ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Лекция № | / **04**
04.03.2019 г.

VIRTUAL DESTRUCTOR

```
class Shape{
    int x, y;
public:
    Shape(int x, int y);
    ~Shape(){
        printf("Dtor shape!\n");
    }
    ...
};
```

```
class Circle: public Shape{
    int radius;
public:
    Circle(int x, int y, int radius);
    ~Circle(){
        printf("Dtor circle!\n");
    }
    ...
};
```

```
int main(){
    Shape* shapePtr = new Circle(0, 0, 1);

    delete shapePtr;     // Dtor shape!

    return 0;
}
```

VIRTUAL DESTRUCTOR

```
class Shape{
    int x, y;
public:
    Shape(int x, int y);
    virtual ~Shape(){
        printf("Dtor shape!\n");
    }
    ...
};
```

```
class Circle: public Shape{
    int radius;
public:
    Circle(int x, int y, int radius);
    ~Circle(){
        printf("Dtor circle!\n");
    }
    ...
};
```

```
int main(){
    Shape* shapePtr = new Circle(0, 0, 1);

    delete shapePtr;     // Dtor circle!
                         // Dtor shape!
    return 0;
}
```

Rule: если предполагается наследование, то всегда делайте деструктор виртуальным!

KEYWORD FINAL

Specifies that a virtual function cannot be overridden in a derived class or that a class cannot be inherited from.

C++ 11

KEYWORD FINAL

```
class Shape{
    int x, y;
public:
    Shape(int x, int y);
    virtual ~Shape() final{
        printf("Dtor shape!\n");
    }

    ...
};
```

```
class Circle: public Shape{
    int radius;
public:
    Circle(int x, int y, int radius);
    ~Circle(){
        printf("Dtor circle!\n");
    }

    ...
};
```

```
int main(){
    Shape* shapePtr = new Circle(0, 0, 1);

    delete shapePtr;

    return 0;
}
```

Error compilation.

KEYWORD FINAL

```
class Shape final{
    int x, y;
public:
    Shape(int x, int y);
    ~Shape(){
        printf("Dtor shape!\n");
    }

    ...
};
```

```
class Circle: public Shape{
    int radius;
public:
    Circle(int x, int y, int radius);
    ~Circle(){
        printf("Dtor circle!\n");
    }

    ...
};
```

```
int main(){
    Shape* shapePtr = new Circle(0, 0, 1);

    delete shapePtr;

    return 0;
}
```

Error compilation.

KEYWORD FINAL

```
class Shape{
    int x, y;
public:
    Shape(int x, int y);
    virtual ~Shape(){
        printf("Dtor shape!\n");
    }
    ...
};
```

```
class Circle final : public Shape{
    int radius;
public:
    Circle(int x, int y, int radius);
    ~Circle(){
        printf("Dtor circle!\n");
    }
    ...
};
```

```
int main(){
    Shape* shapePtr = new Circle(0, 0, 1);

    delete shapePtr;     // Dtor circle!
                         // Dtor shape!
    return 0;
}
```

Rule: классы, от которых не планируете наследоваться, необходимо объявлять со спецификатором **final**!

KEYWORD OVERRIDE

Specifies that a virtual function overrides another virtual function.

C++ 11

KEYWORD OVERRIDE

```
struct A
{
    virtual void foo();
    void bar();
};

struct B : A
{
    void foo() const override;   // Error: B::foo does not override A::foo
                                 // (signature mismatch)

    void foo() override;         // OK: B::foo overrides A::foo
    void bar() override;         // Error: A::bar is not virtual
};
```

Rule: все переопределенные виртуальные функции объявляйте со спецификатором **override**!

ABSTRACT CLASS

Defines an abstract type which cannot be instantiated, but can be used as a base class.

ABSTRACT CLASS

```
struct Abstract {
    virtual void f() = 0;    // pure virtual
}; // "Abstract" is abstract
```

ABSTRACT CLASS

```
struct Abstract {
    virtual void f() = 0;    // pure virtual
}; // "Abstract" is abstract

struct Concrete : Abstract {
    void f() override {}     // non-pure virtual
    virtual void g();        // non-pure virtual
}; // "Concrete" is non-abstract
```

ABSTRACT CLASS

```
struct Abstract {
    virtual void f() = 0;    // pure virtual
}; // "Abstract" is abstract

struct Concrete : Abstract {
    void f() override {}     // non-pure virtual
    virtual void g();        // non-pure virtual
}; // "Concrete" is non-abstract

struct Abstract2 : Concrete {
    void g() override = 0;  // pure virtual overrider
}; // "Abstract2" is abstract
```

