C++. Уровень 2

Урок 3

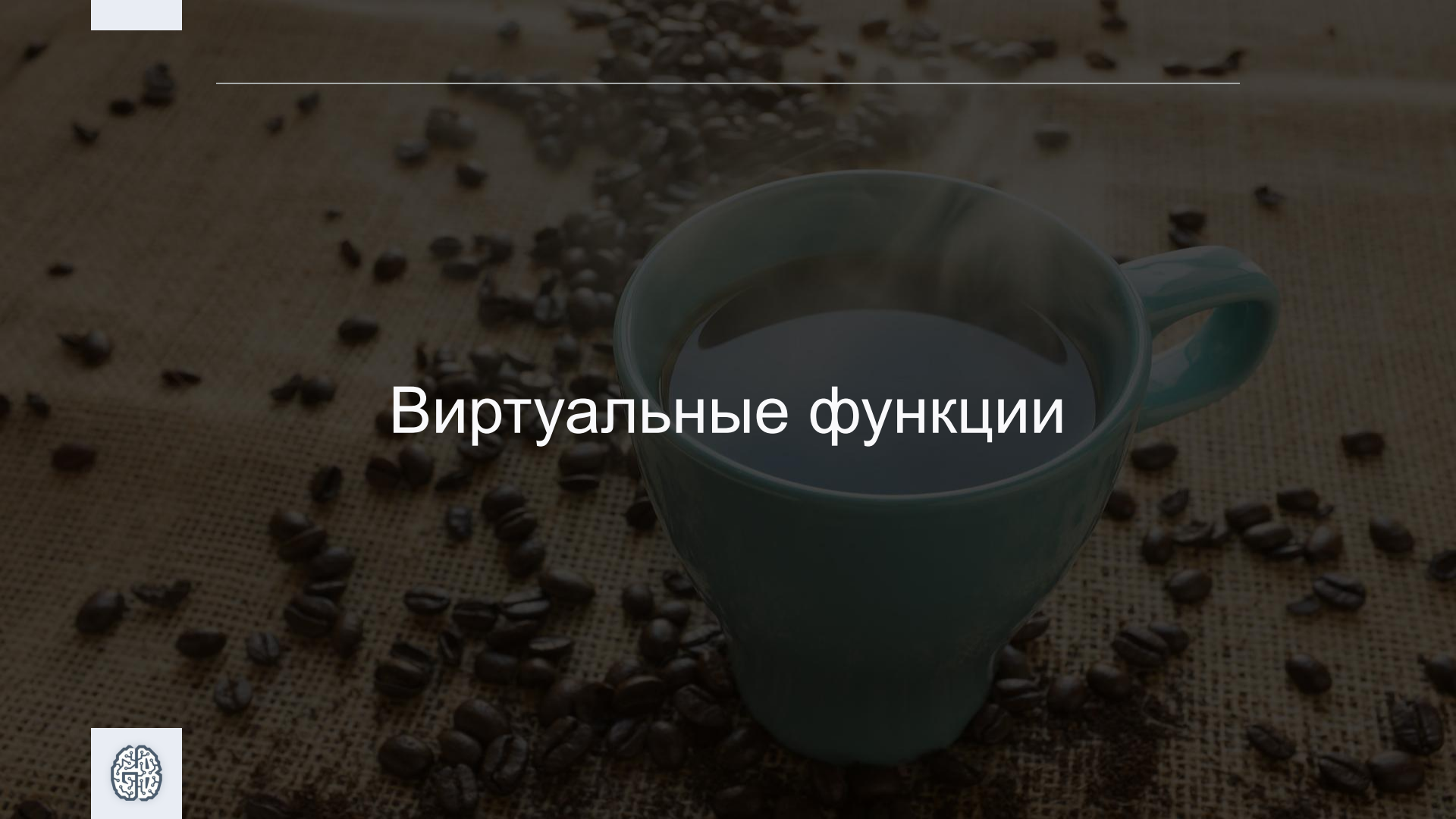
Виртуальные функции и полиморфизм

Абстрактные классы. Интерфейсные классы. Раннее и позднее связывание.

План урока

- Виртуальные функции.
- Абстрактные и интерфейсные классы.
- Раннее и позднее связывание.
- Перегрузка операторов.

