Лекция 3. Структуры данных и классы

Илья Макаров

CS центр
21 сентября 2017
Санкт-Петербург

Зачем группировать данные?

Какая должна быть сигнатура у функции, которая вычисляет длину отрезка на плоскости?

```
double length(double x1, double y1,
              double x2, double y2);
```

А сигнатура функции, проверяющей пересечение отрезков?

```
bool intersects(double x11, double y11,
                double x12, double y12,
                double x21, double y21,
                double x22, double y22,
                double * xi, double * yi);
```

Координаты точек являются логически связанными данными, которые всегда передаются вместе.
Аналогично связанны координаты точек отрезка.

Структуры

Структуры — это способ синтаксически (и физически) сгруппировать логически связанные данные.

```
struct Point {
    double x;
    double y;
};
struct Segment {
    Point p1;
    Point p2;
};

double length(Segment s);

bool intersects(Segment s1,
                Segment s2, Point * p);
```

Работа со структурами

Доступ к полям структуры осуществляется через оператор '.':

```
#include <cmath>

double length(Segment s) {
    double dx = s.p1.x - s.p2.x;
    double dy = s.p1.y - s.p2.y;
    return sqrt(dx * dx + dy * dy);
}
```

Для указателей на структуры используется оператор '->'.

```
double length(Segment * s) {
    double dx = s->p1.x - s->p2.x;
    double dy = s->p1.y - s->p2.y;
    return sqrt(dx * dx + dy * dy);
}
```

Инициализация структур

Поля структур можно инициализировать подобно массивам:

```
Point p1 = { 0.4, 1.4 };
Point p2 = { 1.2, 6.3 };
Segment s = { p1, p2 };
```

Структуры могут хранить переменные разных типов.

```
struct IntArray2D {
    size_t a;
    size_t b;
    std::vector<int> data;
};
```

```
IntArray2D a = {n, m, create_array2d(n, m)};
```

POD типы

POD-типы в языке C++ это аббревиатура от Plain Old Data, что можно трактовать как простые данные в стиле C.
К POD типам относятся:

- все встроенные арифметические типы (включая wchar_t и bool);
- перечисления, т.е. типы, объявленные с помощью ключевого слова enum;
- указатели;
- POD-структуры (struct или class) и POD-объединения (union).

POD типы

Чтобы структура была POD-типом, нужно выполнение следующих условий:

- не иметь пользовательских конструкторов, деструктора или копирующего оператора присваивания;
- не иметь базовых классов;
- не иметь виртуальных функций;
- не иметь защищенных (protected) или закрытых (private) нестатических членов данных;
- не иметь нестатических членов данных не-POD-типов (или массивов из таких типов), а также ссылок.

Structure binding

- Определим простейшую POD структуру:

```
struct Point {
    int x;
    int y;
};
```

- Structure binding:

```
Point p = { 1, 2 };
auto [x, y] = p; // C++17

foo(x); // equal to foo(p.x);
bar(y); // equal to bar(p.y);
```

Structure binding

Такие конструкции удобно использовать вместе с алгоритмами из стандартной библиотеки, которые часто возвращают `std::pair` в качестве результата своей работы.

```
std::unordered_map<std::string, std::string> map;

auto [it, inserted] = map.emplace("key", "value");

if (inserted) {
    // do something with it
}
else {
    // do something else with it
}
```

Методы

Метод — это функция, определённая внутри структуры.

```
struct Segment {
    Point p1;
    Point p2;
    double length() {
        double dx = p1.x - p2.x;
        double dy = p1.y - p2.y;
        return sqrt(dx * dx + dy * dy);
    }
};
int main() {
    Segment s = { { 0.4, 1.4 }, { 1.2, 6.3 } };
    cout << s.length() << endl;
    return 0;
}
```

Методы

Методы реализованы как функции с неявным параметром `this`, который указывает на текущий экземпляр структуры.

```
struct Point
{
    double x;
    double y;

    void shift(/* Point * this, */
               double x, double y) {
        this->x += x;
        this->y += y;
    }
};
```

Методы: объявление и определение

Методы можно разделять на объявление и определение:

```
struct Point
{
    double x;
    double y;

    void shift(double x, double y);
};
```

```
void Point::shift(double x, double y)
{
    this->x += x;
    this->y += y;
}
```

Конструкторы

Конструкторы — это методы для инициализации структур.

```
struct Point {
    Point() {
        x = y = 0;
    }
    Point(double x, double y) {
        this->x = x;
        this->y = y;
    }
    double x;
    double y;
};
```

```
Point p1;
Point p2(3,7);
```

Список инициализации

Список инициализации позволяет проинициализировать поля до входа в конструктор.

```
struct Point {
    Point() : x(0), y(0)
    {}
    Point(double x, double y) : x(x), y(y)
    {}

    double x;
    double y;
};
```

Инициализации полей в списке инициализации происходит в *порядке объявления полей* в структуре.