ABSTRACT CLASS

```
struct Abstract {
    virtual void f() = 0;    // pure virtual
}; // "Abstract" is abstract

struct Concrete : Abstract {
    void f() override {}     // non-pure virtual
    virtual void g();        // non-pure virtual
}; // "Concrete" is non-abstract

struct Abstract2 : Concrete {
    void g() override = 0;  // pure virtual overrider
}; // "Abstract2" is abstract

int main()
{
    // Abstract a;       // Error: abstract class
    Concrete b;          // OK
    Abstract& a = b;     // OK to reference abstract base
    a.f();               // virtual dispatch to Concrete::f()
    // Abstract2 a2;     // Error: abstract class (final overrider of g() is pure)
}
```

INTERFACE

```
class Shape {

public:
    virtual void move(int x, int y) = 0;          // pure virtual
    virtual void rotate(double angle) = 0;        // pure virtual
    virtual double square(double angle) = 0;      // pure virtual
    virtual double perimeter(double angle) = 0;   // pure virtual
    virtual ~Shape() {}                           // non-pure virtual

}; // "Shape" is abstract
```

ACCESS MODIFIERS

public	private	protected
Все, кто имеет доступ к описанию класса	Функции-члены класса	Функции-члены класса
	«Дружественные» функции	«Дружественные» функции
	«Дружественные классы»	«Дружественные классы»
		Функции-члены производных классов

ACCESS MODIFIERS

```
class Base {
public:
    Base();

protected:
    int getSecret() { return secret; }

private:
    int secret;
};
```

ACCESS MODIFIERS

```
class Base {
public:
    Base();

protected:
    int getSecret() { return secret; }

private:
    int secret;
};

class Derived : public Base {
public:
    void doSomething() {
        int x = getSecret();
        // ...
    }

    void error() {
        secret++;   // ERROR
    }
};
```

Protected доступен для наследника.

Нет доступа к private.

FRIEND DECLARATION

```
class Base {
public:
    Base();

private:
    int privateField;

    friend void globalFunc(Base& x);
    friend class FriendlyClass;
};
```

FRIEND DECLARATION

```
class Base {
public:
    Base();

private:
    int privateField;

    friend void globalFunc(Base& x);
    friend class FriendlyClass;
};

void globalFunc(Base& x) {
    x.privateField = -1;
}

class FriendlyClass {
public:
    int func(Base& x) {
        return x.privateField;
    }
};
```

INHERITANCE AND ACCESS SPECIFIERS

	Исходный модификатор доступа члена класса A		
Вид наследования	**public**	**private**	**protected**
`class B : public A {};`	public	Inaccessible	protected
`class B : private A {};`	private	Inaccessible	private
`class B : protected A {};`	protected	Inaccessible	protected

