



Лекция № 3

Углубленное программирование на языке C / C++

Алексей Петров

Лекция №3.

Основные вопросы ООП на языке C++



1. Инкапсуляция и ответственность класса. Принципы SRP, OCP. Идиома RAII.
2. Праводопустимые выражения. Конструкторы (операции) переноса и иные расширения объектной модели в C++11.
3. Инкапсуляция и вопросы производительности.
4. **Постановка задач к практикуму №3.**

Рекомендуемая литература:

модуль №2 (1 / 2)



- Дейтел Х., Дейтел П. Как программировать на C++. — Бином-Пресс, 2009. — 800 с.
- Липпман С., Лажойе Ж. Язык программирования C++. Вводный курс. — Невский Диалект, ДМК Пресс. — 1104 с.
- Липпман С., Лажойе Ж., Му Б. Язык программирования C++. Вводный курс. — Вильямс, 2007. — 4-е изд. — 896 с.
- Прата С. Язык программирования C++. Лекции и упражнения. — Вильямс, 2012. — 6-е изд. — 1248 с.: ил.
- Саттер Г. Новые сложные задачи на C++. — Вильямс, 2005. — 272 с.
- Саттер Г. Решение сложных задач на C++. — Вильямс, 2008. — 400 с.
- Саттер Г., Александреску А. Стандарты программирования на C++. — Вильямс, 2008. — 224 с.
- Страуструп Б. Программирование. Принципы и практика использования C++. — Вильямс, 2011. — 1248 с.
- Страуструп Б. Язык программирования C++. — Бином, 2011. — 1136 с.

Рекомендуемая литература:

модуль №2 (2 / 2)



- Шилдт Г. С++: базовый курс. — Вильямс, 2008. — 624 с.
- Шилдт Г. С++. Методики программирования Шилдта. — Вильямс, 2009. — 480 с.
- Шилдт Г. Полный справочник по С++. — Вильямс, 2007. — 800 с.
- Abrahams, D., Gurtovoy, A. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond* (Addison Wesley Professional, 2004).

Инкапсуляция — базовый принцип ООП



Инкапсуляция, или сокрытие реализации, является фундаментом объектного подхода к разработке ПО.

- Следуя данному подходу, программист рассматривает задачу в терминах предметной области, а создаваемый им продукт видит как **совокупность абстрактных сущностей — классов** (в свою очередь формально являющихся пользовательскими типами).
- Инкапсуляция **предотвращает прямой доступ** к внутреннему представлению класса из других классов и функций программы.
- Без нее теряют смысл остальные основополагающие принципы объектно-ориентированного программирования (ООП): наследование и полиморфизм. Сущность инкапсуляции можно отразить формулой:

Открытый интерфейс + скрытая реализация

Класс: в узком или широком смысле?



Принцип инкапсуляции распространяется не только на классы (`class`), но и на структуры (`struct`), а также объединения (`union`). Это связано с расширительным толкованием понятия «класс» в языке C++, трактуемом как в узком, так и широком смысле:

- **класс в узком смысле** — *одноименный* составной пользовательский тип данных, являющийся контейнером для данных и алгоритмов их обработки. Вводится в текст программы определением типа со спецификатором `class`;
- **класс в широком смысле** — *любой* составной пользовательский тип данных, агрегирующий данные и алгоритмы их обработки. Вводится в текст программы определением типа с одним из спецификаторов `struct`, `union` или `class`.

Каждое определение класса вводит **новый тип данных**. Тело класса определяет **полный перечень его членов**, который не может быть расширен после закрытия тела.

Указатель `this`



Указатель `this` — **неявно определяемый константный указатель** на объект класса, через который происходит вызов соответствующего нестатического метода (**чьим «нулевым» неявным параметром он является**).

Для неконстантных устойчивых методов класса `T` имеет тип `T *const`, для константных — имеет тип `const T *const`, для неустойчивых — `volatile T *const`.

