C++

Лекции

2024–2025

2024-09-13

Введение 2
Классы 2

2024-09-20

Обобщенное программирование (шаблоны) 4
- Правила вывода типов шаблонов 4
- Виды шаблонов 5
- Специализация 5
  - Полная специализация 5
  - Частичная специализация 5
  - Контроль подставляемых типов 6
  - Предикаты времени компиляции 6
  - Концепты 6
  - Требования 6

2024-09-27

Полиморфизм 6
- Наследование 7
  - Наследование и права доступа 7

2024-09-13

Введение

Препод: Сергей Александрович

Оценка:

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

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 Matrx& 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 консруктор
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-а — 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

```cpp
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

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

- Классы

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

- Переменные

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

- Типы (псевдонимы типов)

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

- Концепты (будет позднее)

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

Специализация

Специализации должны быть написаны до первого использования

Полная специализация

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

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

Частичная специализация

```cpp
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
```

```cpp
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

```cpp
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_assinable_v<T>,
        "Swap requires copying"
    );

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

Предикаты времени компиляции

```cpp
#include <type_traits>
```

TODO: ...

Концепты

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

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

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

Требования

```cpp
#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`