Структуры и классы

Спасибо CSCenter

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;
    int ** data;
};
```

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

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;
}
```

Абстракция и инкапсуляция

Использование методов позволяет объединить данные и функции для работы с ними.

```
struct IntArray2D {
    int & get(size_t i, size_t j) {
        return data[i * b + j];
    }
    size_t a;
    size_t b;
    int * data;
};
```

```
    IntArray2D m = foo();
    for (size_t i = 0; i != m.a; ++i )
        for (size_t j = 0; j != m.b; ++j)
            if (m.get(i, j) < 0) m.get(i,j) = 0;
```

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);
```

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));
```

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 IntArray {
    explicit IntArray(size_t size)
        : size(size)
        , data(new int[size])
    { }

    ~IntArray() {
        delete [] data;
    }

    size_t size;
    int *  data;
};
```

Время жизни

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

```
void foo()
{
    IntArray a1(10); // создание a1
    IntArray a2(20); // создание a2
    for (size_t i = 0; i != a1.size; ++i) {
        IntArray a3(30); // создание a3
        ...
    } // удаление a3
} // удаление a2, потом a1
```

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*.

```
struct IntArray {
    explicit IntArray(size_t size);
    ~IntArray();
    int & get(size_t i);

    size_t size;
    int *  data;
};
```

```
    IntArray a(10);
    IntArray b = {20, new int[20]}; // ошибка
```

Объекты в динамической памяти

Создание

Для создания объекта в динамической памяти используется оператор `new`, он отвечает за вызов конструктора.

```
struct IntArray {
    explicit IntArray(size_t size);
    ~IntArray();

    size_t size;
    int *  data;
};
```

```
    // выделение памяти и создание объекта
    IntArray * pa = new IntArray(10);
    // только выделение памяти
    IntArray * pb =
        (IntArray *)malloc(sizeof(IntArray));
```

Объекты в динамической памяти

Удаление

При вызове оператора `delete` вызывается деструктор объекта.

```
// выделение памяти и создание объекта
IntArray * pa = new IntArray(10);

// вызов деструктора и освобождение памяти
delete pa;
```

Операторы `new []` и `delete []` работают аналогично

```
// выделение памяти и создание 10 объектов
// (вызывается конструктор по умолчанию)
IntArray * pa = new IntArray[10];

// вызов деструкторов и освобождение памяти
delete [] pa;
```

Placement new

```
// выделение памяти
void * p = myalloc(sizeof(IntArray));

// создание объекта по адресу p
IntArray * a = new (p) IntArray(10);

// явный вызов деструктора
a->~IntArray();

// освобождение памяти
myfree(p);
```

Проблемы с выравниванием:

```
char b[sizeof(IntArray)];
new (b) IntArray(20); // потенциальная проблема
```

Модификаторы доступа

Модификаторы доступа позволяют ограничивать доступ к методам и полям класса.

```
struct IntArray {
    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_;
};
```

Ключевое слово class

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

```
class 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_;
};
```

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_;
};
```

Определение констант

- Ключевое слово `const` позволяет определять типизированные константы.

```
double const pi = 3.1415926535;
int const day_seconds = 24 * 60 * 60;
// массив констант
int const days[12] = {31, 28, 31,
                      30, 31, 30,
                      31, 31, 30,
                      31, 30, 31};
```

- Попытка изменить константные данные приводит к неопределённому поведению.

```
    int * may = (int *) &days[4];
    *may = 30;
```

Указатели и const

В C++ можно определить как константный указатель, так и указатель на константу:

```
int a = 10;
const int * p1 = &a; // указатель на константу
int const * p2 = &a; // указатель на константу
*p1 = 20; // ошибка
p2  = 0;  // OK

int * const p3 = &a; // константный указатель
*p3 = 30; // OK
p3  = 0;  // ошибка

// константный указатель на константу
int const * const p4 = &a;
*p4 = 30; // ошибка
p4  = 0;  // ошибка
```

Указатели и const

Можно использовать следующее правило:

“слово const делает неизменяемым тип слева от него”.

```
int a = 10;
int * p = &a;

// указатель на указатель на const int
int const ** p1 = &p;

// указатель на константный указатель на int
int * const * p2 = &p;

// константный указатель на указатель на int
int ** const p3 = &p;
```

Ссылки и const

- Ссылка сама по себе является неизменяемой.

```
int a = 10;
int & const b = a; // ошибка
int const & c = a; // ссылка на константу
```

- Использование константных ссылок позволяет избежать копирования объектов при передаче в функцию.

```
Point midpoint(Segment const & s);
```

- По константной ссылке можно передавать rvalue.

```
    Point p = midpoint(Segment(Point(0,0),
                               Point(1,1)));
```

Константные методы

- Методы классов могут быть объявлены как `const`.

```
struct IntArray {
    size_t size() const;
};
```

- Такие методы не могут менять поля объекта (тип `this` — указатель на `const`).
- У константных объектов (через указатель или ссылку на константу) можно вызывать только константные методы:

```
IntArray const * p = foo();
p->resize(); // ошибка
```

- Внутри константных методов можно вызывать только константные методы.