Указатель `this` **допускает разыменование** (`*this`) и его применение внутри методов допустимо, но чаще всего излишне. Исключение составляют две ситуации:

- сравнение адресов объектов:

```
if (this != someObj) /* ... */
```

- оператор `return`:

```
return *this;
```

Класс как область видимости



Класс — наряду с блоком, функцией и пространством имен — **является конструкцией C++**, которая **вводит** в состав программы одноименную **область видимости**. (Строго говоря, область видимости вводит определение класса, а именно его тело.)

- Все члены класса видны в нем самом с момента своего объявления. Порядок объявления членов класса важен: нельзя ссылаться на члены, которые предстоит объявить позднее. Исключение составляет разрешение имен в определениях встроенных методов, а также имен (**статических членов**), используемых как аргументы по умолчанию.

В области видимости класса находится не только его тело, но и внешние определения его членов: методов и статических атрибутов.

Конструкторы и деструкторы (1 / 2)



Конструктор — метод класса, автоматически применяемый к каждому экземпляру (объекту) класса перед первым использованием (в случае динамического выделения памяти — после успешного выполнения операции `new`).

Освобождение ресурсов, захваченных в конструкторе класса либо на протяжении времени жизни соответствующего экземпляра, осуществляет **деструктор**.

В связи с принятым по умолчанию почленным порядком инициализации и копирования объектов класса в большинстве случаев возникает необходимость в реализации, — наряду с конструктором по умолчанию, — **конструктора копирования** и перегруженной **операции-функции присваивания** `operator=`.

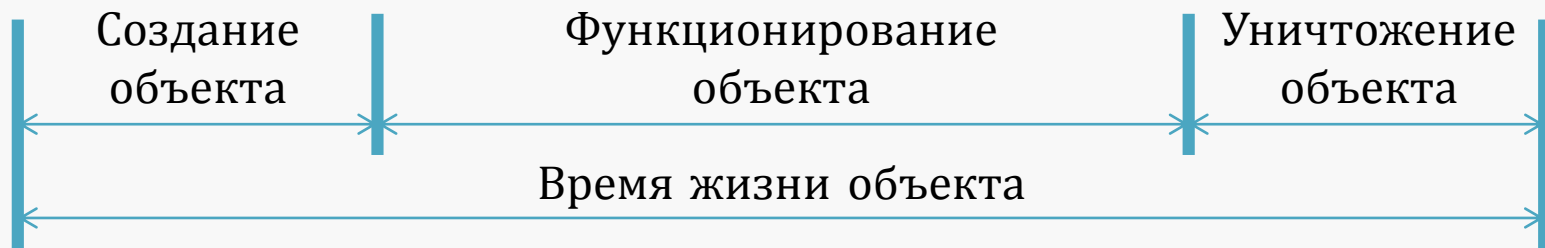
Конструкторы и деструкторы (2 / 2)



Выполнение любого конструктора состоит из двух фаз:

- фаза явной (неявной) инициализации (**обработка списка инициализации**);
- фаза вычислений (**исполнение тела конструктора**).

Конструктор **не может определяться** со спецификаторами **const** и **volatile**. Константность и неустойчивость объекта устанавливается по завершении работы конструктора и снимается перед вызовом деструктора.



Инициализация без конструктора (1 / 2)



Класс, все члены которого открыты, может задействовать механизм **явной позиционной инициализации**, ассоциирующий значения в списке инициализации с членами данных в соответствии с их порядком.

```
struct Sample {  
    int          int_prm;  
    double       dbl_prm;  
    std::string  str_prm;  
};  
  
// ...  
  
Sample sample = { 1, -3.14, "dictum factum" };
```

Инициализация без конструктора (2 / 2)



Преимуществами такой техники выступают:

- скорость и эффективность, особо значимые при выполнении во время запуска программы (для глобальных объектов).

Недостатками инициализации без конструктора являются:

- пригодность только для классов, члены которых открыты;
- отсутствие поддержки инкапсуляции и абстрактных типов;
- требование предельной точности и аккуратности в применении.

