Технология программирования

1. Односвязные и двусвязные списки. Очереди и стеки.

Односвязный список

Односвязный список - структура данных, состоящая из узлов, каждый из которых содержит данные и ссылку на следующий узел.

Структура узла:

```
struct Node {
    int data;        // данные
    Node* next;      // указатель на следующий узел
};
```

Основные операции:

1. **Вставка в начало**:

```
void insertAtBeginning(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = head;
    head = newNode;
}
```

2. **Вставка в конец**:

```
void insertAtEnd(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
        return;
    }

    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
}
```

3. **Удаление по значению**:

```
void deleteNode(Node*& head, int value) {
    if (head == nullptr) return;

    if (head->data == value) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    Node* current = head;
    while (current->next && current->next->data != value) {
        current = current->next;
    }

    if (current->next) {
        Node* temp = current->next;
        current->next = temp->next;
        delete temp;
    }
}
```

Преимущества и недостатки:

Преимущества:

- Вставка и удаление в начале за O(1)
- Динамический размер

Недостатки:

- Доступ к элементу за O(n)
- Нет обратного обхода

Двусвязный список

Двусвязный список - структура данных, где каждый узел содержит ссылки на следующий и предыдущий узлы, позволяя двигаться в обоих направлениях.

Структура узла:

```
struct Node {
    int data;
    Node* next;     // указатель на следующий узел
    Node* prev;     // указатель на предыдущий узел
};
```

Основные операции:

1. **Вставка в начало**:

```
void insertAtBeginning(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->prev = nullptr;
    newNode->next = head;

    if (head) head->prev = newNode;
    head = newNode;
}
```

2. **Удаление узла**:

```
void deleteNode(Node*& head, Node* nodeToDelete) {
    if (!head || !nodeToDelete) return;

    // Если удаляем первый узел
    if (head == nodeToDelete) {
        head = nodeToDelete->next;
    }

    // Обновляем указатели соседних узлов
    if (nodeToDelete->next) {
        nodeToDelete->next->prev = nodeToDelete->prev;
    }
    if (nodeToDelete->prev) {
        nodeToDelete->prev->next = nodeToDelete->next;
    }

    delete nodeToDelete;
}
```

Преимущества:

- Обход в обоих направлениях
- Эффективное удаление узла без поиска предыдущего

Недостатки:

- Больше памяти (2 указателя на узел)
- Сложнее реализация

Стек

Стек - абстрактный тип данных, работающий по принципу LIFO (Last In, First Out - "последним пришел, первым вышел").

Реализация на основе массива:

```
class Stack {
private:
    int* array;
    int capacity;
    int top;

public:
    Stack(int size) {
        array = new int[size];
        capacity = size;
        top = -1;
    }

    bool isEmpty() { return top == -1; }
    bool isFull() { return top == capacity - 1; }

    void push(int value) {
        if (isFull()) return;
        array[++top] = value;
    }

    int pop() {
        if (isEmpty()) return -1;
        return array[top--];
    }

    int peek() {
        if (isEmpty()) return -1;
        return array[top];
    }
};
```

Реализация на основе связного списка:

```
class Stack {
private:
    struct Node {
        int data;
        Node* next;
    };
    Node* top;

public:
    Stack() : top(nullptr) {}

    bool isEmpty() { return top == nullptr; }

    void push(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = top;
        top = newNode;
    }

    int pop() {
        if (isEmpty()) return -1;

        int value = top->data;
        Node* temp = top;
        top = top->next;
        delete temp;
        return value;
    }
};
```

Применения стека:

- Проверка сбалансированности скобок
- Преобразование выражений
- Обход деревьев (DFS)
- Управление вызовами функций

Очередь

Очередь - абстрактный тип данных, работающий по принципу FIFO (First In, First Out - "первым пришел, первым вышел").

Реализация на основе массива:

```
class Queue {
private:
    int* array;
    int capacity;
    int front;  // индекс первого элемента
    int rear;   // индекс последнего элемента
    int size;   // текущий размер

public:
    Queue(int capacity) {
        this->capacity = capacity;
        array = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    bool isEmpty() { return size == 0; }
    bool isFull() { return size == capacity; }

    void enqueue(int value) {
        if (isFull()) return;

        rear = (rear + 1) % capacity;  // круговой буфер
        array[rear] = value;
        size++;
    }

    int dequeue() {
        if (isEmpty()) return -1;

        int value = array[front];
        front = (front + 1) % capacity;
        size--;
        return value;
    }
};
```

Реализация на основе связного списка:

```
class Queue {
private:
    struct Node {
        int data;
        Node* next;
    };
    Node* front;  // указатель на первый элемент
    Node* rear;   // указатель на последний элемент

public:
    Queue() : front(nullptr), rear(nullptr) {}

    bool isEmpty() { return front == nullptr; }

    void enqueue(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = nullptr;

        if (isEmpty()) {
            front = rear = newNode;
            return;
        }

        rear->next = newNode;
        rear = newNode;
    }

    int dequeue() {
        if (isEmpty()) return -1;

        int value = front->data;
        Node* temp = front;

        front = front->next;
        if (front == nullptr) rear = nullptr;  // если очередь
опустела

        delete temp;
        return value;
    }
};
```

Применения очереди:

- Планирование задач
- Обход деревьев (BFS)
- Буферизация данных
- Моделирование очередей в обслуживании

2. Определение класса. Создание и уничтожение объектов класса. Компоненты класса. Конструкторы и деструкторы.

Определение класса

Класс — пользовательский тип данных, объединяющий данные (поля) и функции (методы).

```
class Rectangle {
private:
    double width;
    double height;

public:
    // Конструкторы
    Rectangle();
    Rectangle(double w, double h);

    // Методы
    double area() const;
    double perimeter() const;

    // Деструктор
    ~Rectangle();
};
```

Создание и уничтожение объектов

Статические объекты (в стеке):

```
Rectangle rect;              // вызов конструктора по умолчанию
Rectangle rect2(5.0, 3.0);   // вызов параметризованного конструктора
```

Динамические объекты (в куче):

```
Rectangle* pRect = new Rectangle();       // конструктор по умолчанию
Rectangle* pRect2 = new Rectangle(5, 3); // параметризованный
конструктор

delete pRect;    // вызов деструктора и освобождение памяти
delete pRect2;
```

Компоненты класса

1. Поля (данные-члены)

Переменные внутри класса, хранящие состояние объекта.

```
class Person {
private:
    std::string name;  // поле
    int age;           // поле
};
```

2. Методы (функции-члены)

Функции, определяющие поведение объекта.

```
class Circle {
private:
    double radius;

public:
    double getRadius() const { return radius; }  // метод
    void setRadius(double r) { radius = r; }     // метод
    double area() const { return 3.14 * radius * radius; }  // метод
};
```

3. Статические члены

Члены, общие для всех объектов класса.

```
class Counter {
private:
    static int count;  // статическое поле
    int id;

public:
    Counter() { id = count++; }

    static int getCount() {  // статический метод
        return count;
    }
};

// Определение статического поля
int Counter::count = 0;
```

Конструкторы и деструкторы

Конструкторы

Специальные методы, вызываемые при создании объекта.

