

Технологии программирования

Лекция 4 | Управление ресурсами и памятью



Управление памятью и исключения

И связь данных механизмов с ООП

ООП как основа архитектуры

Проблемы проектирования, которые решает ООП

Шаблоны и их связь с ООП

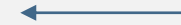
Варианты их применения

Оптимизация шаблонов

Встраивание, выражения шаблонов и сокращение временных объектов

Post-OOP

Новые подходы к проектированию



Мы здесь



Зачем управлять ресурсами

Управление памятью – это процесс распределения, использования и освобождения оперативной памяти (RAM) во время выполнения программы

Можно выделить три основных действия:

Выделение памяти
под объекты, переменные,
структуры данных

Контроль доступа
к памяти (чтение / запись)

Освобождение памяти,
когда она больше не нужна,
чтобы избежать утечек

! С помощью грамотного подхода к управлению памятью можно увеличить производительность и безопасность программы

Зачем управлять ресурсами

1

Эффективность ресурсов:

Оперативная память ограничена
Некорректное управление приводит к её исчерпанию
(например, утечки памяти)

Пример: Серверное приложение, работающее 24/7, должно минимизировать утечки, иначе через несколько дней/недель произойдёт сбой

2

Безопасность:

Уязвимости в коде могут скомпрометировать данные
всего приложения

Пример: Уязвимости в C/C++ программах
(например, Heartbleed в OpenSSL)

3

Производительность:

Неоптимальное выделение памяти (например,
фрагментация) замедляет выполнение программы

Пример: Игры или системы реального времени
требуют предсказуемого времени выполнения
операций

4

Детерминизм:

В embedded-системах или робототехнике важно
гарантировать, что память будет доступна в критический
момент

Пример: Микрокод процессора, контроллеры памяти,
управление двигателем

Виды управления памятью

Автоматическое

(высокоуровневые языки)

Гибридное

Ручное

(низкоуровневые языки)

Автоматическое управление памятью

Python

Java

C#

JavaScript

Присутствует в высокоуровневых языках

Механизмы:

- **Сборка мусора**
Автоматическое освобождение неиспользуемой памяти
- **Менеджер памяти в рантайме**
Выделение памяти скрыто от разработчика (например, list в Python)

Плюсы

Упрощает разработку, снижает риск утечек

Минусы

Накладные расходы на GC,
меньше контроля над производительностью

Ручное управление памятью

C

C++

Rust

Присутствует в низкоуровневых языках

Механизмы:

- Явное выделение/освобождение памяти (new / delete)
- RAII
- Прямая работа с указателями

Плюсы

Полный контроль, высокая производительность

Минусы

Риск ошибок (утечки, висячие указатели)

Гибридное управление памятью

Rust

Может присутствовать как в низкоуровневых языках, так и в высокоуровневых

Механизмы:

- Система владения (ownership) и заимствования (borrowing)
Контроль памяти на этапе компиляции без сборщика мусора
- Автоматическое освобождение через Drop-траиты

Плюсы

Гарантия безопасности
и высокой производительности кода

Минусы

Сложность проектирование кода

RAII в C++

RAII – это способ автоматизировать управление ресурсами в C++, когда ресурс выделяется при создании объекта и автоматически освобождается при его уничтожении

Это работает за счет конструкторов и деструкторов:

1. Конструктор при создании объекта выделяет ресурс и связывает ее с объектом
2. Деструктор при уничтожении объекта автоматически освобождает связанный ресурс

Примеры использования:

- Умные указатели (`std::unique_ptr`, `std::shared_ptr` автоматически освобождают память при выходе из области видимости)
- Объекты, представляющие файлы (файл открывается в конструкторе и закрывается в деструкторе)
- Сокеты (соединение устанавливается в конструкторе и разрывается в деструкторе)

Проблемы, которые решает RAII

Утечки памяти

Без RAII разработчику приходится полностью вручную отслеживать и освобождать выделенную память, а забытая операция освобождения может привести к утечкам

Неопределенное поведение

Если ресурсы не управляются должным образом, это может привести к неопределенному поведению программы
RAII гарантирует, что ресурсы всегда находятся в определенном состоянии

Исключения и безопасность

RAII позволяет обрабатывать исключения более элегантно и безопасно
Ресурсы будут автоматически освобождены даже в случае возникновения исключительных ситуаций

Читаемость кода

Использование RAII делает код более читаемым и понятным, так как связь между ресурсами и объектами становится очевидной

Поддержка стандартных контейнеров

Многие стандартные контейнеры C++ (например, `std::vector` и `std::string`) используют RAII для управления памятью, что обеспечивает их безопасность и эффективность