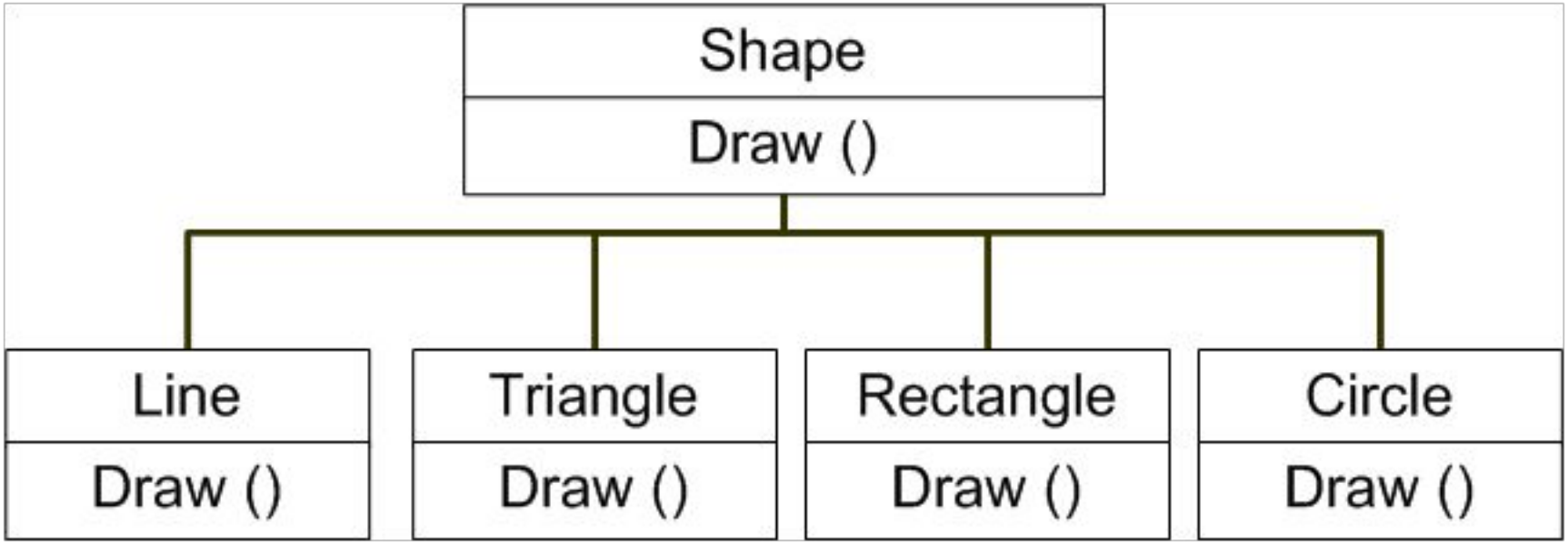
Виртуальные функции



Виртуальные функции

Виртуальная функция – это метод класса, который может быть переопределен в дочерних классах так, что конкретная реализация метода для вызова будет определяться во время исполнения.

Виртуальные функции



Полиморфизм

Один интерфейс, много реализаций

Действие:
бегать и есть

```
void doSomething(Animal animal)
{
animal.move();
//…
animal.eat();
}

Calf calf = new Calf();
Pig pig = new Pig();
Lamb lamb = new Lamb();
doSomething(calf);
doSomething(pig);
doSomething(lamb);
```

