ФТФ, 2-ый курс, ИВТ-Б(2), Королёв Алексей.

**Ссылка на GitHub:**

**ЛАБОРАТОРНАЯ РАБОТА №4**

**Тема:** Агрегация и композиция. Дружественные функции и классы. Исключения.

**Цель:** Научиться реализовывать на С++ межклассовое отношение агрегации / композиции, писать код генерации и обработки исключений, перегружать операторы с помощью дружественных функций.

**Индивидуальные задания:**

1. Написать простой консольный вариант карточной игры BlackJack для игры один-наодин с дилером, в соответствии с вариантом. В приложении А приведены вариации и особенности правил игры для каждого варианта. В ходе работы необходимо сделать как минимум следующее:

* создать и связать отношением агрегации/композиции и/или наследования классы КАРТА, КОЛОДА, ДИЛЕР, ИГРОК, ИГРА. В целом, Вы можете предлагать здесь свои варианты объектно-ориентированного проектирования;
* в начале игры генерировать случайным образом 4 колоды с 36 или 52 картами, в зависимости от варианта;
* имитировать действия дилера, в соответствии с вариантом игры;
* запрограммировать обработку всех потенциально возможных вариантов исхода: блек-джек, перебор, ровно, выигрыш по очкам, проигрыш по очкам;
* бросать и отлавливать исключение при «переборе» («перебор» рассматривать как исключительную ситуацию);
* перегрузить операцию потокового вывода объекта класса карты на экран с помощью дружественной функции. Выводить карту в виде 2♠, Q♦ и т.д. (символы карточных мастей имеют ASCII-коды 3, 4, 5, 6 и UNICODE-коды "\u2665", "\u2666", "\u2663", "\u2660", соответственно).

*Вариант 5. 17 + 4, базовый, Сплит*

1. Реализуйте паттерн GOF «Адаптер» в обоих видах (адаптер класса и адаптер объекта) на следующем примере. Напишите интерфейс IFormattable с методом std::string format() для форматирования объекта, реализующего данный интерфейс, а также функцию void prettyPrint(const IFormattable& object), которая выводит на экран объект-параметр в отформатированном виде. Сделайте так, чтобы в эту функцию можно было передавать ссылку на Вашу колоду карт, разработанную в задании 1. Алгоритм форматирования придумайте произвольный сами.

**Ход работы:**

1. Код программы:

#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

#include <random>

#include <stdexcept>

#include <iomanip>

#include <memory>

#include <map>

#include <windows.h>

#include <fcntl.h>

#include <corecrt\_io.h>

// Пользовательское исключение для ситуации превышения максимальной суммы очков

// Наследуется от стандартного runtime\_error для более информативной обработки ошибок

class OverflowException : public std::runtime\_error {

public:

// Конструктор передает сообщение об ошибке в базовый класс исключений

OverflowException(const std::string& message) : std::runtime\_error(message) {}

};

// Перечисление для масти карт с Unicode-символами

// Используется для типизации и красивого отображения карт

enum class Suit {

Hearts, // ♥ Красные сердца

Diamonds, // ♦ Красные бубны

Clubs, // ♣ Черные трефы

Spades // ♠ Черные пики

};

// Перечисление для номинала карт

// Позволяет преобразовывать номинал в числовое значение

enum class Rank {

Two = 2, // Двойка

Three, // Тройка

Four, // Четверка

Five, // Пятерка

Six, // Шестерка

Seven, // Семерка

Eight, // Восьмерка

Nine, // Девятка

Ten, // Десятка

Jack, // Валет

Queen, // Дама

King, // Король

Ace // Туз

};

// Класс карты - основной элемент игры

class Card {

private:

Rank rank; // Номинал карты

Suit suit; // Масть карты

bool isFaceUp; // Состояние карты (лицом вверх/вниз)

public:

// Конструктор для создания карты с заданным номиналом и мастью

Card(Rank r, Suit s) : rank(r), suit(s), isFaceUp(true) {}

// Определение очков для каждой карты по правилам BlackJack

int getValue() const {

switch (rank) {

case Rank::Jack: // Картинки стоят 10 очков

case Rank::Queen:

case Rank::King:

return 10;

case Rank::Ace: // Туз может стоить 11 или 1

return 11;

case Rank::Ten: // Десятка стоит 10

return 10;

default:

// Для остальных карт - их номинальное значение

return static\_cast<int>(rank);

}

}

// Дружественная функция для вывода карты в консоль

friend std::ostream& operator<<(std::ostream& os, const Card& card);

// Геттеры для получения информации о карте

Rank getRank() const { return rank; }

Suit getSuit() const { return suit; }

bool getFaceUp() const { return isFaceUp; }

// Метод для изменения видимости карты

void setFaceUp(bool up) { isFaceUp = up; }

};

// Перегрузка оператора вывода для красивой печати карты

std::ostream& operator<<(std::ostream& os, const Card& card) {

// Если карта закрыта, выводим заглушку

if (!card.getFaceUp()) {

os << "??";

return os;

