

# C++

## Лекции

Автор конспектов: Чубий Савва Андреевич

Преподаватель: Афонин Сергей Александрович

2024–2025

2024-09-13	
Введение .....	3
Классы .....	3
2024-09-20	
Обобщенное программирование (шаблоны) .....	5
Правила вывода типов шаблонов .....	6
Виды шаблонов .....	6
Специализация .....	6
Полная специализация .....	6
Частичная специализация .....	6
Контроль подставляемых типов .....	7
Предикаты времени компиляции .....	7
Концепты .....	7
Требования .....	7
2024-09-27	
Полиморфизм .....	8
Наследование .....	8
Наследование и права доступа .....	8
Полиморфные функции .....	9
Виртуальные функции .....	10
Перегрузка, переопределение, сокрытие .....	10
Интерфейсы и чистые виртуальные функции .....	11
Принципы проектирования .....	11
2024-10-04	
Шаблоны проектирования .....	12
Некоторые стандартные приемы (не шаблоны) .....	12
The Curiously Recurring Template Pattern (CRTP) .....	12
Примеси .....	12
Паттерны создания .....	13
Зависимости и ограничения .....	13
Абстрактная фабрика .....	13
Саморегистрирующиеся классы .....	14
Фабричный метод .....	15

Паттерн Строитель (Builder) .....	15
Одиночка (Singleton) .....	15
<hr/>	
2024-10-11 .....	
Структурные шаблоны .....	16
Адаптер .....	16
Мост (Bridge, Pimpl) .....	16
Компоновщик (Composite) .....	16
Декоратор (Decorator) .....	17
Фасад .....	18
Заместитель (проху) .....	18
Интератор .....	18
Шаблоны поведения .....	18
Цепочка ответственности .....	18
Команда .....	18
Наблюдатель .....	19
Состояние .....	19
Шаблонный метод .....	19
Стратегия .....	19
Посетитель .....	20
<hr/>	
2024-10-18 .....	
Параллельное программирование .....	20
Параллельные алгоритмы STL .....	20
OpenMP .....	21
С процессом .....	21
Потоки (thread) .....	21
Передача аргументов .....	22
Проблемы с life-time-ом .....	22
Полезные вещи .....	22
Проблема совместного доступа .....	22
Алгоритм Дейкстры (нет, другой) .....	22
std::mutex и std::lock_guard .....	23
<hr/>	
2024-11-01 .....	
Async: Асинхронные вызовы .....	23
promise и future .....	23
Проблема совместного доступа .....	23
Методы решения .....	23
Mutex .....	23
Mutex — не панацея .....	24
Сколько ставить Mutex-ов .....	24
std::shared_mutex — блокировки чтения и записи .....	24
std::scoped_lock .....	25
Проверка корректности .....	25
Ожидание оповещения .....	25
Ожидание с future/ promise .....	25
Защелки, барьеры, семафоры .....	26

2024-09-13

---

## Введение

Препоод: Афонин Сергей Александрович

Оценка:

$$\text{Итог} = 0.5 \cdot (0.1 \cdot A + 0.2 \cdot \text{Дз1} + 0.35 \cdot \text{Дз2} + 0.35 \cdot \text{Дз3}) + 0.5 \cdot (0.3 \cdot \text{Кр} + 0.7 \cdot \text{Экз})$$

Будет:

- ООП
- Параллельное и конкурентное программирование
- Функциональное программирование
- Всякое

## Классы

Классы — исторически первое отличие C++ от C

```
class Matrix {
    private:
        size_t n_rows_;
        size_t n_cols_;
        double *data_;

    public:
        Matrix(size_t n_rows, size_t n_cols);
        Matrix(const Matrix& other);
        Matrix() = delete; // Явно удаляем default-ный конструктор,
                           // хотя он, и так, не создается

        int rank() const;

        size_t n_rows() const { return n_rows_; }
}

int main() {
    Matrix m(10, 10);

    m.rank();
}
```

«Программы надо писать для людей»

Методы, реализованные внутри объявления класса, часто становятся inline-овыми.

