Курс-интенсив

Программирование на C++

ООП в C++

academy.rubius.com
sergey@prohanov.com
Сергей Проханов

Структуры

```
double distance(const Point& a, const Point& b)
{
   return std::sqrt(
      std::pow(b.x - a.x, 2) + std::pow(b.y - a.y, 2)
   );
}
```

```
uint8_t quadrant(const Point& point)
{
   return (point.x < 0)
       ? (point.y < 0 ? 3 : 2)
       : (point.y < 0 ? 4 : 1);
}
```

```
void move(Point& point, double dx, double dy)
{
   point.x += dx;
   point.y += dy;
}
```

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

Классы и объекты

- **Класс** — это фундамент, на котором построена в C++ поддержка объектно-ориентированного программирования.
- **Класс** — это определяемый пользователем тип.
- **Класс** — это логическая абстракция описывающая методы, свойства ещё не существующих объектов.

Классы и объекты

- Класс включает как данные, так и код, предназначенный для выполнения действий над этими данными.

- В классе данные объявляются в виде переменных, а код оформляется в виде функций

- Функции и переменные, составляющие класс, называются его членами

  Переменные – это **свойства** класса

  Функции – это **методы** класса

Классы и объекты

- **Объекты** - конкретное представление абстракции, имеющее свои свойства и методы.
- Созданные объекты на основе одного класса называются **экземплярами** этого класса.

Классы и объекты

- **Инкапсуляция** - это свойство, позволяющее объединить в классе и данные и методы, работающие с ними, и скрыть детали реализации от пользователя.

- **Наследование** - это свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.

- **Полиморфизм** - свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Классы и объекты

```
class Point
{
public:
    double x() const { return x_; }
    void setX(double x) { x_ = x; }

    double y() const { return y_; }
    void setY(double y) { y_ = y; }

    double distanceTo(const Point& point) const;
    uint8_t quadrant() const;
    void move(double dx, double dy);

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

Методы

```
double Point::distanceTo(const Point& point) const
{
    return std::sqrt(
        std::pow(point.x() - this->x_, 2) + std::pow(point.y() - this->y_, 2)
    );
}
```

```
uint8_t Point::quadrant() const
{
    return (this->x_ < 0)
        ? (this->y_ < 0 ? 3 : 2)
        : (this->y_ < 0 ? 4 : 1);
}

void Point::move(double dx, double dy)
{
    x_ += dx;
    y_ += dy;
}
```

Статические члены

```
class Point
{
public:
    // ...
    static Point origin() {
        return origin_;
    }

    double distanceToOrigin() const {
        return distanceTo(origin_);
    }

private:
    // ...
    static Point origin_;
};
```

```
Point p(1, 3);
p.distanceToOrigin();

p.origin();
Point::origin();
```

Конструкторы и деструкторы

```
class Date
{
public:
    explicit Date(int d = 1, int m = 1, int y = 1970);
    explicit Date(uint64_t timestamp);
    Date(const DateTime& dateTime);
    ~Date();
};
```

```
Date::Date(int d, int m, int y)
    : d_(d)
    , m_(m)
    , y_(y)
{
    //...
}
```

```
Date::~Date()
{
    //...
}
```

Делегирующие конструкторы

```
DateTime(const Date& date, const Time& time);

DateTime(int days, int months, int years, int hours, int minutes)
    : DateTime(Date(days, months, years), Time(hours, minutes))
{}
```

Копирование и перемещение

```
class ByteArray
{
private:
    size_t size_;
    std::byte *bytes_;

public:
    // копирование
    ByteArray(const ByteArray& other)
      : size_(other.size_)
    {
      std::memcpy(bytes_, other.bytes_, size_);
    }

    // перемещение
    ByteArray(ByteArray&& other)
      : size_(other.size_)
      , bytes_(other.bytes_)
    {}
}
```

```
ByteArray byteArray(1024);

// копия исходного объекта
ByteArray copied(byteArray);

// "захват" временного объекта
ByteArray captured(readFile(path));

// "перемещение" исходного объекта
ByteArray moved(std::move(byteArray))
```

Перегрузка операторов