}

// Словари для преобразования номинала и масти в строковое представление

std::map<Rank, std::string> rankNames = {

{Rank::Two, "2"}, {Rank::Three, "3"}, {Rank::Four, "4"},

{Rank::Five, "5"}, {Rank::Six, "6"}, {Rank::Seven, "7"},

{Rank::Eight, "8"}, {Rank::Nine, "9"}, {Rank::Ten, "10"},

{Rank::Jack, "J"}, {Rank::Queen, "Q"}, {Rank::King, "K"},

{Rank::Ace, "A"}

};

// Unicode-символы для мастей

std::map<Suit, std::string> suitSymbols = {

{Suit::Hearts, "\u2665"}, // ♥

{Suit::Diamonds, "\u2666"}, // ♦

{Suit::Clubs, "\u2663"}, // ♣

{Suit::Spades, "\u2660"} // ♠

};

// Вывод номинала и масти карты

os << rankNames[card.getRank()] << suitSymbols[card.getSuit()];

return os;

}

// Класс колоды для управления картами

class Deck {

protected:

std::vector<Card> cards; // Вектор для хранения карт

int numDecks; // Количество используемых колод

// Метод для создания и инициализации колоды

void createDecks() {

cards.clear(); // Очищаем текущую колоду

// Создание указанного числа колод

for (int d = 0; d < numDecks; ++d) {

// Перебираем все масти

for (int s = 3; s <= 6; ++s) {

// Перебираем все значения карт

for (int r = 2; r <= 14; ++r) {

cards.emplace\_back(

// Преобразуем значения для корректной игры

static\_cast<Rank>(r <= 10 ? r : (r == 14 ? 11 : 10)),

static\_cast<Suit>(s)

);

}

}

}

shuffle(); // Перемешиваем колоду после создания

}

public:

// Конструктор колоды с возможностью указать число колод

Deck(int decks = 4, bool is36 = false) : numDecks(decks) {

createDecks();

}

// Геттеры для получения информации о колоде

int getNumDecks() const { return numDecks; }

const std::vector<Card>& getAllCards() const { return cards; }

// Метод для случайного перемешивания карт

void shuffle() {

std::random\_device rd; // Генератор случайных чисел

std::mt19937 g(rd()); // Вихрь Мерсенна для качественной случайности

std::shuffle(cards.begin(), cards.end(), g); // Перемешивание

}

// Метод для взятия карты из колоды

Card drawCard() {

if (cards.empty()) {

createDecks(); // Если карты кончились, пересоздаем колоду

}

Card drawn = cards.back(); // Берем последнюю карту

cards.pop\_back(); // Удаляем ее из колоды

return drawn;

}

// Метод для подсчета оставшихся карт

int remainingCards() const { return cards.size(); }

};

// Абстрактный базовый класс для игроков

class Player {

protected:

std::vector<Card> hand; // Рука игрока (набор карт)

int score; // Текущее количество очков

bool isBust; // Признак превышения 21 очка

int balance; // Баланс игрока

int currentBet; // Текущая ставка

public:

// Конструктор с начальным балансом

Player(int initialBalance = 10000) :

score(0), isBust(false), balance(initialBalance), currentBet(0) {}

// Метод для установки ставки

void placeBet(int bet) {

if (bet > balance) {

throw std::runtime\_error("Недостаточно средств");

}

currentBet = bet;

balance -= bet;

}

// Метод для начисления выигрыша (удвоение ставки)

void win() {

balance += currentBet \* 2;

}

// Метод возврата ставки в случае ничьей

void push() {

balance += currentBet;

}

// Геттеры для баланса и ставки

int getBalance() const { return balance; }

int getCurrentBet() const { return currentBet; }

// Добавление карты в руку с пересчетом очков

void addCard(Card card) {

hand.push\_back(card);

updateScore();

}

// Виртуальный метод обновления очков с учетом правил BlackJack

virtual void updateScore() {

score = 0;

int aceCount = 0;

// Подсчет очков с особой логикой для тузов

for (const auto& card : hand) {

if (card.getRank() == Rank::Ace) {

aceCount++;

}

score += card.getValue();

}

// Корректировка очков, если сумма больше 21

while (score > 21 && aceCount > 0) {

score -= 10; // Туз может считаться как 1 вместо 11

aceCount--;

}

// Проверка на превышение 21 очка

if (score > 21) {

isBust = true;

throw OverflowException("Перебор!");

}

}

// Геттеры для получения информации о руке

int getScore() const { return score; }

bool getBust() const { return isBust; }

const std::vector<Card>& getHand() const { return hand; }

// Чисто виртуальный метод для логики взятия карт

virtual bool shouldTakeCard() = 0;

virtual ~Player() = default;

};

// Класс дилера с особой стратегией взятия карт

class Dealer : public Player {

public:

// Дилер берет карты, пока сумма меньше 17

bool shouldTakeCard() override {

return score < 17;