Использование RAII для управления ресурсами

```
#include <fstream>

void processFile(const std::string& filename) {
    std::ifstream file(filename); // Файл автоматически закроется при выходе из области
видимости

    // Работа с файлом
    // ...

    if (!file.is_open()) {
        throw std::runtime_error("Failed to open file");
    }
}
```

Использование RAII для управления ресурсами

```
class ArrayWrapper {  
public:  
    ArrayWrapper(int size) : arr(new int[size]) {} // Выделение памяти  
    ~ArrayWrapper() { delete[] arr; } // Освобождение памяти  
private:  
    int* arr;  
};  
  
// Использование  
{  
    ArrayWrapper arr(10); // Память выделена  
    // ... работа с массивом ...  
}
```

Правило 3

Говоря о работе с классами, следует следовать следующим правилам 3 и 5:

Если классу нужен один из следующих трех методов, то, скорее всего, ему понадобятся и два других

Деструктор

Конструктор копирования

Оператор присваивания
копированием

Правило 3

Было

```
class BrokenResource {
public:
    char* data;

    BrokenResource(const char* input) {
        data = new char[strlen(input) + 1];
        strcpy(data, input);
    }

    ~BrokenResource() {
        delete[] data;
    }
    // конструктор копирования
    и оператор присваивания
    не определены!
};
```

Должно быть

```
class FixedResource {
public:
    char* data;

    FixedResource(const char* input) {
        data = new char[strlen(input) + 1];
        strcpy(data, input);
    }

    ~FixedResource() {
        delete[] data;
    }

    // конструктор копирования
    FixedResource(const FixedResource&
other) {
        data = new char[strlen(other.data) +
1];
```

```
        strcpy(data, other.data);
    }

    // оператор присваивания
    FixedResource& operator=(const
FixedResource& other) {
        if (this != &other) { //
предотвращение самоприсваивания
            delete[] data; // освобождаем
существующий ресурс
            data = new
char[strlen(other.data) + 1];
            strcpy(data, other.data);
        }
        return *this;
    }
};
```

Правило 5

Если явно определяется один из следующих пяти специальных методов класса, вам, скорее всего, нужно явно определить и остальные четыре

Деструктор

Конструктор копирования

Оператор присваивания
копированием

Конструктор перемещения

Оператор присваивания
перемещения

Правило 5

Было

```
class ResourceHolder {
public:
    int* data;

    ResourceHolder(int value) : data(new
int(value)) {}
    ~ResourceHolder() { delete data; }

    // Правило пяти не соблюдено:
    отсутствуют конструктор
    копирования,
    // оператор присваивания
    копированием, конструктор
    перемещения и
    // оператор присваивания
    перемещением.
};
```

Должно быть

```
class ProperResource {
public:
    int* data;

    ProperResource(int value) : data(new
int(value)) {}
    ~ProperResource() { delete data; }

    ProperResource(const
ProperResource& other) : data(new
int(*other.data)) {}

    // оператор присваивания
    копированием
    ProperResource& operator=(
ProperResource& other) {
        *data = *other.data;
    }
}
```

```
ProperResource(ProperResource&&
other) noexcept : data(other.data) {
    other.data = nullptr;
}

// оператор присваивания
перемещением
ProperResource&
operator=(ProperResource&& other)
noexcept {
    if (this != &other) {
        delete data;
        data = other.data;
        other.data = nullptr;
    }
    return *this;
};
```


Проблемы Ручного управления памятью

```
void leak() {  
    int* ptr = new int[100];  
    // ... не delete[] ptr ...  
} // Утечка 400 байт (если sizeof(int) = 4)
```

```
int* ptr = new int;  
delete ptr;  
delete ptr; // Ошибка: undefined behavior!
```

```
int* ptr = new int(5);  
int* ptr2 = ptr;  
delete ptr;  
*ptr2 = 10; // Опасное обращение к несуществующей памяти!
```

Умные указатели

Умные указатели – это объекты, которые автоматически управляют временем жизни других объектов, они помогают предотвращать утечки памяти и упрощают управление ресурсами

`std::unique_ptr:`

Уникальное владение объектом
в данный момент времени

`std::shared_ptr:`

Разделяемое владение объектом
Удаление, когда счетчик ссылок будет 0

`std::weak_ptr:`

Слабая ссылка на объект, управляемый
`std::shared_ptr`

```
class Resource {  
public:  
    Resource() { std::cout << "Resource created\n"; }  
    ~Resource() { std::cout << "Resource destroyed\n"; }  
};  
  
int main() {  
    std::unique_ptr<Resource> resource(new Resource);  
}
```



