Федеральное агентство связи (Россвязь)

Федеральное государственное бюджетное образовательное учреждение

высшего образования «Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Курсовая работа

«Объектно-ориентированное программирование»

«Blackjack»

Выполнил:

студент группы ИП-312

Дорогин Н.С

Проверил:

Старший преподаватель кафедры прикладной математики и кибернетики

Дементьева К.И.

Новосибирск 2024

Оглавление:

Задание на курсовую………………………………………………………….2

Содержание работы…………………………………………………………...2

Описание иерархии объектов………………………………………………...3

Наследование………………………………………………………………….3

Классы и методы……………………………………………………………...3

Доступность компонентов классов………………………………………….5

Конструкторы и деструкторы………………………………………………..5

Перегрузка операторов……………………………………………………….6

Описание алгоритма основной программы…………………………………7

Скриншоты…………………………………………………………………….8

Вывод………………………………………………………………………….13

Код программы………………………………………………………………..15

Литература…………………………………………………………………….24

**Задание на курсовую**

С использованием технологий объектно-ориентированного программирования программно реализовать упрощённую версию азартной карточной игры Blackjack.

Суть игры:

Игрокам раздаются карты, за каждую из которых начисляются очки в соответствии с указанным номиналом. Валет, дама, король дают 10 очков, а туз, в зависимости от того, что больше подходит, даёт 1 или 11 очков. Каждый игрок пытается заработать 21 очко и не больше.

Роль дилера выполняет компьютер. Помимо него в игре может присутствовать от 1 до 7 игроков.

В начале игры все участники получают по 2 карты. Все карты видны, кроме первой карты дилера.

Далее все игроки по очереди получают возможность брать карты из колоды пока не надоест или пока не случится перебор. Когда наступает очередь дилера, его первая карта открывается.

Если у дилера перебор, все игроки без перебора побеждают. В противном случае сумма очков каждого игрока сравнивается с суммой очков дилера. Если сумма больше и нет перебора, игрок выигрывает. В противном случае игрок проигрывает. Если суммы очков равны случается пуш – ничья.

**Содержание работы:**

* Виртуальные методы
* Инкапсуляция (все поля данных не доступны из внешних функций)
* Наследование (7 классов, один из которых - абстрактный)
* Полиморфизм
* Конструкторы, Списки инициализации
* Перегрузка операторов

**Описание иерархии объектов.**  
  
В данной курсовой работе имеется 7 классов, один из которых абстрактный.

Класс *Card* представляет собой реальные карты, не наследует ни от кого. При раздаче карт мы их не копируем, а перемещаем, следовательно в классе *Hand*, представляющим собой руку с картами будет иметь место в качестве члена данных вектор, где будут храниться указатели на объекты типа *Card*. Когда карта раздаётся, указатели будут копироваться и удаляться.

Игроки и дилер в рамках данной курсовой работы представляют собой ничто иное, как руки, имеющие имена, поэтому классы *Player* и *House* являются наследниками класса *Hand*.

*GenericPlayer* определяется как абстрактный класс, в котором размещается общая функциональность классов *Player* и *House*, чтобы не дублировать её в обоих классах.

Так же колода карт рассматривается отдельно от дилера и представлена в отдельном классе *Deck*, который с помощью метода, отвечающего за раздачу карт, будет на равных условиях выдавать карты игрокам и дилеру. Этот метод будет полиморфным – он будет работать как для объектов типа *Player*, так и для объекта *House*.

И наконец, сама игра будет представлена отдельным классом *Game*. Так же, как и *Card* игра ни от кого не наследует.

**Наследование:**

Классы *Card, Game* и *Hand* ни от кого не наследуют.

Классы *Deck* и *GenericPlayer* наследуют от Hand.

Классы *Player* и *House* наследуют от класса *GenericPlayer*.

**Классы и методы:**

***Card:***

*rank m\_Rank* – поле, значение карты (туз, два, три и т.д), rank – enum(перечисление) куда входят все 13 значений.

*suit m\_Suit* – поле, значение масти (крести, бубны, черви, пики), suit – enum(перечисление) куда входят все 4 масти.

*bool m\_IsFaceUp* – поле, логическое значение, указывающее, лежит ли карта лицом вверх или вниз.