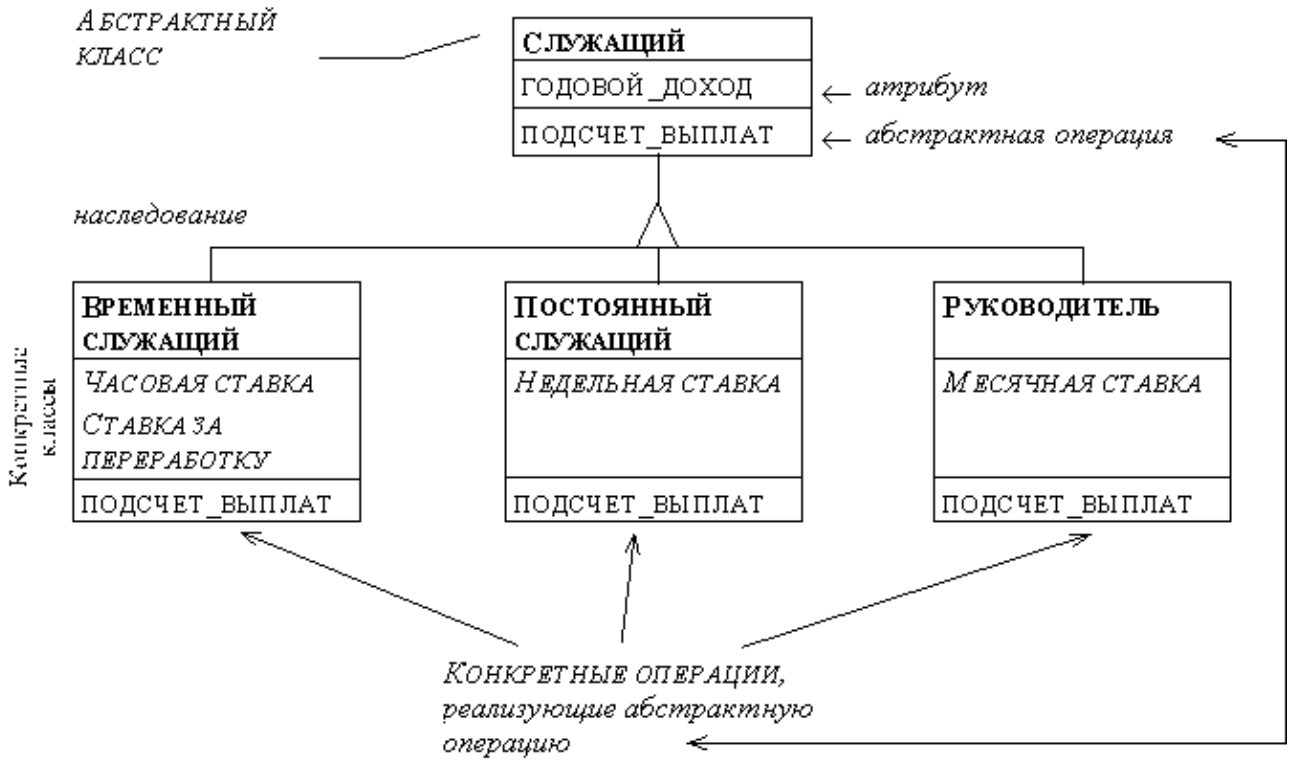
Виртуальные деструкторы

Если деструктор объявлен как **виртуальный**, то при вызове его через указатель на объект базового класса (через **delete**) будет вызван вначале деструктор производного класса, а затем деструктор базового класса.

Правило: при работе с наследованием ваши деструкторы должны быть виртуальными.

Абстрактные и интерфейсные классы

Абстрактный класс

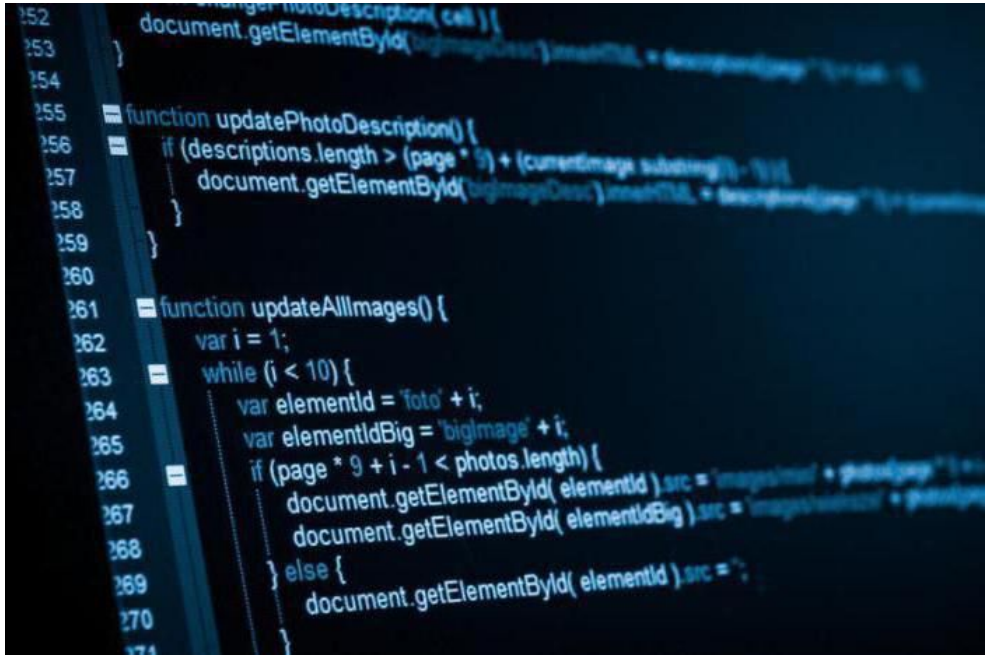


Чисто виртуальные функции

```
virtual void say()=0;
```

- Нельзя создавать объекты абстрактного класса.
- Все производные классы должны переопределять все чисто виртуальные функции.