Управление памятью и исключения

И связь данных механизмов с ООП

ООП как основа проектирования

Проблемы проектирования, которые решает ООП

Шаблоны и их связь с ООП

Варианты их применения

Оптимизация шаблонов

Встраивание, выражения шаблонов и сокращение временных объектов

Post-OOP

Новые подходы к проектированию



Мы здесь

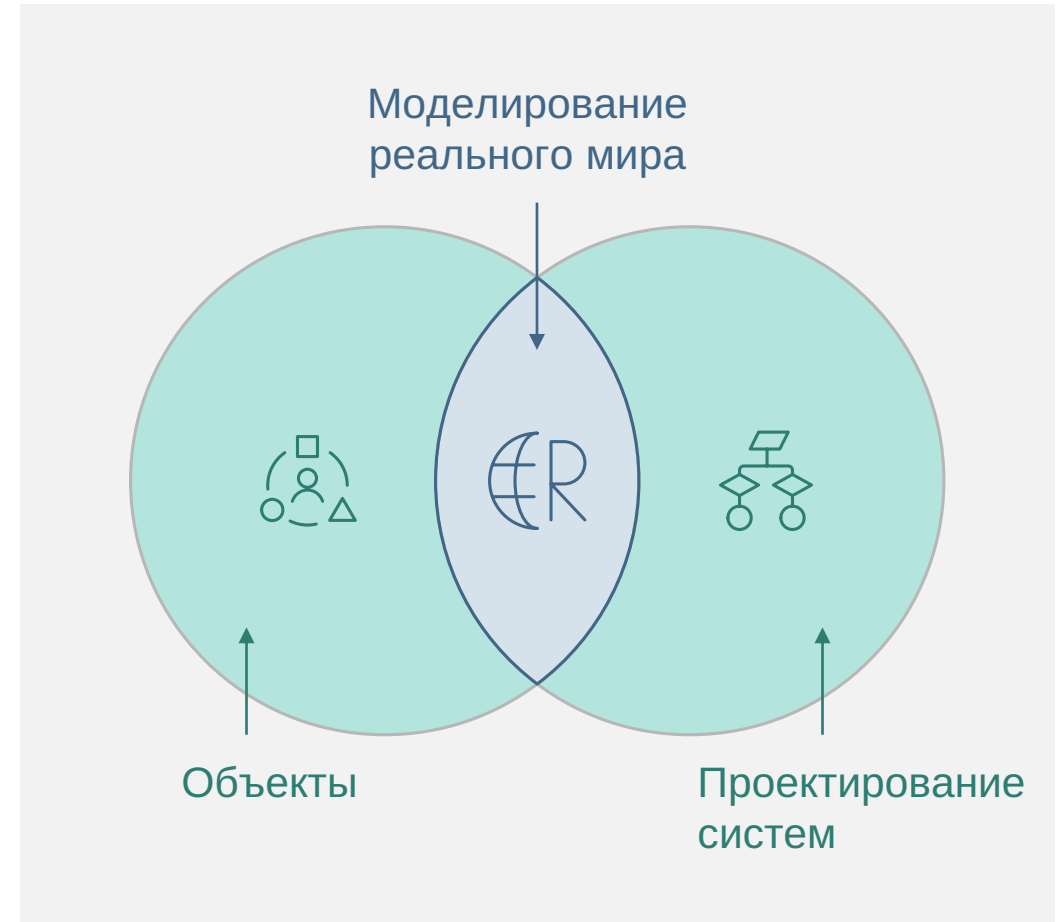


ООП как основа проектирования

Со временем ООП переросло определение «парадигма» и стало скорее набором правил и подходу к проектированию систем

ООП можно сравнить с любой сложной системой, где:

- Сущности имеют чёткие роли
- Взаимодействия регулируются стандартами
- Изменения в одной части не ломают целое



Проблемы с ООП при проектировании

1

Создание глубоких иерархий классов или излишняя абстракция могут усложнить код

2

Жесткая связь между родительскими и дочерними классами усложняет изменения (хрупкие иерархии)

3

Наследование часто приводит к дублированию кода, если иерархия спроектирована плохо

4

В некоторых сценариях накладные расходы на создание объектов и виртуальные методы могут быть критичны (например, высоконагруженные системы)

5

Попытки «впихнуть» всё в ООП-парадигму иногда приводят к ненужной сложности там, где достаточно процедурного подхода

6

Неправильное использование геттеров, сеттеров или публичных полей может превратить объекты в «структуры с функциями», лишив смысла ООП

Проблемы, которые решает ООП

1

Разделение на классы упрощает понимание

2

Наследование и композиция делает код масштабируемым

3

Паттерны и полиморфизм позволяют адаптировать код

Пример задачи, где ООП не идеален:

Обработка данных в стиле pipeline
(лучше подходит функциональный подход)

```
# Функциональный стиль
data = read_csv("data.csv") \
    .filter(lambda x: x > 0) \
    .map(lambda x: x * 2) \
    .to_list()
```

Принципы ООП как проектирования

Управление
сложностью
через абстракцию

Инкапсуляция:
защита данных,
упрощение поддержки

Наследование
и повторное
использование кода

Полиморфизм
для гибкости
и расширяемости

Модульность
и масштабируемость

Поддержка
и эволюция кода

Стандартизация
и интеграция
с экосистемой

Управление сложностью через абстракцию

! ООП позволяет моделировать сложные системы в виде объектов, которые соответствуют сущностям реального мира (например, «Пользователь», «Заказ», «Транзакция»)

В банковском приложении класс **Account** инкапсулирует данные (баланс) и методы (пополнение, списание), что делает код интуитивно понятным

Польза:

Разработчики работают с абстракциями, а не с низкоуровневыми операциями, что снижает когнитивную нагрузку

Инкапсуляция: защита данных и упрощение поддержки

! Объекты скрывают внутреннюю реализацию, предоставляя только интерфейсы для взаимодействия

Класс **DatabaseConnector** может скрывать детали подключения к БД, а иметь публичными только методы **getData()** и **saveData()**

Польза:

Изменения внутри класса не затрагивают другие части системы, уменьшая риск ошибок и упрощая рефакторинг

Наследование и повторное использование кода

! Наследование позволяет создавать иерархии классов, избегая дублирования

Класс **Vehicle** с общими свойствами (скорость, вес) может быть унаследован классами **Car** и **Bicycle**, где добавляются специфичные методы (например, **honk()** для **Car**)

Польза:

Экономия времени, единая точка для изменений и соблюдение принципа DRY (Don't Repeat Yourself)

Полиморфизм для гибкости и расширяемости

! Объекты разных классов могут обрабатываться единообразно через общие интерфейсы

В графическом редакторе классы **Circle** и **Square** реализуют интерфейс **Drawable** с методом **draw()** – это позволяет рендерить все фигуры в цикле, не зная их конкретного типа

Польза:

Легкое добавление новых типов объектов без изменения существующего кода
(принцип открытости / закрытости)

Модульность и масштабируемость

! ООП поощряет разбиение системы на независимые модули (классы), которые можно разрабатывать и тестировать отдельно

В веб-приложении модули «Аутентификация», «Оплата» и «Логирование» реализуются как отдельные классы с четкими API

Польза:

Упрощение командной работы, параллельная разработка и возможность замены компонентов без остановки всей системы

Поддержка и эволюция кода

! Архитектура на основе ООП проще адаптируется к изменениям требований

Добавление новой платежной системы (например, криптовалюты) требует лишь создания класса **CryptoPayment**, наследующего базовый класс **Payment**

Польза:

Снижение стоимости изменений и ускорение вывода новых фич

Стандартизация и интеграция с экосистемой

! Многие современные фреймворки (Django, Spring, .NET) и паттерны проектирования (Factory, Observer) основаны на ООП

Использование Spring Boot для создания микросервисов с инъекцией зависимостей

Польза:

Быстрое внедрение проверенных решений и сокращение времени разработки



Управление памятью и исключения

И связь данных механизмов с ООП

ООП как основа архитектуры

Проблемы проектирования, которые решает ООП

Шаблоны и их связь с ООП

Варианты их применения

Оптимизация шаблонов

Встраивание, выражения шаблонов и сокращение временных объектов

Post-OOP

Новые подходы к проектированию



Мы здесь

Варианты применения шаблонов в ООП

Обобщённые контейнеры

Стандартная библиотека шаблонов (STL)

Шаблонные функции и классы

Реализация паттернов проектирования

Полиморфизм без наследования

Шаблонные функции и классы

При определении шаблона класса компилятор сам генерирует по этому шаблону классы, которые применяют определенные типы

Однако мы сами можем определить подобные классы для конкретного набора параметров шаблона – подобные определения классов называются **специализацией шаблона**

Разделяется на два типа:

Полная специализация (Explicit Specialization)

создание отдельной версии для конкретного типа

Частичная специализация (Partial Specialization)

применяется только к шаблонам классов

Полная специализация (Explicit Specialization)

Полная специализация – это создание отдельной версии шаблона для конкретного типа (или набора типов)

Используется, когда общий шаблон не подходит для определённого случая

```
// Общий шаблон
template <typename T>
void printType() {
    std::cout << "Unknown type: "
<< typeid(T).name() << std::endl;
}
```

```
// Полная специализация
для int
template <>
void printType<int>() {
    std::cout << "Type: int" <<
std::endl;
}
```

```
// Полная специализация
для double
template <>
void printType<double>() {
    std::cout << "Type: double" <<
std::endl;
}
```

