНИЕ ЗАДАНИЯ

1. Спецификация классов.

Класс Cardatch содержит один публичный метод для консольного взаимодействия с объектом. В приватной секции имеется массив из объектов класса Card, пара методов для ввода и вывода карт в консоль, метод для сравнения двух карт, метод для перемешивания карт, метод для симуляции игры четырех человек, вспомогательные методы. Класс Card имеет два приватных поля для ранга и масти карты, а также сеттеры и геттеры. Структуры node, list, table нужны для реализации игры на 4. Идея заключается в том что каждый игрок, описанный в структуре table, включает в себя односвязный список, который хранит и осуществляет поиск карт.

2. Диаграмма классов.



3. Листинг

1.Main.cpp

```
#define _CRTDBG_MAP_ALLOC
#define _CRT_SECURE_NO_WARNINGS
#include <stdlib.h>
#include <crtdbg.h>
#include <iostream>
#include "Cardatch.h"

int main()
{
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
    Cardatch deck;
    deck.Console();
}
```

2. Cardatch.h

```
#pragma once
#include <iostream>
#include <ctime>
#include <exception>
#include "Card.h"
#include "Stack.h"

struct Table
{
 List first_player;
 List second_player;
 List third_player;
 List fourth_player;
};

class Cardatch
{
public:
 void Console();
private:
 const static int _deck_size = 32;
 Card deck[_deck_size];
 void Delete_table(Table* table) const;
 void Print_one_card(Card card) const;
 void Ask_card(char* rank, char* suit, Card* new_card) const;
 //Return 1 - first card win, 2 - second win, 3 - diffrent suits
 int Comparison_couple_cards() const;
 void Read_all();
 void Print_all();
 void Shuffle_deck();
 Card Identify_winner(const int how, Table* table, const Card* all_cards) const;
 Table* Create_four_places(const int how_much_cards, const Card* all_cards) const;
 int Calculate_sum(char* rank) const;
};
```