Инкапсуляция — скрытие внутреннего состояния (private в классах). Позволяет:

1. меньше косячить в программах
2. отделять реализацию от интерфейса

## Листинг 1. Инкапсуляция в Си

```

public.h:

typedef void* Matrix;
Matrix matrix_create();
int matrix_rank(Matrix m);

public.h:

Matrix matrix_create() { ... }
int matrix_rank(Matrix m) {
    struct MatrixData* = (MatrixData*)m;
    ...
}

```

const — после называния метода, значит метод не меняет экземпляр

```

Matrix m; // default-ный конструктор
Matrix m(1, 1); // конструктор
Matrix m(); // объявление функции
Matrix m{}; // default-ный конструктор
Matrix m2 = m; // конструктор копирования

```

Удалять, как создавали:

```

Matrix* pm = new Matrix(1, 1);
Matrix* a = new Matrix[100];

delete pm;
delete[] a;

```

New по уже выделенной памяти:

```

void* addr = malloc(...);
new (addr) Matrix(1, 1);
a->~Matrix();

```

Если у полей нет default-ного конструктора или поля константы или ссылки, то делать так:

```

class X {...};

X::X(int y) : a(y) { ... }

```

Поля инициализируются в том порядке, в котором указаны в классе

X&& — r-value

```

X::X(X&& other) {...}

```

Нельзя перегрузить оператор внутри класса, если первый аргумент другого типа

Правило трех:

- TODO
- TODO
- TODO

Правило пяти:

- .. правило трех
- TODO
- TODO

Не стоит бросать exception в деструкторе тк exception во время обработки exception-a — плохо

exception в конструкторе — можно

Хорошо делать exception только с типами, унаследованными от std::exception

2024-09-20

## Обобщенное программирование (шаблоны)

**Опр. Обобщенное программирование** — набор методов для создания структур и алгоритмов, которые могут работать в различных ситуациях и с различными исходными данными.

Пример:

```
double total(const double* data, size_t len) {
    double sum = 0;
    for (size_t i = 0; i < len; ++i) {
        sum += data[i];
    }
    return sum;
}
```

Плохой вариант: трижды сделать Ctrl-C, Ctrl-V

Мета программирование — программы, которые пишут программы

Пример с шаблонами:

```
template <typename V>
V total(const V* data, size_t len) {
    V sum = 0;
    for (size_t i = 0; i < len; ++i) {
        sum += data[i];
    }
    return sum;
}
```

Обращение к функции от конкретного типа создает реализацию перегруженной функции. Т.е. шаблон создает семейство функций.

Улучшение. Из

```
V sum = 0;
```

в

```
V sum{};
```

Посчитать сумму:

```
auto result = std::accumulate(A.begin(), A.end(), decltype(A)::value_type(0));
```

```
auto result = std::reduce(A.begin(), A.end());
```

```
std::for_each(A.begin(), A.end(), [&](int n) {
    result += n;
});
```

## Правила вывода типов шаблонов

```
template<typename T>
void f(const T& param);

int x = 1;
f(x); // Чему равно T?
      // T = int
      // ParamType = const int&
```

Правила:

1. Если в `f(expr)`, `expr` — ссылка, то ссылка отбрасывается
2. Тип `T` получается из сопоставления (pattern matching) типа `expr` и `ParamType`

**TODO:** см презентацию

## Виды шаблонов

- Функции

```
template<typename T> void f(T arg);
```

- Классы

```
template<typename T> class Matrix;
```

- Переменные

```
template<class T>
T pi = T(3.1415926L);
```

- Типы (псевдонимы типов)

```
template<typename T> using ptr = T*;
ptr<int> x;
```

- Концепты (будет позднее)

```
template<typename T>
concept C1 = sizeof(T) != sizeof(int);
```

## Специализация

Специализации должны быть написаны до первого использования

### Полная специализация

```
// Общая реализация
template<typename T>
class Matrix {...};

// Более эффективная
// реализация для bool-ок
template<>
class Matrix<bool> {...};
```

### Частичная специализация

```
template<class T1, class T2, int I>
class A {}; // основной шаблон

template<class T, int I>
```

```
class A<T, T*, I> {}; // T2 --- указатель на T1

template<class T, class T2, int I>
class A<T*, T2, I> {}; // T1 --- указатель

template<class T>
class A<int, T*, 5> {}; // T1 = int, T2 --- указатель, I = 5
```

### · Контроль подставляемых типов ·

```
template<typename T>
void swap(T& a, T& b) noexcept {
    static_assert(std::is_copy_constructable_v<T>, "Swap requires copying");
    static_assert(
        std::is_nothrow_copy_constructable_v<T> &&
        std::is_nothrow_copy_assignable_v<T>,
        "Swap requires copying"
    );

    auto c = b;
    b = a;
    a = c;
}
```

### · Предикаты времени компиляции ·

```
#include <type_traits>
```

TODO: ...

