Методы и стандарты программирования

КМБО-02-19

21 августа 2021 г.

1 Качественное понимание основ

Замечание 1 *Не очень понятно, что он имел в виду под этой формулировкой, но из сказанного следует, что, по сути, он может по рандому задать любой из вопросов по материалу, который Макс разбирал на очке. Поэтому данный раздел будет подходить и для ЗРЛ.*

1.1 ООП

Определение 1 ***Объектно-ориентированный подход*** *- подход, при котором предметная область представлена совокупностью объектов, взаимодействующих между собой с помощью сообщений.*

Определение 2 ***Предметная область*** *- множество предметов и условий, в рамках которых происходит работа и выполнение задачи.*

Определение 3 ***Объект*** *- описание сущности из предметной области.*

Определение 4 ***Объектно-ориентированное программирование*** *- методология программирования, основанная на представлении программы в виде совокупности объектов.*

Определение 5 ***Класс*** *- множество объектов, обладающих общими свойствами и поведением.*

1.2 Свойства объекта

- **Определение 6** ***Состояние*** *- каждая уникальная комбинация свойств объекта(атрибутов) и связей с другими объектами. Меняется со временем.*
- **Определение 7** ***Поведение*** *- определяется методами - определяет действия объекта относительно внешних связей и манипуляций с собственными свойствами.*
- **Определение 8** ***Идентичность*** *- свойство объекта, отличающее его ото всех других объектов. В C++ это адрес.*

1.3 Принципы объектной модели

Основные принципы:

- **Определение 9** ***Абстракция*** *- выделение наиболее существенных характеристик некоторого объекта, отличающих его от всех других объектов, важных с точки зрения дальнейшего рассмотртения.*

- **Определение 10** ***Инкапсуляция*** *- объединение данных и кода, манипулирующего этими данными, а также защищита и того, и другого от внешнего вмешательства или неправильного использования путем сокрытия.*

- **Определение 11** ***Модульность*** *- возможность спроектировать взаимодействия объектов так, чтобы объекты между собой взаимодействовали неинтенсивно, но внутри самих объектов происходила интенсивная работа. Используется для переиспользования объектов.*

- **Определение 12** ***Иерархия*** *- упорядочивание абстракций по уровням.*

Дополнительные принципы:

- **Определение 13** ***Типизация*** *- защита от неправильного использования объекта, т.е. от ситуации, когда объект одного класса используется вместо другого.*

- **Определение 14** ***Устойчивость*** *- возможность объекта переживать породивший его процесс.*

- **Определение 15** ***Параллелизм*** *- наличие в системе нескольких потоков управления одновременно.*

1.4 Принципы объектно-ориентированного программирования

1.4.1 Сами принципы

1. **Абстракция** - см. выше.

2. **Инкапсуляция** - см. выше.

3. **Определение 16** ***Наследование*** *- механизм создания новых объектов на основе уже существующих путём сохранения свойств и поведения с возможностью расширения функциональности и переопределения.*

4. **Определение 17** ***Полиморфизм*** *- возможность создавать объекты с одинаковым интерфесом и различной реализацией.*

1.4.2 Примеры и особенности реализации принципов инкапсуляции, наследования и полиморфизма в C++

Инкапсуляция в C++ реализуется за счёт разделения атрибутов и методов класса(объекта) на публичные, защищённые и скрытые области видимости, реализующиеся с помощью спецификаторов public, protected и private соответственно.

В листинге ниже представлен класс *Contact*, публичные переменные и методы доступны из основной программы (*main*). Приватные переменные и методы могут прочитаны, вызваны или изменены только самим классом.Попытка напечатать или изменить приватную переменную *mobile_number* из основной программы (*main*) вызовет ошибку при компиляции потому как доступ к приватным данным в классе ограничен.

```cpp
#include <iostream>
using namespace std;

class Contact
{
    private:
        int mobile_number;              // private variable
        int home_number;                // private variable
    public:
        Contact()                       // constructor
        {
            mobile_number = 12345678;
            home_number = 87654321;
        }
        void print_numbers()
        {
            cout << "Mobile number: " << mobile_number;
            cout << ", home number: " << home_number << endl;
        }
};

int main()
{
    Contact Tony;
    Tony.print_numbers();
    // cout << Tony.mobile_number << endl;
    // will cause compile time error
    return 0;
}
```

Наследование в C++ подразделяется на публичное(*public*), защищённое(*protected*) и приватное(*private*).

Разница в следующем:

- **Публичное наследование** - публичные и защищённые данные наследуются без изменения доступа к ним. Т.е. если в исходном классе было поле в защищённой области, то и для наследника это поле останется в защищённой области. Если у предка был публичный метод - он останется публичным и у наследника.
- **Защищённое наследование** - все поля из public и protected родителя становятся protected-полями потомка.
- **Приватное наследование** - поля public и protected предка становятся полями private потомка.