Интерфейсный класс

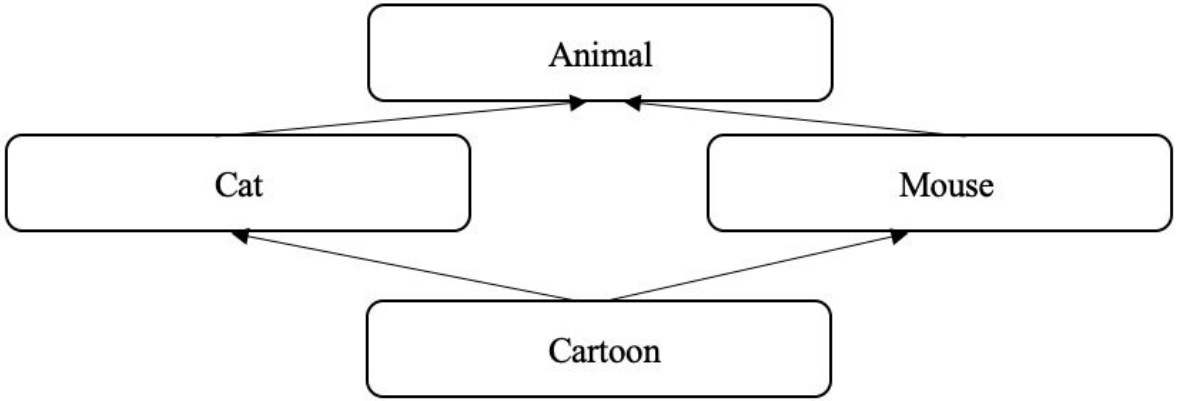


- Нет переменных-членов.
- Все функции чисто виртуальные.

Пример

```
class IPerson {
public:
    virtual ~IPerson();
    virtual string name() const = 0;
    virtual string birthDate() const = 0;
};
```

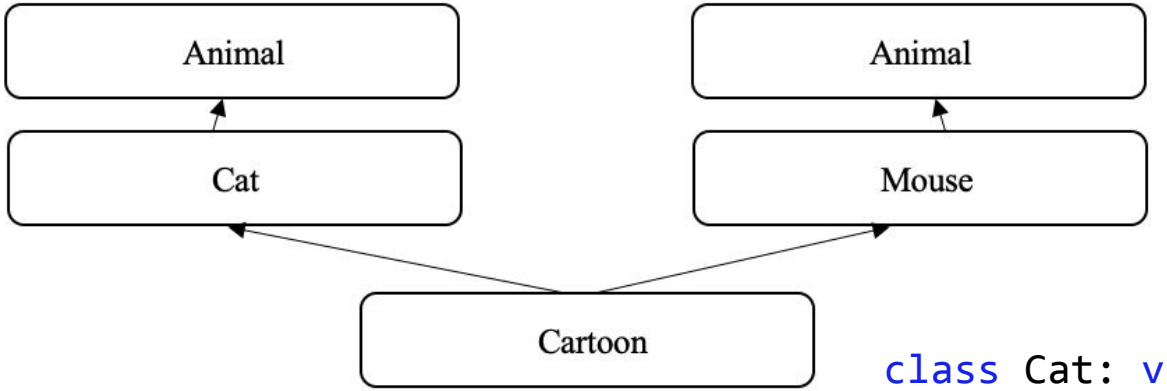
Виртуальный базовый класс



Пример «алмаза смерти»

