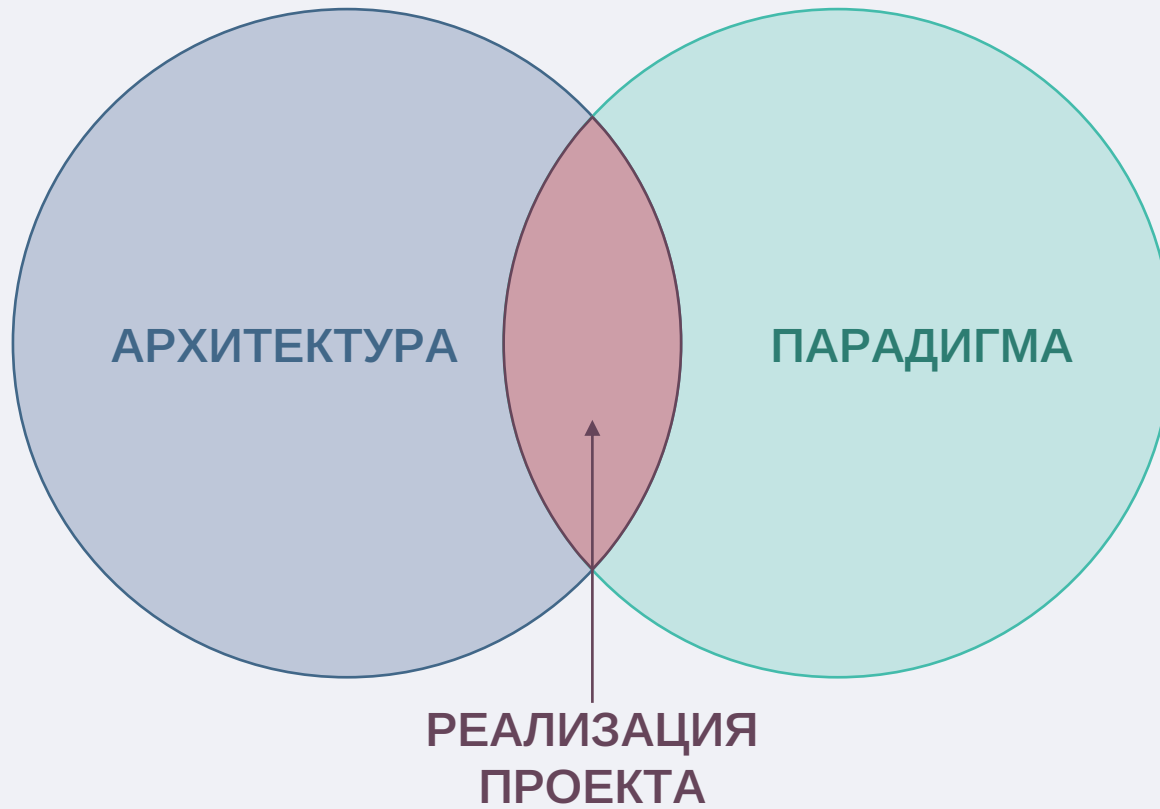


Технологии программирования

Лекция 3 | ООП и подходы к программированию

ООП и подходы к программированию



Выбор парадигмы напрямую влияет на реализацию выбранного архитектурного стиля



Парадигмы программирования

Процедурное, функциональное и объектно-ориентированное

Синтез ООП и других парадигм

И почему именно на ООП делают акцент?

Основные принципы ООП

Наследование, полиморфизм, инкапсуляция

Перегрузка операторов в C++

Почему это необходимо, и какие бывают частые ошибки?



Мы здесь



Парадигмы программирования определяют весь подход к написанию кода



Одна и та же задача может быть решена по-разному:
каждая парадигма подразумевает своё решение

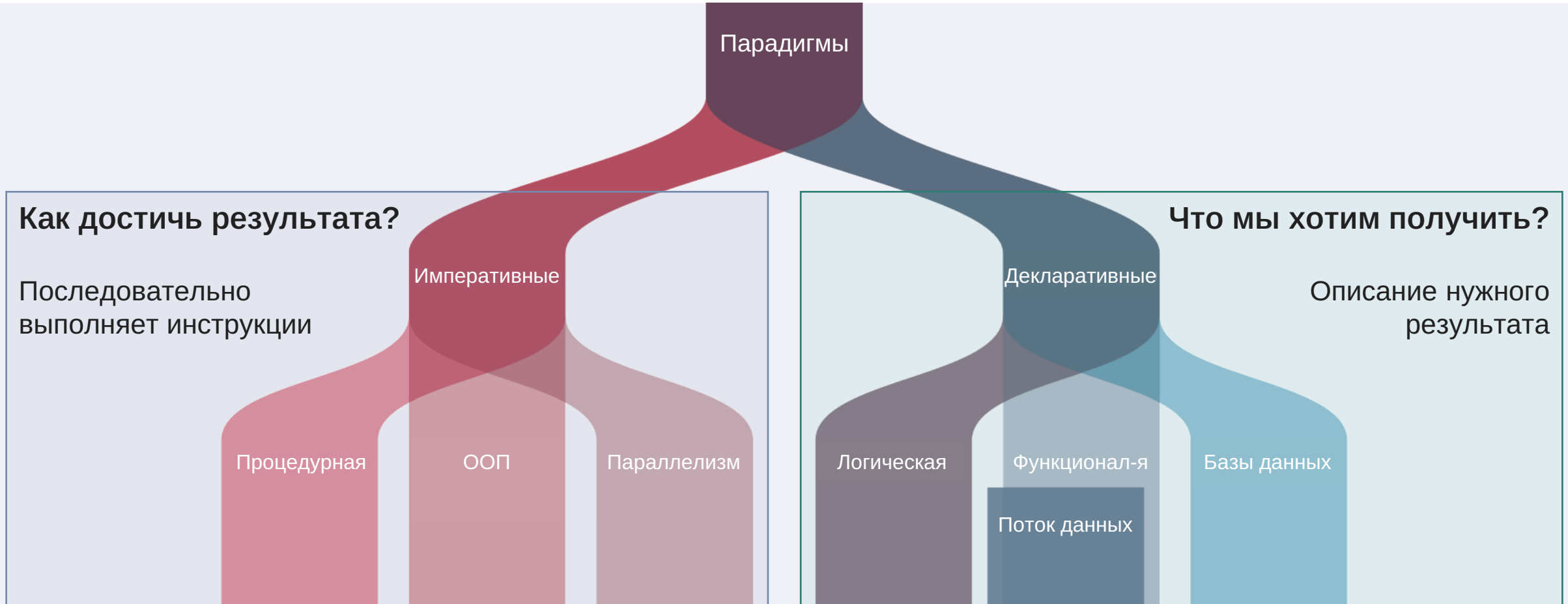
Парадигма – ориентир, состоящий из набора образцов (как надо) и запретов (на определенные действия внутри кода программы)

Языки программирования конструируются на основе парадигм

Если вы хорошо знакомы с парадигмами, то легко перейдете почти на любой язык, отвечающий знакомым принципам

Приемы из одной парадигмы могут использоваться совместно с приемами из другой

Выбор парадигмы – это важно



Процедурное программирование

C

Pascal

Fortran

Basic



- Программа делится на логические блоки, где каждый компонент выполняет свою задачу
- Все процессы могут иметь доступ к одним и тем же данным
- **Недетерминированный ответ блоков** (при одних и тех же входных данных повторный результат не гарантируется)

Например, в программе управления библиотекой:

При процедурном подходе работу с книгами выполняют отдельные блоки кода
Данные будут храниться в **глобальных переменных, используемых файлах или модуле**

Процедурное программирование

C

Pascal

Fortran

Basic

Образцы

Определение и вызов процедур

Последние могут принимать аргументы и возвращать значения

Глобальные переменные для хранения данных,
которые могут использоваться в разных частях программы

Конструкции для проверки условий, выполнения циклов
(if-else или switch-case, циклы for, while или do-while)

Встроенные типы данных и структуры
(массивы, перечни или записи, для хранения и организации сведений)

Организация кода программы в модули или файлы

Запреты

Неоправданные глобальные переменные

Слишком глубокая вложенность процедур

Процедурное программирование

- + Простота – процедурный код часто легко понять и отладить
- + Эффективность – для многих задач процедурный стиль обеспечивает высокую производительность
- + Широкая поддержка – большинство языков программирования поддерживают процедурный стиль
- Сложность для больших проектов – трудно поддерживать множество взаимосвязанных переменных и процедур
- Ограниченная абстракция – менее удобен для моделирования сложных систем и объектов реального мира