3.Cardatch.cpp

```
48:  #define _CRT_SECURE_NO_WARNINGS
49:
50:  #include "Cardatch.h"
51:
52:  void Cardatch::Console()
53:  {
54:         try
     //just check file
55:         {
     //just check file
56:                 Read_all();                                                    //just
     check file
57:         }
     //just check file
58:         catch (std::exception& e)                                  //just check file
59:         {
     //just check file
60:                 std::cout << e.what();                                         //just check
     file
61:         }
     //just check file
62:         int command = 1;
63:         std::cout << "1 - Shuffle\n";
64:         std::cout << "2 - Comparison couple cards\n";
65:         std::cout << "3 - Simulation game\n";
66:         std::cout << "4 - Print all cards\n";
67:         std::cout << "0 - Exit.\n";
68:         while (command)
69:         {
70:                 std::cout << "Enter a command: ";
71:                 std::cin >> command;
72:
73:                 switch (command)
74:                 {
75:                 case 0:
76:                         break;
77:                 case 1:
78:                         Shuffle_deck();
79:                         std::cout << "The deck is shuffled" << std::endl;
80:                         break;
81:                 case 2:
82:                 {
83:                         std::cout << "Type two cards: rank and suit (Rank max 2 symbols and
     capital letter, Suit max 7 symbols and lowercase letter):\n";
84:                         int out = Comparison_couple_cards();
85:                         if (out == 1)
86:                         {
87:                                 std::cout << "First player WIN!\n";
88:                         }
89:                         if (out == 2)
90:                         {
91:                                 std::cout << "Second player WIN!\n";
92:                         }
93:                         if (out == 3)
94:                         {
95:                                 std::cout << "Different suits\n";
96:                         }
97:                         break;
98:                 }
99:                 case 3:
100:                        std::fprintf(stdout, "How much cards for one player? (1-8)\n");
101:                        int how;
102:                        std::cin >> how;
103:                        if (how >= 1 && how <= 8)
104:                        {
105:                                std::fprintf(stdout, "1 - automode\n2 - handmode\n");
```

```
106:                                int as;
107:                                std::cin >> as;
108:                                Card* all_cards = new Card[how * 4];
109:                                if (as == 2)
110:                                {
111:                                       char rank[3], suit[8];
112:                                       for (int i = 0; i < how * 4; i++)
113:                                       {
114:                                              Ask_card(rank, suit, &all_cards[i]);
115:                                       }
116:                                }
117:                                if (as == 1)
118:                                {
119:                                       for (int i = 0; i < how * 4; i++)
120:                                       {
121:                                              all_cards[i] = deck[i];
122:                                       }
123:                                }
124:                                Table* table = Create_four_places(how, all_cards);
125:                                Card max = Identify_winner(how, table, all_cards);
126:                                Print_one_card(max);
127:                                //int* a = new int[222]; //тест на утечку памяти
128:                                delete[] all_cards;
129:                                Delete_table(table);
130:                         }
131:                         else
132:                         {
133:                                std::cout << "Impossible\n";
134:                         }
135:                         break;
136:                  case 4:
137:                         Print_all();
138:                         break;
139:                  default:
140:                         std::cout << "Invalid command. Try again.\n";
141:                         break;
142:                  }
143:           }
144:    }
145:
146:    const static int _deck_size = 32;
147:
148:    Card deck[_deck_size];
149:
150:    void Cardatch::Delete_table(Table* table) const
151:    {
152:           table->first_player.~List();
153:           table->second_player.~List();
154:           table->third_player.~List();
155:           table->fourth_player.~List();
156:           delete table;
157:    }
158:
159:    void Cardatch::Print_one_card(Card card) const
160:    {
161:           std::fprintf(stdout, "|------------|\n|%2s          |\n|            |\n
       |\n|  %8s  |\n|            | <----WIN\n|            |\n|            |\n|------------
       |\n\n", card.GetRank(), card.GetSuit());
162:    }
163:
164:    void Cardatch::Ask_card(char* rank, char* suit, Card* new_card) const
165:    {
166:           std::cout << "Type card: rank and suit (Rank max 2 symbols and capital letter, Suit
       max 7 symbols and lowercase letter):\n";
167:           std::cin >> rank >> suit;
168:           while (Calculate_sum(rank) == 0)
169:           {
```

```
170:                std::cout << "Incorrect data. Please try again.\n";
171:                std::cout << "Type card: rank and suit (Rank max 2 symbols and capital
    letter, Suit max 7 symbols and lowercase letter):\n";
172:                std::cin >> rank >> suit;
173:        }
174:        new_card->SetRank(rank);
175:        new_card->SetSuit(suit);
176: }
177: //Return 1 - first card win, 2 - second win, 3 - diffrent suits
178: int Cardatch::Comparison_couple_cards() const
179: {
180:        //Card first = deck[rand() % 32];                          //automode
181:        //Card second = deck[rand() % 32];                         //automode
182:        Card first, second;
183:        char rank[3], suit[8];
184:        Ask_card(rank, suit, &first);
185:        Ask_card(rank, suit, &second);
186:        if (strcmp(first.GetSuit(), second.GetSuit()) == 0)
187:        {
188:               if (Calculate_sum(first.GetRank()) < Calculate_sum(second.GetRank()))
189:               {
190:                      return 1;
191:               }
192:               else
193:               {
194:                      return 2;
195:               }
196:        }
197:        else
198:        {
199:               return 3;
200:        }
201: }
202:
203: void Cardatch::Read_all()
204: {
205:        const char* path = "OriginDeck.txt";
206:        char rank[3], suit[8];
207:        FILE* Input;
208:        if ((Input = fopen(path, "r")) == NULL)
209:        {
210:               throw "File not found.";
211:        }
212:        else
213:        {
214:               for (int i = 0; i < _deck_size; i++)
215:               {
216:                      fscanf(Input, "%s %s\n", rank, suit);
217:                      deck[i].SetRank(rank);
218:                      deck[i].SetSuit(suit);
219:               }
220:               fclose(Input);
221:        }
222: }
223:
224: void Cardatch::Print_all()
225: {
226:        for (int i = 0; i < _deck_size; i++)
227:        {
228:               std::fprintf(stdout, "|------------|\n|%2s          |\n|            |\n|
    |\n|  %8s  |\n|            |\n|            |\n|            |\n|------------|\n\n",
    deck[i].GetRank(), deck[i].GetSuit());
229:        }
230: }
231:
232: void Cardatch::Shuffle_deck()
233: {
```

```
234:         srand(time(0));
235:         int n = _deck_size;
236:         while (n > 1)
237:         {
238:                 int ind = rand() % _deck_size;
239:                 n--;
240:                 Card temp = deck[n];
241:                 deck[n] = deck[ind];
242:                 deck[ind] = temp;
243:         }
244: }
245:
246: Card Cardatch::Identify_winner(const int how, Table* table, const Card* all_cards) const
247: {
248:         Card temp = table->first_player.Find_best();
249:         char* suit = temp.GetSuit();
250:         Card four_best_cards[4] =
251:         {
252:                 table->first_player.Find_best(),
253:                 table->second_player.Find_best(suit),
254:                 table->third_player.Find_best(suit),
255:                 table->fourth_player.Find_best(suit)
256:         };
257:         suit = nullptr;
258:         delete suit;
259:         Card max = four_best_cards[0];
260:         for (int i = 1; i < 4; i++)
261:         {
262:                 if (Calculate_sum(max.GetRank()) <
    Calculate_sum(four_best_cards[i].GetRank()))
263:                 {
264:                         max = four_best_cards[i];
265:                 }
266:         }
267:         return max;
268: }
269:
270: Table* Cardatch::Create_four_places(const int how_much_cards, const Card* all_cards) const
271: {
272:         Table* table = new Table;
273:         for (int j = 0; j < how_much_cards; j += 4)
274:         {
275:                 table->first_player.Add(all_cards[j]);
276:                 table->second_player.Add(all_cards[j + 1]);
277:                 table->third_player.Add(all_cards[j + 2]);
278:                 table->fourth_player.Add(all_cards[j + 3]);
279:         }
280:         return table;
281: }
282:
283: int Cardatch::Calculate_sum(char* rank) const
284: {
285:         int sum = 0;
286:         switch (*rank)
287:         {
288:         case 54:
289:                 sum += 6;
290:                 break;          //seven = 7,
291:         case 55:                        //eight = 8,
292:                 sum += 7;               //nine = 9,
293:                 break;                  //ten = 10,
294:         case 56:                        //jack = 11,
295:                 sum += 8;               //queen = 12,
296:                 break;                  //king = 13,
297:         case 57:                        //ace = 14
298:                 sum += 9;
299:                 break;
```

```
300:          case 49:
301:                 sum += 10;
302:                 break;
303:          case 74:
304:                 sum += 11;
305:                 break;
306:          case 81:
307:                 sum += 12;
308:                 break;
309:          case 75:
310:                 sum += 13;
311:                 break;
312:          case 65:
313:                 sum += 14;
314:                 break;
315:          }
316:          return sum;
317:   }
```