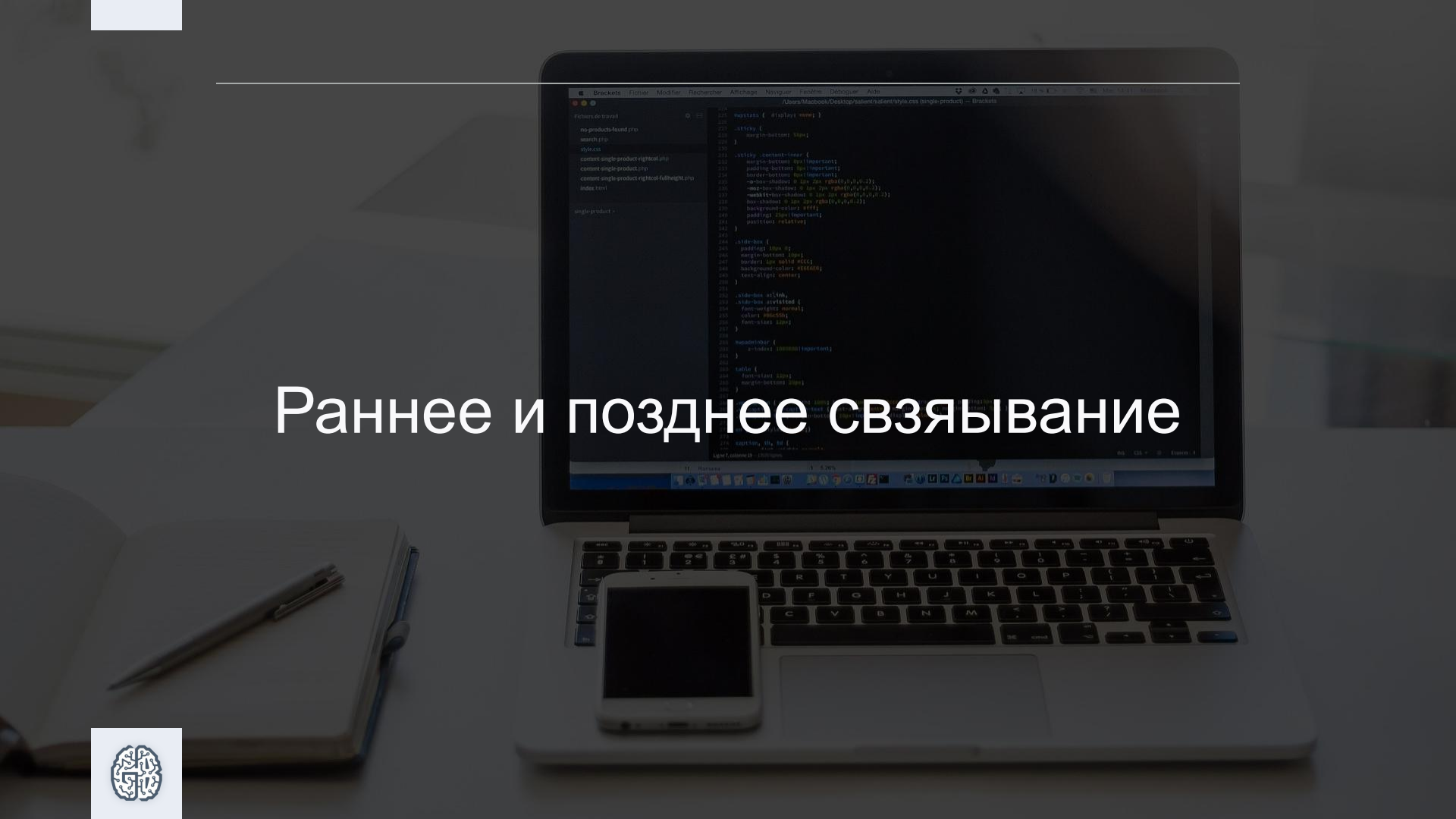
```
class Cat: public Animal
class Mouse: public Animal
class Cartoon: public Cat, public Mouse
```

Виртуальный базовый класс




```
class Cat: virtual public Animal
class Mouse: virtual public Animal
class Cartoon: public Cat, public Mouse
```

Раннее и позднее свзяывание



Раннее связывание

Раннее связывание означает, что компилятор может напрямую связать имя идентификатора (например, имя функции или переменной) с машинным адресом.

```
#include <iostream>
using namespace std;


void printValue(int value)
{
    cout << value;
}


int main()
{
    printValue(7); // Это прямой вызов функции
    return 0;
}
```