**Операционные системы
и системное ПО**

Веб-приложения и сайты
(PHP для динамических страниц)

Научные вычисления
(Fortran и Matlab для расчетов)

Процедурное программирование

```
#include <iostream>

using namespace std;

// Функция для вычисления факториала
int factorial(int n) {
    if (n == 0) {
        return 1; // Факториал 0 равен 1
    } else {
        return n * factorial(n - 1); // Рекурсивный
        вызов функции
    }
}
```

```
int main() {
    int num;
    cout << "Введите число: ";
    cin >> num;

    if (num < 0) {
        cout << "Факториал не определен для
отрицательных чисел." << endl;
    } else {
        int result = factorial(num);
        cout << "Факториал числа " << num << " равен "
        << result << endl;
    }

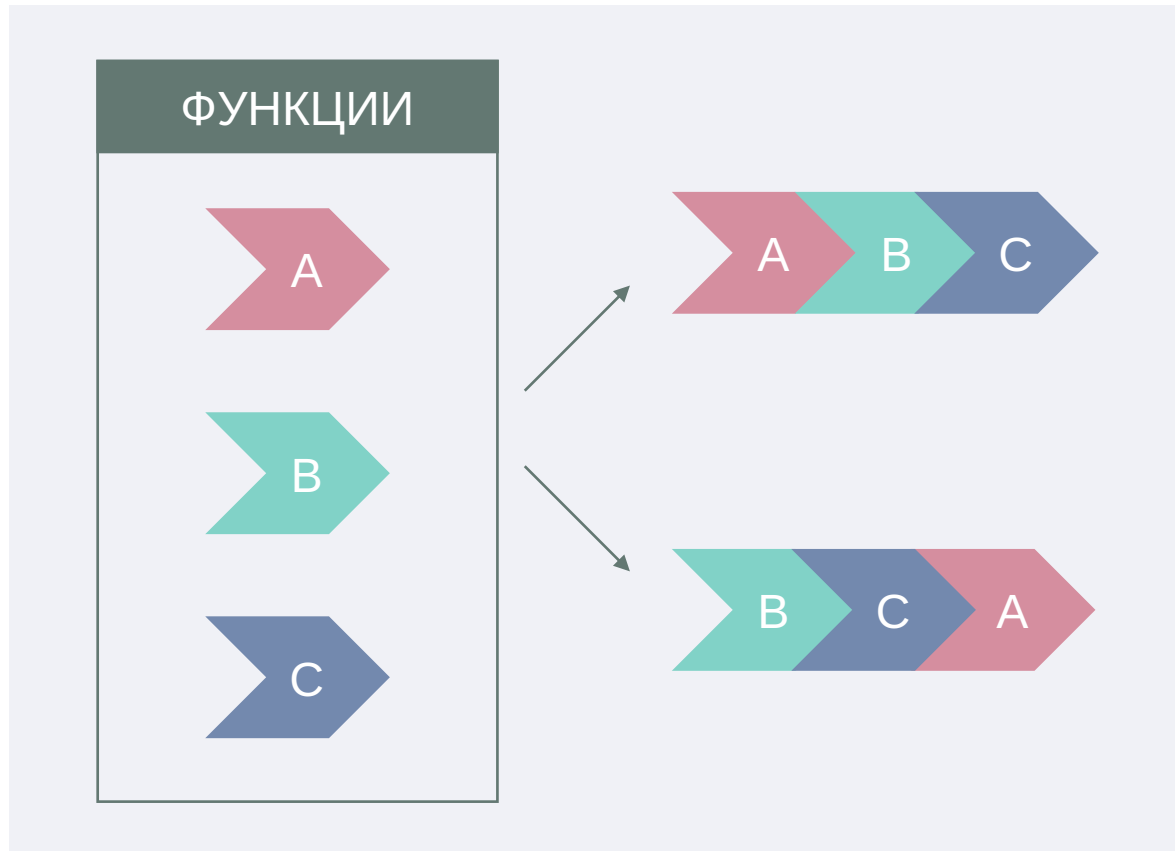
    return 0;
}
```

Функциональное программирование

Haskell

Erlang

F#



- **Весь код состоит из функций,** а также сам может быть воспринят как функция
- **Функции – обособленные объекты**
При одних и тех же входных данных выдают одинаковый результат
- Переменные определяются либо **глобальными константами**, либо **внутри функций**

Например, в программе управления библиотекой:

Работу с книгами выполняют отдельные функции (одна – находит нужную, другая – обновляет данные)
Данные будут локальными внутри вызываемой функции

Функциональное программирование

Haskell

Erlang

F#

Образцы

Функции – полноценные объекты

Их можно присваивать переменным

Функции можно использовать в качестве аргументов

или возвращаемого значения других функций

Функции работают с неизменяемыми структурами данных

Новые создаются при необходимости после операций с существующими

Функции не имеют побочных эффектов

всегда возвращают один и тот же результат для одних и тех же входных данных

Организация кода программы в изолированные функции

Запреты

Изменение данных

Функция не должна влиять ни на что за своими пределами

Циклы

(часто заменяются рекурсией)

Функциональное программирование

- + Чистота и предсказуемость – функции не имеют побочных эффектов
- + Параллелизм – функции могут выполняться независимо друг от друга, что упрощает их распараллеливание
- + Композиция – функции можно легко комбинировать для создания более сложных
- Производительность – может работать медленнее с большими объемами данных или сложными алгоритмами
- Ограничения при необходимости модульности ПО

Прикладные программы

WhatsApp

Поддержка 900 млн. пользователей
50 специалистами с помощью Erlang

Discord

Обработка свыше 1 млн. запросов
каждую минуту с применением Elixir

Функциональное программирование

```
#include <vector>
#include <algorithm>

std::vector<int> filterEvenNumbers(const std::vector<int>& numbers) {
    std::vector<int> evenNumbers;
    std::copy_if(numbers.begin(), numbers.end(), std::back_inserter(evenNumbers),
        [](int num) { return num % 2 == 0; });
    return evenNumbers;
}

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
    auto evenNumbers = filterEvenNumbers(numbers);

    for (int num : evenNumbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}
```

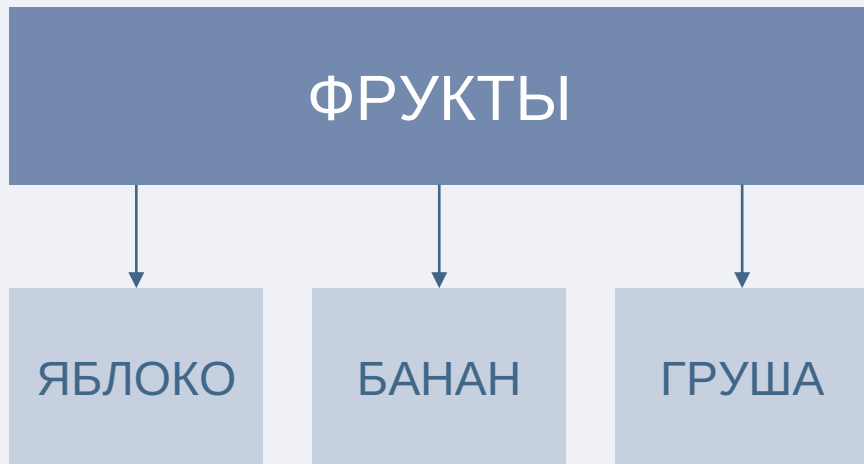
	Процедурное	Функциональное
	Последовательное выполнение инструкций, изменение состояния данных	Преобразование данных с помощью функций, неизменяемость данных
Пример	Сортировка массива пузырьком: последовательно сравниваем элементы и меняем их местами	Фильтрация списка: создаем новый список, отбирая элементы по определенному условию
Фокус	На процессе вычислений: как достичь результата	На результате вычислений: что нужно получить
Изменение данных	Часто изменяет состояние данных (например, массивы)	Стремится к неизменяемости данных, создавая новые данные
Побочные эффекты	Допускает побочные эффекты (например, вывод на экран)	Старается минимизировать побочные эффекты
Области применения	Системное программирование, алгоритмы, игры	Обработка данных, математические вычисления, параллельные вычисления

Объектно-ориентированное программирование (ООП)

C++

Python

C#



- Программы – **совокупности объектов**, взаимодействующих между собой
- Каждый объект – экземпляр определенного класса, который **определяет свойства и поведение** объекта
- Условно основан на **моделировании реального мира**