```
struct Vector2D {
    double x;
    double y;

    Vector2D operator+(const Vector2D& other) {
       return {x + other.x, x + other.y};
    }

    Vector2D& operator=(const Vector2D& other) {
        x = other.x;
        y = other.y;
        return *this;
    }
};

Vector2D operator*(double c, const Vector2D& vector)
{
    return {c * vector.x, c * vector.y};
}

bool operator==(const Vector2D& lhs, const Vector2D& rhs)
{
    return (lhs.x == rhs.x) && (lhs.x == rhs.y);
}
```

default и delete

```
Vector2D() = default;
~Vector2D() = default;

Vector2D(const Vector2D& other) = delete;
Vector2D(Vector2D&& other) = default;

Vector2D& operator=(const Vector2D& other) = delete;
Vector2D& operator=(Vector2D&& other) = default;
```

RAII

Получение ресурса есть инициализация (англ. Resource Acquisition Is Initialization (**RAII**)) — программная идиома объектно-ориентированного программирования, смысл которой заключается в том, что с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение — с уничтожением объекта.

```
class ByteArray {
public:
    explicit ByteArray(size_t size)
        : bytes_(new std::byte[size])
    {}

    ~ByteArray() {
        delete[] bytes_;
    }

private:
    std::byte *bytes_;
};
```

```
class TempDirectory {
public:
    explicit TempDirectory()
        : path_(create_dir(get_unique_name()))
    {}

    ~TempDir() {
        remove_dir(path_);
    }

private:
    std::string path_;
};
```

Друзья класса

Дружественная функция — это функция, которая не является членом класса, но имеет доступ к членам класса, объявленным в полях private или protected.

```
class Vector {
   float v[4];
   friend Vector operator*(const Matrix&, const Vector&);
};
class Matrix {
   float v[4][4];
   friend Vector operator*(const Matrix&, const Vector&);
};

Vector operator*(const Matrix& matr, const Vector& vec){
   Vector r;
   for (int i = 0; i < 4; i++) {
      r.v[i] = 0;
      for (int j = 0; j < 4; j++) {
           r.v[i] += matr.v[i][j] * vec.v[j];
      }
   }

   return r;
}
```

Наследование

Наследование — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.

```
struct Person {
    std::string name;
    int age;
};

struct Employee : Person {
    std::string position;
};

struct Student : Person {
    std::string university;
    int course;
};
```

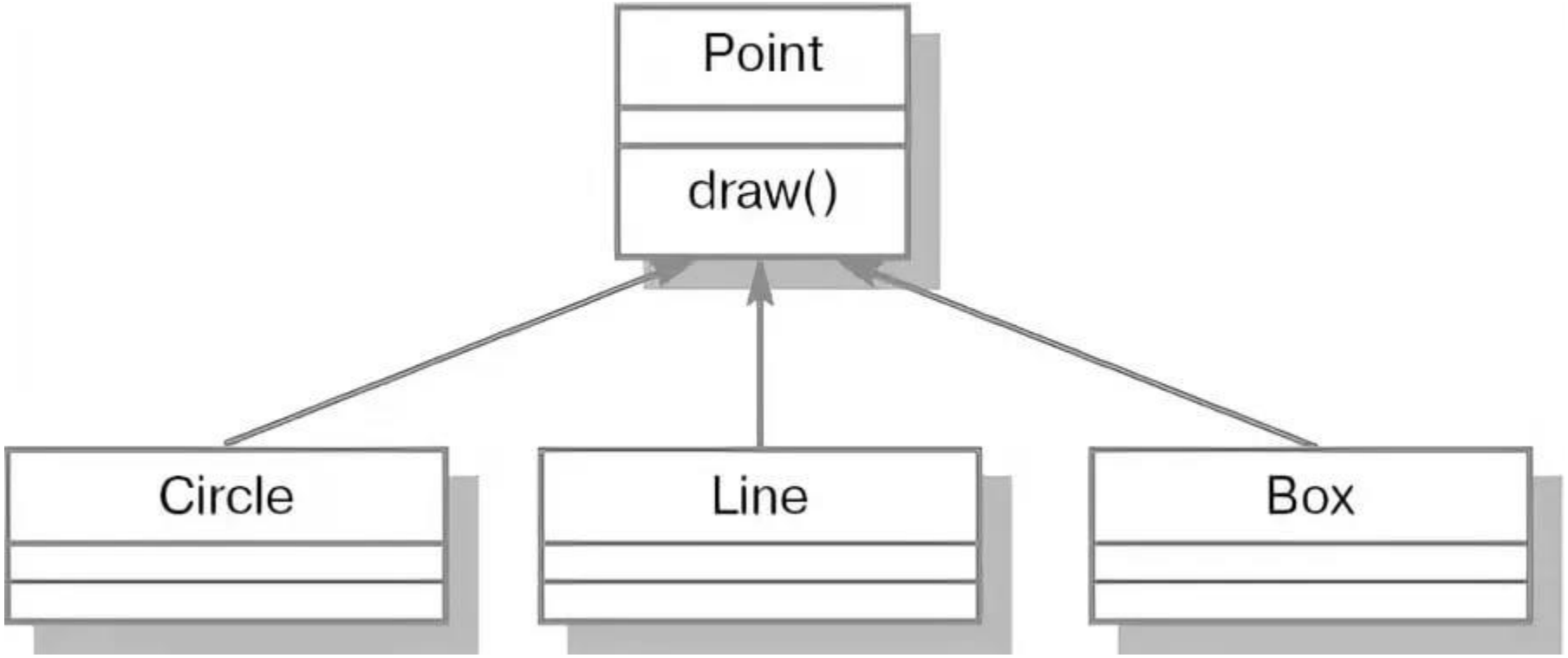
```
struct Doctor : public Employee {
    ...
};