Конструкторы по умолчанию (1 / 2)



Явный конструктор по умолчанию **не требует задания значений** его параметров, хотя таковые могут присутствовать в сигнатуре (но в таком случае должны иметь значения по умолчанию).

```
struct Sample {  
    Sample(int ipr = 0, double dpr = 0.0);  
    // ...  
};
```

Наличие формальных параметров в конструкторе по умолчанию позволяет **сократить общее число конструкторов** и объем исходного кода.

Конструкторы по умолчанию (2 / 2)



Если в классе определен хотя бы один конструктор с параметрами, то при использовании класса со стандартными контейнерами и динамическими массивами экземпляров конструктор по умолчанию **обязателен**.

```
Sample *samples = new Sample[NUM_OF_SAMPLES];
```

Если конструктор по умолчанию **не определен**, но существует хотя бы один конструктор с параметрами, в определении объектов должны присутствовать аргументы. Если ни одного конструктора не определено, объект класса не инициализируется (память под статическими объектами по общим правилам обнуляется).



Конструкторы с параметрами: пример



```
struct Sample {  
    Sample(int prm) : _prm (prm) {}  
private:  
    int _prm;  
};
```

```
// все вызовы конструктора допустимы и эквивалентны  
Sample sample1(10),  
    sample2 = Sample(10),  
    sample3 = 10; // для одного аргумента
```



Массивы объектов: пример



// массивы объектов класса определяются

// аналогично массивам объектов базовых типов

// для конструктора с одним аргументом

```
Sample array1[] = { 10, -5, 0, 127 };
```

// для конструктора с несколькими аргументами

```
Sample array2[5] = {  
    Sample(10, 0.1),  
    Sample(-5, -3.6),  
    Sample(0, 0.0),  
    Sample() // если есть конструктор по умолчанию  
};
```


Закрытые и защищенные конструкторы



Описание конструктора класса как **защищенного** или **закрытого** дает возможность ограничить или полностью запретить отдельные способы создания объектов класса.

В большинстве случаев закрытые и защищенные конструкторы используются для:

- **предотвращения копирования** одного объекта в другой;
- указания на то, что конструктор **должен вызываться** только **для создания подобъектов** базового класса в объекте производного класса, а не создания объектов, непосредственно доступных в коде программы.

Почленная инициализация и присваивание (1 / 2)



Почленная инициализация по умолчанию — механизм инициализации одного объекта класса другим объектом того же класса, который активизируется независимо от наличия в определении класса явного конструктора.

Почленная инициализация по умолчанию происходит в следующих ситуациях:

- явная инициализация одного объекта другим;
- передача объекта класса в качестве аргумента функции;
- передача объекта класса в качестве возвращаемого функцией значения;
- определение непустого стандартного последовательного контейнера;
- вставка объекта класса в стандартный контейнер.

Почленная инициализация и присваивание (2 / 2)



Почленная инициализация по умолчанию **подавляется** при наличии в определении класса конструктора копирования.

Запрет почленной инициализации по умолчанию осуществляется одним из следующих способов:

- описание закрытого конструктора копирования (**не действует для методов класса и дружественных объектов**);
- описание конструктора копирования без его определения (**действует всюду**).

Почленное присваивание по умолчанию — механизм присваивания одному объекту класса значения другого объекта того же класса, отличный от почленной инициализации по умолчанию использованием копирующей операции-функции присваивания вместо конструктора копирования.

Конструкторы копирования



Конструктор копирования принимает в качестве первого формального параметра **ссылку** на существующий объект класса. Другими словами, этот параметр имеет тип `T&`, `const T&`, `volatile T&` или `const volatile T&`.

Второй и последующие параметры конструктора копирования, если есть, должны иметь значения по умолчанию.

В случае отсутствия явного конструктора копирования в определении класса производится почленная инициализация объекта по умолчанию.

```
struct Sample {  
    Sample(const Sample &rhs);  
    // ...  
};
```

Конструкторы и операции преобразования



Конструкторы преобразования служат для построения объектов класса по одному или нескольким значениям иных типов.