Например, в программе управления библиотекой:

В ООП будет класс "Книга", который описывает свойства (название, автор, год издания) и методы (например, метод для выдачи книги)

Каждый экземпляр этого класса – конкретная книга

Объектно-ориентированное программирование (ООП)

C++

Python

C#

Образцы

Декомпозиция системы на слабо связанные модули (классы)

Каждый класс отвечает за свою одну конкретную задачу

Методы внутри класса связаны

логически, по смыслу и функционалу

Изменение в одном классе не затрагивают другие классы

Исключение – родительские/наследуемые классы

Создание новых классов на основе существующих,

наследуя их свойства и методы

Ограничение доступа к внутренним членам класса

для обеспечения их целостности

Запреты

Создание God-классов

Неоправданное наследование

Обилие статических методов

Слишком глубокая иерархия классов (6+)

Объектно-ориентированное программирование (ООП)

- + Модульность и повторное использование кода – каждый объект представляет собой самостоятельную сущность
- + Объекты разных классов реагируют на один и тот же вызов по-разному – это делает код гибким и расширяемым
- + ООП позволяет создавать более интуитивно понятные модели реальных объектов и процессов
- Сложность – ООП может быть трудным для понимания и освоения, особенно для начинающих программистов
- Громоздкость – по сравнению с процедурным подходом ООП часто требует написания бОльшего объема кода

Прикладные программы

Научные расчеты
(симуляции и моделирование
физических процессов)

Gamedev

Объектно-ориентированное программирование (ООП)

```
struct Author {
    std::string name, nationality;
};

struct Book {
    std::string title;
    Author author;
    int year;
};

class Library {
public:
    std::vector<Book> books;

    void addBook(const Book& b) { books.push_back(b); }

    Book* findBookByAuthor(const std::string& name) {
        return std::find_if(books.begin(), books.end(),
            [name](const Book& b) { return b.author.name == name;
        })
        .base();
    }
};
```

```
int main() {
    Library library;
    library.addBook({"Преступление и наказание", {"Фёдор
Достоевский", "русский"}, 1866});
    // ... другие книги

    auto book = library.findBookByAuthor("Фёдор Достоевский");
    if (book)
        std::cout << "Найдена книга: " << book->title << '\n';
}
```



Парадигмы программирования

Процедурное, функциональное и объектно-ориентированное

Синтез ООП и других парадигм

И почему именно на ООП делают акцент?

Основные принципы ООП

Наследование, полиморфизм, инкапсуляция

Перегрузка операторов в C++

Почему это необходимо, и какие бывают частые ошибки?



← Мы здесь

Почему ООП остается важным?

Помимо перечисленных плюсов, ООП распространено в сообществе программистов и является знакомым для большинства подходом

А также ООП имеет большую базу инструментов: огромное количество библиотек, фреймворков и даже **обобщенных шаблонов**

А также ООП можно дополнить принципами других парадигм

Ключевые вопросы при выборе парадигмы:

?

Какова природа задачи?

Для числовых вычислений может подойти функциональное программирование, для моделирования реальных объектов – ООП

?

Какие требования к производительности?

Некоторые парадигмы могут быть более эффективны в определенных сценариях

?

Кто будет поддерживать код?

Если команда знакома с определенной парадигмой, это упростит разработку и поддержку

?

Какие инструменты и библиотеки доступны?

Выбор парадигмы может быть ограничен доступными инструментами

?

Какие сроки выполнения?

Важно понимать, что порой написать функциональный код будет преимуществом только на короткой дистанции

Использование **процедурной** парадигмы в ООП

Может улучшить модульность и эффективность ООП-кода

Методы классов

Процедуры могут быть реализованы как методы классов, обеспечивая доступ к данным объекта

Статические методы

Для функций, не связанных с конкретными объектами, можно использовать статические методы классов

Функции-помощники

Вне классов могут быть созданы вспомогательные функции для выполнения общих задач

Lambda-выражения

В некоторых языках (пр-р C++, C#) можно использовать лямбда-выражения для создания анонимных функций

Главное – найти баланс между ООП и процедурным подходом, чтобы не нарушить целостность системы, ясность кода и поддерживаемость

Использование функциональной парадигмы в ООП

Особое значение имеет использование функций внутри классов для чистых и тестируемых методов

Лямбда-выражения

Для создания анонимных функций прямо внутри кода

Высшие порядки функций

Передавать функции как аргументы и возвращать их из функций

Неизменяемость

Оставлять данные неизменяемыми внутри методов для безопасности и скорости работы кода

Рекурсия

Для решения задач, которые естественно выражаются в рекурсивной форме

Важно верно оценить необходимость сохранения промежуточных или неизменяемых данных
Функциональный код уместен, когда мы передаем интерфейсы как аргументы

Пример с Tesla и BigData

Если рассматривать пример с Tesla, то можно выделить:

1. ООП – для моделирования объектов

Автомобиль, датчики, алгоритмы машинного обучения – все это естественно моделируется как объекты реального мира

2. Функциональное программирование – для чистых вычислений

Многие алгоритмы обработки данных могут быть выражены в функциональном стиле, что повышает их чистоту и тестируемость

3. Процедурное программирование – для низкоуровневых операций

Некоторые части системы могут требовать более процедурного подхода для оптимизации производительности



Парадигмы программирования

Процедурное, функциональное и объектно-ориентированное

Синтез ООП и других парадигм

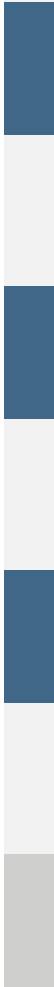
И почему именно на ООП делают акцент?

Основные принципы ООП

Наследование, полиморфизм, инкапсуляция

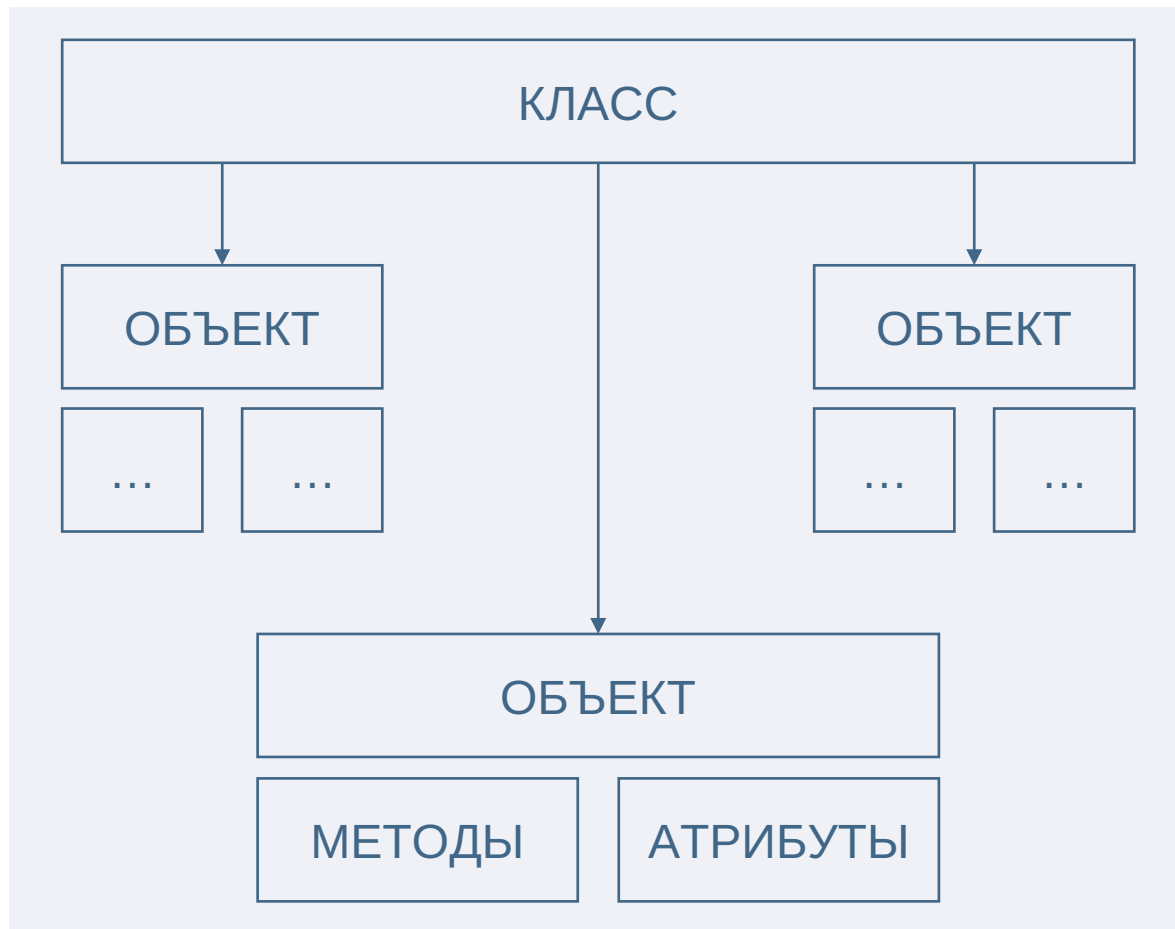
Перегрузка операторов в C++

Почему это необходимо, и какие бывают частые ошибки?