Значения по умолчанию

- Функции могут иметь значения параметров *по умолчанию*.
- Значения параметров по умолчанию нужно указывать в *объявлении функции*.

```
struct Point {
    Point(double x = 0, double y = 0)
        : x(x), y(y)
    {}
    double x;
    double y;
};
```

```
Point p1;
Point p2(2);
Point p3(3,4);
```

Значения по умолчанию

- Определим структуру:

```
struct Point {
    int x;
    int y;
};
```

- Попробуем инициализировать ее значения:

```
Point p1 = { 1, 2 }; // ok
Point p2; // undefined
```

Значения по умолчанию

- Исправим нашу структуру:

```
struct Point {
    int x = 0;
    int y = 0;
};
```

- Попробуем инициализировать ее значения:

```
Point p1 = { 1, 2 }; // ok
Point p2; // now ok
```

Конструкторы от одного параметра

Конструкторы от одного параметра задают *неявное* пользовательское преобразование:

```
struct Segment {
    Segment() {}
    Segment(double length)
        : p2(length, 0)
    {}
    Point p1;
    Point p2;
};
```

```
    Segment s1;
    Segment s2(10);
    Segment s3 = 20;
```

Конструкторы от одного параметра

Для того, чтобы запретить *неявное* пользовательское преобразование, используется ключевое слово `explicit`.

```
struct Segment {
    Segment() {}
    explicit Segment(double length)
        : p2(length, 0)
    {}
    Point p1;
    Point p2;
};
```

```
    Segment s1;
    Segment s2(10);
    Segment s3 = 20; // error
```

Конструкторы от одного параметра

Неявное пользовательское преобразование, задаётся также конструкторами, которые могут принимать один параметр.

```
struct Point {
    explicit Point(double x = 0, double y = 0)
        : x(x), y(y)
    {}
    double x;
    double y;
};
```

```
    Point p1;
    Point p2(2);
    Point p3(3,4);
    Point p4 = 5; // error
```

Конструктор по умолчанию

Если у структуры нет конструкторов, то конструктор без параметров, *конструктор по умолчанию*, генерируется компилятором.

```
struct Segment {
    Segment(Point p1, Point p2)
        : p1(p1), p2(p2)
    {}
    Point p1;
    Point p2;
};
```

```
    Segment s1; // error
    Segment s2(Point(), Point(2,1));
```

Конструктор по умолчанию

Существует возможность добавить *конструктор по умолчанию*. В таком случае компилятор сгенерирует конструктор самостоятельно.

```
struct Segment {
    Segment() = default; // C++11
    Segment(Point p1, Point p2)
        : p1(p1), p2(p2)
    {}

    Point p1;
    Point p2;
};
```

```
    // ok if Point type has default constructor
    Segment s1;
```

Удаление конструктора

Существует возможность удалить *конструктор по умолчанию*.

```
struct Segment {
    Segment() = delete;

    Point p1;
    Point p2;
};
```

```
    Segment s1; // error
    Segment s2(Point(), Point(2,1)); // error
    Segment s2{ Point(), Point(2,1) }; // ok until C++20
```

Особенности синтаксиса C++

“Если что-то похоже на объявление функции, то это и есть объявление функции.”

```
struct Point {
    explicit Point(double x = 0, double y = 0)
        : x(x), y(y) {}
    double x;
    double y;
};
```

```
    Point p1;   // определение переменной
    Point p2(); // объявление функции

    double k = 5.1;
    Point p3(int(k)); // объявление функции
    Point p4((int)k); // определение переменной
```

Деструктор

Деструктор — это метод, который вызывается при удалении структуры, генерируется компилятором.

```
struct Point {
    Point() {
        std::cout << "A" << std::endl;
    }

    ~Point() {
        std::cout << "B" << std::endl;
    }
};

void foo() { Point p; }

foo(); // prints AB
```

Время жизни

Время жизни — это временной интервал между вызовами конструктора и деструктора.

```
void foo() {
    Point p1; // создание p1
    Point p2; // создание p2

    {
        Point p3; // создание p3
    } // удаление p3
} // удаление p2, потом p1
```

Деструкторы переменных на стеке вызываются в обратном порядке (по отношению к порядку вызова конструкторов).

Объекты и классы

- Структуру с методами, конструкторами и деструктором называют *классом*.
- Экземпляр (значение) класса называется *объектом*.

Модификаторы доступа

Модификаторы доступа позволяют ограничивать доступ к методам и полям класса.

```
struct Point {
    explicit Point(double x = 0, double y = 0)
        : x(x), y(y) {}

    int x() { return x; }
    void set_x(int x_new) { x = x_new; }

    // y methods here
private:
    double x;
    double y;
};
```

Ключевое слово `class`

Ключевое слово `struct` можно заменить на `class`, тогда поля и методы по умолчанию будут private.

```
class Point {
    double x;
    double y;

public:
    explicit Point(double x = 0, double y = 0)
        : x(x), y(y) {}

    int x() { return x; }
    void set_x(int x_new) { x = x_new; }

    // y methods here
};
```

Ключевое слово `class`

```
class Point {
public:
    explicit Point(double x = 0, double y = 0)
        : x_(x), y_(y) {}

    int x() { return x_; }
    void set_x(int x_new) { x_ = x_new; }

    // y methods here

private:
    double x_;
    double y_;
};
```

Инварианты класса

- Выделение *публичного интерфейса* позволяет поддерживать *инварианты класса*
(сохранять данные объекта в согласованном состоянии).

```
struct IntArray {
    ...
    size_t size_;
    int * data_; // массив размера size_
};
```

- Для сохранения инвариантов класса:

1. все поля должны быть закрытыми,
2. публичные методы должны сохранять инварианты класса.