*int getValue()* – метод возвращающий количество очков за карту.

*void Flip()* – метод, переворачивающий карту.

***Hand:***

*vector<Card\*> m\_Cards* – поле, вектор, хранящий указатели на объекты типа *Card* - карты в руке.

*void Add(Card\* pCard)* – метод, добавляющий карту в руку, то есть указатель в вектор.

*void Clear()* – метод, очищающий руку от карт, удаляя все указатели из вектора.

*int getTotal()* – метод, возвращающий сумму очков карт руки.

***GenericPlayer(абстрактный):***

*string m\_Name* – поле, имя игрока.

*virtual bool IsHitting() const = 0* – виртуальный метод, указывающий, нужна ли игроку новая карта.

*bool IsBusted() const* – метод, указывающий что у игрока перебор.

*void Bust() const* – метод, объявляющий перебор у игрока.

***Player:***

*virtual bool IsHitting() const* – метод, указывающий нужна ли игроку ещё одна карта.

*void Win() const* – метод, объявляющий победу игрока.

*void Lose() const* – метод, объявляющий поражение игрока.

*void Push() const* – метод, объявляющий что игрок сыграл вничью.

***House:***

*virtual bool IsHitting() const* – метод, указывающий, нужна ли игроку ещё одна карта.

*void FlipFirstCard()* – метод, переворачивающий первую карту дилера.

***Deck:***

*void Populate()* – метод, создающий колоду из 52 карт.

*void Shuffle()* – метод, тасующий карты в колоде.

*void Deal(Hand& aHand)* – метод, выдающий карту из колоды в руку.

*void AdditionalCards(GenericPlayer& aGenericPlayer)* – метод, выдающий игроку карты до тех пор, пока тот не откажется брать новую или у него не случится перебор.

***Game:***

*Deck m\_Deck* – поле, объект класса *Deck*, колода карт.

*House m\_House* – поле, объект класса House, дилер (или рука дилера).

*vector<Player> m\_Players* – поле, вектор состоящий из объектов класса Player, игроки.

*void Play()* – метод, проводящий кон игры Blackjack.

**Доступность компонентов класса:**

Класс *Card* имеет спецификатор доступа *public* для своих методов и двух перечислений и *private* для своих полей.

Класс *Hand* имеет спецификатор доступа *public* для своих методов и *protected* для своего единственного поля.

Абстрактный класс *GenericPlayer* имеет спецификатор доступа *public* для своих методов и *protected* для своего единственного поля.

Класс *Player* имеет спецификатор доступа *public* для своих методов, своих полей не имеет.

Класс *House* имеет спецификатор доступа *public* для своих методов, своих полей не имеет.

Класс *Deck* имеет спецификатор доступа *public* для своих методов, своих полей не имеет.

Класс *Game* имеет спецификатор доступа *public* для своих методов и *private* для своих полей.

**Конструкторы и деструкторы:**

Каждый класс в программе имеет по одному конструктору и деструктору.

*Card(rank r = ACE, suit s = SPADES, bool ifu = true);* - конструктор карты, принимает на вход номинал, масть и повёрнута ли карта лицом и записывает эти значения в поля объекта класса через список инициализации.

Деструктор карты по умолчанию.

*Hand();* - конструктор руки, не принимает ничего на вход, резервирует 7 мест для вектора из карт.

*virtual ~Hand();* - деструктор руки, применяет метод *Clear()*, очищая руку от карт и освобождая динамическую память.

*GenericPlayer(const string &name = "");* - конструктор условного игрока, принимает на вход имя и записывает его в поле объекта класса через список инициализации.

*virtual ~GenericPlayer();* - деструктор условного игрока, деструктор по умолчанию.

*Player(const string &name = "");* - конструктор игрока, принимает на вход имя и подаёт его на вход конструктору условного игрока.

*virtual ~Player();* - деструктор игрока, деструктор по умолчанию.

*House(const string &name = "House");* - конструктор дилера, принимает на вход имя и подаёт его на вход конструктору условного игрока.

*virtual ~House();-* деструктор дилера, деструктор по умолчанию.