← Мы здесь

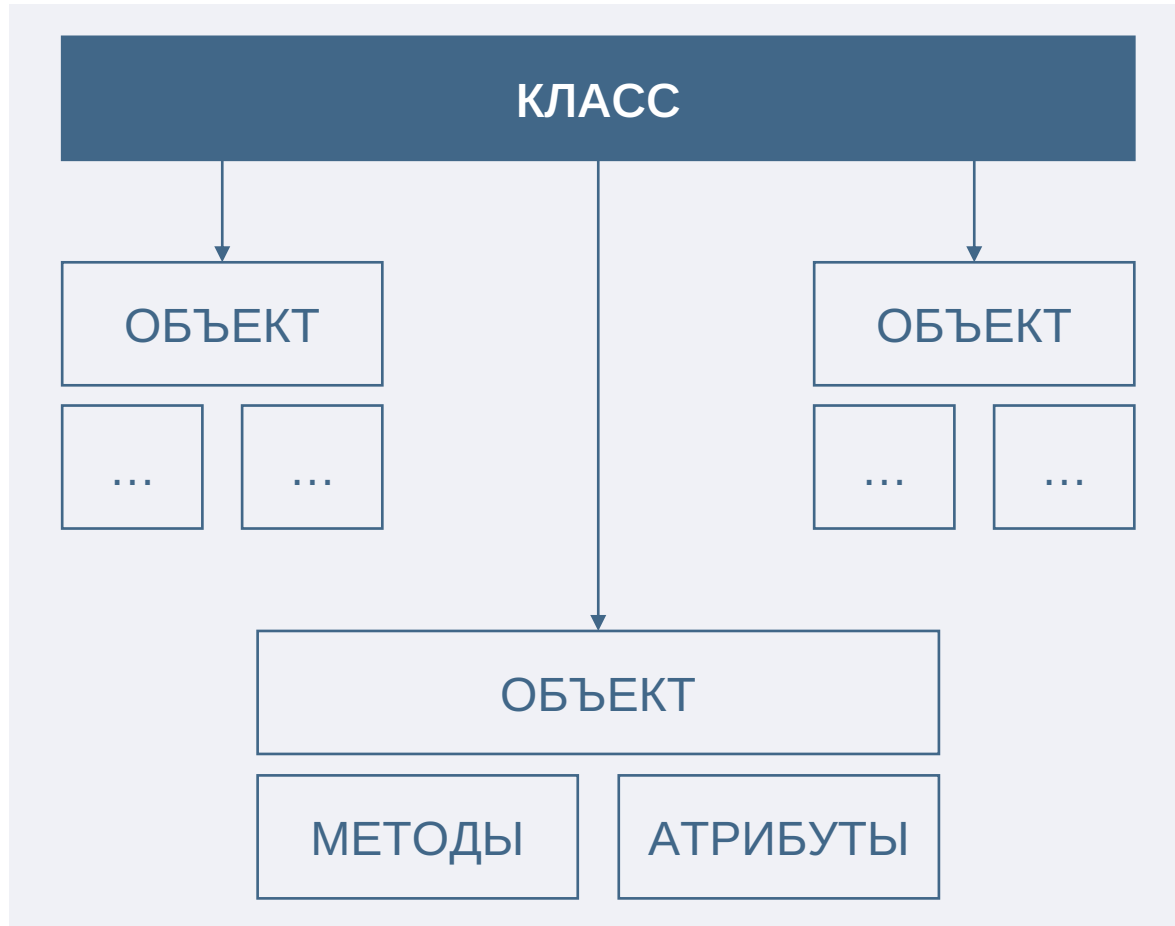
Структура ООП



Выделяют 4 основных элемента:

1. Класс
2. Объект
3. Атрибуты
4. Методы

Структура ООП



Класс – шаблон, на базе которого можно построить Объект в программировании

Например, у интернет-магазина может быть класс «Карточка товара», который описывает общую структуру всех карточек

Классы могут наследоваться друг от друга

Например, есть общий класс «Карточка товара» и вложенные классы, или подклассы: «Карточка бытовой техники», «Карточка ноутбука», «Карточка смартфона»

Структура ООП

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

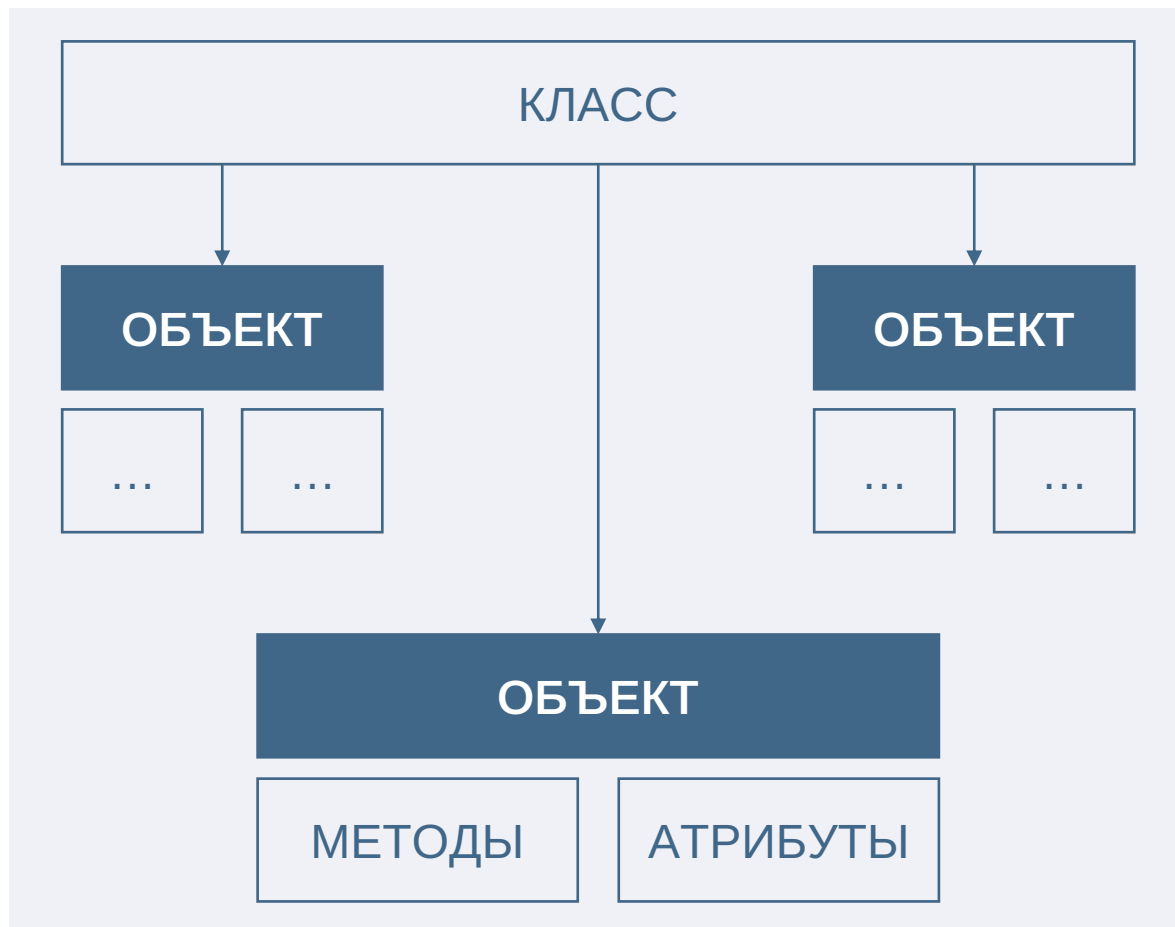
class AbstractProduct {
    virtual void show() const = 0;
};
```