}

// Метод для сокрытия первой карты дилера

void showInitialHand() {

if (!hand.empty()) {

hand[0].setFaceUp(false); // Первая карта закрыта

}

}

};

// Класс игрока-человека с интерактивным выбором действий

class HumanPlayer : public Player {

public:

HumanPlayer(int initialBalance = 10000) : Player(initialBalance) {}

// Интерактивный метод выбора действия

bool shouldTakeCard() override {

std::cout << "\n1. Хватит\n";

std::cout << "2. Еще\n";

// Добавляем опцию сплита, если позволяет структура руки

if (canSplit()) {

std::cout << "3. Сплит?\n";

}

int choice;

std::cin >> choice;

switch (choice) {

case 1: return false; // Не брать карту

case 2: return true; // Взять карту

case 3:

if (canSplit()) {

// Логика сплита (пока не реализована)

std::cout << "Сплит пока не реализован.\n";

return false;

}

return false;

default: return false;

}

}

// Проверка возможности сплита

bool canSplit() const {

// Сплит возможен, если в руке 2 карты одного номинала

return (hand.size() == 2 &&

hand[0].getRank() == hand[1].getRank());

}

};

// Основной класс игры BlackJack

class BlackJackGame {

private:

Deck deck; // Колода карт

std::unique\_ptr<Dealer> dealer; // Указатель на дилера

std::unique\_ptr<HumanPlayer> player; // Указатель на игрока

// Метод вывода руки дилера с возможностью скрытия первой карты

void printDealerHand(bool hideFirst = true) {

std::cout << "Дилер: ";

const auto& dealerHand = dealer->getHand();

for (size\_t i = 0; i < dealerHand.size(); ++i) {

if (i == 0 && hideFirst) {

std::cout << "?? "; // Скрытая карта

}

else {

std::cout << dealerHand[i] << " ";

}

}

std::cout << std::endl;

}

// Метод вывода руки игрока

void printPlayerHand() {

std::cout << "Вы: ";

for (const auto& card : player->getHand()) {

std::cout << card << " ";

}

std::cout << std::endl;

}

public:

// Конструктор игры с инициализацией колоды и игроков

BlackJackGame() : deck(4) {

dealer = std::make\_unique<Dealer>();

player = std::make\_unique<HumanPlayer>();

}

// Основной метод игры

void play() {

try {

// Запрос и установка ставки

int bet;

std::cout << "Ваш баланс: " << player->getBalance() << std::endl;

std::cout << "Ваша ставка? ";

std::cin >> bet;

player->placeBet(bet);

// Начальная раздача карт

initialDeal();

printDealerHand();

printPlayerHand();

// Ход игрока

playerTurn();

// Ход дилера, если игрок не перебрал

if (!player->getBust()) {

dealerTurn();

determineWinner();

}

}

catch (const std::exception& e) {

std::cerr << "Ошибка: " << e.what() << std::endl;

}

}

void initialDeal() {

player->addCard(deck.drawCard());

dealer->addCard(deck.drawCard());

player->addCard(deck.drawCard());

dealer->addCard(deck.drawCard());

}

void playerTurn() {

try {

while (player->shouldTakeCard()) {

Card newCard = deck.drawCard();

player->addCard(newCard);

printDealerHand();

printPlayerHand();

}

}

catch (const OverflowException& e) {

std::cout << "Перебор! Ваши карты: ";

for (const auto& card : player->getHand()) {

std::cout << card << " ";

}

std::cout << "\nОбщая сумма: " << player->getScore() << std::endl;

}

}

void dealerTurn() {

try {

while (dealer->shouldTakeCard()) {

dealer->addCard(deck.drawCard());

}

}

catch (const OverflowException& e) {

std::cout << "Дилер перебрал!\n";

}

}

void determineWinner() {

int playerScore = player->getScore();

int dealerScore = dealer->getScore();

std::cout << "\n--- Результаты ---\n";

std::cout << "Ваши очки: " << playerScore << std::endl;

std::cout << "Очки дилера: " << dealerScore << std::endl;

if (player->getBust()) {

std::cout << "Вы проиграли (перебор).\n";

}

else if (dealer->getBust()) {

std::cout << "Вы победили (дилер перебрал)!\n";

player->win();

}

else if (playerScore > dealerScore) {

std::cout << "Вы победили!\n";

player->win();

}

else if (playerScore < dealerScore) {

std::cout << "Вы проиграли.\n";

}

else {

std::cout << "Ничья.\n";

player->push();

}

std::cout << "Ваш выигрыш: " << player->getCurrentBet()

<< ". Всего: " << player->getBalance() << std::endl;

}

};

// Главная функция программы

int main() {

// Установка локали для корректного отображения русских символов

setlocale(LC\_ALL, "");

SetConsoleOutputCP(CP\_UTF8);