1. **Конструктор по умолчанию**:

```
class Point {
public:
    Point() {
        x = 0;
        y = 0;
    }
private:
    int x, y;
};
```

2. **Параметризованный конструктор**:

```
class Point {
public:
    Point(int xVal, int yVal) {
        x = xVal;
        y = yVal;
    }
private:
    int x, y;
};
```

3. **Конструктор копирования**:

```
class Point {
public:
    Point(const Point& other) {
        x = other.x;
        y = other.y;
    }
private:
    int x, y;
};
```

4. **Список инициализации**:

```
class Point {
public:
    Point(int xVal, int yVal) : x(xVal), y(yVal) {
        // Тело конструктора (если нужно)
    }
private:
    int x, y;
};
```

Деструктор

Вызывается при уничтожении объекта, освобождает ресурсы.

```
class DynamicArray {
private:
    int* array;
    int size;

public:
    DynamicArray(int s) : size(s) {
        array = new int[size];
    }

    ~DynamicArray() {  // деструктор
        delete[] array;  // освобождение памяти
    }
};
```

3. Наследование классов. Базовый и производный классы. Правила доступа к элементам.

Наследование

Наследование — механизм, позволяющий создавать новые классы на основе существующих.

```
class Shape {  // базовый класс
protected:
    double x, y;  // координаты

public:
    Shape(double xPos = 0, double yPos = 0) : x(xPos), y(yPos) {}

    virtual double area() const { return 0.0; }

    virtual void draw() const {
        std::cout << "Drawing a shape at (" << x << ", " << y << ")"
<< std::endl;
    }
};

class Circle : public Shape {  // производный класс
private:
    double radius;

public:
    Circle(double xPos, double yPos, double r)
        : Shape(xPos, yPos), radius(r) {}

    double area() const override {
        return 3.14159 * radius * radius;
    }

    void draw() const override {
        std::cout << "Drawing a circle with radius " << radius <<
std::endl;
    }
};
```

Правила доступа

Доступ к членам базового класса зависит от спецификатора наследования:

1. Публичное наследование (`public`):

- `public` члены → `public` в производном
- `protected` члены → `protected` в производном
- `private` члены → недоступны в производном

```
class Base {
public:
    int publicVar;
protected:
    int protectedVar;
private:
    int privateVar;
};

class Derived : public Base {
    void access() {
        publicVar = 1;      // OK
        protectedVar = 2;   // OK
        // privateVar = 3;  // Ошибка
    }
};
```

2. Защищенное наследование (`protected`):

- `public` члены → `protected` в производном
- `protected` члены → `protected` в производном
- `private` члены → недоступны в производном

3. Приватное наследование (`private`):

- `public` члены → `private` в производном
- `protected` члены → `private` в производном
- `private` члены → недоступны в производном

4. Одиночное и множественное наследование классов.

Одиночное наследование

Одиночное наследование — когда класс наследует от одного базового класса.

```
class Animal {
protected:
    std::string name;

public:
    Animal(const std::string& n) : name(n) {}

    void eat() const {
        std::cout << name << " is eating." << std::endl;
    }
};

class Dog : public Animal {
private:
    std::string breed;

public:
    Dog(const std::string& n, const std::string& b)
        : Animal(n), breed(b) {}

    void bark() const {
        std::cout << name << " barks: Woof!" << std::endl;
    }
};
```

Множественное наследование

Множественное наследование — когда класс наследует от нескольких базовых классов.

```
class Engine {
public:
    void start() {
        std::cout << "Engine started" << std::endl;
    }
};

class Vehicle {
public:
    void drive() {
        std::cout << "Vehicle is moving" << std::endl;
    }
};

class Car : public Vehicle, public Engine {
public:
    void useHorn() {
        std::cout << "Beep! Beep!" << std::endl;
    }
};

int main() {
    Car myCar;
    myCar.start();  // от Engine
    myCar.drive();  // от Vehicle
    myCar.useHorn(); // собственный метод
}
```

Проблема ромбовидного наследования

Возникает, когда два класса наследуют от одного базового, а затем третий класс наследует от этих двух.

```
    A
   / \
  B   C
   \ /
    D
```

```
class A {
public:
    int data;
};

class B : public A { };
class C : public A { };

// D наследует две копии A
class D : public B, public C { };
```

Виртуальное наследование

Решает проблему ромбовидного наследования, гарантируя одну копию базового класса.

```
class A {
public:
    int data;
};

class B : virtual public A { };
class C : virtual public A { };

// D наследует только одну копию A
class D : public B, public C { };
```

2. Определение класса. Создание и уничтожение объектов класса. Компоненты класса. Конструкторы и деструкторы. Правила преобразования указателей. Способы реализации инкапсуляции.

Определение класса

Класс в C++ — это пользовательский тип данных, который объединяет данные (поля, атрибуты) и функции (методы), которые работают с этими данными. Класс является основой объектно-ориентированного программирования и используется для моделирования объектов реального мира.