```
// Класс товара
class Product : public
AbstractProduct {
    /* Реализация */
};
```

```
// Класс заказа
class Order {
    std::vector<Product> products;
public:
    void addProduct(const Product&
product) {
        /* Реализация */
    }

    void showOrder() const {
        /* Реализация */
    }
};
```

Структура ООП



Объект – часть кода, которая описывает элемент с конкретными характеристиками и функциями по шаблону Класса

Конкретная карточка товара в каталоге интернет-магазина – это объект
Кнопка «заказать» – тоже

Структура ООП

```
// Класс товара
class Product: public
AbstractProduct {
    std::string name;
    double price;
public:
    Product(std::string n, double p) :
name(n), price(p) {}

    void show() const { /* cout the
Product*/ }

    double getPrice() const { return
price; }
};
```

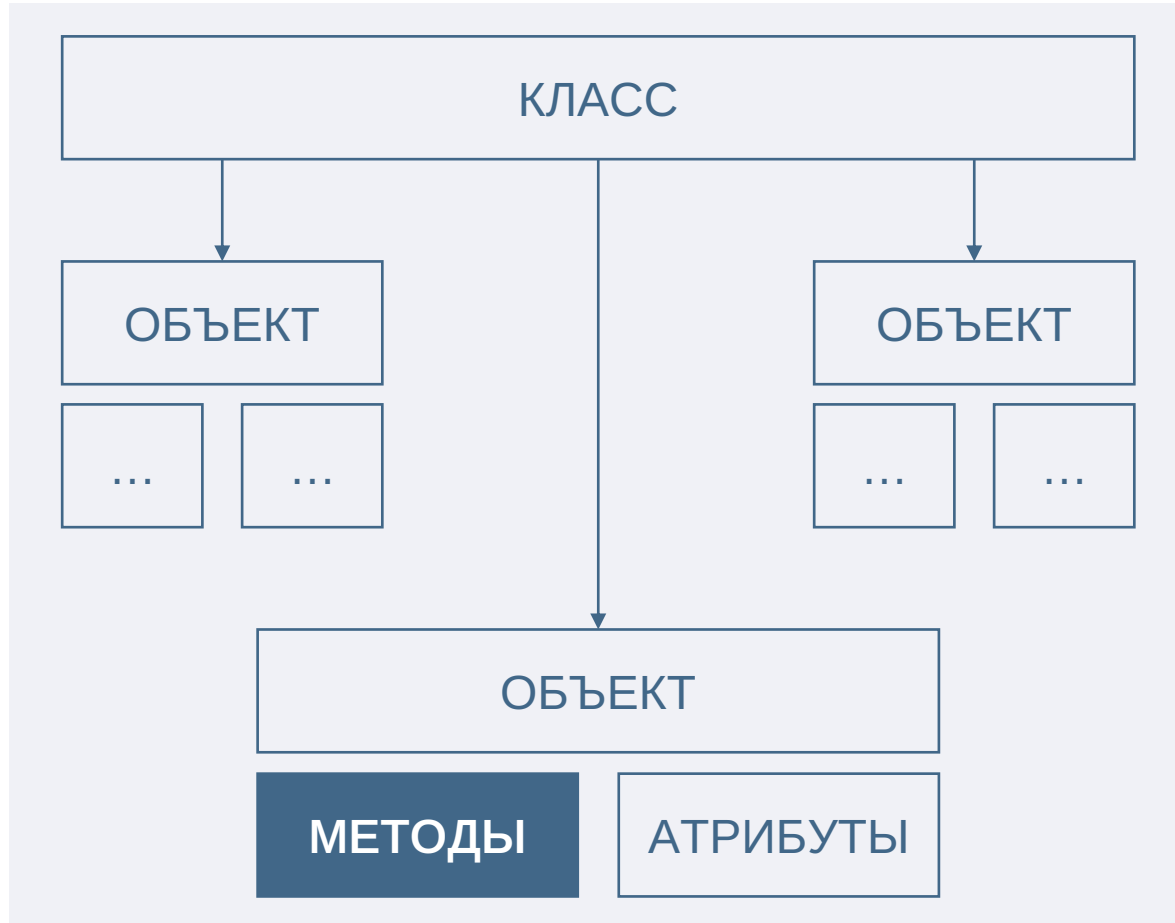
```
// Класс заказа
class Order {
    std::vector<Product> products;
public:
    void addProduct(const Product&
product) {
        products.push_back(product);
    }

    void showOrder() const {
        std::cout << "Заказ
содержит:\n";
        for (const auto& p : products) {
            p.show();
        }
    }
};
```

```
// Класс пользователя
class User {
    std::string name;
public:
    User(std::string name) :
name(name) {}

    void makeOrder(Order& order) {
        std::cout << name << "
оформил заказ.\n";
        order.showOrder();
    }
};
```

Структура ООП

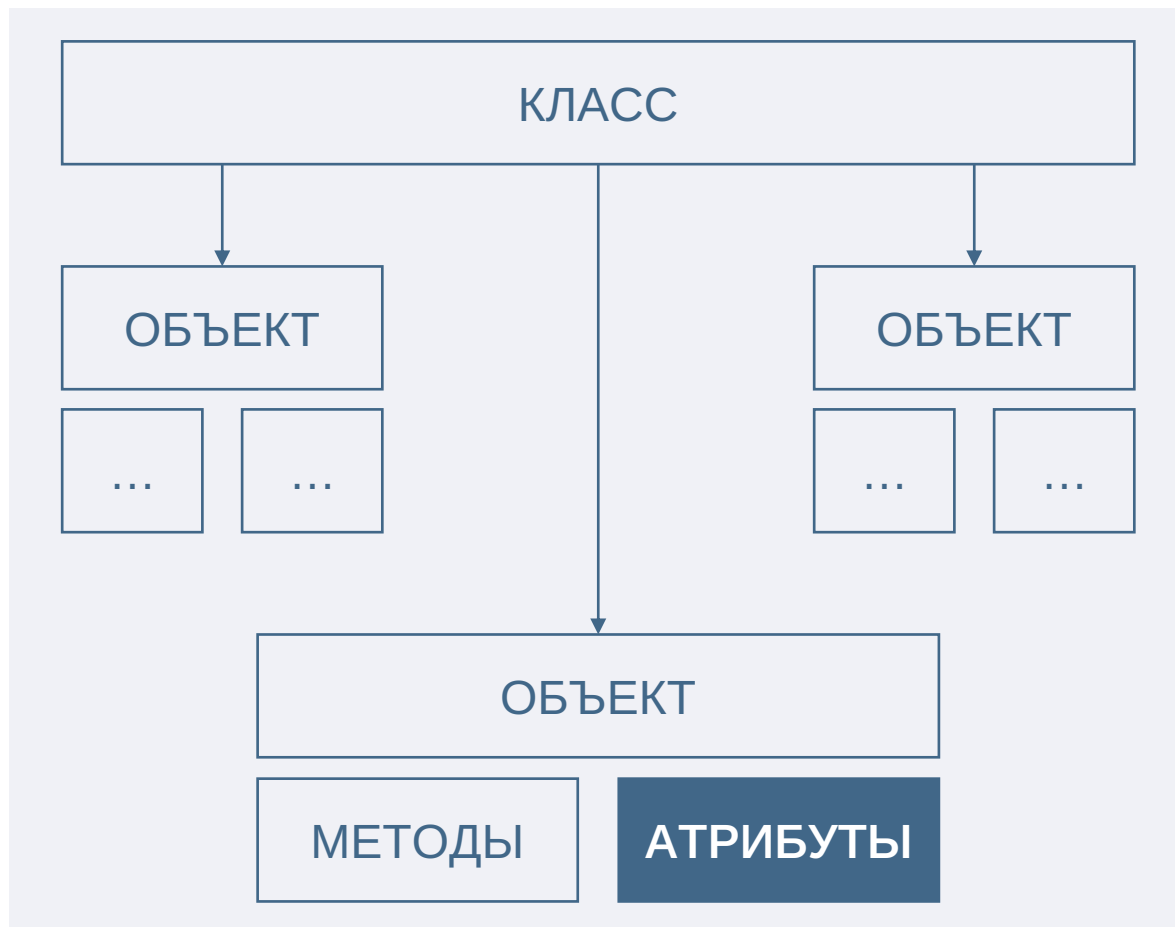


Метод – функция внутри объекта или класса, которая позволяет взаимодействовать с ним или другой частью кода

В примере с карточками товара метод может:

- Обновить количество товара в наличии, сверясь с БД
- Сравнить два товара между собой
- Предложить купить похожие товары

Структура ООП



Атрибут – характеристика объекта
(например, цена, производитель)

В классе прописывают, какие атрибуты могут быть, а в объектах с помощью конструктора и методов заполняют эти атрибуты данными

Структура ООП

Атрибуты

Методы