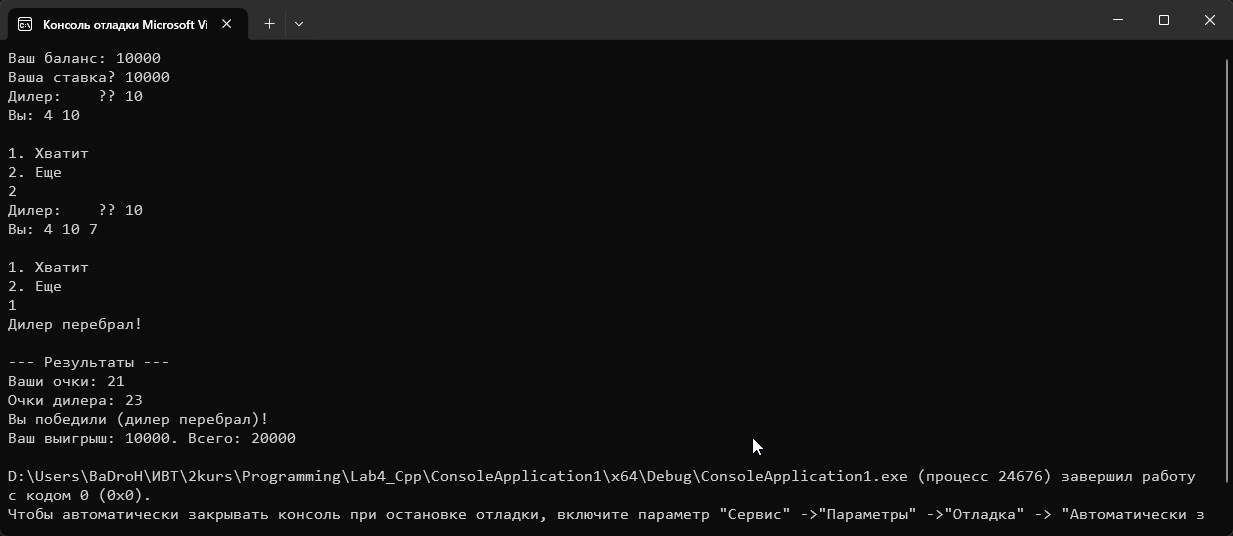
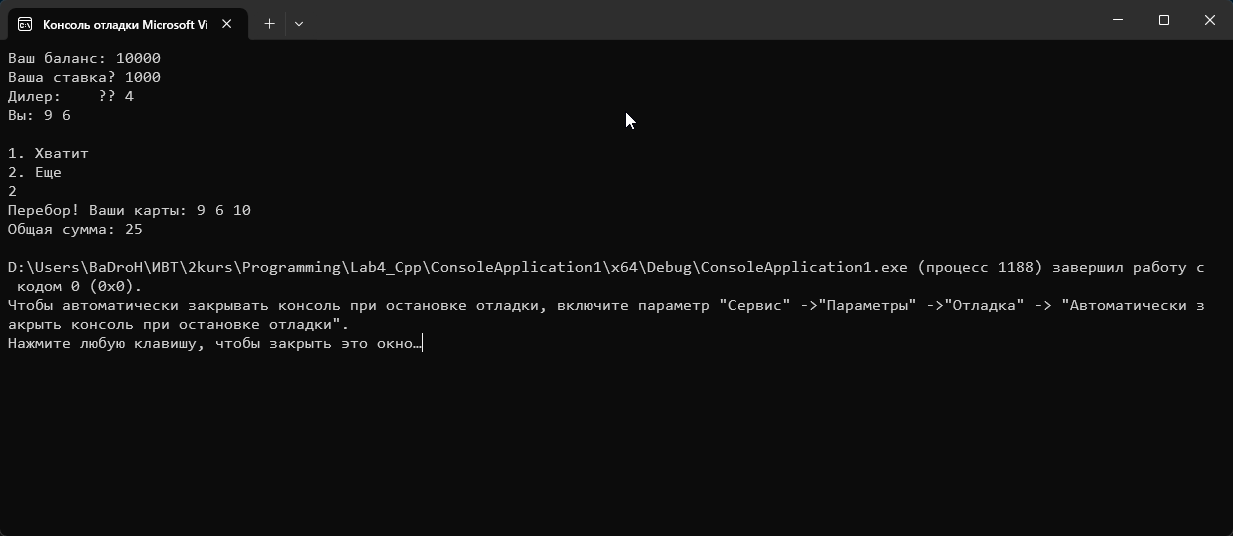
// Создание и запуск игры

BlackJackGame game;

game.play();

}

Результат программы:



1. Код программы:

#include <iostream>

#include <vector>

#include <string>

#include <windows.h> // Для UTF-8 вывода в консоли

using namespace std;

// Интерфейс для форматирования

class IFormattable {

public:

virtual ~IFormattable() = default;

virtual std::string format() const = 0;

};

// Масти карт

enum Suit {

Spades, // Пики

Hearts, // Черви

Diamonds, // Бубны

Clubs // Крести

};

// Класс карты (из предыдущего задания)

class Card {

private:

std::string rank;

Suit suit;

public:

Card(const std::string& rank, Suit suit) : rank(rank), suit(suit) {}

std::string getRank() const { return rank; }

Suit getSuit() const { return suit; }

// Метод для получения символа масти

std::string getSuitSymbol() const {

switch (suit) {

case Spades: return "\u2660"; // Пики

case Hearts: return "\u2665"; // Черви

case Diamonds: return "\u2666"; // Бубны

case Clubs: return "\u2663"; // Крести

default: return "?";

}

}

};

// Адаптер класса (Class Adapter)

class ClassAdapter : public Card, public IFormattable {

public:

ClassAdapter(const std::string& rank, Suit suit) : Card(rank, suit) {}

// Реализация интерфейса IFormattable

std::string format() const override {

return "Карта: " + getRank() + getSuitSymbol();

}

};

// Адаптер объекта (Object Adapter)

class ObjectAdapter : public IFormattable {

private:

const Card& card; // Ссылка на адаптируемый объект

public:

ObjectAdapter(const Card& c) : card(c) {}

// Реализация интерфейса IFormattable

std::string format() const override {

return "Карта: " + card.getRank() + card.getSuitSymbol();

}

};

// Функция для вывода в отформатированном виде

void prettyPrint(const IFormattable& object) {

cout << object.format() << endl;

}

// Тестирование адаптеров

int main() {

setlocale(LC\_ALL, "");

// Устанавливаем UTF-8 кодировку для корректного вывода

SetConsoleOutputCP(CP\_UTF8);

// Пример карт

Card card1("2", Spades);

Card card2("Q", Diamonds);

// Класс-адаптер

ClassAdapter classAdapter("A", Hearts);

// Объект-адаптер

ObjectAdapter objectAdapter(card1);

// Вывод с помощью функции prettyPrint

cout << "Используем адаптер класса:\n";

prettyPrint(classAdapter); // Карта: A♥

cout << "\nИспользуем адаптер объекта:\n";

prettyPrint(objectAdapter); // Карта: 2♠

// Можно создать адаптер объекта для другой карты

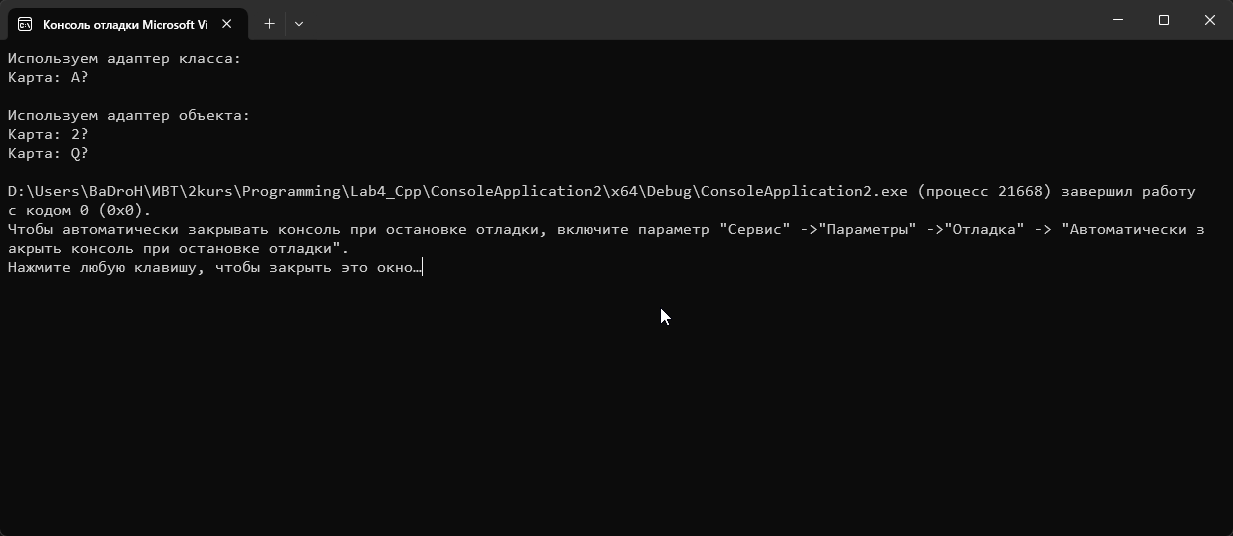
ObjectAdapter objectAdapter2(card2);