Операции преобразования позволяют преобразовывать содержимое объектов класса к требуемым типам данных.

```
struct Sample {  
    // конструкторы преобразования  
    Sample(const char *);  
    Sample(const std::string &);  
    // операции преобразования  
    operator int      () { return int_prm; }  
    operator double () { return dbl_prm; }  
    // ...  
};
```

Список инициализации нестатических членов данных



Выполнение любого конструктора состоит из двух фаз:

- фаза явной (неявной) инициализации (**обработка списка инициализации**) — предполагает **начальную инициализацию** членов данных;
- фаза вычислений (**исполнение тела конструктора**) — предполагает **присваивание значений** (в **предварительно инициализированных областях памяти**).

Присваивание значений членам данных — объектам классов в теле конструктора **неэффективно** ввиду ранее произведенной инициализации по умолчанию. Присваивание значений членам данных, представляющих «старые» базовые типы, **по эффективности равнозначно** инициализации.

К началу исполнения тела конструктора все **константные члены и члены-ссылки** должны быть инициализированы.

Деструкторы.

Виртуальные деструкторы (1 / 2)



Деструктор — не принимающий параметров и не возвращающий результат метод класса, автоматически вызываемый при выходе объекта из области видимости и применении к указателю на объект класса операции `delete`.

```
struct Sample
{
    // ...
    virtual ~Sample();
};
```

Примечание: деструктор не вызывается при выходе из области видимости ссылки или указателя на объект.

Деструкторы.

Виртуальные деструкторы (2 / 2)



Типичные задачи деструктора:

- сброс содержимого программных буферов в долговременные хранилища;
- освобождение (возврат) системных ресурсов, главным образом — оперативной памяти;
- закрытие файлов или устройств;
- снятие блокировок, останов таймеров и т.д.

Для обеспечения корректного освобождения ресурсов объектами производных классов деструкторы в полиморфных иерархиях, как правило, определяют как **виртуальные**.

ЯВНЫЙ ВЫЗОВ ДЕСТРУКТОРОВ



Потребность в явном вызове деструктора обычно связана с необходимостью **уничтожить** динамически размещенный объект **без освобождения памяти**.

```
char *buf = new char[sizeof(Sample)];  
// "размещающий" вариант new  
Sample *psmp1 = new (buf) Sample(100);  
// ...  
psmp1->~Sample();      // ВЫЗОВ 1  
Sample *psmp2 = new (buf) Sample(200);  
// ...  
psmp2->~Sample();      // ВЫЗОВ 2  
delete [] buf;
```



Скобочные инициализаторы членов данных (C++11): пример (1 / 2)



```
#include <iostream>

int counter = int();

struct Sample {
    // скобочные и приравнивающие инициализаторы
    // (brace-or-equal initializers)
    std::string msg{"Abeunt studia in mores"}; // форма 1
    int id = ++counter;                        // форма 2
    int n{42};                                // форма 3

    // приравнивающий инициализатор
    // с невычисляемым операндом (n)
    static const std::size_t sz = sizeof n;
```



Скобочные инициализаторы членов данных (C++11): пример (2 / 2)



```
// struct Sample
    Sample() {} // msg == "Abeunt studia in mores", n == 42
    Sample(int _n) : n(_n) {} // msg == "Abeunt... ", n == _n
};

int main() {
    Sample sample;
    std::cout << sample.id << "\t"           // 1
               << sample.n << std::endl      // 42
               << sample.msg << std::endl;    // "Abeunt..."
    return 0;
}
```

Семантика переноса (C++11)



Введение в C++11 семантики переноса ([англ. move semantics](#)) обогащает язык возможностями более **тонкого и эффективного управления памятью данных**, устраняющего копирование объектов там, где оно нецелесообразно. Технически семантика переноса реализуется при помощи **ссылок на праводопустимые выражения** ([англ. expiring value, xvalue](#)) и **конструкторов переноса**.

Конструкторы переноса не создают точную копию своего параметра, а «отнимают» его ресурсы (указатели на участки программной кучи, дескрипторы файлов, потоки ввода-вывода, потоки исполнения, TSP-сокеты и т.д.), передавая права владения ими вновь создаваемому объекту. **Параметр конструктора переноса остается в корректном, но неопределенном состоянии.**

Отсутствие поддержки классом семантики переноса — не ошибка, но упущенная возможность оптимизации.