Частичная специализация (Partial Specialization)

Частичная специализация применяется только к шаблонам классов и позволяет специализировать шаблон для части параметров или условий (например, для указателей, массивов, шаблонных типов)

```
template <typename T, typename U>
class Pair {
public:
    void print() { std::cout << "Generic Pair<T, U>" <<
std::endl; }
};
```

```
// Частичная специализация для случая, когда
оба типа одинаковы
template <typename T>
class Pair<T, T> {
public:
    void print() { std::cout << "Pair<T, T>" << std::endl; }
};
```

Реализация паттернов проектирования

Фабричный метод через шаблоны:

```
template <typename T>
T* FactoryMethod() {
    return new T();
}
```

Одиночка (Singleton) с шаблоном:

```
template <typename T>
class Singleton {
public:
    static T& getInstance() {
        static T instance;
        return instance;
    }

private:
    Singleton() = default;
};
```

Полиморфизм без наследования

C RTP (Curiously Recurring Template Pattern) – интересный приём, когда класс наследуется от шаблонного базового класса, параметризованного самим наследником

Он помогает реализовывать **компилируемый полиморфизм** без виртуальных функций

Оптимизация:

Избегает накладных расходов виртуальных вызовов

```
template <typename Derived>
class Interface {
public:
    void execute() {
        static_cast<Derived*>(this)->run();
    }
};

class Task : public Interface<Task> {
public:
    void run() {
        std::cout << "Task is running" << std::endl;
    }
};

int main() {
    Task t;
    t.execute(); // Выведет "Task is running"
}
```

Rust Generics как интерпретация шаблонов C++

Синтаксис:

Угловые скобки (<>) + трейты (where T: Trait)

```
fn max<T: PartialOrd>(a: T, b: T) -> T {  
    if a > b { a } else { b }  
}  
  
// Использование с пользовательским  
типом  
#[derive(PartialEq, PartialOrd)]  
struct Point(f32, f32);  
  
let p1 = Point(1.0, 2.0);  
let p2 = Point(3.0, 4.0);  
println!("{}", max(p1, p2).0); // 3.0
```

Ограничения на типы задаются через трейты (аналоги абстрактных классов)

Мономорфизация
(как в C++, код генерируется для каждого типа)

Система владения и заимствования предотвращает ошибки памяти и повышает безопасность

- Безопасность: Нет неопределённого поведения
- Чёткие сообщения об ошибках благодаря трейтам
- Высокая производительность (как в C++)

Менее гибкие, чем шаблоны C++ (например, нет специализации) и сложны для новичков

Go Generics как интерпретация шаблонов C++

Синтаксис:

Минималистичный с квадратными скобками []

```
func Max[T constraints.Ordered](a, b T) T {  
    if a > b {  
        return a  
    }  
    return b  
}
```

```
type Stack[T any] struct {  
    items []T  
}
```

```
func (s *Stack[T]) Push(item T) {  
    s.items = append(s.items, item)  
}
```

Ограничения задаются через интерфейсы (например, comparable, any)

Нет мономорфизации – дженерики работают через боксы и интерфейсы

Нет перегрузки операторов, минимальная поддержка сложных сценариев

- Простота использования и читаемость
- Быстрая компиляция
- Подходит для большинства задач общего назначения

Низкая производительность и ограниченная гибкость (например, нельзя специализировать функции)

Сравнение

Критерий	C++	Rust	Go
Производительность	Максимальная (мономорфизация)	Максимальная (мономорфизация)	Средняя (рантайм-боксы)
Безопасность	Зависит от программиста	Максимальная (владение, трейты)	Средняя (сборщик мусора)
Синтаксис	Сложный (<>, SFINAE, концепты)	Умеренный (<>, трейты)	Простой ([], интерфейсы)
Гибкость	Очень высокая (TMP, специализация)	Высокая (трейты, макросы)	Низкая (базовые операции)
Сообщения об ошибках	Запутанные (шаблоны)	Чёткие (проверка трейтов)	Простые, но менее информативные
Области применения	Высокопроизводительные системы	Системное ПО, безопасные приложения	Веб-сервисы, утилиты



Управление памятью и исключения

И связь данных механизмов с ООП

ООП как основа архитектуры

Проблемы проектирования, которые решает ООП

Шаблоны и их связь с ООП


Варианты их применения

Оптимизация шаблонов

Встраивание, выражения шаблонов и сокращение временных объектов

Post-OOP

Новые подходы к проектированию



Мы здесь

Инструменты оптимизации шаблонов

**Инлайнинг (встраивание)
и специализация**

**Выражения шаблонов
(Expression Templates)**

Используются в библиотеках линейной алгебры (например, Eigen)