prettyPrint(objectAdapter2); // Карта: Q♦

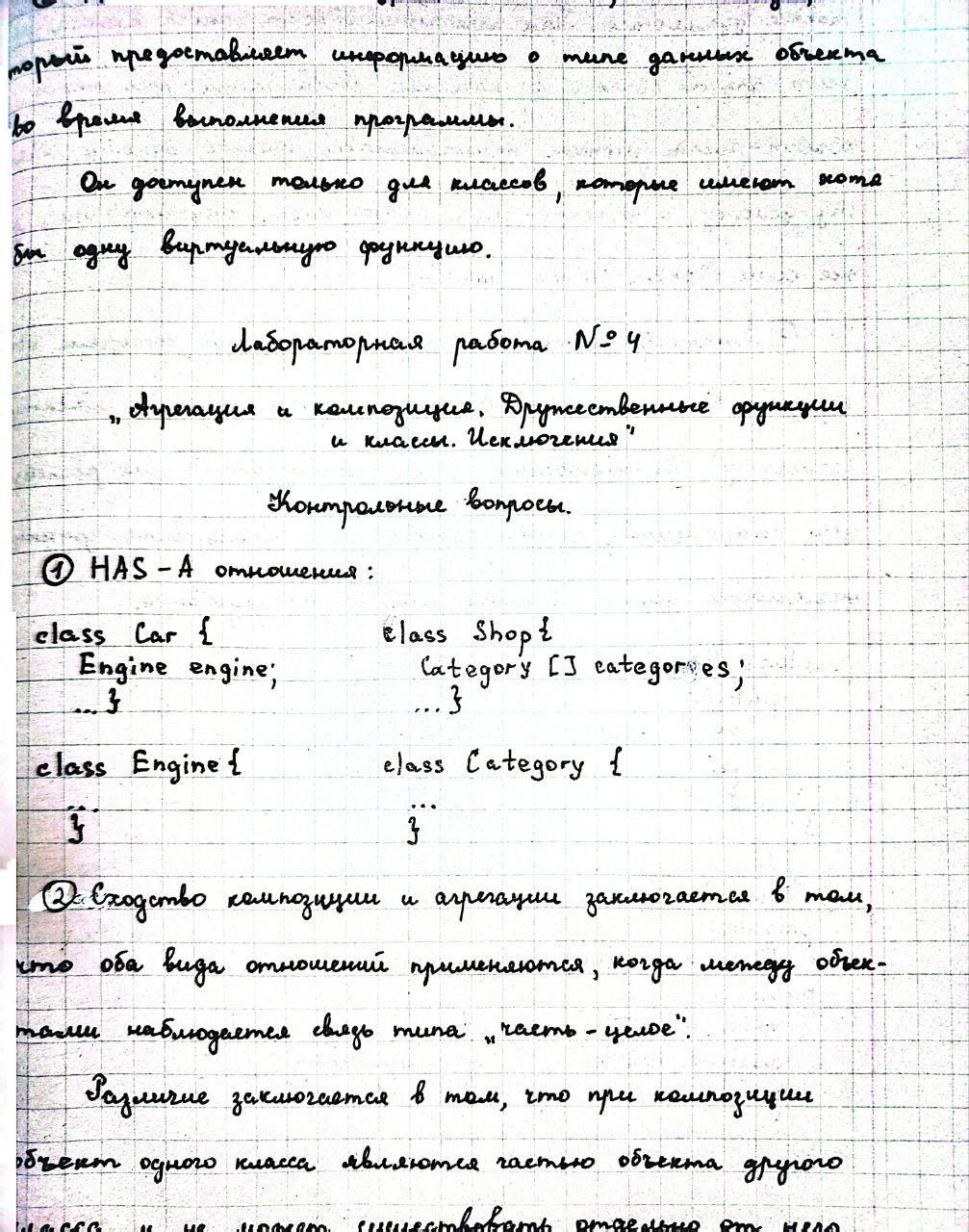
return 0;