Конструктор переноса: пример (C++11, 1 / 3)



```
class Alpha {  
public:  
    Alpha();  
    Alpha(const Alpha &a); // конструктор копирования  
    Alpha(Alpha &&a);      // конструктор переноса  
    ~Alpha();  
private:  
    std::size_t sz;  
    double *d;  
};  
  
Alpha::Alpha() : sz(0), d(NULL) { }  
Alpha::~~Alpha() { delete [] d; }
```



Конструктор переноса: пример (C++11, 2 / 3)



```
// конструктор копирования
Alpha::Alpha(const Alpha &a) : sz(a.sz) {
    d = new double[sz];
    // ...
    for(std::size_t i = 0; i < sz; i++)
        d[i] = a.d[i];
}

// конструктор переноса
Alpha::Alpha(Alpha &&a) : sz(a.sz) {
    d = a.d;
    a.d = nullptr; // перенастройка параметра, C++11
    a.sz = 0;
}
```



Конструктор переноса: пример (C++11, 3 / 3)



```
Alpha foo(Alpha arg) { return arg; }  
void bar(Alpha arg) { }  
int main(void) {  
    // вызов Alpha::Alpha(Alpha&&) при возврате из функции  
    // с сигнатурой Alpha f(/* ... */)   
    Alpha alp1 = foo(Alpha());  
    // вызов Alpha::Alpha(Alpha&&) при инициализации  
    Alpha alp2 = std::move(alp1); // alp2(std::move(alp1))  
    // вызов Alpha::Alpha(Alpha&&) при передаче параметра  
    // функции void g(Alpha)  
    bar(std::move(alp2));  
    return 0;  
}
```



Операция-функция присваивания с переносом: пример (C++11, 1 / 2)



```
#include <iostream>

struct Beta {
    // явно определенная операция-функция присваивания
    // делает конструкторы T::T(), T::T(T&) удаленными
    Beta() = default;
    Beta(const Beta&) = default;
    Beta& operator=(Beta &&rhs) {
        msg = std::move(rhs.msg);
        return *this;
    }
    std::string msg;
};
```




Операция-функция присваивания с переносом: пример (C++11, 2 / 2)



```
int main() {  
    Beta beta = {"Per aspera ad astra"}, gamma;  
    gamma = std::move(beta);  
  
    std::cout << gamma.msg << std::endl      // "Per asper..."  
              << beta.msg << std::endl;     // ""  
}
```



Реализация присваивания через вызов конструктора (C++11)



```
class T {
public: // функции ::acquire/::release являются вымышленными
    explicit T(const std::string& _name) :
        handle {::acquire(_name)} {}
    T(T&& rhs) : handle {rhs.handle} {rhs.handle = nullptr;}
    T& operator=(T&& rhs) { // ресурсом владеет rhs
        T copy{std::move(rhs)}; // ресурсом владеет copy
        std::swap(handle, copy.handle); // владеет *this
        return *this; // для copy вызывается T::~~T()
    }
    ~T() { ::release(handle); }
private:
    resource_t handle;
};
```

Автоматически генерируемый конструктор по умолчанию



В случае отсутствия в классе явных конструкторов любого типа **компилятор самостоятельно неявно определяет** конструктор по умолчанию как встраиваемый (*inline*) открытый (*public*) метод данного класса.

В ходе трансляции неявно определенный конструктор, — если он не удален и не является тривиальным, — по умолчанию генерируется (формируется на уровне тела функции) компилятором и работает точно так же, как явно определенный конструктор **с пустым телом и пустым списком инициализации**.

Примечание. В случае участия класса в иерархии наследования автоматически генерируемый конструктор по умолчанию вызывает конструкторы по умолчанию базовых классов и своих членов, не являющихся статическими.