```
// Класс товара
class Product {
    std::string name;
    double price;
public:
    Product(std::string n, double p) : name(n), price(p) {}

    void show() const { /* cout the Product*/ }

    double getPrice() const { return price; }
};
```

```
// Класс пользователя
class User {
    std::string name;
public:
    User(std::string name) : name(name) {}

    void makeOrder(Order& order) {
        std::cout << name << " оформил заказ.\n";
        order.showOrder();
    }
};
```

```
// Класс заказа
class Order {
    std::vector<Product> products;
public:
    void addProduct(const Product& product) {
        products.push_back(product);
    }

    void showOrder() const {
        std::cout << "Заказ содержит:\n";
        for (const auto& p : products) {
            p.show();
        }
    }
};
```

```
// Демонстрация работы
int main() {
    User user("Иван");
    Product phone("Смартфон", 39990);
    Product laptop("Ноутбук", 79990);

    Order order;
    order.addProduct(phone);
    order.addProduct(laptop);

    user.makeOrder(order);
    return 0;
}
```

Структура ООП

```
class AbstractProduct {
    virtual void show() const = 0;
};
```

```
// Класс товара
class Product: public AbstractProduct {
    std::string name;
    double price;
public:
    Product(std::string n, double p) : name(n), price(p) {}
    void show() const { /* cout the Product */ }
    double getPrice() const { return price; }
};
```

```
// Класс пользователя
class User {
    std::string name;
public:
    User(std::string name) : name(name) {}

    void makeOrder(Order& order) {
        std::cout << name << " оформил заказ.\n";
        order.showOrder();
    }
};
```

```
// Класс заказа
class Order {
    std::vector<Product> products;
public:
    void addProduct(const Product& product) {
        products.push_back(product);
    }

    void showOrder() const {
        std::cout << "Заказ содержит:\n";
        for (const auto& p : products) {
            p.show();
        }
    }
};
```

```
// Демонстрация работы
int main() {
    User user("Иван");
    Product phone("Смартфон", 39990);
    Product laptop("Ноутбук", 79990);

    Order order;
    order.addProduct(phone);
    order.addProduct(laptop);

    user.makeOrder(order);
    return 0;
}
```

Реализация ООП

```
class Item {  
    private:  
        string name;  
        string price;  
  
    public:  
        Item(const std::string &name_,  
             const std::string &price_): name(name_),  
            price(price_) {}  
  
        void card() {  
            std::cout << this->name << " стоит " << this->  
            price << std::endl;  
        }  
}
```

```
class Phone: public Item {  
    public:  
        Phone(const std::string &price): Item("СУПЕР  
ТЕЛЕФОН", price) {}  
  
    }  
  
    ///...  
  
    Phone phone("1 руб");  
    phone.card();  
  
    // СУПЕР ТЕЛЕФОН стоит 1 руб
```

Основные принципы ООП

Объектно-ориентированное программирование базируется на 3 основных принципах:

1. Инкапсуляция

Вся информация, которая нужна для работы конкретного объекта, должна храниться внутри этого объекта

2. Наследование

Каждый дочерний элемент наследует методы и атрибуты, прописанные в родительском

3. Полиморфизм

Один и тот же метод работает по-разному в зависимости от объекта, в котором он вызван, и данных, которые ему передали

Именно они обеспечивают удобство использования этой парадигмы

Основные принципы ООП | Какая цель?

Объектно-ориентированное программирование базируется на 3 основных принципах:

- | | |
|-----------------|---|
| 1. Инкапсуляция | Изоляция реализации от клиента, чтобы не предоставлять лишние данные |
| 2. Наследование | Используется прежде всего для расширения, переопределения и использования готовых функций |
| 3. Полиморфизм | Для создания гибкого и универсального кода |

Именно они обеспечивают удобство использования этой парадигмы

Инкапсуляция

- Вся информация, которая нужна для работы конкретного объекта, должна храниться внутри этого объекта
- Если нужно вносить изменения, методы для этого тоже должны лежать в самом объекте — посторонние объекты и классы этого делать не могут
- Для внешних объектов доступны только публичные атрибуты и методы

В C++ для управления доступом к членам класса используются модификаторы доступа:

public: Члены, объявленные как `public`, доступны из любой части программы.

private: Члены, объявленные как `private`, доступны только внутри класса.

protected: Члены, объявленные как `protected`, доступны внутри класса и его производных классов

Скрытие данных, то есть объявление большинства полей класса как `private`, обеспечивает безопасность и целостность данных, упрощенную отладку и улучшенную модульность

Инкапсуляция – геттеры и сеттеры

для контролируемого доступа к приватным данным

Геттер (getter) возвращает значение поля

```
class Person {  
  
    private:  
        std::string name;  
        int age;  
  
    public:  
        Person(const std::string& n, int a) : name(n), age(a) {}  
  
        std::string getName() const { return name; }
```

Сеттер (setter) устанавливает новое значение

```
void setName(const std::string& n) { name = n; }  
  
int getAge() const { return age; }  
void setAge(int a) {  
    if (a >= 0) {  
        age = a;  
    } else {  
        // Обработка ошибки  
    }  
};
```

В этом примере поля `name` и `age` объявлены как `private`, а для доступа к ним используются геттеры `getName()` и `setAge()`. Сеттер `setAge()` проверяет, что новое значение возраста неотрицательно, обеспечивая таким образом целостность данных.

Инкапсуляция

```
class BankAccount {  
private:  
    double balance; // Приватное поле, недоступное напрямую извне  
  
public:  
    BankAccount(double initialBalance) {  
        if (initialBalance < 0) {  
            balance = 0;  
        } else {  
            balance = initialBalance;  
        }  
    }  
  
    void deposit(double amount) {  
        if (amount > 0) {  
            balance += amount;  
            std::cout << "Пополнение на " << amount << ". Новый баланс: " << balance <<  
std::endl;  
        } else {  
            std::cout << "Сумма пополнения должна быть положительной!" << std::endl;  
        }  
    }  
}
```

```
bool withdraw(double amount) {  
    if (amount > 0 && amount <= balance) {  
        balance -= amount;  
        std::cout << "Снятие " << amount << ". Остаток: " << balance << std::endl;  
        return true;  
    } else {  
        std::cout << "Недостаточно средств или некорректная сумма!" << std::endl;  
        return false;  
    }  
}  
  
double getBalance() const {  
    return balance;  
}  
};  
  
int main() {  
    BankAccount account(1000); // Создаем счет с начальным балансом 1000  
    account.deposit(500); // Пополняем на 500  
    account.withdraw(200); // Снимаем 200  
    account.withdraw(2000); // Пытаемся снять больше, чем есть на счету  
}
```


Наследование – вся суть ООП

- Каждый дочерний элемент наследует методы и атрибуты, прописанные в родительском
- Он может использовать их все, отбросить часть или добавить новые
- При этом заново прописывать эти атрибуты и методы не нужно

Разработчик создаёт класс X с определёнными свойствами



Подкласс на основе класса X, который берёт свойства класса и добавляет свои



Объект подкласса, который также копирует свойства класса X и добавляет свои

Многоуровневое наследование

Допускается C++, когда класс может наследовать свойства от нескольких базовых классов

Однако, управление иерархией классов усложняется, так что следует использовать с осторожностью

Наследование

```
// Базовый класс Animal
class Animal {
protected:
    std::string name;

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

    void introduce() {
        std::cout << "Меня зовут " << name << ". " << std::endl;
    }

    virtual void makeSound() { // Виртуальный метод для полиморфизма
        std::cout << "Какой-то звук..." << std::endl;
    }
};

// Производный класс Dog (наследуется от Animal)
class Dog : public Animal {
public:
    Dog(std::string n) : Animal(n) {}

    void makeSound() override { // Переопределение метода
        std::cout << "Гав-гав!" << std::endl;
    }
};
```