### · Концепты ·

**Опр. Концепт** — семейство типов, обладающих определенными свойствами («утинная типизация»)

```
template<typename T>
concept C1 = sizeof(T) != sizeof(int);

template<C1 T>
struct S1 {...};
```

### · Требования ·

```
#include <type_traits>

template<typename T>
requires std::is_copy_constructible_v<T>
T get_copy(T* pointer) {
    if (!pointer) {
        throw std::runtime_error{"Null-pointer dereference"};
    }
    return *pointer;
}
```

2024-09-27

## ———— Полиморфизм ————

**Опр. Полиморфизм** — возможность написания кода, которым можно использовать для разных типов («форм»).

В C++ есть два полиморфизма:

- Времени компиляции (шаблоны)
- Времени исполнения (с помощью наследования и виртуальных функций)

## ———— Наследование ————

**Опр. Наследование** — иерархическое отношение между классами. Механизм повторного использования и расширения класса без модификации его кода.

Обычно отражает отношение «общее–частное».

Как без наследования:

```
struct A {
    void f();
};

struct B {
    A a;
    void something_new();
};
```

```
B obj;
obj.something_new();
obj.a.f();
// ^^
```

С наследованием:

```
struct A {
    void f();
};

struct B : public A {
    void something_new();
};
```

```
B obj;
obj.something_new();
obj.f();
// ^
```

## · Наследование и права доступа ·

- `class D : public B { ... }`
  - `public` -> `public`
  - `protected` -> `protected`
- `class D : private B { ... }`
  - `public`, `protected` -> `private`
- `class D : protected B { ... }`
  - `public`, `protected` -> `protected`

Изменить права доступа при наследовании:



```
struct B { void f(); };

class D : public B {
private:
    using B::f; // делаем f приватным
}

D obj;
obj.f(); // ошибка
```

### —— Полиморфные функции ——

Функцию `pmf` можно вызвать для объекта класса `D`, несмотря на то, что она была объявлена раньше самого класса:

```
struct B {
    void f() {
        std::cout << "B::f()\n";
    }
};

void pmf(B& br) {
    br.f();
}

class D : public B {};

// Можем вызывать функцию pmf для класса
// объявленного после неё самой
D obj;
pmf(obj);
```

При вызове `pmf` объект `cast`-уется к типу `B`, вызывается изначальная функция (из `B`), а не переопределение (из `D`):

```
struct B {
    void f() {
        std::cout << "B::f()\n";
    }
};

void pmf(B& br) {
    br.f();
}

class D : public B {
    void f() {
        std::cout << "D::f()\n";
    }
};

B b;
D d;
pmf(b); // -> "B::f()"
pmf(d); // -> "B::f()"
```

Виртуальные функции

Если сделать функцию `virtual`, то будет вызываться переопределенная функция, а не изначальная.

```
struct B {
    virtual void f() {
        std::cout << "B::f()\n";
    }
};

void pmf(B& br) {
    br.f();
}

class D : public B {
    void f() override {
        std::cout << "D::f()\n";
    }
};

B b;
D d;
pmf(b); // -> "B::f()"
pmf(d); // -> "D::f()"
```

———— Перегрузка, переопределение, сокрытие ————

- Перегрузка (overload): несколько функций с одним именем в одной области видимости
- Переопределение (override) виртуальной функции: объявление в дочернем классе функции с той же сигнатурой
- Сокрытие: объявление функции с тем же именем во вложенной области (в подклассе/дочернем классе)

```
struct B {
    virtual void f(int) { ... }
    virtual void f(double) { ... }
    virtual void g(int i = 20) { ... }
};

struct D : public B {
    void f(complex<double>);
    void g(int i = 20);
}

B b;
D d;
B* pb = new D;
b.f(1.0); // B::f(double)
d.f(1.0); // D::f(complex<double>) неявно кастуемся к double
pb->f(1.0); // B::f(double) нет более специальной реализации для double
b.g(); // B::g(int) 10
d.g(); // D::g(int) 20
pb->g(); // D::g(int) 10 <-- так не надо
```

`override` — ключевое слово, которое проверяет, что данная функция, и правда, является переопределением. Иначе выкидывает `compile error`. Его хорошо писать везде, где оно подходит.