Тривиальный конструктор по умолчанию



Конструктор по умолчанию является **тривиальным**, если одновременно соблюдаются **все следующие условия**:

- конструктор определен неявно или определен как `default`;
- класс не имеет виртуальных методов;
- класс не имеет виртуальных базовых классов;
- каждый непосредственный предок класса имеет тривиальный конструктор по умолчанию;
- каждый нестатический член класса имеет тривиальный конструктор по умолчанию.

Тривиальный конструктор по умолчанию не совершает никаких действий. Объекты классов с таким конструктором — при условии соблюдения требований к выравниванию — могут создаваться при помощи `reinterpret_cast` в любом подходящем месте, к примеру, на участках программной кучи, запрошенных вызовом `std::malloc()`.

Автоматически генерируемые специальные методы иных видов



Сказанное выше справедливо также для конструкторов копирования, конструкторов переноса и деструктора класса:

- в отсутствие явно определенного конструктора копирования конструктор формируется компилятором автоматически как встраиваемый открытый метод с сигнатурой `T::T(const T&)` или `T::T(T&)`;
- в отсутствие явно определенных конструктора переноса, конструкторов копирования, операций присваивания путем копирования, операций присваивания путем переноса, а также деструкторов конструктор переноса формируется компилятором автоматически как встраиваемый открытый метод с сигнатурой `T::T(T&&)`;
- в отсутствие явно определенного деструктора деструктор формируется компилятором автоматически как встраиваемый открытый метод с сигнатурой `T::~~T()`.

Принуждение и подавление генерации конструкторов и деструкторов (C++11)



Автоматическая генерация конструктора по умолчанию, конструктора копирования, конструктора переноса или деструктора компилятором может быть как подавлена программистом, так и, наоборот, форсирована.

```
class Sample {  
public:  
    // запрет автогенерации конструктора по умолчанию  
    Sample() = delete;  
    Sample(int ipr, double dpr);  
    // принудительная автогенерация деструктора по умолчанию  
    ~Sample() = default;  
};
```

Подавление копирования



Для обеспечения эффективности объектного кода компилятору разрешено пропускать необязательные («лишние») вызовы конструкторов копирования и переноса, реализуя так называемую **семантику передачи по значению без копирования** (англ. *zero-copy pass-by-value semantics*).

- Такому поведению компиляторов, известному как **подавление копирования** (англ. *copy elision*), не препятствует даже реализация конструкторами **наблюдаемых внешне побочных действий**. Примером ситуаций такого рода является оптимизация времени компиляции, известная как RVO и NRVO.

Подавление копирования — единственная разрешенная стандартом форма оптимизации, легально нарушающая **правило “as-if”** и способная повлиять на побочные действия вызова функций.

Программы, полагающиеся на побочные действия конструкторов и деструкторов классов, не являются переносимыми.

Оптимизации RVO и NRVO



Оптимизация NRVO (англ. Named Return Value Optimization) — ситуация, в которой функция возвращает объект класса по значению, а выражение в операторе `return` есть идентификатор устойчивого объекта с автоматической продолжительностью хранения, не являющегося параметром самой функции и имеющего тот же тип без квалификаторов `const` / `volatile`, что и тип результата функции.

Оптимизация RVO (англ. Return Value Optimization) — ситуация, в которой анонимный временный объект является аргументом `return`.

- Расширением RVO является оптимизация, при которой анонимный временный объект, не связанный с какой-либо ссылкой, копируется или переносится в объект того же типа без квалификаторов `const` / `volatile`.

Примечание: подавление копирования в GCC может быть отключено флагом компиляции `-fno-elide-constructors`.

«Правило трех»



«Правило трех» (англ. *Rule of Three*) — если класс требует написания явно определенного деструктора, такого же конструктора копирования или операции присваивания, он почти наверное требует написания всех трех названных методов. ■

- Неявно определенные специальные методы класса, как правило, **неверно решают возложенную задачу**, если класс управляет ресурсом, который доступен по описателю, не являющемуся классом (*указатель T**, *POSIX-дескриптор файла и пр.*); деструктор не выполняет никаких действий, а конструктор копирования / операция присваивания осуществляет «поверхностное копирование» (*описателя, а не управляемого / адресуемого ресурса*).