```
// Производный класс Cat (наследуется от Animal)
class Cat : public Animal {
public:
    Cat(std::string n) : Animal(n) {}

    void makeSound() override {
        std::cout << "Мяу-мяу!" << std::endl;
    }
};

int main() {
    Dog dog("Бобик");
    Cat cat("Мурка");

    dog.introduce();
    dog.makeSound();

    cat.introduce();
    cat.makeSound();

    return 0;
}
```

Полиморфизм

Полиморфизм позволяет объектам одного типа вести себя как объекты другого типа, предоставляя возможность использовать один интерфейс для работы с разными типами данных и делая его более универсальным

- Один и тот же метод работает по-разному в зависимости от объекта, в котором он вызван, и данных, которые ему передали

Разделяется на два типа:

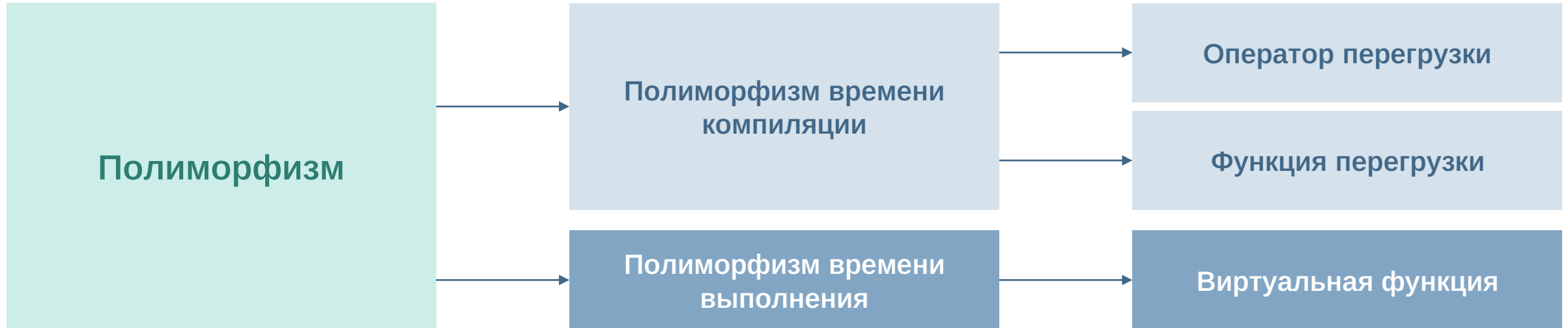
Статический

Определяется во время компиляции
(через перегрузку функций и операторов)

Динамический

Определяется во время выполнения программы
(через виртуальные функции и наследование)

Полиморфизм



Виртуальные функции позволяют реализовывать способность объектов разных классов реагировать на один и тот же вызов метода по-разному

Динамическое связывание позволяет определить, какая именно функция будет вызвана во время выполнения программы, в зависимости от фактического типа объекта.

Статический Полиморфизм

Перегрузка функций и операторов (Function|Operator Overloading)

Это механизм, который позволяет в одном классе или области видимости иметь несколько функций с одинаковым именем, но с разными параметрами (по количеству или типу)

Компилятор выбирает подходящую функцию на основе типов переданных аргументов на этапе компиляции

```
int add(int a, int b) {  
    return a + b;  
}  
  
double add(double a, double b) {  
    return a + b;  
}  
  
int main() {  
    cout << add(3, 4) << endl;    // int  
    cout << add(3.5, 4.2) << endl; // double  
    return 0;  
}
```

Статический Полиморфизм

Шаблоны функций (Function Templates)

Позволяют создавать универсальные функции, которые могут работать с любыми типами данных

Шаблон функции создается с использованием параметров типа, которые затем заменяются конкретными типами на этапе компиляции

```
// Шаблон функции для нахождения максимума
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << max(3, 5) << endl;    // для int
    cout << max(3.2, 5.1) << endl; // для double
    cout << max('a', 'z') << endl; // для char
    return 0;
}
```

Статический Полиморфизм

Шаблоны классов (Class Templates)

Аналогичны шаблонам функций, но создают универсальные классы

Это позволяет создавать типы данных, которые могут работать с разными типами данных, обеспечивая их гибкость

```
// Шаблон класса для контейнера
template <typename T>
class Box {
private:
    T value;
public:
    Box(T v) : value(v) {}
    T getValue() { return value; }
};

int main() {
    Box<int> intBox(10);
    Box<double> doubleBox(3.14);
    cout << intBox.getValue() << endl;
    cout << doubleBox.getValue() << endl;
    return 0;
}
```

Динамический полиморфизм | Основные понятия

Виртуальные функции:

- Функции, которые могут быть переопределены в производных классах
- Для того чтобы механизм полиморфизма работал, нужно пометить метод как виртуальный с помощью ключевого слова **virtual** в базовом классе

Переопределение функций:

- В производных классах можно изменить поведение виртуальных функций, используя ключевое слово **override**

Использование указателей или ссылок на базовый класс:

- Для использования динамического полиморфизма необходимо работать с указателями или ссылками на базовый класс, чтобы вызвать переопределенную функцию производного класса

Динамический полиморфизм

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

// Базовый класс
class Animal {
public:
    virtual void speak() const {
        cout << "Animal makes a sound" <<
endl;
    }

    // Виртуальный деструктор
    virtual ~Animal() = default;
};

// Производный класс Dog
class Dog : public Animal {
public:
    void speak() const override {
        cout << "Woof!" << endl;
    }
};
```

```
// Производный класс Cat
class Cat : public Animal {
public:
    void speak() const override {
        cout << "Meow!" << endl;
    }
};

// Функция, использующая полиморфизм
void makeSound(const Animal& animal) {
    animal.speak(); // Вызов функции speak
на основе реального типа объекта
}

int main() {
    Dog dog;
    Cat cat;

    makeSound(dog); // Woof!
    makeSound(cat); // Meow!
```

```
// Создаем объекты через указатели
Animal* animal1 = new Dog();
Animal* animal2 = new Cat();

animal1->speak(); // Woof!
animal2->speak(); // Meow!

// Очистка памяти
delete animal1;
delete animal2;

return 0;
}
```

Динамический полиморфизм

Например, вызов функции `area()` для объекта типа `Circle` будет вызывать реализацию этой функции, определенную в классе `Circle`, а для объекта типа `Square` – реализацию из класса `Square`

```
class Figure {  
    protected:  
        std::string color;  
  
    public:  
        Figure(const std::string& c) : color(c) {}  
  
        virtual double area() = 0; // Чисто виртуальная  
        функция  
  
        std::string getColor() const { return color; }  
};
```

```
class Circle : public Figure {  
  
    private:  
        double radius;  
    public:  
        Circle(const std::string& c, double r) : Figure(c),  
            radius(r) {}  
  
        double area() override { return 3.14 * radius *  
            radius; }  
};
```

Динамический полиморфизм

Абстрактный класс – это главный компонент для реализации динамического полиморфизма из-за возможности замены реализации при не изменении типа переменной

```
#include <vector>

std::vector<Figure*> figures;

figures.push_back(new Circle("красный", 5));
figures.push_back(new Square("синий", 4));

for (Figure* figure : figures) {
    std::cout << "Площадь фигуры: " << figure->area() << std::endl;
}
```



Парадигмы программирования

Процедурное, функциональное и объектно-ориентированное

Синтез ООП и других парадигм

И почему именно на ООП делают акцент?

Основные принципы ООП

Наследование, полиморфизм, инкапсуляция

Перегрузка операторов в C++

Почему это необходимо, и какие бывают частые ошибки?



Мы здесь

Зачем перегружать операторы?

Перегрузка операторов в C++ позволяет придать привычным арифметическим, логическим и другим операторам новое значение для пользовательских типов данных (классов)

Это делает код более читаемым и интуитивно понятным

- **Интуитивность**

Когда мы видим выражение $a + b$, где a и b — объекты класса, мы ожидаем, что результатом будет сумма этих объектов. Перегрузка оператора $+$ позволяет реализовать такое поведение

- **Упрощение кода**

Вместо громоздких вызовов функций, мы можем использовать привычную арифметическую запись

- **Совместимость с существующим кодом**

Наши пользовательские типы данных можно интегрировать в стандартные алгоритмы, которые работают со встроенными типами

Подходы и ошибки

- **Сохранение интуитивности**

Перегруженный оператор должен вести себя так, как ожидается от него в контексте данного типа данных

- **Симметрия**

Если оператор коммутативен (например, +), то перегрузка должна быть симметричной

- **Консистентность**

Перегруженные операторы должны работать согласованно друг с другом

- **Эффективность**

Перегрузка не должна существенно снижать производительность

Необходимо быть аккуратным при перегрузке операторов присваивания и копирования

Неправильная перегрузка этих операторов может привести к утечкам памяти и другим проблемам



Доступные операторы для перегрузки

В C++17 стандарт разрешает перегружать следующие операторы:

+	-	*	/	%	^	&		~	!	=
<	>	<=	>=	++	--	<<	>>	==	!=	&&
	+=	--	/=	%=	^=	&=	=	*=	<<=	>>=
[]	()	->	->*	new	new[]	delete	delete[]			

Хороший пример перегрузки – использование операторов + и += для конкатенации экземпляров `std::basic_string<>`

Интересное решение используется в классе `std::filesystem::path` (C++17)

В этом классе операторы / и /= перегружены для управлением элементов пути в файловой системе

Операторы, которые не рекомендуется перегружать

Три бинарных оператора:

, (запятая)

&&

||

В блоке if нарушен порядок исполнения оператора &&

Вместо того, чтобы опустить вызов перегруженного оператора (так как проверяемое выражение всегда False) и сразу перейти в блок False, цикл сначала вызвал перегруженный оператор, исполнил его и уже после запустил тело if

Нарушается порядок исполнения других операций
и не гарантируется корректное исполнение выражений

```
class BoolWrapper {
public:
    bool value;

    BoolWrapper(bool val) : value(val) {}

    // Перегружаем оператор &&
    bool operator&&(const BoolWrapper& other) const {
        std::cout << "Вызван оператор &&\n";
        return (value && other.value);
    }
};

int main() {
    BoolWrapper a(false);
    BoolWrapper b(true);

    if (a && b) { // Ожидаем, что b не будет вычисляться
        std::cout << "Истина!\n";
    } else {
        std::cout << "Ложь!\n";
    }
}
```


Корректный код

Вместо перегрузки оператора приводим класс к bool

```
class BoolWrapper {  
public:  
    bool value;  
  
    BoolWrapper(bool val) : value(val) {}  
  
    // Приведение к bool  
    explicit operator bool() const {  
        return value;  
    }  
};
```

```
int main() {  
    BoolWrapper a(false);  
    BoolWrapper b(true);  
  
    if (a && b) { // Ожидаем, что b не будет  
        // вычисляться  
        std::cout << "Истина!\n";  
    } else {  
        std::cout << "Ложь!\n";  
    }  
}
```

Реализация операторов

```
#include <iostream>
#include <cmath>

class Power {
private:
    double exponent;
public:
    Power(double exponent) :
        exponent(exponent) {}

    // Перегруженный оператор ()
    double operator()(double base) const {
        return std::pow(base, exponent);
    }
};
```

```
int main() {
    Power square(2); // Объект для
    возведения в квадрат
    Power cube(3); // Объект для
    возведения в куб

    std::cout << "5^2 = " << square(5) <<
    std::endl; // 25
    std::cout << "4^3 = " << cube(4) <<
    std::endl; // 64
}

#include <iostream>

class Wrapper {
private:
    int value;
public:
    Wrapper(int val) : value(val) {}
    void show() { std::cout << "Value: " <<
        value << std::endl; }
};
```

```
class SmartPointer {
private:
    Wrapper* ptr;
public:
    SmartPointer(Wrapper* p) : ptr(p) {}
    ~SmartPointer() { delete ptr; }

    // Перегрузка ->
    Wrapper* operator->() { return ptr; }
};

int main() {
    SmartPointer sp(new Wrapper(42));
    sp->show(); // Развозначно (*sp).show()
}
```

Примеры перегрузки операторов

Класс комплексных чисел

```
#include <iostream>

class Complex {
public:
    double real, imag;

    // Конструктор
    Complex(double r = 0, double i = 0) : real(r), imag(i)
    {}

    // Перегрузка оператора +
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag +
other.imag);
    }
}
```

```
// Перегрузка оператора << для вывода на
консоль
friend std::ostream& operator<<(std::ostream& os,
const Complex& c) {
    os << c.real << " + " << c.imag << "i";
    return os;
}

int main() {
    Complex c1(2, 3), c2(4, -1);
    Complex c3 = c1 + c2;
    std::cout << c3 << std::endl; // Вывод: 6 + 2i
}
```

Примеры перегрузки операторов

Сравнение строк, игнорируя регистр

```
#include <string>

class CaseInsensitiveString {
public:
    std::string str;

    // Перегрузка оператора ==
    bool operator==(const CaseInsensitiveString& other) const {
        return std::equal(str.begin(), str.end(), other.str.begin(), other.str.end(),
            [](char a, char b) { return std::tolower(a) == std::tolower(b); });
    }
};
```

SpaceShip Оператор C++20 (<=>)

Оператор трехстороннего сравнения <=> был введен в C++20 для сокращения шаблонного кода при перегрузке операторов сравнения

`std::strong_ordering`

- Среди наборов сопоставимых объектов существует полный порядок
- Несравнимые объекты не допускаются
- Два эквивалентных объекта также являются взаимозаменяемыми, поскольку ведут себя одинаково (вероятно, все члены также эквивалентны)

Допустимые возвращаемые значения:

- `std::strong_ordering::less`
- `std::strong_ordering::equivalent`
- `std::strong_ordering::greater`

SpaceShip Оператор C++20 (<=>)

Оператор трехстороннего сравнения <=> был введен в C++20 для сокращения шаблонного кода при перегрузке операторов сравнения

`std::weak_ordering`

- Подобно `std::strong_ordering`, но два объекта, которые эквивалентны, не обязательно являются взаимозаменяемыми
- Поскольку они могут вести себя по-разному (вероятно, сравнение включает только подмножество членов, а другие члены могут быть разными)

Допустимые возвращаемые значения:

- `std::weak_ordering::less`
- `std::weak_ordering::equivalent`
- `std::weak_ordering::greater`

SpaceShip Оператор C++20 (<=>)

Оператор трехстороннего сравнения <=> был введен в C++20 для сокращения шаблонного кода при перегрузке операторов сравнения

`std::partial_ordering`

- Среди наборов сопоставимых объектов существует частичный порядок
- Несравнимые объекты разрешены, и для них есть специальное возвращаемое значение
- Как и `std::weak_ordering`, два объекта, которые эквивалентны, не обязательно являются взаимозаменяемыми

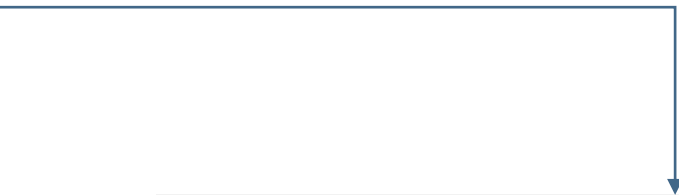
Допустимые возвращаемые значения:

- `std::partial_ordering::less`
- `std::partial_ordering::equivalent`
- `std::partial_ordering::greater`
- `std::partial_ordering::unordered` (в случае несравнимости элементов)

SpaceShip Оператор C++20 (<=>)

Оператор трехстороннего сравнения <=> был введен в C++20 для сокращения шаблонного кода при перегрузке операторов сравнения

```
struct MyStruct {  
    int x;  
  
    std::weak_ordering operator<=>(const MyStruct& o) {  
        if (x < o.x) {  
            return std::weak_ordering::less;  
        } else if (x == o.x) {  
            return std::weak_ordering::equivalent;  
        } else {  
            return std::weak_ordering::greater;  
        }  
    }  
    bool operator==(const MyStruct& o) const {  
        return x == o.x;  
    }  
};
```



```
auto operator<=>(const MyStruct&) const = default;
```


SpaceShip Оператор C++20 (<=>)

Оператор трехстороннего сравнения <=> был введен в C++20 для сокращения шаблонного кода при перегрузке операторов сравнения

```
struct MyStruct {  
    int x;  
  
    auto operator<=>(const MyStruct& o) const =  
    default;  
  
    bool operator==(const MyStruct& o) const {  
        return x == o.x;  
    }  
};
```

```
int main() {  
    MyStruct x{1}, y{0};  
    std::cout << (x == y) << ' ';  
    std::cout << (x != y) << ' ';  
    std::cout << (x < y) << ' ';  
    std::cout << (x <= y) << ' ';  
    std::cout << (x > y) << ' ';  
    std::cout << (x >= y) << std::endl;  
    return 0;  
}
```

ИТОГОВЫЕ СОВЕТЫ

Перегружайте только те операторы, которые имеют смысл для вашего типа данных

Тщательно продумывайте поведение перегруженного оператора

Тестируйте код с перегруженными операторами

Используйте константные ссылки для аргументов, когда это возможно

Избегайте перегрузки операторов, которые могут привести к неоднозначности

Спасибо за внимание!
Ваши вопросы?