Если есть хотя бы одна виртуальная функция, то деструктор тоже должен быть виртуальным.

### —— Интерфейсы и чистые виртуальные функции ——

Пример (как делать не надо): Для добавления каждого нового logger-а приходится много и тривиально менять метод logger:

```
struct ConsoleLogger {
    void log_tx(long from, long to, double amount) { ... }
};

struct DBLogger {
    void log_tx(long from, long to, double amount) { ... }
    DBLogger(...) {...}
    ~DBLogger() { ... }
};

struct Processor {
    void transfer(long from, long to, double amount) {
        // ...
        switch (logger_type) {
            // ...
        }
        // ...
    }
};
```

Пример (как делать надо): Сделать интерфейс Logger:

```
struct Logger {
    virtual void log_tx(long from, long to, double amount) = 0;
    // ^^^^^
    //      делает виртуальную функцию чистой
};

class Processor {
public:
    Processor(Logger* logger) { ... }
    void transfer() { ... }

private:
    Logger* logger_;
};
```

**Опр. Абстрактные классы (интерфейсы)** — классы с чистыми виртуальными функциями.

Абстрактный класс нельзя создать, можно только унаследовать.

### —— Принципы проектирования ——

- Минимизация зависимостей между частями системы (классами)
- DRY (Don't repeat yourself) – не WET (write everything twice/ we enjoy typing)
- KISS (Keep it simple, stupid)
- YAGNI (You aren't gonna need it)
- SOLID
  - класс должен отвечать за одну конкретную сущность
  - разделение интерфейсов
  - открытость к расширению

- принцип подстановки: класс ведет себя, как базовый

2024-10-04

## Шаблоны проектирования

Ctrl+C, Ctrl+V — плохо

При разработке систем стараются предусматривать возможность будущего расширения:

- Framework — общее решение в некоторой ограниченной области
- Библиотека классов. Например, STL, Boost
- Шаблоны (patterns) проектирования

Виды паттернов проектирования:

- Паттерн создания (как создавать новые объекты): фабрика, прототип, одиночка
- Структурные паттерны (как компоновать сущности): адаптер, мост, проху
- Паттерны поведения: итератор, команда, цепочка ответственности

### Некоторые стандартные приемы (не шаблоны)

- Интерфейс (базовый класс определяет набор чистых виртуальных функций, производные классы их реализуют)
- CRTP (см. дальше)
- Примеси (см. дальше)

#### THE CURIOUSLY RECURRING TEMPLATE PATTERN (CRTP)

```
template <class T>
struct Base {
    void generic_fun() {
        static_cast<T*>(this)->implementation();
    }
};

struct Derived : public Base<Derived> {
    void implementation();
};
```

Как виртуальные функции, но выбор функции происходит в compile time.

#### Примеси

```
struct Number {
    int n;
    void set(int v) { n = v; }
    int get() const { return n; }
};
```

Хотим добавить undo;

```
struct Number {
    int n;
    int old_n;
    void set(int v) { old_n = n; n = v; }
    void undo() { n = old_n; } // undo на один шаг
    int get() const { return n; }
};
```

Хотим это для произвольного класса.

Если состояние — `int`:

```
template <typename T>
struct Undoable : public B {
    int before;
    void set(int v) { before = B::get(); B::set(v); }
    void undo() { B::set(before); }
};
```

```
using UNumber = Undoable<Number>;
```

Если состояние любого типа:

```
template <typename B, typename T = typename B::value_type>
struct Undoable : public B {
    using value_type = T;
    T before;
    void set(T v) { before = B::get(); B::set(v); }
    void undo() { B::set(before); }
};
```

```
using UNumber = Undoable<Number>;
```

```
template <typename B, typename T = typename B::value_type>
struct Redoable : public B {
    using value_type = T;
    T after;
    void set(T v) { after = v; B::set(v); }
    void redo() { B::set(after); }
};
```

```
using RUNumber = Redoable<Undoable<Number>>;
```

## —— Паттерны создания ——

### · Зависимости и ограничения ·

Для создания объекта нужно указать класс. Конструкторы могут требовать сложных аргументов.