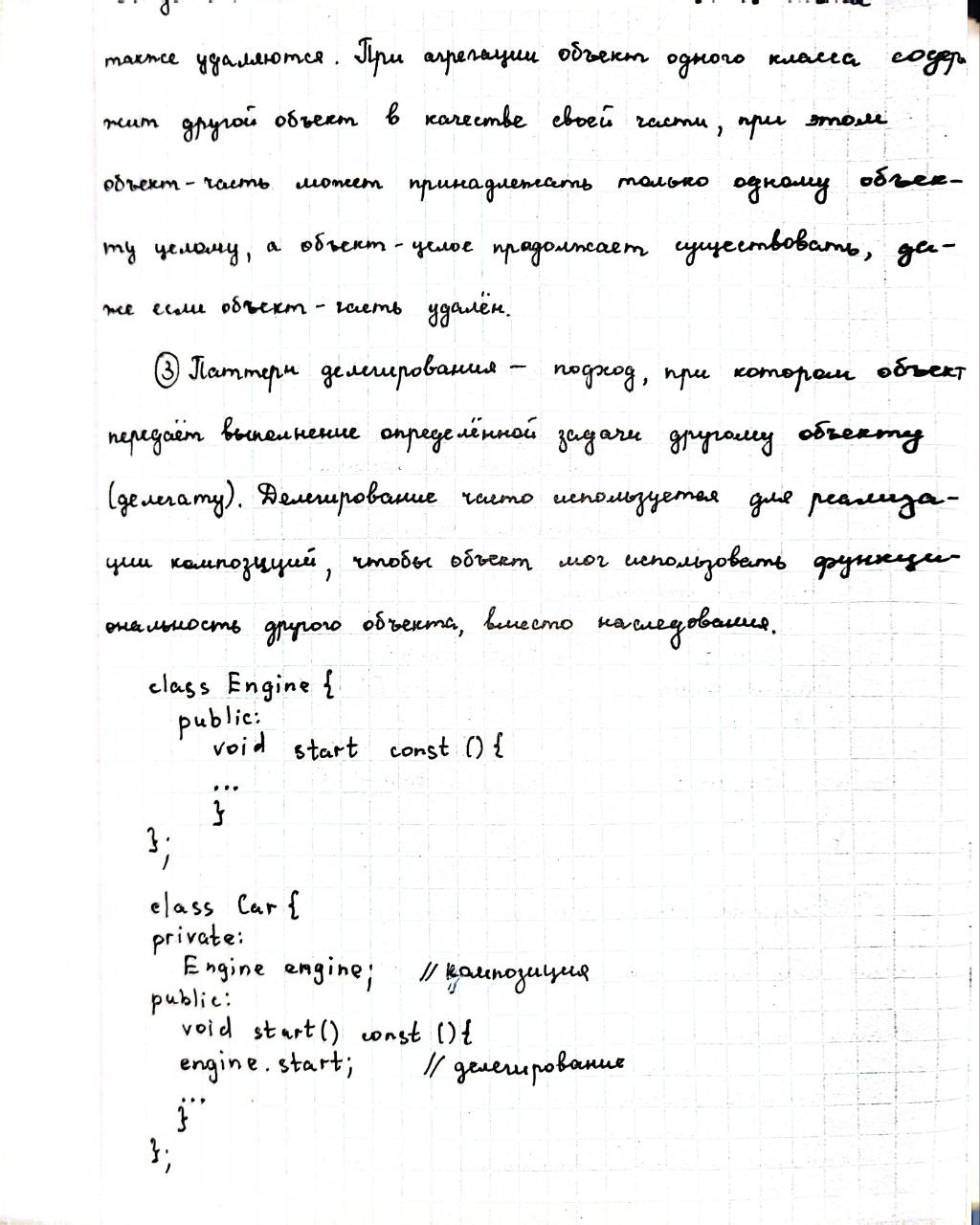
}

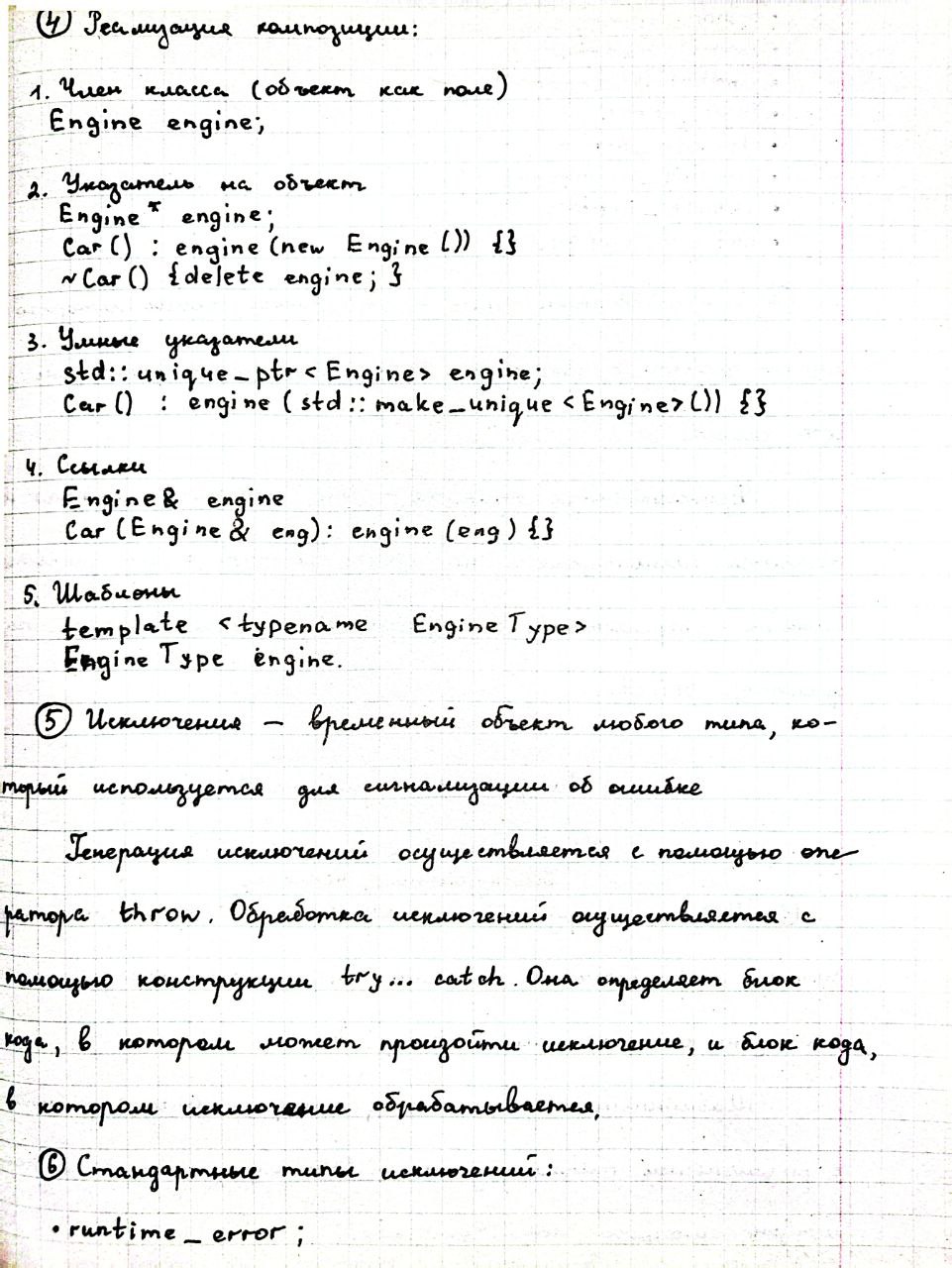
Результат программы:

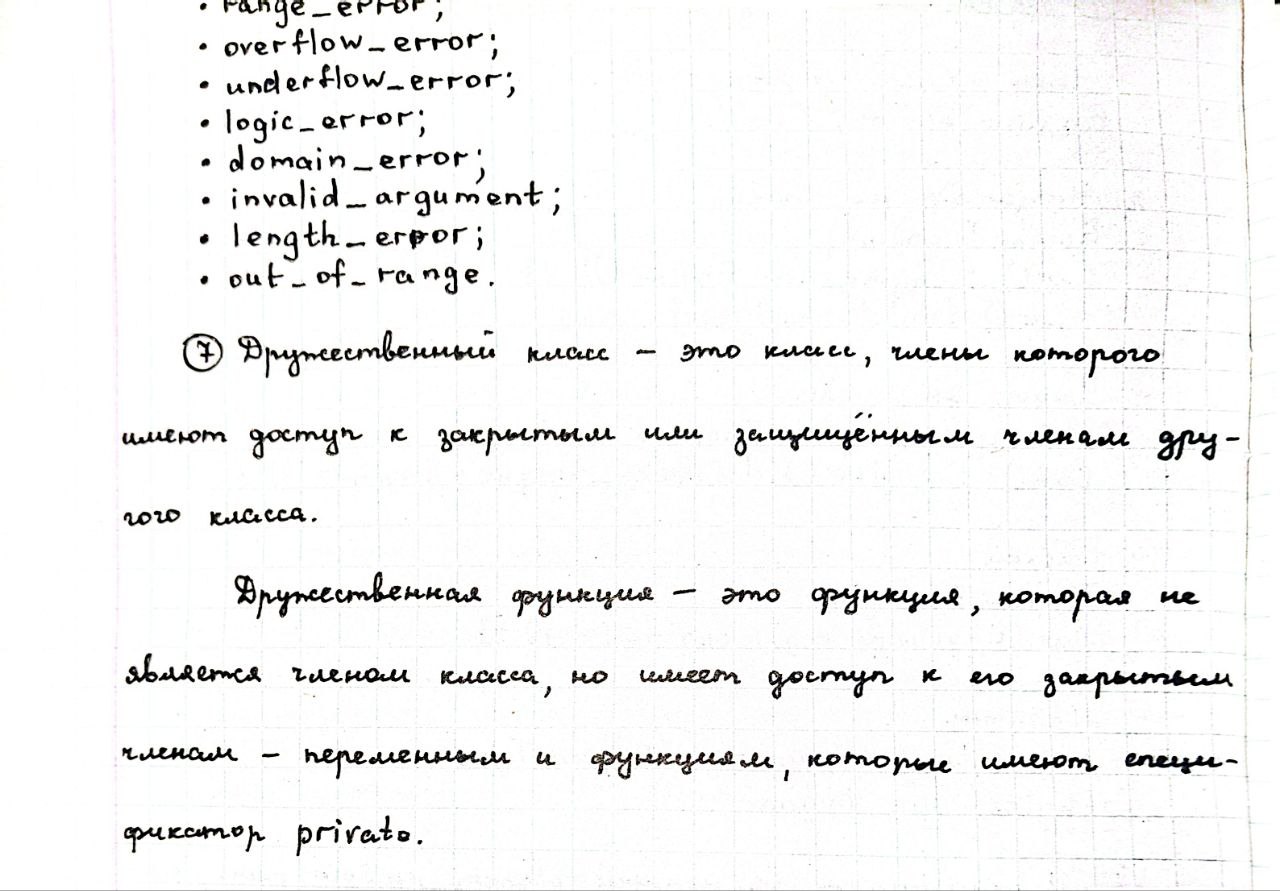


Контрольные вопросы:









Вывод: научился реализовывать на С++ межклассовое отношение агрегации / композиции, писать код генерации и обработки исключений, перегружать операторы с помощью дружественных функций.