*Deck(); -* конструктор колоды, очищает колоду, вызывая метод *Clear()*, затем резервирует место в векторе на 52 карты, а после в цикле заполняет колоду методом *Populate()*,а тот,в свою очередь,через *Add().*

*virtual ~Deck();* - деструктор колоды, деструктор по умолчанию.

*Game(const vector<string> &names);* - конструктор игры, принимает на вход вектор из имён, которые присваивает игрокам, также вызывает у колоды методы заполнения и перемешки.

*~Game();-* деструктор игры, деструктор по умолчанию.

**Перегрузка операторов:**

*ostream &operator<<(ostream &os, const Card &aCard);* - перегрузка оператора вывода для класса *Card*, выводит номинал и масть карты, или “XX”, если карта лежит лицом вниз.

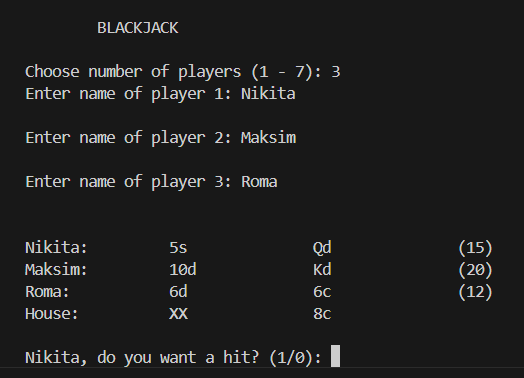
*ostream &operator<<(ostream &os, const GenericPlayer &aGenericPlayer);* - перегрузка оператора вывода для условного игрока, выводит имя, все карты игрока (с использованием прошлой перегрузки), а также сумму очков, если у игрока есть карты.

**Описание алгоритма основной программы**

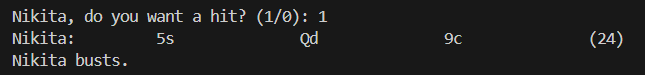
Функция *main()* получает имена всех игроков и помещает их в вектор строк, а затем создаёт объект класса Game и передаёт в него ссылку на вектор. Функция *main()* вызывает метод *Play()* объекта класса Game до тех пор, пока игроки не покажут, что не хотят играть.

**Скриншоты**

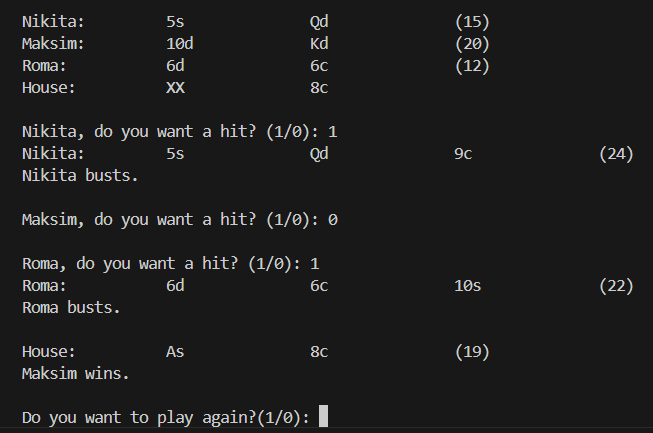
**Без графики:**



Скриншот 1. Инициализация игроков, вывод игроков, их карт и очков.