4.Card.h

```
318:  #pragma once
319:  class Card
320:  {
321:  public:
322:   char* const GetSuit();
323:   char* const GetRank();
324:   void SetSuit(char suit[]);
325:   void SetRank(char rank[]);
326:  private:
327:   char _suit[8];
328:   char _rank[3];
329:  };
```

5.Card.cpp

```
330:  #define _CRT_SECURE_NO_WARNINGS
331:  #include <cstring>
332:  #include "Card.h"
333:
334:  char* const Card::GetSuit()
335:  {
336:   return _suit;
337:  }
338:  char* const Card::GetRank()
339:  {
340:   return _rank;
341:  }
342:  void Card::SetSuit(char suit[])
343:  {
344:   strcpy(_suit, suit);
345:  }
346:  void Card::SetRank(char rank[])
347:  {
348:   strcpy(_rank, rank);
349:  }
```

6.Stack.h

```
#pragma once
#include "Card.h"
#include <cstring>

//СТРУКТУРА БУДЕТ ЭЛЕМЕНТОМ СПИСКА

struct Node
{
    Card card;
    Node* next;
};

class List
{
    Node* head, * tail;
public:
    List() :head(NULL), tail(NULL) {};
    ~List();
    void Add(Card card);
    void Del();
    Card Find_best();
    Card Find_best(char suit[]);
};
```

7.Stack.cpp

```
#include "Stack.h"
List::~List() //ДЕСТРУКТОР ДЛЯ ОЧИСТКИ ПАМЯТИ
{
    Node* temp = head;                      //Временный указатель на начало списка
    while (temp != NULL)                    //Пока в списке что-то есть
    {
        temp = head->next;                  //Резерв адреса на следующий элемент списка
        delete head;                        //Освобождение памяти от первой структуры как элемента списка
        head = temp;                        //Сдвиг начала на следующий адрес, который берем из резерва
    }
}


//ФУНКЦИЯ ЗАПОЛНЕНИЯ ИНФОРМАЦИОННЫХ ПОЛЕЙ СТРУКТУРЫ NODE И ДОБАВЛЕНИЯ ЭТОЙ СТРУКТУРЫ В СПИСОК

void List::Add(Card card)
{
    Node* temp = new Node;              //Выделение памяти для нового звена списка
    temp->card = card;                      //Временное запоминание принятого параметра
    temp->next = NULL;                  //Указание, что следующее звено новосозданной структуры пока пустое
    if (head != NULL)                   //Если список не пуст
    {
        tail->next = temp;              //Указание, что следующее звено списка это новосозданная структура
        tail = temp;
    }
    else
    {
        head = tail = temp;      //Если список не пуст, добавление первого элемента
    }
}

//ФУНКЦИЯ ИЗЪЯТИЯ ЭЛЕМЕНТА ИЗ ОЧЕРЕДИ
void List::Del()
{
    if (head != NULL)
    {
        Node* temp = head;              //Обращаемся к началу списка с помощью вспомогательного указателя
        head = head->next;              //Сдвиг начала списка
        delete temp;                    //Освобождение памяти от предыдущего звена списка
    }
}

Card List::Find_best()
{
    Card max;
    max = head->card;
    Node* temp = head;                  //Временный указатель на начало списка
    while (temp != NULL)                //Пока в списке что-то встречается
    {
        if (temp->card.GetRank() > max.GetRank())
        {
            max = temp->card;
        }
        temp = temp->next;             //Сдвигаем указатель на начало на адрес следующего элемента
    }
    return max;
}

Card List::Find_best(char suit[])
```

```
{
    Card max;
    bool flag = 0;
    Node* temp = head;
    while (temp != NULL)
    {
        if (strcmp(suit, temp->card.GetSuit()) == 0)
        {
            flag = 1;
            max = temp->card;
        }
        temp = temp->next;
    }
    if (!flag)
    {
        char a[3] = "0";
        char b[8] = "no suit";
        max.SetRank(a);
        max.SetSuit(b);
    }
    return max;
}
```