Синтаксис определения класса:

```
class ИмяКласса {
    // Спецификатор доступа
private:
    // Приватные члены (данные и методы)

public:
    // Публичные члены (данные и методы)

protected:
    // Защищенные члены (данные и методы)
};
```

Пример определения класса:

```
class Rectangle {
private:
    double width;
    double height;

public:
    // Конструкторы
    Rectangle();
    Rectangle(double w, double h);

    // Методы
    double getWidth() const;
    double getHeight() const;
    void setWidth(double w);
    void setHeight(double h);
    double area() const;
    double perimeter() const;

    // Деструктор
    ~Rectangle();
};
```

Создание и уничтожение объектов класса

Создание объектов:

1. **Статические объекты** (в стеке):

```
Rectangle rect;                    // Вызывается конструктор по умолчанию
Rectangle rect2(5.0, 3.0);         // Вызывается параметризованный конструктор
```

2. **Динамические объекты** (в куче):

```
Rectangle* pRect = new Rectangle();         // Вызывается конструктор по умолчанию
Rectangle* pRect2 = new Rectangle(5.0, 3.0); // Вызывается параметризованный конструктор
```

Уничтожение объектов:

1. **Статические объекты**: Автоматически уничтожаются при выходе из области видимости, где они определены.
2. **Динамические объекты**: Требуют явного освобождения памяти с помощью оператора `delete`:

```
delete pRect;   // Вызывается деструктор
delete pRect2;  // Вызывается деструктор
```

Компоненты класса

1. Поля (данные-члены)

Переменные, описанные внутри класса, которые хранят состояние объекта.

```
class Person {
private:
    std::string name;   // Поле
    int age;            // Поле
};
```

2. Методы (функции-члены)

Функции, описанные внутри класса, которые определяют поведение объекта.

```
class Person {
private:
    std::string name;
    int age;

public:
    void setName(const std::string& n) {  // Метод
        name = n;
    }

    std::string getName() const {         // Метод
        return name;
    }
};
```

3. Статические члены

Члены, общие для всех объектов класса. Существуют в единственном экземпляре независимо от количества созданных объектов.

```
class Counter {
private:
    static int count;     // Статическое поле
    int id;

public:
    Counter() {
        id = count++;
    }

    static int getCount() {  // Статический метод
        return count;
    }
};

// Определение статического поля
int Counter::count = 0;
```

4. Константные методы

Методы, которые гарантируют, что не будут изменять состояние объекта.

```
class Circle {
private:
    double radius;

public:
    double getRadius() const {  // Константный метод
        return radius;
    }

    double area() const {       // Константный метод
        return 3.14159 * radius * radius;
    }
};
```

Конструкторы и деструкторы

Конструкторы

Специальные методы, вызываемые при создании объекта. Они инициализируют объект и обычно имеют то же имя, что и класс.

1. **Конструктор по умолчанию** (не принимает параметров):

```
class Point {
private:
    int x, y;

public:
    Point() {
        x = 0;
        y = 0;
    }
};
```

2. **Параметризованный конструктор**:

```
class Point {
private:
    int x, y;

public:
    Point(int xVal, int yVal) {
        x = xVal;
        y = yVal;
    }
};
```

3. **Конструктор копирования**:

```
class Point {
private:
    int x, y;

public:
    Point(const Point& other) {
        x = other.x;
        y = other.y;
    }
};
```

4. **Список инициализации** (предпочтительный способ инициализации членов):

```
class Point {
private:
    int x, y;

public:
    Point(int xVal, int yVal) : x(xVal), y(yVal) {
        // Тело конструктора (если нужно)
    }
};
```

Деструктор

Специальный метод, вызываемый при уничтожении объекта. Он освобождает ресурсы, выделенные объектом во время его жизни.

```
class DynamicArray {
private:
    int* array;
    int size;

public:
    DynamicArray(int s) : size(s) {
        array = new int[size];
    }

    ~DynamicArray() {  // Деструктор
        delete[] array;  // Освобождаем память
    }
};
```

Правила преобразования указателей

В C++ существуют правила неявного преобразования указателей между классами в иерархии наследования:

1. **Указатель на производный класс может быть неявно преобразован в указатель на базовый класс**:

```
class Base {};
class Derived : public Base {};

Derived* derivedPtr = new Derived();
Base* basePtr = derivedPtr;  // Неявное преобразование вверх по
иерархии
```

2. **Обратное преобразование (от базового к производному) требует явного приведения**:

```
Base* basePtr = new Derived();
Derived* derivedPtr = static_cast<Derived*>(basePtr);  // Явное
преобразование
// или
Derived* derivedPtr2 = dynamic_cast<Derived*>(basePtr);  //
Безопасное преобразование с проверкой
```

3. **Виртуальное наследование влияет на преобразование указателей**:

```
class A {};
class B : public virtual A {};
class C : public virtual A {};
class D : public B, public C {};

D* dPtr = new D();
A* aPtr = dPtr;  // Указатель на единственный экземпляр A в D
```

Способы реализации инкапсуляции

Инкапсуляция — один из основных принципов ООП, который заключается в скрытии внутренней реализации объекта и предоставлении интерфейса для взаимодействия с ним.

1. Спецификаторы доступа

Основной механизм инкапсуляции в C++:

```
class BankAccount {
private:
    double balance;  // Приватные данные, недоступные извне

public:
    // Публичный интерфейс
    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    bool withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
            return true;
        }
        return false;
    }

    double getBalance() const {
        return balance;
    }

protected:
    // Доступно для производных классов, но не для внешнего кода
    void setBalance(double newBalance) {
        balance = newBalance;
    }
};
```

2. Геттеры и сеттеры

Методы, которые контролируют доступ к приватным полям:

```
class Person {
private:
    std::string name;
    int age;

public:
    // Геттеры
    std::string getName() const {
        return name;
    }

    int getAge() const {
        return age;
    }

    // Сеттеры с валидацией
    void setName(const std::string& newName) {
        if (!newName.empty()) {
            name = newName;
        }
    }

    void setAge(int newAge) {
        if (newAge >= 0 && newAge <= 150) {
            age = newAge;
        }
    }
};
```

3. Приватные методы реализации

Методы, которые используются только внутри класса:

```
class TextProcessor {
private:
    std::string text;

    // Приватный метод, используемый только внутри класса
    std::string removeExtraSpaces(const std::string& input) {
        // Реализация
        return result;
    }

public:
    void setText(const std::string& newText) {
        text = removeExtraSpaces(newText);
    }

    std::string getText() const {
        return text;
    }
};
```

4. Дружественные функции и классы

Механизм, позволяющий предоставить доступ к приватным членам выбранным функциям или классам:

```
class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r, double i) : real(r), imag(i) {}

    friend std::ostream& operator<<(std::ostream& os, const Complex&
c);
    friend class ComplexCalculator;  // Дружественный класс
};

// Дружественная функция имеет доступ к приватным членам
std::ostream& operator<<(std::ostream& os, const Complex& c) {
    os << c.real << " + " << c.imag << "i";
    return os;
}
```