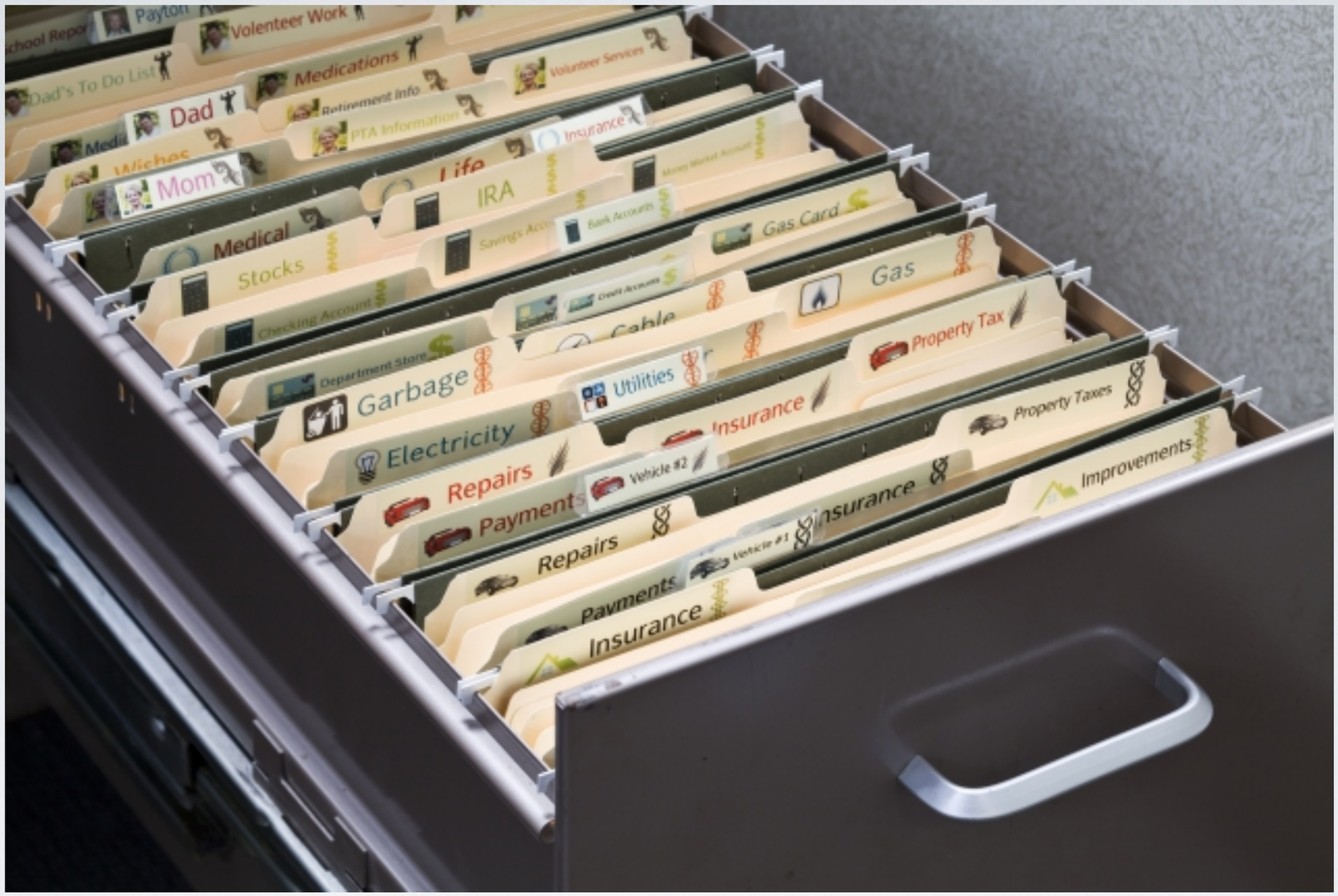
STRUCT VS CLASS

```
struct X
{
    int a;   // public
};
```

```
class DerivedX : X // : public X
{
    ...
};
```

```
class Y
{
    int a;   // private
};
```

```
class DerivedY : Y // : private Y
{
    ...
};
```



NAMESPACE (ПРОСТРАНСТВО ИМЕН)

Способ создать отдельные области видимости для классов, констант, функций…

Модуль Awesome

```
class Something {
public:
    Something(const char *name);
    // ...
};
```

Модуль Joe

```
class Something {
public:
    Something();
    // ...
};
```

Конфликт имён

```
class AwesomeSomething {
public:
    AwesomeSomething(const char *name);
    // ...
};

typedef AwesomeSomething *AwesomeSomethingPtr;

const int AwesomeNumber = 42;

enum AwesomeTypes {
    AWESOME_FOO,
    AWESOME_BAR,
    /* ... */
};

void AwesomeGlobalFunction(AwesomeSomething *);
```

Так себе решение

```
// awesome.h
namespace Awesome {
    class Something {
    public:
        Something(const char *name);
        // ...
    };

    typedef Something *SomethingPtr;

    const int Number = 42;

    enum Types { FOO, BAR, /* ... */ };

    void GlobalFunction(Something *);
}
```

Настоящее решение — создать пространство имён

```cpp
// awesome.cxx

#include "awesome.h"

Awesome::Something::Something(const char *name) {
    int n = Number;   // здесь префикс не нужен!

    // ...
}

void Awesome::GlobalFunction(Awesome::Something *ps) {
    /* ... */
}
```

Для адресации используется префикс Awesome::

```cpp
// client.cxx

#include "awesome.h"



void f(int q) {
    Awesome::Something x;
    int n = Awesome::Number;

    if (q == Awesome::FOO) {
        ...
    }

    Awesome::GlobalFunction(&x);
}
```

```cpp
// client.cxx

#include "awesome.h"

using namespace Awesome;

void f(int q) {
    Something x;
    int n = Number;

    if (q == FOO) {
        ...
    }

    GlobalFunction(&x);
}
```

Rule: не используйте using namespace в header файлах!!!