struct Teacher : Employee {
    ...
};
```

Модификаторы доступа

	private	protected	public
Доступ из тела класса	открыт	открыт	открыт
Доступ из производных классов	закрыт	открыт	открыт
Доступ из внешних функций и классов	закрыт	закрыт	открыт

Полиморфизм

Полиморфи́зм (от греч. πολὺ- – много, и μορφή – форма) в языках программирования – возможность объектов с одинаковой спецификацией иметь различную реализацию.



Виртуальные функции

- Виртуальные функции решают проблему, связанную с полем типа, предоставляя возможность программисту объявить в базовом классе функции, которые можно заместить в каждом производном классе.

- Компилятор и загрузчик гарантируют правильное соответствие между объектами и функциями, применяемыми к ним.

Виртуальные функции

```
struct Animal {
    virtual ~Animal();

    std::string name() const;
    int age() const;

    virtual void voice() {}
};

struct Cat : Animal {
    void voice() override {
      print("Meow");
    }
};

struct Dog : Animal {
    void voice() override {
      print("Woof");
    }
};
```

```
std::vector<Animal*> animals = {
     new Cat,
     new Dog,
     ...
     new Cow,
};

for (Animal* animal : animals) {
    animal->voice();
}
```

Абстрактные классы

```
class Shape : public IDrawable {
public:
    Shape() = delete;
    void draw() override;

    Rect boundingRect() const;

private:
    PenStyle penStyle() const override;

private:
    Rect boundingRect_;
};
```

```
class Rectangle : public Shape {
public:
    void draw() override final;

    int width() const;
    int height() const;
};
```

```
class IDrawable {
public:
    // виртуальный деструктор
    virtual ~IDrawable() = default;

    // чисто виртуальный метод
    virtual void draw() = 0;

private:
    virtual PenStyle penStyle() const = 0;

    // реализация по умолчанию
    virtual Color color() const {
       return DEFAULT_COLOR;
    }
};
```