Пример полной реализации класса

```
#include <iostream>
#include <string>

class Book {
```

```
private:
    std::string title;
    std::string author;
    int year;
    double price;

    // Приватный метод для внутренней валидации
    bool isValidYear(int y) const {
        return y >= 1450 && y <= 2030; // книгопечатание изобретено
около 1450 года
    }

public:
    // Конструктор по умолчанию
    Book() : title("Unknown"), author("Unknown"), year(2000),
price(0.0) {
        std::cout << "Default constructor called" << std::endl;
    }

    // Параметризованный конструктор
    Book(const std::string& t, const std::string& a, int y, double p)
        : title(t), author(a), price(p) {
        if (isValidYear(y)) {
            year = y;
        } else {
            year = 2000; // значение по умолчанию
        }
        std::cout << "Parameterized constructor called" << std::endl;
    }

    // Конструктор копирования
    Book(const Book& other)
        : title(other.title), author(other.author),
          year(other.year), price(other.price) {
        std::cout << "Copy constructor called" << std::endl;
    }

    // Деструктор
    ~Book() {
        std::cout << "Destructor called for " << title << std::endl;
    }

    // Геттеры
    std::string getTitle() const { return title; }
    std::string getAuthor() const { return author; }
    int getYear() const { return year; }
    double getPrice() const { return price; }

    // Сеттеры
    void setTitle(const std::string& t) { title = t; }
    void setAuthor(const std::string& a) { author = a; }

    void setYear(int y) {
        if (isValidYear(y)) {
            year = y;
        }
```

```
    }

    void setPrice(double p) {
        if (p >= 0) {
            price = p;
        }
    }

    // Функциональные методы
    void display() const {
        std::cout << "Title: " << title << std::endl;
        std::cout << "Author: " << author << std::endl;
        std::cout << "Year: " << year << std::endl;
        std::cout << "Price: $" << price << std::endl;
    }

    bool isOld() const {
        return year < 1950;
    }

    // Перегрузка оператора
    friend std::ostream& operator<<(std::ostream& os, const Book&
book);
};

// Определение дружественной функции
std::ostream& operator<<(std::ostream& os, const Book& book) {
    os << book.title << " by " << book.author << " (" << book.year <<
") - $" << book.price;
    return os;
}

int main() {
    // Создание объектов различными способами
    Book book1; // использует конструктор по умолчанию
    book1.setTitle("The C++ Programming Language");
    book1.setAuthor("Bjarne Stroustrup");
    book1.setYear(1985);
    book1.setPrice(59.99);

    Book book2("Effective C++", "Scott Meyers", 2005, 49.99); //
использует параметризованный конструктор

    Book book3 = book2; // использует конструктор копирования
    book3.setTitle("More Effective C++");

    // Создание объекта в динамической памяти
    Book* pBook = new Book("Design Patterns", "Gang of Four", 1994,
54.99);

    // Использование объектов
    std::cout << "\nBook 1 details:" << std::endl;
    book1.display();

    std::cout << "\nBook 2 details:" << std::endl;
```

```
    std::cout << book2 << std::endl;

    std::cout << "\nBook 3 details:" << std::endl;
    std::cout << book3 << std::endl;

    std::cout << "\nDynamic Book details:" << std::endl;
    pBook->display();

    // Проверка методов
    if (book1.isOld()) {
        std::cout << "\n" << book1.getTitle() << " is considered an
old book." << std::endl;
    } else {
        std::cout << "\n" << book1.getTitle() << " is a relatively
recent book." << std::endl;
    }

    // Освобождение динамически выделенной памяти
    delete pBook;

    return 0;
}
```

3. Наследование классов. Базовый и производный классы. Правила доступа к элементам производного класса. Иерархия классов.

Наследование классов

Наследование — один из основных принципов объектно-ориентированного программирования, который позволяет создавать новые классы на основе существующих. Новый класс (производный или дочерний) наследует поля и методы базового (родительского) класса и может добавлять новые или переопределять существующие.

Синтаксис объявления наследования:

```
class ПроизводныйКласс : [спецификатор_доступа] БазовыйКласс {
    // Дополнительные члены
};
```

где `спецификатор_доступа` может быть `public`, `protected` или `private`.

Базовый и производный классы

Базовый класс

Базовый класс предоставляет общую функциональность, которую будут наследовать производные классы.

```
class Shape {
protected:
    int x, y;  // Координаты центра

public:
    Shape(int xPos = 0, int yPos = 0) : x(xPos), y(yPos) {}

    void move(int newX, int newY) {
        x = newX;
        y = newY;
    }

    virtual double area() const {
        return 0.0;  // Базовая реализация
    }

    virtual void draw() const {
        std::cout << "Drawing a shape at (" << x << ", " << y << ")"
<< std::endl;
    }

    virtual ~Shape() {}  // Виртуальный деструктор для корректного
удаления
};
```

Производный класс

Производный класс наследует и расширяет функциональность базового класса.

```
class Circle : public Shape {
private:
    double radius;

public:
    Circle(int xPos = 0, int yPos = 0, double r = 1.0)
        : Shape(xPos, yPos), radius(r) {}

    double getRadius() const {
        return radius;
    }

    void setRadius(double r) {
        if (r > 0) {
            radius = r;
        }
    }

    // Переопределение виртуальных методов
    double area() const override {
        return 3.14159 * radius * radius;
    }

    void draw() const override {
        std::cout << "Drawing a circle at (" << x << ", " << y
                  << ") with radius " << radius << std::endl;
    }
};
```

Правила доступа к элементам производного класса

Доступ к членам базового класса из производного класса зависит от спецификатора доступа при наследовании и спецификаторов доступа членов базового класса:

1. Публичное наследование (`public`):

- `public` члены базового класса становятся `public` в производном
- `protected` члены базового класса становятся `protected` в производном
- `private` члены базового класса недоступны напрямую в производном

```
class Base {
public:
    int publicVar;
protected:
    int protectedVar;
private:
    int privateVar;
};

class Derived : public Base {
    void access() {
        publicVar = 1;      // OK - публичный член доступен
        protectedVar = 2;   // OK - защищенный член доступен
        // privateVar = 3;   // Ошибка - приватный член недоступен
    }
};
```

2. Защищенное наследование (`protected`):

- `public` члены базового класса становятся `protected` в производном
- `protected` члены базового класса становятся `protected` в производном
- `private` члены базового класса недоступны напрямую в производном

```
class Derived : protected Base {
    void access() {
        publicVar = 1;      // OK - становится защищенным
        protectedVar = 2;   // OK - остается защищенным
        // privateVar = 3;   // Ошибка - приватный член недоступен
    }
};

void externalAccess(Derived& d) {
    // d.publicVar = 1;     // Ошибка - в производном классе это
защищенный член
}
```

3. Приватное наследование (`private`):

- `public` члены базового класса становятся `private` в производном
- `protected` члены базового класса становятся `private` в производном
- `private` члены базового класса недоступны напрямую в производном

```
class Derived : private Base {
    void access() {
        publicVar = 1;      // OK - становится приватным
        protectedVar = 2;   // OK - становится приватным
        // privateVar = 3;   // Ошибка - приватный член недоступен
    }
};

class GrandChild : public Derived {
    void access() {
        // publicVar = 1;    // Ошибка - в Derived это приватный член
        // protectedVar = 2; // Ошибка - в Derived это приватный член
    }
};
```

Иерархия классов

Иерархия классов представляет собой древовидную структуру, отражающую отношения наследования между классами.