**Сокращение
временных объектов**

Концепты (C++20)

**Статический
полиморфизм**

**Явные инстанцииции
и extern-шаблоны**

Инлайнинг (встраивание)

Это оптимизация, при которой компилятор заменяет вызов функции её телом, устраняя накладные расходы на передачу аргументов и управление стеком

Шаблонные функции и классы инстанцируются на этапе компиляции для конкретных типов, что позволяет компилятору лучше оптимизировать код

Данные трюки позволяют использовать статический полиморфизм

```
// Обобщённый шаблонный алгоритм
template <typename Iter>
void my_sort(Iter begin, Iter end) {
    // Инлайнинг оператора сравнения для
    // конкретного типа
    std::sort(begin, end, [](const auto& a, const auto& b) {
        return a < b; });
}

// Полиморфный подход (менее эффективный)
struct Base {
    virtual bool operator<(const Base&) const = 0;
};

void polymorphic_sort(std::vector<Base*>& vec) {
    std::sort(vec.begin(), vec.end(), [](Base* a, Base* b) {
        return *a < *b; });
    // Виртуальный вызов operator< — инлайнинг
    // невозможен!
}
```

Выражения шаблонов (Expression Templates)

Это техника, которая позволяет объединять несколько операций в одну, избегая создания промежуточных объектов (например, математические выражения)

Связь с ООП:

- Объекты-выражения инкапсулируют операции, но вместо их немедленного выполнения, они строят «ленивое» представление выражения
- Позволяет совместить удобный ООП-интерфейс с низкоуровневой оптимизацией

```
Matrix operator+(const Matrix& a, const Matrix& b)
{
    Matrix result(a.rows(), a.cols());
    for (size_t i = 0; i < a.rows(); ++i)
        for (size_t j = 0; j < a.cols(); ++j)
            result[i][j] = a[i][j] + b[i][j];
    return result; // Создаётся временный объект
}
```

// Вычисление: A + B создаёт временную матрицу, затем копирует её в C.

Matrix C = A + B;

Выражения шаблонов (Expression Templates)

Шаблонное выражение «сумма матриц»

```
// Шаблонное выражение "сумма матриц"
template <typename E1, typename E2>
class MatrixSum {
    const E1& a;
    const E2& b;
public:
    MatrixSum(const E1& a, const E2& b) : a(a), b(b) {}
    double operator()(size_t i, size_t j) const {
        return a(i, j) + b(i, j);
    }
};

// Перегрузка оператора +
template <typename E1, typename E2>
MatrixSum<E1, E2> operator+(const E1& a, const E2& b) {
    return MatrixSum<E1, E2>(a, b);
}
```

Оператор присваивания для Matrix

```
// Оператор присваивания для Matrix
class Matrix {
public:
    template <typename Expr>
    Matrix& operator=(const Expr& expr) {
        for (size_t i = 0; i < rows(); ++i)
            for (size_t j = 0; j < cols(); ++j)
                data[i][j] = expr(i, j); // Вычисление "на лету"
        return *this;
    }
};

// Теперь A + B не создаёт временный объект, а сразу
// вычисляется в C.
Matrix C = A + B;
```

Сокращение временных объектов

Шаблоны позволяют избегать копирования через move-семантику и perfect forwarding (например, `std::make_unique<T>`, `emplace` в контейнерах)

Позволяет передавать аргументы без лишнего копирования (`std::forward`)
Полезно для фабричных методов

Как это работает:

- `Args&&... args` принимает параметры в виде универсальных ссылок
- `std::forward<Args>(args)...` передаёт аргументы без лишних копирований

```
template <typename T, typename... Args>
T* create(Args&&... args) {
    return new T(std::forward<Args>(args)...);
}

class Example {
public:
    Example(int a, double b) {
        std::cout << "Example(" << a << ", " << b << ")" <<
std::endl;
    }
};

int main() {
    Example* e = create<Example>(10, 3.14); // Выведет
    "Example(10, 3.14)"
    delete e;
}
```



Управление памятью и исключения

И связь данных механизмов с ООП

ООП как основа архитектуры

Проблемы проектирования, которые решает ООП

Шаблоны и их связь с ООП

Варианты их применения

Оптимизация шаблонов

Встраивание, выражения шаблонов и сокращение временных объектов

Post-OOP

Новые подходы к проектированию



Мы здесь

Post OOP – Что дальше?