Проблемы:

- Хотим минимизировать количество зависимостей между разными частями кода.
- Ограничивает множество классов

### · Абстрактная фабрика ·

Взаимодействие следующих сущностей:

- `AbstractFactory` (интерфейс), `ConcreteFactory` (несколько классов, конкретные реализации)
- `AbstractProduct` (интерфейс), `ConcreteProduct` (несколько классов, конкретные реализации)

```
class Shape { // AbstractProduct
public:
    virtual std::string text() = 0; // имя
    virtual double area() const = 0; // площадь
    virtual ~Shape();
};
```

```
class Rectangle : public Shape {
```

```

public:
    std::string text() override { ... }
    double area() const override { ... }
private:
    double w_, h_;
}

class ShapeFactory {
public:
    // ...TODO
};

// ...TODO

```

Нужна функция для создания фабрик:

```

ShapeFactory* makeShapeFactory(std::string shape) {
    if (shape == "triangle") {
        return new TriangleFactory();
    } else if (shape == "rectangle") {
        return new RectangleFactory();
    } else {
        throw std::invalid_argument("wrong shape name");
    }
}

```

### Саморегистрирующиеся классы

Идея:

- Все фабрики наследуются от базового класса
- В этом базовом классе создается статический реестр фабрик
- При создании фабрики регистрируют себя

```

class AbstractFactory {
public:
    using create_f = std::unique_ptr<AbstractFactory>();

    static void registrate(std::string const& name, create_f* fp) {
        registry[name] = fp;
    }

    static std::unique_ptr<AbstractFactory> make(std::string const& name) {
        auto it = registry.find(name);
        return it == registry.end() ? nullptr : (it->second)();
    }

    template <typename F>
    struct Registrar {
        explicit Registrar(std::string const& name) {
            AbstractFactory::registrate(name, &F::create);
        }
    }

private:
    static std::map<std::string, create_f*> registry;
};

```

Конкретная фабрика:

```

class ConcreteFactory : public AbstractFactory {
    static std::unique_ptr<AbstractFactory> create() {
        return std::make_unique<ConcreteFactory>();
    }
};

// В cpp-файле
namespace {
    ConcreteFactory::Registrar<ConcreteFactory> reg("my_name");
}

```

### · Фабричный метод ·

- Product, ConcreteProduct
- Creator, ConcreteCreator

```

class Creator {
public:
    virtual Product* Create() = 0;
}

```

### · Паттерн Строитель (BUILDER) ·

Строим сложный объект по частям

- Builder, ConcreteBuilder
- Director — распорядитель (вызывает методы Строителя)
- Product

```

class DocBuilder {
public:
    virtual DocBuilder& build_title(std::string& title) { ... }
    ...
    virtual Product* build() { ... }
};

class HTMLBuilder : public DocBuilder { ... };
class LaTeXBuilder : public DocBuilder { ... };

```

Пример использования:

```
Doc transformer(const string& src, Builder& builder);
```

### · Одиночка (SINGLETON) ·

```

template <class T>
class Singleton {
public:
    T& get() {
        static T* obj;
        return obj;
    }
};

```

2024-10-11

## —— Структурные шаблоны ——

### · Адаптер ·

Хотим штуку с одним интерфейсом засунуть в функцию, которая ожидает другой интерфейс

Есть такое:

```
void draw(const PointsShapel& ps); // ps.begin(), ps.end()
class Line { Point a, b; };
```

Хотим использовать в таком:

```
using V1 = std::vector<Point>::iterator;
void draw_points(V1 start, V1 end);
```

Делаем класс-обертку (адаптер) с правильным интерфейсом:

```
using V1 = std::vector<Point>::iterator;
class LinePointShape : public Shapel {
public:
    V1 begin() override { ... }
    V1 end() override { ... }
private:
    Line* line_;
}
```