Пример простой иерархии классов:

```
         Shape
        /     \
   Circle     Rectangle
                 |
              Square
```

```
class Shape {
public:
    virtual double area() const = 0;
    virtual void draw() const = 0;
    virtual ~Shape() {}
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    double area() const override {
        return 3.14159 * radius * radius;
    }

    void draw() const override {
        std::cout << "Drawing Circle" << std::endl;
    }
};

class Rectangle : public Shape {
protected:
    double width;
    double height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    double area() const override {
        return width * height;
    }

    void draw() const override {
        std::cout << "Drawing Rectangle" << std::endl;
    }
};

class Square : public Rectangle {
public:
    Square(double side) : Rectangle(side, side) {}

    void draw() const override {
        std::cout << "Drawing Square" << std::endl;
    }
};
```

Использование иерархии классов:

```
void printArea(const Shape& shape) {
    std::cout << "Area: " << shape.area() << std::endl;
    shape.draw();
}

int main() {
    Circle circle(5.0);
    Rectangle rectangle(4.0, 6.0);
    Square square(3.0);

    printArea(circle);      // Полиморфный вызов для Circle
    printArea(rectangle);   // Полиморфный вызов для Rectangle
    printArea(square);      // Полиморфный вызов для Square

    // Работа с указателями
    Shape* shapes[3];
    shapes[0] = new Circle(2.0);
    shapes[1] = new Rectangle(3.0, 4.0);
    shapes[2] = new Square(5.0);

    for (int i = 0; i < 3; i++) {
        printArea(*shapes[i]);
        delete shapes[i];  // Вызов виртуального деструктора
    }

    return 0;
}
```

Примеры более сложных иерархий

Пример 1: Иерархия транспортных средств

```
        Vehicle
       /       \
  LandVehicle   WaterVehicle
    /     \          |
 Car     Bicycle    Boat
  |
SUV
```

```
class Vehicle {
protected:
    std::string manufacturer;
    int year;

public:
    Vehicle(const std::string& make, int y)
        : manufacturer(make), year(y) {}

    virtual void move() const = 0;
    virtual ~Vehicle() {}
```

```
};

class LandVehicle : public Vehicle {
protected:
    int wheels;

public:
    LandVehicle(const std::string& make, int y, int w)
        : Vehicle(make, y), wheels(w) {}

    int getWheels() const { return wheels; }
};

class WaterVehicle : public Vehicle {
protected:
    double displacement;  // водоизмещение в тоннах

public:
    WaterVehicle(const std::string& make, int y, double d)
        : Vehicle(make, y), displacement(d) {}

    void move() const override {
        std::cout << manufacturer << " is sailing" << std::endl;
    }
};

class Car : public LandVehicle {
private:
    int doors;

public:
    Car(const std::string& make, int y, int d)
        : LandVehicle(make, y, 4), doors(d) {}

    void move() const override {
        std::cout << manufacturer << " is driving" << std::endl;
    }
};

class Bicycle : public LandVehicle {
public:
    Bicycle(const std::string& make, int y)
        : LandVehicle(make, y, 2) {}

    void move() const override {
        std::cout << manufacturer << " is cycling" << std::endl;
    }
};

class SUV : public Car {
private:
    bool fourWheelDrive;

public:
    SUV(const std::string& make, int y, bool fwd)
```

```
        : Car(make, y, 5), fourWheelDrive(fwd) {}

    bool hasFourWheelDrive() const { return fourWheelDrive; }
};

class Boat : public WaterVehicle {
private:
    int maxPassengers;

public:
    Boat(const std::string& make, int y, double d, int p)
        : WaterVehicle(make, y, d), maxPassengers(p) {}
};
```

Пример 2: Иерархия геометрических фигур с интерфейсами

```
          IDrawable           IRotatable
             |                    |
           Shape                 /
          /       \             /
    Circle      Rectangle ----/
                   |
                Square
```

```
// Интерфейсы
class IDrawable {
public:
    virtual void draw() const = 0;
    virtual ~IDrawable() {}
};

class IRotatable {
public:
    virtual void rotate(double angle) = 0;
    virtual ~IRotatable() {}
};

// Базовый класс
class Shape : public IDrawable {
protected:
    double x, y;

public:
    Shape(double xPos = 0, double yPos = 0) : x(xPos), y(yPos) {}

    virtual double area() const = 0;
    void moveTo(double newX, double newY) {
        x = newX;
        y = newY;
    }

    virtual ~Shape() {}
```

```
};

// Конкретные классы
class Circle : public Shape {
private:
    double radius;

public:
    Circle(double xPos, double yPos, double r)
        : Shape(xPos, yPos), radius(r) {}

    double area() const override {
        return 3.14159 * radius * radius;
    }

    void draw() const override {
        std::cout << "Drawing Circle at (" << x << ", " << y
                  << ") with radius " << radius << std::endl;
    }
};

class Rectangle : public Shape, public IRotatable {
protected:
    double width;
    double height;
    double angle;  // угол поворота в градусах

public:
    Rectangle(double xPos, double yPos, double w, double h)
        : Shape(xPos, yPos), width(w), height(h), angle(0) {}

    double area() const override {
        return width * height;
    }

    void draw() const override {
        std::cout << "Drawing Rectangle at (" << x << ", " << y
                  << ") with width " << width << ", height " <<
height
                  << ", and rotation " << angle << " degrees" <<
std::endl;
    }

    void rotate(double a) override {
        angle = fmod(angle + a, 360.0);
        if (angle < 0) angle += 360.0;
    }
};

class Square : public Rectangle {
public:
    Square(double xPos, double yPos, double side)
        : Rectangle(xPos, yPos, side, side) {}

    void draw() const override {
```

```
        std::cout << "Drawing Square at (" << x << ", " << y
                  << ") with side " << width
                  << " and rotation " << angle << " degrees" <<
std::endl;
    }
};
```