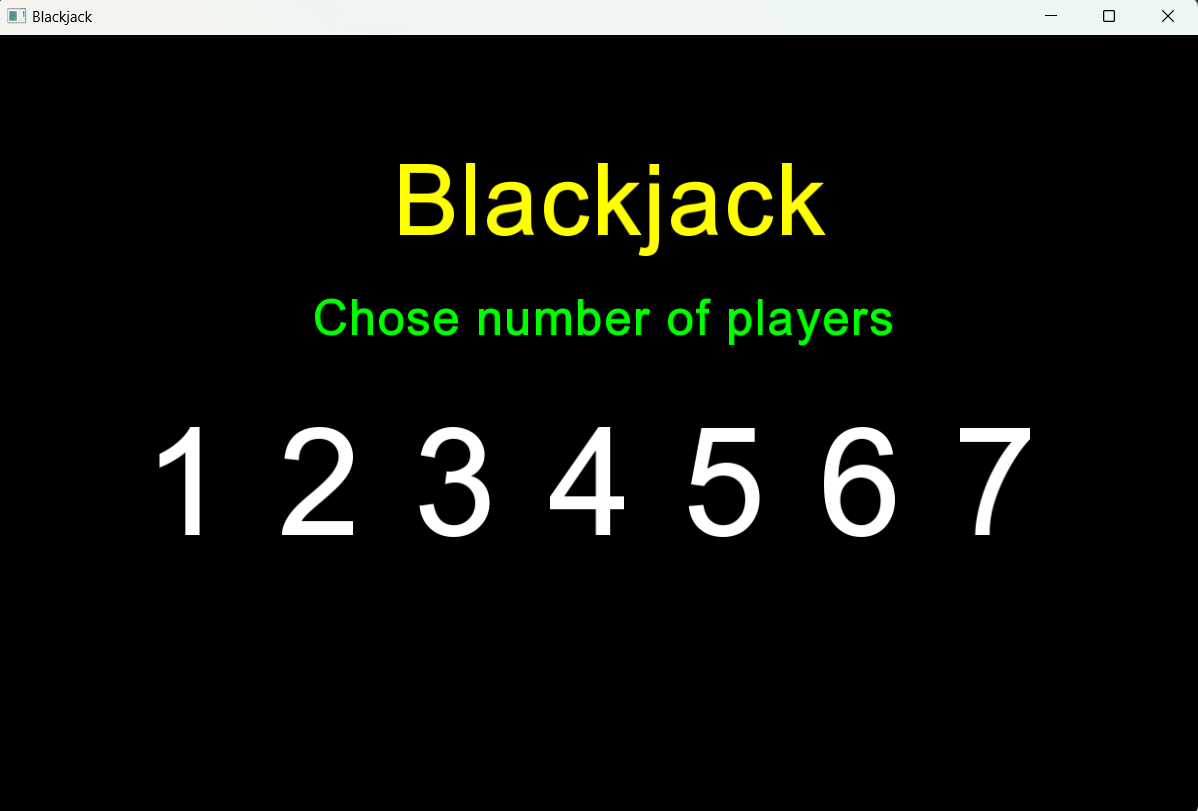


Скриншот 2. Первый игрок берёт карту и у него случается перебор.