## —— Мост (BRIDGE, PIMPL) ——

- Отделить интерфейс от реализации
- Скрыть детали реализации

Полезно, если интерфейс простой, а релизация сложная

```
class Optimizer {
public:
    using Target = float(Variablies&);
    virtual void set_target(Target* f);
    virtual Soution optimize(); // вызывает методы impl_

private:
    class OptiomizerImpl {
        ...
    };

    std::unique_ptr<OptimizerImpl> impl_; // "Настоящая" реализация
};
```

Внутреннюю реализацию можно «подменять» в зависимости от задачи

## —— Компоновщик (COMPOSITE) ——

Позволяет единым образом трактовать индивидуальные и составные объекты

Пример: в графическом редакторе одни и те же действия можно выполнять как с одним элементом, так сразу и со многими

```
struct Component {
    virtualvoid opl();
    virtual add(Component*);
};
```



```
struct Leaf : Component {
    void op1() override;
};

struct Group : Component {
    void op1() override;
    void add() override;
};
```

### —— Декоратор (DECORATOR) ——

Декоратор — это «примесь объектов»

Объект помещается в другой объект, который повторяет интерфейс первого, но что-то добавляет

Например: Shape, ColorShape, TransparentShape, Circle, Rectangle

```
// Базовый класс
struct Shape {
    virtual void draw() const = 0;
};

// Конкретная реализация
struct Circle : Shape {
    Circle(float r);
    void draw() const override { ... };
};

// Декоратор, дополнительное свойство
struct ColoredShape : Shape {
    Shape& shape;
    float color;
    ColoredShape(Shape& s, float c) : shape(s), color(c) {}
    void draw() const override {
        set_color();
        shape.draw();
    }
};
```

Пример использования:

```
Circle circle;
ColoredShape green_circle(circle, 0x00FF00);
green_circle.draw();
```

Декораторы можно композировать:

```
TransparentShape circle {
    ColoredShape{
        Circle{0.5},
        0x00FF00
    }
}
```

## —— Фасад ——

Фасад — интерфейс для внешних пользователей. Нужен для скрытия деталей реализации для внешних пользователей.

## —— Заместитель (PROXY) ——

Заглушка тяжелого объекта, локальный представитель удаленного объекта.

Реализует интерфейс основного объекта с некоторыми «техническими» улучшениями.

## —— Итератор ——

Как в std

## —— Шаблоны поведения ——

### —— Цепочка ответственности ——

Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают

```
class Handler {
public:
    void add(Handler* h) {
        next = h;
    }

    virtual void handle(Request& rq) {
        if (next) {
            next->handle();
        }
    }

private:
    Handler* next; // указатель на следующего обработчика
};

class SomeHandler : public Handler {
    void handle(Request& r) override {
        Handler::handle(r);
    }
};
```

## —— Команда ——

Представляет действие, как объект.

Команды можно ставить в очередь и отменять.

```
// Класс банковского счета
class Account {
public:
    void deposit(int amount);
    void withdraw(int amount);

private:
    int ammount;
};
```

```
struct Command {  
    virtual void process() = 0;  
};  
  
struct BankAccountCommand : Command {  
    BankAccount& account;  
    enum Action {  
        deposit,  
        withdraw,  
    } action;  
    int amount;  
  
    ...  
}
```

### ———— Наблюдатель ————

Задача: при изменении одного объекта информировать об этом изменении другие. Publish and subscribe.

При изменении поля ввода меняется состояние кнопки.

Участники:

- Объекты, заинтересованные в информации
- Объекты, обладающие информацией

То есть одни объекты «подписываются» на изменения других

```
class Subject {  
    public:  
        virtual ~Subject();  
        virtual void attach(Observer*);  
        virtual void detach(Observer*);  
        virtual void notify();  
  
    protected:  
        Subject();  
  
    private:  
        List<Observer*> observers_;  
};
```

### ———— Состояние ————

Реализуем класс, как конечный автомат

Базовый класс State определяет нужные виртуальные методы, производные классы переопределяют эти методы для конкретного состояния.

Внутри класса храним указатель на нужный State.

### ———— Шаблонный метод ————

В базовом классе алгоритм, который вызывает некоторые виртуальные функции. Эти функции (например, для оптимизации можно переопределить)

### ———— Стратегия ————

Алгоритм решения выделяется в класс.

---

...