Следствие. Для конструктора копирования и операции присваивания справедливо: определение одного метода как закрытого (*private*), удаленного (*delete*) или не имеющего реализации при наличии неявно определенной реализации другого чаще всего является признаком (*влечет за собой*) ошибки. ■

«Правило пяти [умолчаний]»



«Правило пяти [умолчаний]» (англ. Rule of Five [Defaults]) — так как наличие явно определенного деструктора, конструктора копирования или копирующей операции присваивания подавляет неявное определение конструктора переноса и операции присваивания с переносом, класс, требующий поддержки семантики переноса, должен включать определения всех пяти специальных методов (возможно, в виде `= default`). ■

Класс `T`, подчиняющийся «правилу пяти», гарантированно содержит:

- явный конструктор копирования, напр. `T::T(const T&);`
- явный конструктор переноса, напр. `T::T(T&&);`
- явную операцию-функцию присваивания путем копирования, напр. `T& operator=(const T&);`
- явную операцию-функцию присваивания путем переноса, напр. `T& operator=(T&&);`
- явный деструктор `T::~~T()`.

«Правило нуля»



«Правило нуля» (англ. *Rule of Zero*) — единственной зоной ответственности класса (ср.: принцип *SRP*) с нестандартным деструктором, конструктором копирования / переноса или операцией-функцией присваивания путем копирования / переноса должно быть обслуживание ресурса, которым владеет его соответствующий экземпляр (ср.: идиома *RAII*). ■

```
struct ruleOfZero {  
    ruleOfZero(const std::string &_msg) : msg(_msg) {}  
private:  
    std::string msg;  
};
```



Закрепление за конструкторами функции захвата, выделения, блокировки или инициализации ресурсов, а **за деструкторами — функции** их возврата, освобождения и снятия установленных блокировок:

- позволяет **безопасно обрабатывать ошибки и исключения**;
- составляет суть одной из важнейших идиом ОО-программирования RAII (англ. Resource Acquisition Is Initialization — «получение ресурса есть инициализация»).

Работа идиомы RAII в языке C++ основана, главным образом, на **гарантированном вызове деструкторов автоматических переменных**, являющихся экземплярами классов, при выходе из соответствующих областей видимости.

Правильный выбор объекта-владельца соответствующего ресурса — лучшее средство в борьбе с утечкой ресурсов.

Принципы S.O.L.I.D.: начало



Принципы S.O.L.I.D. — устоявшееся обозначение «первой пятерки» принципов объектно-ориентированного программирования и дизайна, сформулированных главным редактором *C++ Report* Р. Мартином (Robert Martin) в начале 2000-х гг.

В число принципов S.O.L.I.D., обобщающих классические результаты 1980 – 1990-х гг., входят:

- Принцип единственной ответственности [Р. Мартин];
- Принцип открытости / закрытости [Б. Мейер (Bertrand Meyer)];
- Принцип подстановки Лисков [Б. Лисков (Barbara Liskov) — Ж. Уинг (Jeannette Wing)];
- Принцип разделения интерфейсов [Р. Мартин];
- Принцип инверсии зависимостей [Р. Мартин].

Принципы SRP и OCP (S.O.L.I.D.)



Принцип единственной ответственности (англ. Single Responsibility Principle, SRP) требует:

Любой класс должен иметь
одну и только одну зону ответственности

Принцип открытости / закрытости (англ. Open / Closed Principle, OCP) гласит:

Программные элементы должны быть
открыты для расширения, но закрыты для изменения

Домашнее задание № 3



- Сформировать команду ([выполнено?!\).](#)
- Предложить собственную тему проекта ([см. блог дисциплины\).](#)
- Построить концептуальную UML-модель предметной области проекта и детализировать состав основных классов.
- **Цель** — спроектировать полиморфную иерархию из трех или более классов с множественным наследованием, семантика и функциональная нагрузка которых определяются темой проекта.

Срок сдачи

Весь семестр



**Спасибо за
внимание!**

Алексей Петров