Клиентский код

```
namespace His_lib {
    class String { /* ... */ };
    class Vector { /* ... */ };
}

namespace Her_lib {
    class String { /* ... */ };
    class Vector { /* ... */ };
}

namespace My_lib {
    using namespace His_lib;
    using namespace Her_lib;

    using His_lib::String;    // разрешение возможных конфликтов
    using His_lib::Vector;    // разрешение возможных конфликтов

    class List { /* ... */ };
}
```

Отбор и селекция пространств имён

```c
// something.c
void public_interface_function() {
    // ...
    internal_function2();
}

void one_more_public_interface_function() {
    internal_function1();
    // ...
}

static void internal_function1() {
    // ...
}

static void internal_function2() {
    // ...
}
```

Сокрытие деталей реализации, C-style

```cpp
// something.cxx
namespace {
    void internal_function1() {
        // ....
    }

    void internal_function2() {
        // ....
    }
}

void public_interface_function() {
    // ....
    internal_function2();
}

void one_more_public_interface_function() {
    internal_function1();
    // ....
}
```

C++ style. Unnamed namespace.

OPERATOR OVERLOADING

```
struct Vector2D {
    double x, y;
    // ....
};

// можно писать так:
Vector2D x, y;
Vector2D z = x.add(y);
z = z.mul(x);
z = z.sub(x.div(y));
```

```
// а хочется писать так:
z = x + y;
z *= x;
z -= x / y;
```

```
// Магия C++:
z = x + y;
z = 2 * x;
z *= x;
z -= x / y;
```

```
// Превращается в:
z = x.operator+(y);
z = operator*(2, x);
z.operator*=(x);
z.operator-=(x.operator/(y));

// Остаётся только определить
// эти функции и методы…
```

OPERATOR OVERLOADING

```
class Vector2D {
public:
    // ...
    Vector2D &operator*=(const Vector2D &other);
    Vector2D &operator*=(double x);
    Vector2D &operator-=(const Vector2D &other);
    Vector2D operator+(const Vector2D &other) const;
    Vector2D operator/(const Vector2D &other) const;
};

Vector2D operator*(double x, const Vector2D &other);
```

ASSIGNMENT OPERATORS

- `+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=`
- Метод класса!

- **operator+=** не генерируется автоматически из **operator=** и **operator+**.

- Нужно возвращать ***this**.

COPY ASSIGNMENT OPERATOR

- Object& operator=(const Object&);
- Только метод класса!
- Существует реализация по умолчанию: копирование всех полей.
- Если в классе присутствуют указатели, реализация по умолчанию приносит проблемы.
- Нужно возвращать ***this**.

AUTOMATIC MEMBER FUNCTION GENERATION

- X()
- X(const X &)
- X& operator=(const X &)
- X(X &&)
- X& operator=(X &&)
- ~X()

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

    Point(double _x, double _y, double _z)
        : x(_x), y(_y), z(_z) {}

    // Не надо такое писать!
    Point(const Point &other) :
        x(other.x), y(other.y), z(other.z)
    {}
};
```

**Такой конструктор можно не писать,
он будет сгенерирован автоматически
(копирование всех полей)**

RULE OF THREE

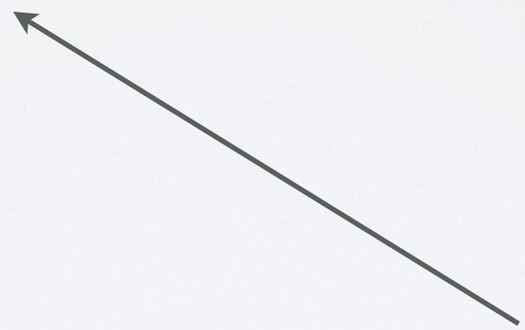
Если класс или структура определяет один из методов (конструктор копий, оператор присваивания или деструктор), то они должны явным образом определить все три метода!

KEYWORD DEFAULT

```
class SomeClass final {

public:
    SomeClass() = default;
    SomeClass(const SomeClass&) = default;
    ~SomeClass() = default;

    SomeClass& operator=(const SomeClass&) = default;
};
```



В этом случае вы точно знаете, что делаете!

KEYWORD DELETE

```
class SomeClass final {

public:
    SomeClass() = delete;
    SomeClass(const SomeClass&) = delete;
    ~SomeClass() = default;

    SomeClass& operator=(const SomeClass&) = delete;
};
```

DEFAULT ARGUMENTS

```
char *format_number(double d, int precision = 2);

format_number(10.0, 3);      // d: 10.0, precision: 3
format_number(10.0, 2);      // d: 10.0, precision: 2
format_number(10.0);         // d: 10.0, precision: 2
```

```
void f(int a, int b = 2, int c = 3);  // OK
void g(int a = 1, int b = 2, int c);  // ERROR!
void h(int a, int b = 3, int c);      // ERROR!
```

КОНЕЦ ЧЕТВЁРТОЙ ЛЕКЦИИ

```
~Lection() = default;
```