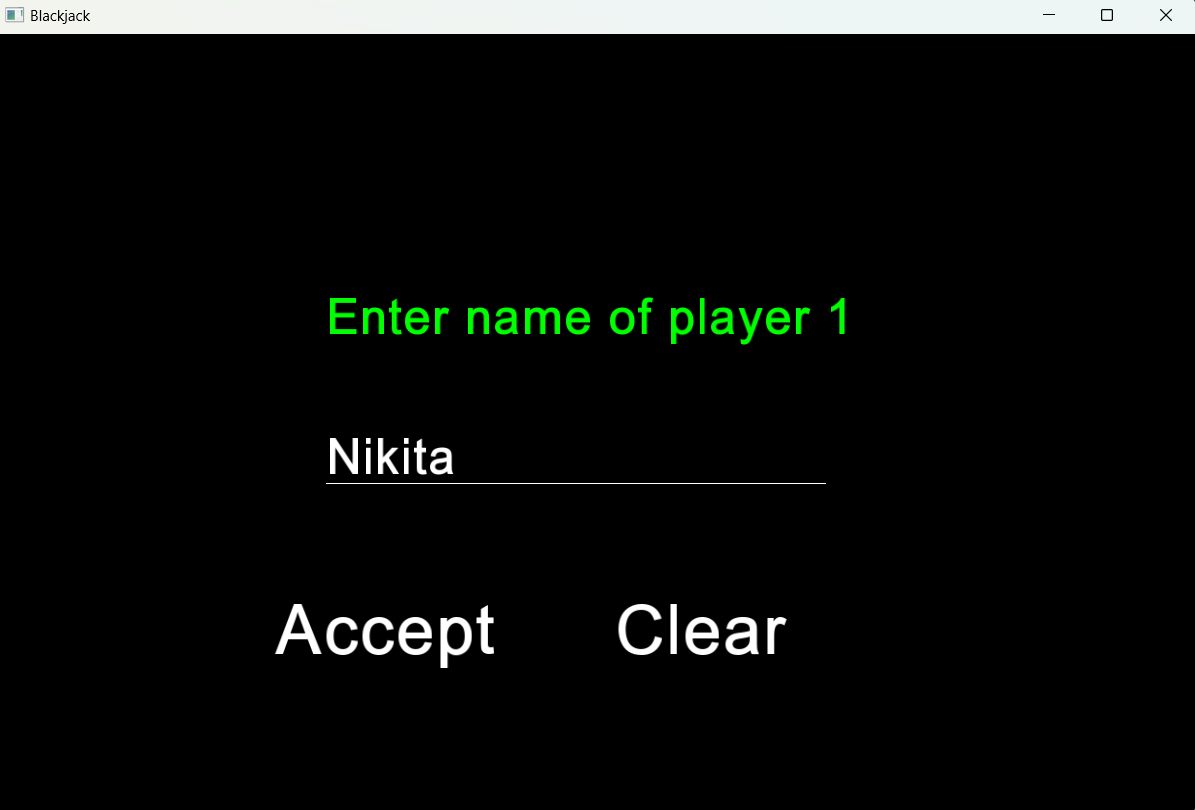


Скриншот 3. Второй игрок не берёт карт, так как уже имеет сумму близкую к 21, третий игрок берёт карту и у него случается перебор, дилер набирает меньше очков, чем второй игрок, второй игрок выигрывает.

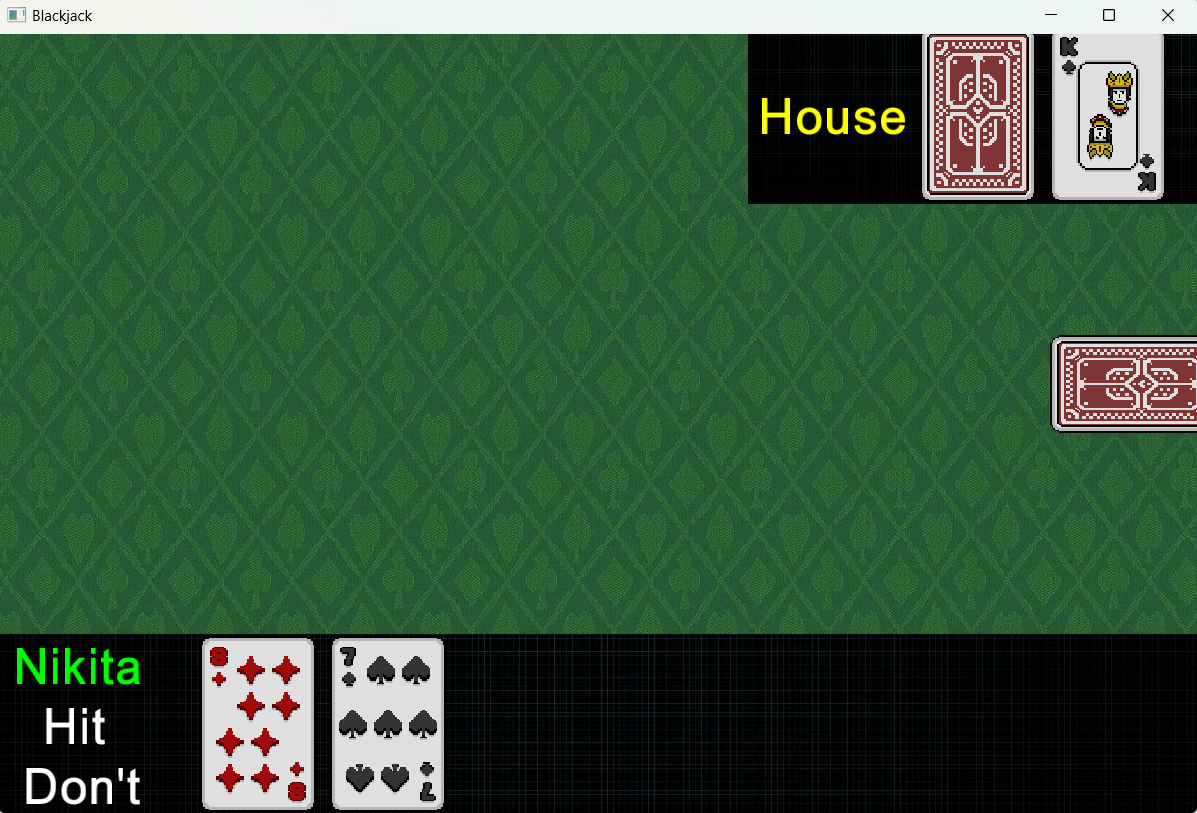
**С графикой:**



Скриншот 4. Выбор количества игроков.