Пример полного приложения, использующего наследование

Создадим систему для банковских счетов с иерархией классов:

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

// Базовый класс для всех счетов
class Account {
protected:
    std::string accountNumber;
    std::string ownerName;
    double balance;

public:
    Account(const std::string& number, const std::string& owner,
double initialBalance = 0.0)
        : accountNumber(number), ownerName(owner),
balance(initialBalance) {}

    std::string getAccountNumber() const { return accountNumber; }
    std::string getOwnerName() const { return ownerName; }
    double getBalance() const { return balance; }

    virtual void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            std::cout << "Deposited $" << amount << " to account " <<
accountNumber << std::endl;
        }
    }

    virtual bool withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
            std::cout << "Withdrew $" << amount << " from account "
<< accountNumber << std::endl;
            return true;
        }
        std::cout << "Withdrawal failed: insufficient funds" <<
std::endl;
        return false;
    }

    virtual void display() const {
        std::cout << "Account: " << accountNumber << std::endl;
```

```
        std::cout << "Account: " << accountNumber << std::endl;
        std::cout << "Owner: " << ownerName << std::endl;
        std::cout << "Balance: $" << balance << std::endl;
    }

    virtual ~Account() {}
};

// Сберегательный счет с процентной ставкой
class SavingsAccount : public Account {
private:
    double interestRate;  // Годовая процентная ставка (в процентах)

public:
    SavingsAccount(const std::string& number, const std::string&
owner,
                   double initialBalance = 0.0, double rate = 1.0)
        : Account(number, owner, initialBalance), interestRate(rate)
{}

    double getInterestRate() const { return interestRate; }

    void setInterestRate(double rate) {
        if (rate >= 0) {
            interestRate = rate;
        }
    }

    // Начисление процентов
    void addInterest() {
        double interest = balance * interestRate / 100.0;
        balance += interest;
        std::cout << "Added $" << interest << " interest to account "
<< accountNumber << std::endl;
    }

    void display() const override {
        Account::display();
        std::cout << "Interest Rate: " << interestRate << "%" <<
std::endl;
    }
};

// Текущий счет с возможностью овердрафта
class CheckingAccount : public Account {
private:
    double overdraftLimit;  // Лимит овердрафта

public:
    CheckingAccount(const std::string& number, const std::string&
owner,
                    double initialBalance = 0.0, double limit = 100.0)
        : Account(number, owner, initialBalance),
overdraftLimit(limit) {}

    double getOverdraftLimit() const { return overdraftLimit; }
```

```
    void setOverdraftLimit(double limit) {
        if (limit >= 0) {
            overdraftLimit = limit;
        }
    }

    // Переопределение метода для учета овердрафта
    bool withdraw(double amount) override {
        if (amount > 0 && balance + overdraftLimit >= amount) {
            balance -= amount;
            std::cout << "Withdrew $" << amount << " from account "
<< accountNumber << std::endl;

            if (balance < 0) {
                std::cout << "Account is now overdrawn by $" << -
balance << std::endl;
            }

            return true;
        }
        std::cout << "Withdrawal failed: exceeds overdraft limit" <<
std::endl;
        return false;
    }

    void display() const override {
        Account::display();
        std::cout << "Overdraft Limit: $" << overdraftLimit <<
std::endl;
    }
};

// Кредитный счет
class CreditAccount : public Account {
private:
    double creditLimit;
    double interestRate;

public:
    CreditAccount(const std::string& number, const std::string&
owner,
                  double limit = 1000.0, double rate = 18.0)
        : Account(number, owner, 0.0), creditLimit(limit),
interestRate(rate) {}

    // Переопределение методов для учета специфики кредитного счета
    void deposit(double amount) override {
        if (amount > 0) {
            balance += amount;
            std::cout << "Paid $" << amount << " towards credit
account " << accountNumber << std::endl;
        }
    }
```

```
    bool withdraw(double amount) override {
        double availableCredit = creditLimit + balance;
        if (amount > 0 && amount <= availableCredit) {
            balance -= amount;
            std::cout << "Charged $" << amount << " to credit account
" << accountNumber << std::endl;
            return true;
        }
        std::cout << "Charge failed: exceeds credit limit" <<
std::endl;
        return false;
    }

    // Начисление процентов на задолженность
    void chargeInterest() {
        if (balance < 0) {
            double interest = -balance * interestRate / 100.0;
            balance -= interest;
            std::cout << "Charged $" << interest << " interest to
credit account " << accountNumber << std::endl;
        }
    }

    void display() const override {
        std::cout << "Credit Account: " << accountNumber <<
std::endl;
        std::cout << "Owner: " << ownerName << std::endl;

        if (balance >= 0) {
            std::cout << "Credit Balance: $" << balance << " (paid
ahead)" << std::endl;
        } else {
            std::cout << "Outstanding Balance: $" << -balance <<
std::endl;
        }

        std::cout << "Credit Limit: $" << creditLimit << std::endl;
        std::cout << "Interest Rate: " << interestRate << "%" <<
std::endl;
        std::cout << "Available Credit: $" << (creditLimit + balance)
<< std::endl;
    }
};

// Банк, управляющий счетами
class Bank {
private:
    std::string name;
    std::vector<Account*> accounts;

public:
    Bank(const std::string& bankName) : name(bankName) {}

    ~Bank() {
        // Освобождаем память при уничтожении банка
```

```
        for (Account* account : accounts) {
            delete account;
        }
    }

    void addAccount(Account* account) {
        accounts.push_back(account);
    }

    Account* findAccount(const std::string& accountNumber) {
        for (Account* account : accounts) {
            if (account->getAccountNumber() == accountNumber) {
                return account;
            }
        }
        return nullptr;
    }

    void displayAllAccounts() const {
        std::cout << "===== " << name << " Accounts =====" <<
std::endl;
        for (const Account* account : accounts) {
            account->display();
            std::cout << "------------------------" << std::endl;
        }
    }
};

int main() {
    Bank bank("Example Bank");

    // Создание разных типов счетов
    Account* acc1 = new Account("A001", "John Doe", 1000.0);
    SavingsAccount* acc2 = new SavingsAccount("S001", "Jane Smith",
2000.0, 2.5);
    CheckingAccount* acc3 = new CheckingAccount("C001", "Bob
Johnson", 500.0, 200.0);
    CreditAccount* acc4 = new CreditAccount("CC001", "Alice Brown",
1500.0, 15.0);