Две версии одного метода

- Слово `const` является частью сигнатуры метода.

```
size_t IntArray::size() const {return size_;}
```

- Можно определить две версии одного метода:

```
struct IntArray {
    int get(size_t i) const {
        return data_[i];
    }
    int & get(size_t i) {
        return data_[i];
    }
private:
    size_t size_;
    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 IntArray {
    void foo() const {
        // нарушение логической константности
         data_[10] = 1;
    }
private:
    size_t size_;
    int *  data_;
};
```

Ключевое слово mutable

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

```
struct IntArray {
    size_t size() const {
        ++counter_;
        return size_;
    }

private:
    size_t size_;
    int *  data_;

    mutable size_t counter_;
};
```

Копирование объектов

```
struct IntArray {
    ...
private:
    size_t size_;
    int *  data_;
};

int main() {
    IntArray a1(10);
    IntArray a2(20);
    IntArray a3 = a1; // копирование
    a2 = a1; // присваивание

    return 0;
}
```

Конструктор копирования

Если не определить конструктор копирования, то он сгенерируется компилятором.

```
struct IntArray {
    IntArray(IntArray const& a)
        : size_(a.size_), data_(new int[size_])
    {
        for (size_t i = 0; i != size_; ++i)
            data_[i] = a.data_[i];
    }
    ...
private:
    size_t size_;
    int *  data_;
};
```

Оператор присваивания

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

```
struct IntArray {
    IntArray & operator=(IntArray const& a)
    {
        if (this != &a) {
            delete [] data_;
            size_ = a.size_;
            data_ = new int[size_];
            for (size_t i = 0; i != size_; ++i)
                data_[i] = a.data_[i];
        }
        return *this;
    }
    ...
};
```

Метод swap

```
struct IntArray {
    void swap(IntArray & a) {
        size_t const t1 = size_;
        size_ = a.size_;
        a.size_ = t1;

        int * const t2 = data_;
        data_ = a.data_;
        a.data_ = t2;
    }
    ...
private:
    size_t size_;
    int *  data_;
};
```

Метод swap

Можно использовать функцию std::swap и файла algorithm.

```
#include <algorithm>

struct IntArray {
    void swap(IntArray & a) {
        std::swap(size_, a.size_);
        std::swap(data_, a.data_);
    }
    ...
private:
    size_t size_;
    int *  data_;
};
```

Реализация оператора = при помощи swap

```
struct IntArray {
    IntArray(IntArray const& a)
        : size_(a.size_), data_(new int[size_]) {
        for (size_t i = 0; i != size_; ++i)
            data_[i] = a.data_[i];
    }
    IntArray & operator=(IntArray const& a) {
        if (this != &a)
            IntArray(a).swap(*this);
        return *this;
    }
    ...
private:
    size_t size_;
    int *  data_;
};
```

Запрет копирования объектов

Для того, чтобы запретить копирование, нужно объявить конструктор копирования и оператор присваивания как `private` и не определять их.

```
struct IntArray {
    ...
private:
    IntArray(IntArray const& a);
    IntArray & operator=(IntArray const& a);

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

Методы, генерируемые компилятором

Компилятор генерирует четыре метода:

1. конструктор по умолчанию,
2. конструктор копирования,
3. оператор присваивания,
4. деструктор.

Если потребовалось переопределить конструктор копирования, оператор присваивания или деструктор, то нужно переопределить и остальные методы из этого списка.

Поля и конструкторы

```
struct IntArray {
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size]) {
        for (size_t i = 0; i != size_; ++i)
            data_[i] = 0;
    }
    IntArray(IntArray const& a)
        : size_(a.size_), data_(new int[size_]) {
        for (size_t i = 0; i != size_; ++i)
            data_[i] = a.data_[i];
    }
    ...
private:
    size_t size_;
    int *  data_;
};
```

Деструктор, оператор присваивания и swap

```
~IntArray() {
    delete [] data_;
}

IntArray & operator=(IntArray const& a) {
    if (this != &a)
        IntArray(a).swap(*this);
    return *this;
}

void swap(IntArray & a) {
    std::swap(size_, a.size_);
    std::swap(data_, a.data_);
}
```

Методы

```
size_t size() const { return size_; }

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

void resize(size_t nsize) {
    IntArray t(nsize);
    size_t n = nsize > size_ ? size_ : nsize;
    for (size_t i = 0; i != n; ++i)
        t.data_[i] = data_[i];
    swap(t);
}
```