ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Лекция № | / **03**
19.02.2018 г.

ТЕСТИРОВАНИЕ

- Статическое *(без запуска)* и динамическое *(с запуском)*.
- Ручное и машинное.
- Модульное *(unit)*, интеграционное, системное, …
- Дымовое *(хоть что-то работает?)*, регрессионное *(не вылезли ли старые баги?)*, приёмочное *(на соответствие спецификации)*, нагрузочное *(выдерживает ли программа нагрузку?)*, …
- …

```c
/********** power.h **********/
// Возвести x в квадрат
long square(int x);

// Возвести x в куб
long cube(int x);

/********** power_test.c **********/
#include <assert.h>
#include "power.h"

void test_square() {
    assert(square(2) == 4);
}

void test_cube() {
    assert(cube(2) == 8);
}

int main() {
    test_square();
    test_cube();
    return 0;
}
```

- Дымовое тестирование:
  - Проверка самых распространённых случаев.
- Полное модульное тестирование:
  - По максимуму проверить все развилки в коде.
  - Выделенные числа: –1, 0, 1, 2.
  - Нулевые указатели.
  - …

КРАТКАЯ ИСТОРИЯ C++

- 1979: «C with classes».
- 1983: вышла первая версия C++.
- 1998: C++98.
- …
- 2011: C++11.
- 2014: C++14, последний стандарт языка.

ВЛАДЕНИЕ C++

- Уровень 1. **«Более удобный C»**.
- Уровень 2. **«Обычный ООП»**. Классы, наследование, полиморфизм.
- Уровень 3. **«Специфичный C++»**. Переопределение операторов, шаблонные классы и т.д.

КЛАССЫ

```
// foo.h
class Foo {
public:
     Foo(int x);
     Foo();
      ~Foo();

     int something();

private:
     int internal();
     int value;
};
```

Конструкторы

Деструктор

Метод

Частный метод

```cpp
// foo.cpp
Foo::Foo(int x)
    : value(x)
{
}

Foo::Foo()
{
    value = 0;
}

int Foo::something()
{
    return value + internal();
}

int Foo::internal()
{
    // ....
}

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

Начальное значение для поля класса

РАБОТА С КЛАССАМИ

```
void f() {
   Foo foo;

   // ....
}

void g() {
   Foo *f = new Foo(123);

   // ....

   h(f);
}

void h(Foo *foo) {
   // ....
   delete foo;
}
```

Вызов конструктора

Вызов деструктора

```
int *arrint = new int[1000];
// int *arrint = (int *)malloc(1000*sizeof(int));


Foo *arrfoo = new Foo[1000];
// Foo *arrfoo = (Foo *)malloc(1000*sizeof(Foo));
//     ... и вызвать Foo() 1000 раз для каждого элемента
```

```
delete arrint;
// free(arrint);


delete[] arrfoo;
// ... вызвать Foo::~Foo() 1000 раз для каждого элемента
// ... затем вызвать free(arrfoo);
```

ГЛОБАЛЬНЫЕ ОБЪЕКТЫ

```
Foo foo;

int main()
{
    // ....
}
```

Вызов конструктора: до **main()**

Вызов деструктора: после **main()**

INLINE

```
// foo.h
class Foo {
    // ...
    int getX();

private:
    int x;
};

inline int Foo::getX() {
    return x;
}

inline int max(int x, int y) {
    return (x < y) ? y : x;
}
```

```
Foo foo;

// Тот же код, что и в:
//     int a = foo.x;
int a = foo.getX();
```

Тело функции объявляется в заголовочном файле!

```
// foo.h
class Foo {
    // ...
    int getX() { return x; }
private:
    int x;
};
```

Более короткая запись inline-методов

СТАТИЧЕСКИЕ ЧЛЕНЫ

```
// box.h
class Box {
public:
    Box(double l, double b, double h) :
        length(l), breadth(b), height(h)
    {
        ++objectCount;
    }

    ~Box() { --objectCount; }

    static int getObjectCount() { return objectCount; }

private:
    double length, height, breadth;

    static int objectCount;
};

//-------------------------------------------------------------------
// box.cpp
int Box::objectCount = 0;
```

Вызов функции:

```
Box::getObjectCount()
```

СОКРЫТИЕ ИМЁН

```
struct Ha {
    Ha(int);

    int x;
};
```

```
inline Ha::Ha(int x) {
    this->x = x;

    for (int x = 0; x < 100; ++x) {
        // ...
    }
}
```

НАСЛЕДОВАНИЕ

```
class Base {
public:
    Base();
    void say();
    virtual void sayVirtual();
};

inline Base::Base()             { puts("Base"); }
inline void Base::say()         { puts("Base::say"); }
inline void Base::sayVirtual() { puts("Base::sayVirtual"); }
```

```
//-------------------------------------------------------------

class Derived : public Base {
public:
    Derived()                   { puts("Derived"); }
    void say()                  { puts("Derived::say"); }
    void sayVirtual()           { puts("Derived::sayVirtual"); }
};

Base *b = new Derived;  // -> Base Derived
b->say();               // -> Base::say
b->sayVirtual();        // -> Derived::sayVirtual
delete b;
```

```
class MyVector {
public:
    MyVector(int size);

    // ...
};

class Vector3D : public MyVector {
public:
    Vector3D() : MyVector(3) {}

    // ...
};
```

Явный вызов конструктора базового класса

```
struct Base {
    virtual void cool();
};

struct Derived : Base {
    /* virtual */ void cool();    // virtual можно не ставить
};
```

```
struct Base {
    virtual void cool(int arg);   // поменяли сигнатуру метода
};

struct Derived : Base {
    /* virtual */ void cool();    // ОЙ. Это теперь другой метод
};
```

```
struct Base {
    virtual void cool();
};

struct Derived : Base {
    void cool() override;          // привязка к базовому методу
};
```

Виртуальные методы требуют аккуратности!

МОДИФИКАТОРЫ ДОСТУПА

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

```
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
    }
};
```

```
class XFiles {
    XFiles();

private:
    friend void mulder(XFiles *x);
    friend class Scully;

    int secrets[100];
};

void mulder(XFiles *x) {
    x->secrets[0]++;
}
```

```
class Scully {
public:
    int inspect(XFiles *x) {
        int s = 0;
        for (int i = 0; i < 100; ++i)
            s += x->secrets[i];
        return s;
    }
};
```

friend-доступ

STRUCT И CLASS

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

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

ВИДЫ НАСЛЕДОВАНИЯ

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

ССЫЛКИ

```
int a = 2;
int &b = a;

b = 3;
printf("%d\n", a);      // 3
```

```
void noop(int p, int q) {
    int r = p;
    p = q;
    q = r;
}

void exch(int *p, int *q) {
    int r = *p;
    *p = *q;
    *q = r;
}

void exch2(int &p, int &q) {
    int r = p;
    p = q;
    q = r;
}
```

ССЫЛКИ И УКАЗАТЕЛИ

```
int *p;
```

```
int &r;
```

Нельзя! Нужна инициализация

```
int *&wat1;
```

```
int &*wat2;
```

Нельзя делать указатель на ссылку

КОНЕЦ ТРЕТЬЕЙ ЛЕКЦИИ

```
Lection::~Lection() { }
```