Позднее связывание

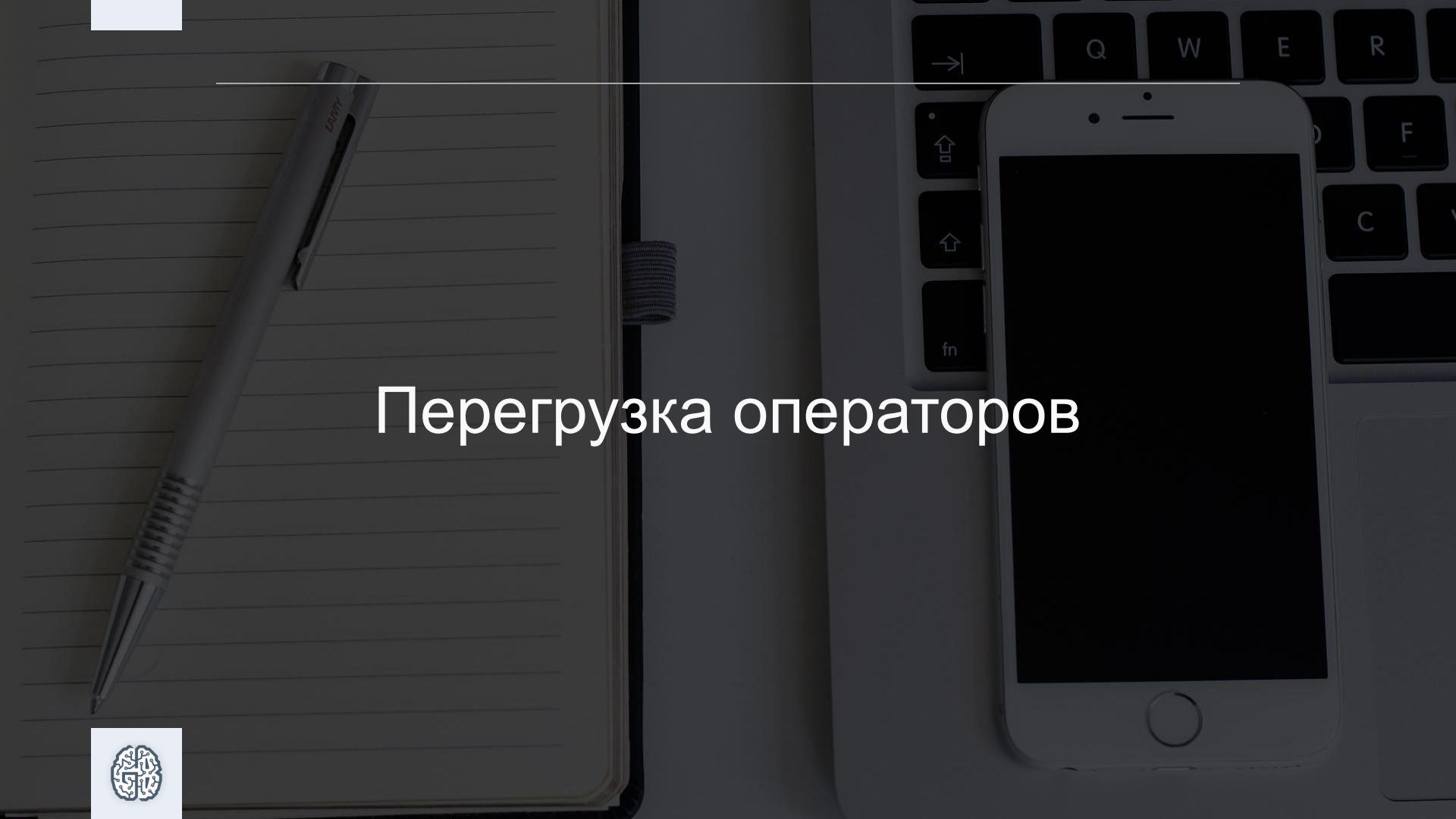
В некоторых программах невозможно заранее знать, какая функция будет вызываться первой. В таком случае используется **позднее связывание.**

```
#include <iostream>


int add(int a, int b)
{
    return a + b;
}


int main()
{
    // Создаем указатель на функцию add
    int (*pFcn)(int, int) = add;
    cout << pFcn(4, 5); //вызов add(4 + 5)

    return 0;
}
```



Перегрузка операторов

Все-таки перегрузка

Операторы

Унарные операторы
(&, |, ~, !)

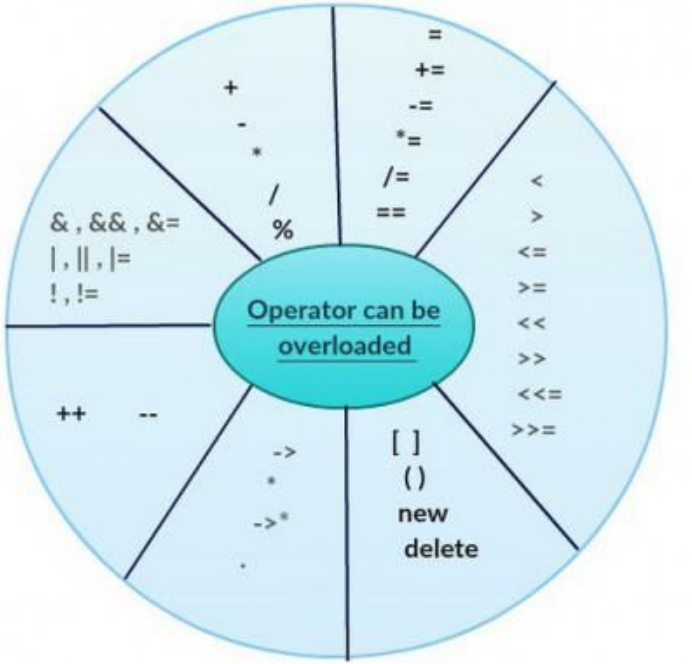
Бинарные операторы
(+, -, >, &&)

Тернарный оператор
(? :)

Способы перегрузки операторов

Операторы	Перегрузка через обычную или дружественную функцию	Перегрузка через метод класса
Унарный оператор	-	+
Бинарный оператор, не изменяющий левый операнд	+	-
Бинарный оператор, изменяющий левый операнд	-	+