- Закрытие полей класса позволяет абстрагироваться от способа хранения данных объекта.

Публичный интерфейс

```
struct IntArray {
    ...
    void resize(size_t nsize) {
        int * ndata = new int[nsize];
        size_t n = nsize > size_ ? size_ : nsize;
        for (size_t i = 0; i != n; ++i)
            ndata[i] = data_[i];
        delete[] data_;
        data_ = ndata;
        size_ = nsize;
    }
private:
    size_t size_;
    int * data_;
};
```

Абстракция

```
struct IntArray {
public:
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size])
    {}
    ~IntArray() { delete[] data_;  }

    int & get(size_t i) { return data_[i]; }
    size_t size() { return size_; }

private:
    size_t size_;
    int * data_;
};
```

Абстракция

```
struct IntArray {
public:
    explicit IntArray(size_t size)
        : data_(new int[size + 1])
    {
        data_[0] = size;
    }
    ~IntArray() { delete[] data_;  }

    int & get(size_t i) { return data_[i + 1]; }
    size_t size() { return data_[0]; }

private:
    int * data_;
};
```

Константные методы

- Методы могут быть объявлены как `const`.

```
struct S
{
    const int & ref() const {
        return data_;
    }
private:
    int data_;
};
```

- Такие методы не могут менять поля объекта.
- У константных объектов (через указатель или ссылку на константу) можно вызывать только константные методы.
- Внутри константных методов можно вызывать только константные методы.

Две версии одного метода

- Слово `const` является частью сигнатуры метода.

```
const int & S::ref() const { return data_; }
```

- Можно определить две версии одного метода:

```
struct S
{
    const int & ref() const {
        return data_;
    }
    int & ref() {
        return data_;
    }
private:
    int data_;
};
```

Синтаксическая и логическая константность

- Синтаксическая константность — константные методы не могут менять поля (обеспечивается компилятором).
- Логическая константность — нельзя менять те данные, которые определяют состояние объекта.

```
struct S
{
    void foo() const {
        data_ = 10; // error
    }
private:
    int data_;
};
```

Ключевое слово mutable

Ключевое слово mutable позволяет определять поля, которые можно изменять внутри константных методов:

```
struct S
{
    void foo() const {
        data_ = 10; // now ok
    }
private:
    mutable int data_;
};
```

Копирование и перемещение объектов

Язык C++ позволяет определить поведение при копировании и перемещении объектов.

```
struct S
{
    S() = default;
    // Копирующий конструктор.
    S(const S &) = default;
    // Перемещающий конструктор.
    S(S &&) = default;

    // Оператор копирующего присваивания.
    S & operator=(const S &) = default;
    // Оператор перемещающего присваивания.
    S & operator=(S &&) = default;
private:
    int data_;
};
```

Копирование объектов

Определим конструктор и оператор копирования:

```
struct S
{
    S() = default;
    S(const S & other)
        : data_(other.data)
    {}
    S & operator=(const S & other) {
        data_ = other.data;
        return *this;
    }
private:
    std::vector<int> data_ = decltype(data_)(100500);
};
```

Копирование объектов

```
int main() {
    S s1;
    S s2(s1); // copy
    S s3 = s1; // copy
    s2 = s1; // operator=
}
```

Копирование объектов

- Если не определить конструктор копирования, то он сгенерируется компилятором.
- Если не определить оператор присваивания, то он тоже сгенерируется компилятором.

Перемещение объектов

Определим конструктор и оператор перемещения:

```
struct S
{
    S() = default;
    S(S && other)
        : data_(std::move(other.data_))
    {}
    S & operator=(S && other) {
        data_ = std::move(other.data_);
        return *this;
    }
private:
    std::vector<int> data_ = decltype(data_)(100500);
};
```

Перемещение объектов

```
int main() {
    S s1;
    S s2(std::move(s1)); // move
    S s3 = std::move(s2); // move

    S s4;
    s4 = std::move(s3); // move operator=

    foo(s1); // undefined use-after-move
}
```

Перемещение объектов

- Если не определить конструктор перемещения, то он сгенерируется компилятором.
- Если не определить оператор перемещения, то он тоже сгенерируется компилятором.

Правило 3/5/0

- Правило 3. Если определен пользовательский деструктор, копирующий конструктор или копирующий `operator=`, то следует определять все 3 сразу.
- Правило 5. Если определен пользовательский деструктор, копирующий конструктор или копирующий `operator=`, то их определение не позволяет компилятору сгенерировать методы для перемещения. Следует определять все 5 методов сразу.
- Правило 0. Если можно обойтись без пользовательских методов копирования/перемещения/деструкторов, то не следует их определять. Следуйте SRP - single responsibility principle.

Замечание: подробнее можно прочитать на cppreference или в cpp core guidelines.