    // Добавление счетов в банк
    bank.addAccount(acc1);
    bank.addAccount(acc2);
    bank.addAccount(acc3);
    bank.addAccount(acc4);

    // Выполнение операций
    acc1->deposit(500.0);
    acc1->withdraw(200.0);

    acc2->deposit(1000.0);
    acc2->addInterest();

    acc3->withdraw(600.0);  // Использование овердрафта

    acc4->withdraw(1000.0);
```

```
    acc4->withdraw(1000.0);
    acc4->chargeInterest();
    acc4->deposit(500.0);

    // Вывод информации о всех счетах
    bank.displayAllAccounts();

    return 0;
}
```

4. Одиночное и множественное наследование классов. Особенности доступа при множественном наследовании.

Одиночное наследование

Одиночное наследование — это форма наследования, при которой класс может наследовать только от одного базового класса. Это самая простая и распространенная форма наследования.

Пример одиночного наследования:

```
class Animal {
protected:
    std::string name;
    int age;

public:
    Animal(const std::string& n, int a) : name(n), age(a) {}

    void eat() const {
        std::cout << name << " is eating." << std::endl;
    }

    void sleep() const {
        std::cout << name << " is sleeping." << std::endl;
    }

    virtual void makeSound() const {
        std::cout << name << " makes a sound." << std::endl;
    }
};

class Dog : public Animal {
private:
    std::string breed;

public:
    Dog(const std::string& n, int a, const std::string& b)
        : Animal(n, a), breed(b) {}

    void makeSound() const override {
        std::cout << name << " barks: Woof! Woof!" << std::endl;
    }

    void fetch() const {
        std::cout << name << " is fetching the ball." << std::endl;
    }

    std::string getBreed() const {
        return breed;
    }
};
```

Множественное наследование

Множественное наследование — это форма наследования, при которой класс может наследовать от нескольких базовых классов. Это позволяет объединить функциональность разных классов, но может привести к проблемам, таким как ромбовидное наследование.

Пример множественного наследования:

```
class Engine {
protected:
    int horsepower;

public:
    Engine(int hp) : horsepower(hp) {}

    void start() const {
        std::cout << "Engine started. " << horsepower << " HP
available." << std::endl;
    }

    void stop() const {
        std::cout << "Engine stopped." << std::endl;
    }
};

class Vehicle {
protected:
    std::string make;
    std::string model;

public:
    Vehicle(const std::string& mk, const std::string& mdl)
        : make(mk), model(mdl) {}

    void drive() const {
        std::cout << make << " " << model << " is being driven." <<
std::endl;
    }

    void park() const {
        std::cout << make << " " << model << " is parked." <<
std::endl;
    }
};

// Множественное наследование
class Car : public Vehicle, public Engine {
private:
    int doors;

public:
    Car(const std::string& mk, const std::string& mdl, int hp, int d)
        : Vehicle(mk, mdl), Engine(hp), doors(d) {}

    void displayInfo() const {
        std::cout << "Car Info: " << make << " " << model <<
std::endl;
        std::cout << "Horsepower: " << horsepower << std::endl;
        std::cout << "Doors: " << doors << std::endl;
    }
};
```

Особенности доступа при множественном наследовании

1. Разрешение конфликтов имен

Если два базовых класса имеют члены с одинаковыми именами, то при обращении к ним из производного класса возникает неоднозначность:

```
class A {
public:
    void show() {
        std::cout << "A::show()" << std::endl;
    }
};

class B {
public:
    void show() {
        std::cout << "B::show()" << std::endl;
    }
};

class C : public A, public B {
    // Наследует show() от обоих классов
};

int main() {
    C c;
    // c.show();  // Ошибка: неоднозначность между A::show() и
B::show()

    // Правильный способ - указать класс явно
    c.A::show();  // Вызов A::show()
    c.B::show();  // Вызов B::show()

    return 0;
}
```

2. Ромбовидное наследование (Diamond Problem)

Ромбовидное наследование возникает, когда класс D наследует от классов B и C, которые, в свою очередь, наследуют от класса A. Это приводит к двум копиям класса A в D.

```
    A
   / \
  B   C
   \ /
    D
```

```
class A {
protected:
    int data;

public:
    A(int d) : data(d) {
        std::cout << "A constructor" << std::endl;
    }
};

class B : public A {
public:
    B(int d) : A(d) {
        std::cout << "B constructor" << std::endl;
    }
};

class C : public A {
public:
    C(int d) : A(d) {
        std::cout << "C constructor" << std::endl;
    }
};

// Обычное множественное наследование
class D : public B, public C {
public:
    D(int d1, int d2) : B(d1), C(d2) {
        std::cout << "D constructor" << std::endl;
    }

    void display() {
        // Ошибка: неоднозначность, какой data использовать - B::data
или C::data
        // std::cout << "data: " << data << std::endl;

        // Правильный способ - указать путь явно
        std::cout << "B::data: " << B::data << std::endl;
        std::cout << "C::data: " << C::data << std::endl;
    }
};
```

3. Виртуальное наследование

Для решения проблемы ромбовидного наследования используется **виртуальное наследование**, которое гарантирует, что в производном классе будет только одна копия общего базового класса.

```
class A {
protected:
    int data;

public:
    A(int d = 0) : data(d) {
        std::cout << "A constructor" << std::endl;
    }
};

// Виртуальное наследование
class B : virtual public A {
public:
    B(int d = 0) : A(d) {
        std::cout << "B constructor" << std::endl;
    }
};

// Виртуальное наследование
class C : virtual public A {
public:
    C(int d = 0) : A(d) {
        std::cout << "C constructor" << std::endl;
    }
};

class D : public B, public C {
public:
    // При виртуальном наследовании D должен явно вызвать конструктор
A
    D(int d = 0) : A(d), B(), C() {
        std::cout << "D constructor" << std::endl;
    }

    void display() {
        // data теперь однозначно определен
        std::cout << "data: " << data << std::endl;
    }
};
```

4. Порядок вызова конструкторов и деструкторов

При множественном наследовании порядок вызова конструкторов и деструкторов следующий:

1. Конструкторы:
    - Сначала вызываются конструкторы базовых классов в порядке их объявления в списке базовых классов
    - Затем вызываются конструкторы вложенных объектов в порядке их объявления
    - В конце вызывается конструктор самого класса

2. Деструкторы:
    - Порядок обратный вызову конструкторов: сначала вызывается деструктор класса, затем деструкторы вложенных объектов, и в конце деструкторы базовых классов в обратном порядке их объявления

5. Приведение типов при множественном наследовании

При множественном наследовании иногда требуется явное приведение типов для разрешения неоднозначностей:

```
int main() {
    D d;

    // Приведение к базовым классам
    B& b = d;        // OK
    C& c = d;        // OK

    A& a1 = b;       // OK
    A& a2 = c;       // OK

    // Без виртуального наследования a1 и a2 указывают на разные
объекты A
    // С виртуальным наследованием a1 и a2 указывают на один и тот же
объект A

    // Приведение между базовыми классами
    B* pb = &d;
    C* pc = &d;

    // Требует явного приведения
    // B* pb2 = pc;            // Ошибка
    B* pb2 = static_cast<B*>(static_cast<void*>(pc));  // Опасно, но
возможно

    // Более безопасно использовать dynamic_cast
    B* pb3 = dynamic_cast<B*>(pc);  // Работает только если классы
имеют виртуальные функции

    return 0;
}
```

Практические рекомендации для множественного наследования

1. **Избегайте при возможности**: Множественное наследование часто можно заменить композицией или агрегацией.
2. **Используйте интерфейсы**: Если нужно множественное наследование, лучше наследовать реализацию только от одного класса, а остальные классы использовать как интерфейсы (абстрактные базовые классы).

3. **Виртуальное наследование**: Всегда используйте виртуальное наследование при наследовании от общего базового класса.
4. **Явное разрешение конфликтов**: При конфликтах имен явно указывайте, какой метод или поле вы хотите использовать.
5. **Видимость и доступ**: Тщательно продумывайте спецификаторы доступа при множественном наследовании.

Пример: Интерфейсы и реализация

Более практичный подход к множественному наследованию — использование интерфейсов:

```cpp
// Интерфейс - абстрактный класс с чисто виртуальными методами
class Drawable {
public:
    virtual void draw() const = 0;
    virtual ~Drawable() {}
};

// Другой интерфейс
class Movable {
public:
    virtual void move(double x, double y) = 0;
    virtual ~Movable() {}
};

// Базовый класс с реализацией
class Shape {
protected:
    double x, y;

public:
    Shape(double xPos = 0, double yPos = 0) : x(xPos), y(yPos) {}

    virtual double area() const = 0;
    virtual ~Shape() {}
};

// Множественное наследование: один класс реализации + несколько
интерфейсов
class Circle : public Shape, public Drawable, public Movable {
private:
    double radius;

public:
    Circle(double xPos, double yPos, double r)
        : Shape(xPos, yPos), radius(r) {}

    double area() const override {
        return 3.14159 * radius * radius;
    }

    void draw() const override {
        std::cout << "Drawing Circle at (" << x << ", " << y
                  << ") with radius " << radius << std::endl;
    }

    void move(double newX, double newY) override {
        x = newX;
        y = newY;
        std::cout << "Moving Circle to (" << x << ", " << y << ")" <<
std::endl;
    }
};
```

Пример полного приложения с множественным наследованием

Рассмотрим пример игровой системы с использованием множественного наследования:

```
#include <iostream>
#include <string>
#include <vector>
#include <cmath>

// Базовый класс для всех игровых объектов
class GameObject {
protected:
    std::string name;
    bool active;

public:
    GameObject(const std::string& n) : name(n), active(true) {}

    virtual void update() {
        // Базовая реализация
    }

    virtual void render() const {
        // Базовая реализация
    }

    std::string getName() const { return name; }
    bool isActive() const { return active; }
    void setActive(bool value) { active = value; }

    virtual ~GameObject() {}
};

// Интерфейс для физических объектов
class PhysicsObject {
protected:
    double x, y;        // Позиция
    double vx, vy;      // Скорость
    double mass;        // Масса

public:
    PhysicsObject(double xPos, double yPos, double m = 1.0)
        : x(xPos), y(yPos), vx(0), vy(0), mass(m) {}

    virtual void applyForce(double fx, double fy) {
        // F = ma => a = F/m
        double ax = fx / mass;
        double ay = fy / mass;

        vx += ax;
        vy += ay;
    }

    virtual void updatePhysics() {
```

```
        x += vx;
        y += vy;

        // Простое затухание
        vx *= 0.98;
        vy *= 0.98;
    }

    double getX() const { return x; }
    double getY() const { return y; }
    double getVelocityX() const { return vx; }
    double getVelocityY() const { return vy; }

    virtual ~PhysicsObject() {}
};

// Интерфейс для объектов, с которыми можно взаимодействовать
class Interactive {
public:
    virtual void interact() = 0;
    virtual bool isInteractable() const = 0;
    virtual ~Interactive() {}
};

// Класс для визуального представления объектов
class Renderable {
protected:
    std::string texture;
    double width, height;
    double rotation;

public:
    Renderable(const std::string& tex, double w, double h)
        : texture(tex), width(w), height(h), rotation(0) {}

    virtual void draw(double x, double y) const {
        std::cout << "Drawing " << texture << " at (" << x << ", " <<
y
                  << ") with size " << width << "x" << height
                  << " and rotation " << rotation << " degrees" <<
std::endl;
    }

    void setRotation(double angle) {
        rotation = angle;
    }

    virtual ~Renderable() {}
};

// Пример множественного наследования: игровой персонаж
class Character : public GameObject, public PhysicsObject, public
Interactive, public Renderable {
private:
    int health;
    int maxHealth;
```

```
    bool interactable;

public:
    Character(const std::string& name, double x, double y, const
std::string& tex, double w, double h)
        : GameObject(name), PhysicsObject(x, y, 70.0),
Renderable(tex, w, h),
          health(100), maxHealth(100), interactable(true) {}

    void update() override {
        if (isActive()) {
            updatePhysics();

            // Проверка здоровья
            if (health <= 0) {
                setActive(false);
            }
        }
    }

    void render() const override {
        if (isActive()) {
            draw(getX(), getY());

            // Отрисовка полоски здоровья
            std::cout << "Health: " << health << "/" << maxHealth <<
std::endl;
        }
    }

    void takeDamage(int amount) {
        if (amount > 0 && isActive()) {
            health -= amount;
            std::cout << getName() << " takes " << amount << "
damage." << std::endl;

            if (health <= 0) {
                health = 0;
                std::cout << getName() << " has been defeated!" <<
std::endl;
            }
        }
    }

    void heal(int amount) {
        if (amount > 0 && isActive()) {
            health = std::min(health + amount, maxHealth);
            std::cout << getName() << " heals for " << amount << "
health." << std::endl;
        }
    }

    // Реализация Interactive
    void interact() override {
        if
```