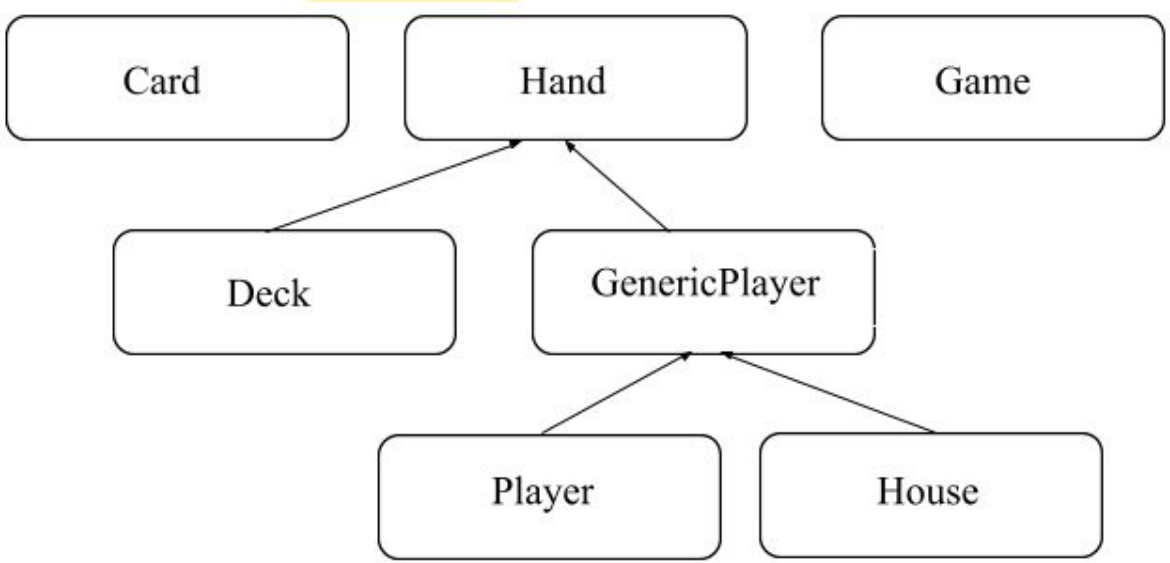
Эти операторы можно перегрузить



Классы в игре Blackjack

Класс	Родительский класс	Описание
Card	нет	Карта
Hand	нет	Набор карт, коллекция объектов класса Card
Deck	Hand	Имеет дополнительную функциональность, которая отсутствует в классе Hand, в частности — тасование и раздачу
GenericPlayer	Hand	Обобщенно описывает игрока. Не является полноценным игроком, а лишь содержит элементы, характерные как для игрока-человека, так и для игрока-компьютера
Player	GenericPlayer	Человек-игрок
House	GenericPlayer	Компьютер-игрок
Game	нет	Игра

Иерархия классов



Решите задачи

Каждая из следующих программ содержит ошибку. Ваша задача — найти ее.

Предполагаемый вывод каждой программы: Child

Продолжение

1

```
#include <iostream>
using namespace std;


class Parent
{
protected:
    int m_value;
public:
    Parent(int value) : m_value(value)
    { }
    const char* getName() const {
      return "Parent";
      }
};
```

```
class Child: public Parent
{
public:
    Child(int value) : Parent(value)
    { }
    const char* getName() const {
      return "Child";
      }
};


int main()
{
    Child ch(7);
    Parent &p = ch;
    cout << p.getName();
    return 0;
}
```

Продолжение

2

```
#include <iostream>
using namespace std;


class Parent
{
protected:
    int m_value;
public:
    Parent(int value) : m_value(value)
    { }
    virtual const char* getName()
    {
        return "Parent";
    }
};
```

```
class Child: public Parent
{
public:
    Child(int value) : Parent(value)
    { }
    virtual const char* getName() const
    {
        return "Child";
    }
};

int main()
{
    Child ch(7);
    Parent &p = ch;
    cout << p.getName();
    return 0;
}
```

Продолжение

3

```
#include <iostream>
using namespace std;


class Parent
{
protected:
    int m_value;
public:
    Parent(int value) : m_value(value)
    { }
    virtual const char* getName()
    {
        return "Parent";
    }
};
```

```
class Child: public Parent
{
public:
    Child(int value) : Parent(value)
    { }
    virtual const char* getName() const
    {
        return "Child";
    }
};

int main()
{
    Child ch(7);
    Parent p = ch;
    cout << p.getName();
    return 0;
}
```

Продолжение

4

```
#include <iostream>
using namespace std;


class Parent
{
protected:
    int m_value;
public:
    Parent(int value) : m_value(value)
    { }
    virtual const char* getName()
    {
        return "Parent";
    }
};
```

```
class Child: public Parent
{
public:
    Child(int value) : Parent(value)
    { }
    virtual const char* getName() = 0;
};

const char* Child::getName()
{
    return "Child";
}

int main()
{
    Child ch(7);
    Parent &p = ch;
    cout << p.getName();
    return 0;
}
```