```
class RedRectangle final : public Rectangle {
    Color color() const override {
        return Red;
    }
};
```

Множественное наследование

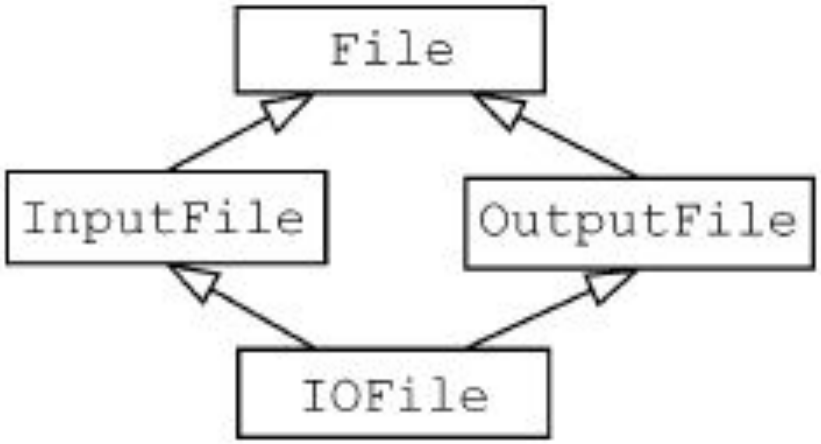
```
struct Rectangle {
    Rectangle(int width, int height)
      : width(width)
      , height(height)
    {}

    int width;
    int height;
};
```

```
struct TextArea {
    explicit TextArea(std::string text)
      : text(std::move(text))
    {}

    std::string text;
};

struct IClickable {
    virtual void onClick() = 0;
};
```

```
struct Button : public Rectangle, public TextArea, public IClickable {
    Button(int width, int height, std::string text)
      : Rectangle(width, height)
      , TextArea(std::move(text))
    {}

    void onClick() override {
      //...
    }
};
```

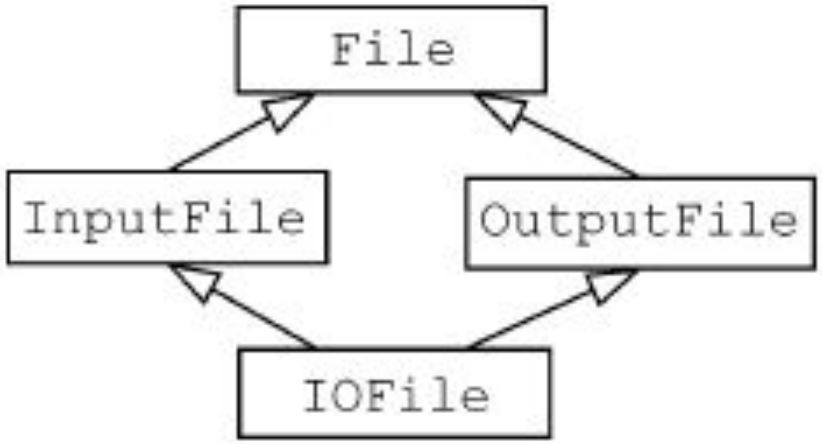
Множественное наследование

```
struct File {
   virtual bool open();
   virtual void close();
};

struct InputFile : File {
   ByteArray read(size_t size);
};

struct OutputFile : File {
   void write(ByteArray data);
};

struct IOFile : InputFile , OutputFile {};
```



Виртуальное наследование

```
struct File {
   virtual bool open();
   virtual void close();
};

struct InputFile : virtual File {
   ByteArray read(size_t size);
};

struct OutputFile : virtual File {
   void write(ByteArray data);
};

struct IOFile : InputFile , OutputFile {};
```



dynamic_cast

```
Shape* shape = currentSelection();

auto* rectangle = dynamic_cast<Rectangle*>(shape);

if (rectangle) {
    // …
}


Shape& shape = currentSelection();
auto& rectangle = dynamic_cast<Rectangle&>(shape);
```

Пространства имён

```
namespace Image
{
    class Converter;
    class Reader;
    class Writer;
}

namespace Video
{
    class Converter;
    class Reader;
    class Writer;
}

Image::Reader imageReader;
Video::Converter converter;

std::string v;

using namespace Image;
Reader imageReader;
```

```
namespace Network
{
    class Configuration;

    namespace Tcp
    {
       class Socket;
    }

    namespace Http
    {
       class Request;
       class Response;
    }
}

namespace Core::Algorithm::Collections
{
    void sort(Collection& collection);
}
```

Курс-интенсив

Программирование на C++

ООП в C++

academy.rubius.com
sergey@prohanov.com
Сергей Проханов