Скриншот 5. Запись имён игроков.



Скриншот 6. Первому игроку предлагается взять карту (Hit) или не брать (Don’t).



Скриншот 7. Первый игрок взял карту и получил перебор. Очередь второго игрока.



Скриншот 8. Второй игрок берёт карту и получает перебор. Очередь третьего игрока.



Скриншот 9. Третий игрок берёт карту и получает в сумме 21 очко.



Скриншот 10. Третий игрок отказывается брать ещё одну карту, наступает очередь дилера, дилер не берёт карту и сумма его очков меньше, чем у третьего игрока. Третий игрок побеждает.

# Вывод

Данная курсовая работы была разработана с использованием основных принципов ООП:

Инкапсуляция, наследование, полиморфизм, также использовались конструкторы, инициализация объектов, перегрузка операторов, виртуальные функции и передача объектов в качестве параметров функций и методов.

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования. Благодаря ООП, мы можем быстро написать программу с таким кодом, который будет понятен остальным.

ООП даёт возможность делить программу на кучу «модулей»-классов, каждый из которых делает свою часть работы. Все объекты, взаимодействуя между собой, образуют работу нашей программы.

Кроме того, написанный нами код можно повторно использовать в другом месте программы, что также экономит большое количество времени.

Мы научились работать с основными принципами ООП: наследование, полиморфизм и инкапсуляция.

Состояние будущих объектов описывается в классе с помощью полей, а их поведение – с помощью методов. Возможность же изменения состояния и поведения осуществляется с помощью модификаторов доступа к полям и методам – private, protected, public.

Для работы с нашим объектом мы оставляем открытыми методы с помощью модификатора public.

Предоставление открытых методов для работы с объектом также является частью механизма инкапсуляции, тат как если полностью закрыть доступ к объекту – он станет бесполезным.

Наследование заключается в использовании уже существующих классов для описания новых.

Принцип в ООП, когда программа может использовать объекты с одинаковым интерфейсом без информации о внутреннем устройстве объекта, называется полиморфизмом.

**Код программы (без графики)**  
#include <iostream>

#include <string>

#include <vector>

#include <algorithm>

#include <ctime>

using namespace std;

class Card

{

public:

enum rank

{

ACE = 1,

TWO,

THREE,

FOUR,

FIVE,

SIX,

SEVEN,

EIGHT,

NINE,

TEN,

JACK,

QUEEN,

KING

};

enum suit

{

CLUBS,

DIAMONDS,

HEARTS,

SPADES

};

friend ostream &operator<<(ostream &os, const Card &aCard);

Card &operator=(const Card &other);

Card(rank r = ACE, suit s = SPADES, bool ifu = true);

int getValue() const;

int getRank() const;

int getSuit() const;

bool getFace() const;

void Flip();

private:

rank m\_Rank;

suit m\_Suit;

bool m\_IsFaceUp;

};

Card::Card(rank r, suit s, bool ifu) : m\_Rank(r), m\_Suit(s), m\_IsFaceUp(ifu) {}

int Card::getRank() const

{

return m\_Rank;

}

int Card::getSuit() const

{

return m\_Suit;

}

bool Card::getFace() const

{

return m\_IsFaceUp;

}

int Card::getValue() const

{

int value = 0;

if (m\_IsFaceUp)

{

value = m\_Rank;

if (value > 10)

{

value = 10;

}

}

return value;

}

void Card::Flip()

{

m\_IsFaceUp = !(m\_IsFaceUp);

}

//------------------------------------------------------------------------------------------------------------

class Hand

{

public:

Hand();

virtual ~Hand();

void Add(Card \*pCard);

void Clear();

int getTotal() const;

protected:

vector<Card \*> m\_Cards;