Продолжение

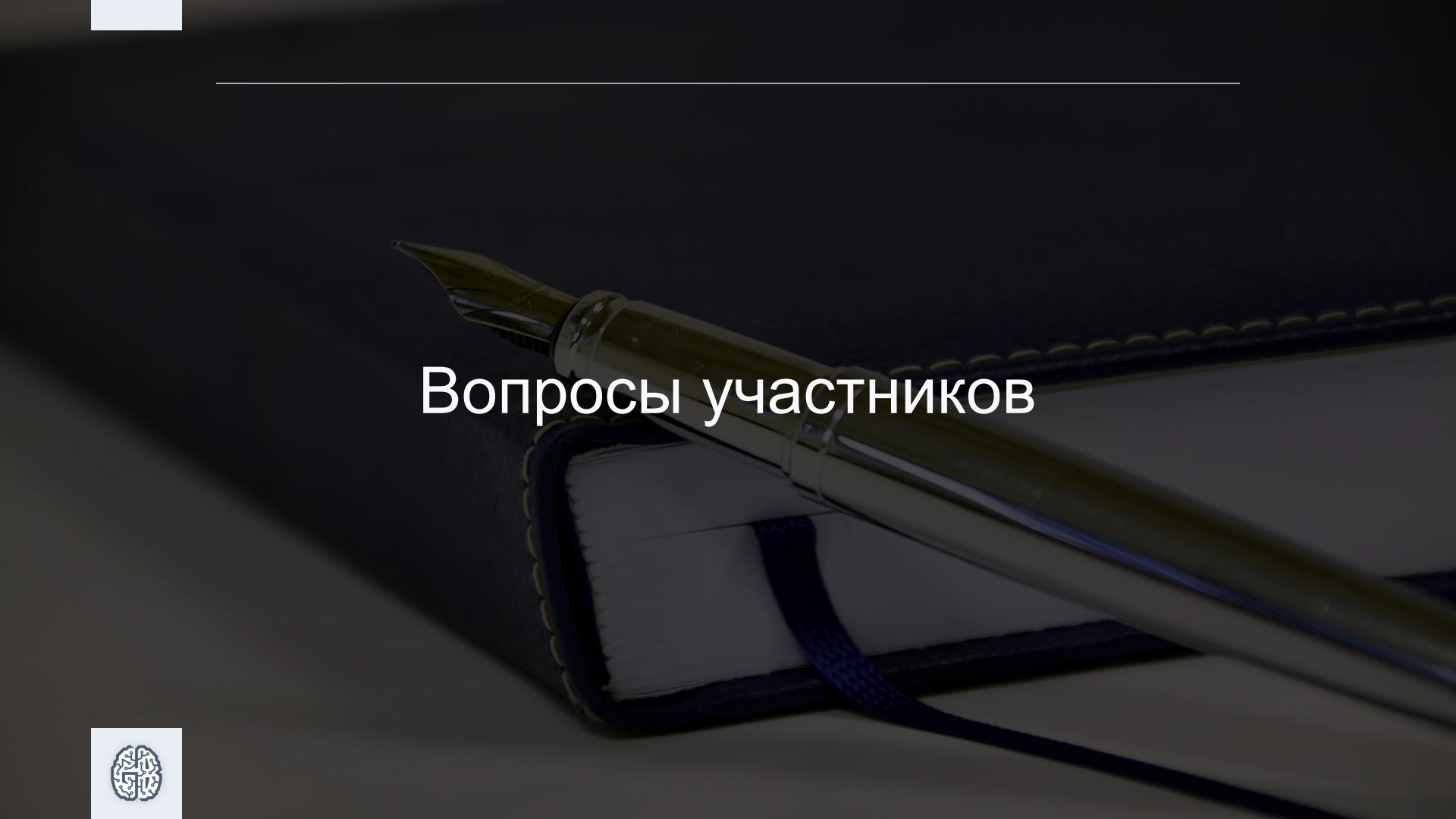
5

```
#include <iostream>
using namespace std;


class Parent
{
protected:
    int m_value;
public:
    Parent(int value) : m_value(value)
    { }
    virtual const char* getName()
    {
        return "Parent";
    }
};
```

```
class Child: public Parent
{
public:
    Child(int value) : Parent(value)
    { }
    virtual const char* getName() const
    {
        return "Child";
    }
};


int main()
{
    Child *ch = new Child(7);
    Parent *p = ch;
    cout << p->getName();
    delete p;
    return 0;
}
```



Вопросы участников