Посетитель

```

class Visitor {
public:
    virtual void visit(ElementA*) = 0;
    virtual void visit(ElementB*) = 0;

protected:
    Visitor();
};

class Element {
public:
    virtual ~Element();
    virtual void accept(Visitor& v) = 0;

protected:
    Element();
};

```

Может, например, быть SaveVisitor.

2024-10-18

## Параллельное программирование

Зачем:

- хотим выполнять разные операции в одно и то же время (например, одновременно считать и взаимодействовать с пользователем)
- ускорение на многоядерной системой

Параллельное или конкурентное:

- Параллельное — хотим решить одну задачу быстрее
- Конкурентное — есть много задач, которые мы хотим решать одновременно

Возможности C++:

- Потоки
- Асинхронный вызов
- Сопрограммы (не совсем конкурентность)
- Параллельные алгоритмы STL

Средства создать параллельных приложений:

- Примитивы C++
- OpenMP (стандарт для написания параллельных программ)
- MPI (для кластеров)
- Сторонние библиотеки (Intel TBB, HPX)

### Параллельные алгоритмы STL

```

#include <vector>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> vec{1, 5, 2, 10, 3};
}

```

```
std::sort(std::execution::par, vec.begin(), vec.end());
}
```

### —— OPENMP ——

```
double res[LEN];
int i;

#pragma omp parallel for num_threads(10)
for (i = 0; i < LEN; ++i) {
    res[i] = long_running(i);
}
```

g++ -openmp file.cpp

### —— С процессом ——

```
int main(void) {
    pid_t pid;
    if (signal(SIGCHLD, SIG_IGN) == SIG_ERR) {
        exit(EXIT_FAILURE);
    }
    pid = fork();
    switch(pid) {
        case -1: // ошибка
        case 0: // родитель
        default: // родитель
    }
}
```

### —— Потоки (THREAD) ——

С функцией:

```
void do_some_work();
std::thread my_thread(do_some_work);
```

С функтором:

```
void do_some_work();
struct X {
    void operator()() { ... }
}
```

```
std::thread my_thread{X()};
```

У нового thread-а и его родителя общее адресное пространство. До удаления std::thread нужно сделать либо:

- join() — блокируемся до завершения
- detach() — отсоединяем процесс, он работает в фоне

Отсоединенный (detach) процесс завершится либо сам, либо вместе с main

```
#include <thread>
#include <iostream>

void f() {
    for (size_t i = 0; i < 100; ++i) {
        std::cout << i << std::endl;
    }
}
```

```
    }  
}  
  
int main() {  
    std::thread t(f);  
    t.detach(); // ничего не выведется  
    // t.join(); // так выведется всё  
}
```

### · Передача аргументов ·

```
void f(int val);  
struct X {  
    f(int val);  
};  
  
// Вызов функции  
std::thread t1(f, 42);  
  
// Вызов метода  
X obj;  
std::thread(&X::f, &obj, 42);
```

Аргументы копируются во внутреннее хранилище, а затем передаются, как r-value.

### · Проблемы с LIFE-TIME-ОМ ·

```
void f(int*);  
  
void caller() {  
    int data[100];  
    std::thread t(f, data); // UB: data умирает раньше завершения f  
    t.detach();  
}
```

### · Полезные вещи ·

- std::thread
- std::thread::hardware\_concurrency()
- std::this\_thread::get\_id()
- std::this\_thread::sleep\_for(/\* time \*/) — надо делать так, ибо обычный sleep усыпит весь процесс (все thread-ы)
- std::this\_thread::sleep\_until(/\* ... \*/)
- std::this\_thread::yield()

### · Проблема совместного доступа ·

Всё хорошо только тогда, когда есть

- либо только много читающих thread-ов
- либо только один пишущий thread

### · Алгоритм Дейкстры (нет, другой) ·

```
status[i] in {competing, out, crit}  
turn in {1, ..., N}  
repeat  
    while turn != i do  
        if status[turn] == out then
```

```

        turn := i
    end if
end while
status[i] = cs
until not exists other : satus[other] = cs
CS
status[i] = out

```

`std::mutex` и `std::lock_guard`

Реализует взаимное исключение

2024-11-01

## ASYNC: Асинхронные вызовы

```

int f() {
    return 42;
}

int main() {
    std::future res = std::async(f); // Запуск
    res.wait(); // Блокирующий вызов. Просто ждет
    int value = res.get(); // get вызывает wait
}