};

Hand::Hand()

{

m\_Cards.reserve(7);

}

Hand::~Hand()

{

Clear();

}

void Hand::Add(Card \*pCard)

{

m\_Cards.push\_back(pCard);

}

void Hand::Clear()

{

vector<Card \*>::iterator iter = m\_Cards.begin();

for (iter = m\_Cards.begin(); iter != m\_Cards.end(); iter++)

{

delete \*iter;

\*iter = 0;

}

m\_Cards.clear();

}

int Hand::getTotal() const

{

if (m\_Cards.empty())

{

return 0;

}

if (m\_Cards[0]->getValue() == 0)

{

return 0;

}

int total = 0;

vector<Card \*>::const\_iterator iter;

for (iter = m\_Cards.begin(); iter != m\_Cards.end(); iter++)

{

total += (\*iter)->getValue();

}

bool containAce = false;

for (iter = m\_Cards.begin(); iter != m\_Cards.end(); iter++)

{

if ((\*iter)->getValue() == Card::ACE)

{

containAce = true;

}

}

if (containAce && total <= 11)

{

total += 10;

}

return total;

}

//----------------------------------------------------------------------------------------------------------------

class GenericPlayer : public Hand

{

friend ostream &operator<<(ostream &os, const GenericPlayer &aGenericPlayer);

public:

GenericPlayer(const string &name = "");

virtual ~GenericPlayer();

virtual bool IsHitting() const = 0;

bool IsBusted() const;

void Bust() const;

protected:

string m\_Name;

};

GenericPlayer::GenericPlayer(const string &name) : m\_Name(name) {}

GenericPlayer::~GenericPlayer() {}

bool GenericPlayer::IsBusted() const

{

return (getTotal() > 21);

}

void GenericPlayer::Bust() const

{

cout << m\_Name << " busts.\n";

}

//-------------------------------------------------------------------------------------------------------------------------------------

class Player : public GenericPlayer

{

public:

Player(const string &name = "");

virtual ~Player();

virtual bool IsHitting() const;

void Win() const;

void Lose() const;

void Push() const;

};

Player::Player(const string &name) : GenericPlayer(name) {}

Player::~Player() {}

bool Player::IsHitting() const

{

cout << m\_Name << ", do you want a hit? (1/0): ";

int response;

cin >> response;

return response;

}

void Player::Win() const

{

cout << m\_Name << " wins.\n";

}

void Player::Lose() const

{

cout << m\_Name << " loses.\n";

}

void Player::Push() const

{

cout << m\_Name << " pushes.\n";

}

//-----------------------------------------------------------------------------------------------------------------------------

class House : public GenericPlayer

{

public:

House(const string &name = "House");

virtual ~House();

virtual bool IsHitting() const;

void FlipFirstCard();

};

House::House(const string &name) : GenericPlayer(name) {}

House::~House() {}

bool House::IsHitting() const

{

return (getTotal() <= 16);

}

void House::FlipFirstCard()

{

if (!(m\_Cards.empty()))

{

m\_Cards[0]->Flip();

}

else

{

cout << "No card to flip!" << endl;

}

}

//-------------------------------------------------------------------------------------------------------------------------------------

class Deck : public Hand

{

public:

Deck();

virtual ~Deck();

void Populate();

void Shuffle();

void Deal(Hand &aHand);

void AdditionalCards(GenericPlayer &aGenericPlayer);

};

Deck::Deck()

{

m\_Cards.reserve(52);

Populate();

}

Deck::~Deck() {}

void Deck::Populate()

{

Clear();

for (int s = Card::CLUBS; s <= Card::SPADES; s++)

{

for (int r = Card::ACE; r <= Card::KING; r++)

{

Add(new Card(static\_cast<Card::rank>(r), static\_cast<Card::suit>(s)));

}

}

}

void Deck::Shuffle()

{

random\_shuffle(m\_Cards.begin(), m\_Cards.end());

}

void Deck::Deal(Hand &aHand)

{

if (!m\_Cards.empty())

{

aHand.Add(m\_Cards.back());

m\_Cards.pop\_back();

}

else

{

cout << "Out of cards. Unable to deal" << endl;