Пост-ООП – набор парадигм и практик, которые развивают или заменяют классическое ООП, решая его проблемы и адаптируясь к современным требованиям: параллелизму, безопасности, модульности и простоте

Он не отвергает ООП полностью, но переосмысливает его принципы, делая акцент на композицию, интерфейсы, неизменяемость и функциональные элементы

Концептуальные отличия от классического ООП

- Композиция вместо наследования

- Интерфейсы и трейты вместо абстрактных классов

- Неизменяемость и функциональный подход

- Отказ от классического полиморфизма

Композиция вместо наследования

ООП

Наследование создаёт жёсткие иерархии классов

Это может приводить к проблеме «хрупкого базового класса» (изменения в родительском классе ломают дочерние)



Пост-ООП

Повторное использование кода достигается через композицию (включение одних объектов в другие) и интерфейсы

Например,
вместо `class Dog extends Animal`



используется `struct Dog { animal: Animal, ... }`

Интерфейсы и трейты вместо абстрактных классов

ООП

Абстрактные классы объединяют состояние и поведение, что может усложнять иерархию



Пост-ООП

Поведение определяется через интерфейсы (Go) или трейты (Rust), которые не привязаны к состоянию

Это уменьшает связанность и упрощает тестирование

Неизменяемость и функциональный подход

ООП

Объекты часто изменяют своё состояние через методы



Пост-ООП

Поощряется использование неизменяемых данных и чистых функций (как в функциональном программировании)

Это упрощает работу в многопоточных средах

Отказ от классического полиморфизма

ООП

Полиморфизм через наследование
и виртуальные методы



Пост-ООП

Полиморфизм достигается через
интерфейсы, дженерики и трейты

Без наследования

Rust

Трейты вместо наследования – они определяют поведение, но не состояние


Их можно реализовать для любых типов

Неизменяемость по умолчанию

Переменные неизменяемы, если не указано `mut`

Композиция и алгебраические типы

Вместо иерархий используются структуры и `enum`:



```
enum Message {  
    Quit,  
    Move { x: i32, y: i32 },  
}
```

Rust и функция сравнения

```
fn max<T: PartialOrd>(a: T, b: T) -> T {  
    if a > b { a } else { b }  
}
```

// Используем концепт для проверки, что T поддерживает оператор >

```
template <typename T>  
requires std::totally_ordered<T> // C++20 концепт  
T max(T a, T b) {  
    return (a > b) ? a : b;  
}
```

// Пример для строк (специализация)

```
template <>  
const char* max(const char* a, const char* b) {  
    return (strcmp(a, b) > 0) ? a : b;  
}
```

```
int main() {  
    std::cout << max(5, 10) << "\n"; // 10  
    std::cout << max("xyz", "abc"); // "xyz"  
}
```

Rust и обобщенный стек

```

struct Stack<T> {
    items: Vec<T>,
}

impl<T> Stack<T> {
    // Создать новый стек
    fn new() -> Self {
        Stack { items: Vec::new() }
    }

    // Добавить элемент в стек
    fn push(&mut self, item: T) {
        self.items.push(item);
    }

    // Извлечь элемент из стека
    fn pop(&mut self) -> Option<T> {
        self.items.pop()
    }
}

```

```

fn main() {
    let mut stack = Stack::new();
    stack.push(42);
    stack.push(100);
    println!("Извлечено: {:?}",
stack.pop()); // Some(100)
}

```

```

template <typename T>
class Stack {
private:
    std::vector<T> items;
public:
    void push(const T& item) {
        items.push_back(item);
    }
    T pop() {
        T item = items.back();
        items.pop_back();
        return item;
    }
};

int main() {
    Stack<int> stack;
    stack.push(42);
    stack.push(100);
    std::cout << stack.pop(); // 100
}

```

Rust | Пример 3

```
trait Summary<T> {  
    fn summarize(&self) -> T;  
}  
  
struct NewsArticle {  
    content: String,  
}  
  
// Реализация трейта для NewsArticle, возвращающая длину текста  
impl Summary<usize> for NewsArticle {  
    fn summarize(&self) -> usize {  
        self.content.len()  
    }  
}  
  
fn main() {  
    let article = NewsArticle { content: String::from("Rust — это круто!") };  
    println!("Длина статьи: {}", article.summarize()); // 15  
}
```


Rust | Пример 4

```
// Шаблонный интерфейс через CRTP (Curiously Recurring  
Template Pattern)
```

```
template <typename T, typename Result>
```

```
class Summary {
```

```
public:
```

```
    Result summarize() const {
```

```
        return static_cast<const T*>(this)->summarize_impl();
```

```
    }
```

```
};
```

```
class NewsArticle : public Summary<NewsArticle, size_t> {
```

```
private:
```

```
    std::string content;
```

```
public:
```

```
    NewsArticle(const std::string& content) : content(content) {}
```

```
    size_t summarize_impl() const {
```

```
        return content.size();
```

```
    }
```

```
};
```

```
int main() {
```

```
    NewsArticle article("C++ — это мощно!");
```

```
    std::cout << article.summarize(); // 19
```

```
}
```