```

### PROMISE И FUTURE

- promise — неготовое значение со стороны поставщика; есть `set_value()`<sup>1</sup>
- future — неготовое значение со стороны получателя

## Проблема совместного доступа

- Чтение в нескольких потоках безопасно
- Плохо, когда
  - один читает другой пишет
  - несколько пишут

### Методы решения

- Блокировки
- lock-free
- Трансляционная память (особо нигде не поддерживается)

## MUTEX

«Mutex блокирует инвариант»: пока mutex lock-нут, инвариант может нарушаться, иначе — нет

```

std::mutex m;
std::list<int> the_list;

void safe_append(int v) {
    std::lock_guard lock(m);
    the_list.push_back(v);
}

```

<sup>1</sup>`set_value()` можно делать только один раз

```
void process_list() {
    std::lock_guard<std::mutex> lock(m); // Полная форма
    // *Что-то делаем*
}
```

### · МУТЕХ — не панацея ·

```
template <typename T>
class stack {
    void push(const T& v); // Защищена mutex-ом
    void empty() const; // Защищена mutex-ом
    const T& top() const; // Защищена mutex-ом
    void pop(); // Защищена mutex-ом
};

stack<int> s;
if (!s.empty()) {
    // Ошибка, если в этом месте другой thread сделал s.pop()
    int v = s.top();
    s.pop();
}
```

Плохое решение — сказать пользователю завернуть if в mutex

Хорошее решение — по-умному изменить интерфейс, добавить «большие» операции:

```
template <typename T>
class safe_stack {
private:
    stack<T> s_;
    mutable std::mutex m_;

public:
    std::shared_ptr<T> pop() {
        std::lock_guard lock(m_);
        if (s_.empty()) throw stack_exception();
        std::shared_ptr<T> res(std::make_shared<T>(s_.top()));
        s_.pop();
        return res;
    }

    bool empty() const {
        std::lock_guard lk(m_);
        return s_.empty();
    }
}
```

### · Сколько ставить МУТЕХ-ов ·

- Один на программу — слишком мало
- Один на структуру — см. прошлый пример
- На каждый элемент

Меньше область блокирования — выше параллелизм, но сложнее программа

### · std::shared\_mutex — блокировки чтения и записи ·

- lock / try\_lock — эксклюзивная блокировка (только один поток)



- `lock_shared / try_lock_shared` — разделяемая блокировка (несколько потоков могут удерживать блокировку)

Эксклюзивная блокировка — изменение объекта, разделяемые — для чтения.

`std::scoped_lock`

Решает проблему взаимных блокировок

Как `std::lock_guard`, но атомарно блокирует все данные mutex-ы

```
std::mutex m1, m2;
std::scoped_lock lk(m1, m2);
```

### Проверка корректности

\*Сюжет про Сети Петри\*: строим модель нашей системы, модель анализируем (на практике обычно идет сложно).

Советы:

- Избегать блокировки двух mutex-ов
- Не выполнять пользовательский код в момент удержания блокировки
- Ставить несколько блокировок в фиксированном порядке
- Иерархические блокировки

### —— Ожидание оповещения ——

```
std::queue<Data> tasks;
std::mutex m;
std::condition_variable cond;
```

```
void task_maker() {
    while(1) {
        auto data = prepare();
        std::scoped_lock lk(m);
        tasks.push(data);
    }
    cond.notify_all();
}

void process() {
    while(1) {
        std::unique_lock lk(m);
        cond.wait(lk);
        auto data = task.front();
        tasks.pop();
        lk.unlock();
    }
}
```

В `wait` можно передать предикат

```
cond.wait(lk, []() { return !tasks.empty(); });
```

### —— Ожидание с FUTURE/ PROMISE ——

```
std::promise<int> p;
std::future<int> f = p.get_future();

f.wait();
```

```
int res = f.get();
```

```
// В другом потоке  
p.set_value(42);
```

### —— Защелки, барьеры, семафоры ——

- Барьер — точка синхронизации, до которой все потоки должны прийти одновременно. Число потоков известно в момент создания барьера.
- Защелка — одноразовый барьер.