}

}

void Deck::AdditionalCards(GenericPlayer &aGenericPlayer)

{

cout << endl;

while (!(aGenericPlayer.IsBusted()) && aGenericPlayer.IsHitting())

{

Deal(aGenericPlayer);

cout << aGenericPlayer << endl;

if (aGenericPlayer.IsBusted())

{

aGenericPlayer.Bust();

}

}

}

class Game

{

public:

Game(const vector<string> &names);

~Game();

void Play();

private:

Deck m\_Deck;

House m\_House;

vector<Player> m\_Players;

};

Game::Game(const vector<string> &names)

{

vector<string>::const\_iterator pName;

for (pName = names.begin(); pName != names.end(); pName++)

{

m\_Players.push\_back(Player(\*pName));

}

srand(time(NULL));

m\_Deck.Populate();

m\_Deck.Shuffle();

}

Game::~Game() {}

void Game::Play()

{

vector<Player>::iterator pPlayer;

for (int i = 0; i < 2; i++)

{

for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); pPlayer++)

{

m\_Deck.Deal(\*pPlayer);

}

m\_Deck.Deal(m\_House);

}

m\_House.FlipFirstCard();

for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); pPlayer++)

{

cout << \*pPlayer << endl;

}

cout << m\_House << endl;

for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); pPlayer++)

{

m\_Deck.AdditionalCards(\*pPlayer);

}

m\_House.FlipFirstCard();

cout << endl;

cout << m\_House;

m\_Deck.AdditionalCards(m\_House);

if (m\_House.IsBusted())

{

for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); pPlayer++)

{

if (!(pPlayer->IsBusted()))

{

pPlayer->Win();

}

}

}

else

{

for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); pPlayer++)

{

if (!(pPlayer->IsBusted()))

{

if (pPlayer->getTotal() > m\_House.getTotal())

{

pPlayer->Win();

}

else if (pPlayer->getTotal() < m\_House.getTotal())

{

pPlayer->Lose();

}

else

{

pPlayer->Push();

}

}

}

}

for (pPlayer = m\_Players.begin(); pPlayer != m\_Players.end(); pPlayer++)

{

pPlayer->Clear();

}

m\_House.Clear();

}

ostream &operator<<(ostream &os, const Card &aCard);

ostream &operator<<(ostream &os, const GenericPlayer &aGenericPlayer);

int main()

{

cout << "\n\tBLACKJACK\n"

<< endl;

int numPlayers = 0;

while (numPlayers < 1 || numPlayers > 7)

{

cout << "Choose number of players (1 - 7): ";

cin >> numPlayers;

}

vector<string> names;

string name;

for (int i = 0; i < numPlayers; i++)

{

cout << "Enter name of player " << i + 1 << ": ";

cin >> name;

names.push\_back(name);

cout << endl;

}

cout << endl;

Game aGame(names);

int again = 1;

while (again)

{

aGame.Play();

cout << "\nDo you want to play again?(1/0): ";

cin >> again;

cout << endl;

}

return 0;

}

ostream &operator<<(ostream &os, const Card &aCard)

{

const string RANKS[] = {"0", "A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"};

const string SUITS[] = {"c", "d", "h", "s"};

if (aCard.m\_IsFaceUp)

{

os << RANKS[aCard.m\_Rank] << SUITS[aCard.m\_Suit];

}

else

{

os << "XX";

}

return os;

}

ostream &operator<<(ostream &os, const GenericPlayer &aGenericPlayer)

{

os << aGenericPlayer.m\_Name << ":\t\t";

vector<Card \*>::const\_iterator pCard;

if (!aGenericPlayer.m\_Cards.empty())

{

for (pCard = aGenericPlayer.m\_Cards.begin(); pCard != aGenericPlayer.m\_Cards.end(); pCard++)

{

os << \*(\*pCard) << "\t\t";

}

if (aGenericPlayer.getTotal() != 0)

{

cout << "(" << aGenericPlayer.getTotal() << ")";

}

}

else

{

os << "<empty>";

}

return os;

}

**Литература:**

1. Dawsom M. Beginning C++ through game programming. Indianapolis.: Wrox Publishing, 2016. 352 p.