GoLang

Интерфейсы и встраивание:

Интерфейсы неявные (тип реализует интерфейс автоматически, если имеет нужные методы)

Встраивание структур заменяет наследование

```
type Writer interface {  
    Write([]byte) (int, error)  
}  
  
type Logger struct {  
    Writer // Встраивание интерфейса  
}  
  
func (l *Logger) Log(s string) {  
    l.Write([]byte(s))  
}
```

Отказ от классов:

Нет классов, только структуры и методы:

```
type User struct {  
    Name string  
}  
  
func (u User) GetName() string {  
    return u.Name  
}
```

GoLang и суммирование #1

```
package main

import (
    "fmt"
    "golang.org/x/exp/constraints"
)

// Ограничение: типы, поддерживающие сложение (числа)
type Number interface {
    constraints.Integer | constraints.Float
}

func Sum[T Number](slice []T) T {
    var total T
    for _, v := range slice {
        total += v
    }
    return total
}
```

```
func main() {
    ints := []int{1, 2, 3}
    floats := []float64{1.5, 2.5, 3.5}
    fmt.Println(Sum(ints)) // 6
    fmt.Println(Sum(floats)) // 7.5
}
```

GoLang и суммирование #2

```
template <typename T>
requires std::integral<T> || std::floating_point<T> // Концепты C++20
T sum(const std::vector<T>& slice) {
    T total = 0;
    for (const auto& v : slice) {
        total += v;
    }
    return total;
}

int main() {
    std::vector<int> ints = {1, 2, 3};
    std::vector<double> floats = {1.5, 2.5, 3.5};
    std::cout << sum(ints) << "\n"; // 6
    std::cout << sum(floats) << "\n"; // 7.5
}
```

GoLang и пустой контейнер #1

```
package main
import "fmt"

type Box[T any] struct {
    Content T
}

func (b Box[T]) Get() T {
    return b.Content
}

func main() {
    intBox := Box[int]{Content: 42}
    stringBox := Box[string]{Content: "Hello"}
    fmt.Println(intBox.Get()) // 42
    fmt.Println(stringBox.Get()) // Hello
}
```

```
template <typename T>
class Box {
private:
    T content;
public:
    Box(const T& content) : content(content) {}
    T get() const { return content; }
};

int main() {
    Box<int> intBox(42);
    Box<std::string> strBox("Hello");
    std::cout << intBox.get() << "\n"; // 42
    std::cout << strBox.get(); // Hello
}
```

GoLang и пустой контейнер #2

```
package main

import (
    "encoding/json"
    "fmt"
)

// Десериализация JSON в тип T
func ParseJSON[T any](data []byte) (*T, error) {
    var result T
    if err := json.Unmarshal(data, &result); err != nil {
        return nil, err
    }
    return &result, nil
}

type User struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}
```

```
func main() {
    data := []byte(`{"name": "Alice", "age": 30}`)
    user, _ := ParseJSON[User](data)
    fmt.Printf("%+v", user) // &{Name:Alice Age:30}
}
```

GoLang и пустой контейнер #3

```
#include <nlohmann/json.hpp>
using json = nlohmann::json;

// Шаблонная функция десериализации
template <typename T>
T parse_json(const std::string& data) {
    auto j = json::parse(data);
    return j.get<T>();
}

struct User {
    std::string name;
    int age;
    // Для десериализации нужен макрос
    NLOHMANN_DEFINE_TYPE_INTRUSIVE
    NLOHMANN_DEFINE_TYPE_INTRUSIVE(User, name, age)
};
```

```
int main() {
    std::string data = R("{\"name\": \"Alice\", \"age\": 30})";
    User user = parse_json<User>(data);
    std::cout << user.name << " " << user.age; // Alice 30
}
```

Ключевые различия на примерах

RUST

Типы и трейты

Дженерики тесно связаны с системой трейтов
Например, `PartialOrd` гарантирует, что тип поддерживает операции сравнения

Мономорфизация

Для каждого типа генерируется отдельный код, что повышает производительность

Безопасность

Компилятор проверяет, что все операции допустимы для типа `T` на этапе компиляции

GO

Интерфейсы и `any`

Для дженериков используется ключевое слово `any` (аналог `interface{}`)
С версии 1.18 есть ограничения через интерфейсы

Рантайм

Нет мономорфизации – дженерики работают через боксы, что может замедлить выполнение

Простота

Минималистичный синтаксис, но ограниченная гибкость (например, нельзя перегружать операторы)

Спасибо за внимание!
Ваши вопросы?