Замечание 2 ***Private поля предка ни в каком случае не наследуются!***

Пример приватного наследования:

```
#include <iostream>
using namespace std;

class Device {
    public:
        int serial_number = 12345678;

        void turn_on() {
            cout << "Device is on" << endl;
        }
};

class Computer: private Device {
    public:
        void say_hello() {
            turn_on();
            cout << "Welcome to Windows 95!" << endl;
        }
};

int main() {
    Device Device_instance;
    Computer Computer_instance;

    cout << "\t Device" << endl;
    cout << "Serial number is: "<< Device_instance.serial_number << endl;
    Device_instance.turn_on();

    // cout << "Serial number is: " << Computer_instance.serial_number << endl;
    // Computer_instance.turn_on();
    // will cause compile time error

    cout << "\t Computer" << endl;
    Computer_instance.say_hello();
    return 0;
}
```

Класс *Computer* теперь использует метод *turn_on*() как и любой приватный метод: *turn_on*() может быть вызван изнутри класса, но попытка вызвать его напрямую из *main* приведет к ошибке во время компиляции. Для базового класса *Device*, метод *turn_on*() остался публичным, и может быть вызван из *main*.

Замечание 3 *Более подробно о наследовании в C++, порядке вызовов конструкторов и деструкторов, виртуальном наследовании и прочем подробно написано в соответствующем разделе в материалах по методам и стандартам программирвания. В этом же разделе приводятся лишь некоторые общетеоретические моменты.*

Полиморфизм представлен следующими видами: **параметрический полиморфизм**, **полиморфизм подтипов**, **Ad-hoc полиморфизм**, **полиморфизм приведения к типу**.

- **Параметрический полиморфизм**. Его суть в том, что для различных типов входных данных объект обеспечивает одинаковое поведение. Реализацию этого вида полиморфизма в C++ предлагает механизм шаблонов.
- **Полиморфизм подтипов**. Ситуация, реализуемая через механизм наследования, когда существует базовый тип и несколько его производных; и в базовом, и в производных типах определён один и тот же метод с одинаковой сигнатурой, однако реализация этого метода отличается в зависимости от того, на ком этот метод используется: на базовом типе или на производном (а если на производном, то нужно уточнять, на каком именно).

  Примером полиморфизма подтипов может быть следующая ситуация: мы можем создать какой-то базовый тип, который отправляет email-ы, и создать его подтип, который вместо отправки email-ов записывает в лог то что мы хотели отправить, после чего уже отправляет данные дальше другому объекту этого типа.

  Замечание 4 *Не стоит путать полиморфизм подтипов и наследование. Наследование - всего лишь механизм реализации, тогда как полиморфизм - принцип.*

  Замечание 5 *С этим полиморфизмом существует ограничение, связанное с принципами **SOLID**. А именно: если мы хотим заменить в системе объект какого-то типа на объект подтипа (грубо говоря наследника), то система не должна сломаться. То есть "другое"поведение нашего подтипа должно быть совместимо в плане интерфейса с базовым типом.*

- **Ad-hoc полиморфизм**. Полиморфизм, при котором поведение функции или метода зависит от входных параметров. Говоря совсем грубо, это ситуация, когда есть две функции с одинаковым названием и разными сигнатурами.

  Замечание 6 *Этот полиморфизм характерен для языков со статической типизацией.*

- **Полиморфизм приведения к типу**. Возникает в ситуациях, когда мы кастуем объекты производных классов до объектов базового класса, чтобы добиться от них одинакового поведения.

Отличным примером применения полиморфизма является наша с вами лабораторная работа по написанию стека(её ведь все сделали, я надеюсь?). У нас есть абстрактный класс *StackImplementation*, содержащий интерфейс, который должна поддерживать структура, на основе которой работает стек. Уже от этого класса наследуются $VectorStack, ListStack$ - данные классы поддерживают уже реализации на конкретных структурах данных: векторе и списке соответственно. На следующей странице приведены примеры объявления класса *StackImplementation* и класса $VectorStack$.

Здесь мы видим использование полиморфизма подтипов (наследование $VectorStack$ от *StackImpl*, в результате чего у $VectorStack$ есть своя собственная реализация методов, объявленных родителем), а также параметрического полиморфизма(используем шаблоны).

Замечание 7 *Если бы мы здесь рассматривали реализацию алгоритма сортировочной станции, а именно работу в нём с токенами, то отметили бы, что для их хранения мы используем полиморфизм приведения к типу: все токены мы храним как объекты некоторого GeneralToken, и уже при работе программы мы кастуем каждый из них, чтобы конкретизировать их поведение.*

```
template <class ValueType>
class StackImplementation {
public:
        // добавление в хвост
        virtual void push(const ValueType& value) = 0;
        // удаление с хвоста
        virtual void pop() = 0;
        // посмотреть элемент в хвосте
        virtual const ValueType& top() const = 0;
        // проверка на пустоту
        virtual bool isEmpty() const = 0;
        // размер
        virtual size_t size() const = 0;
        // виртуальный деструктор
        virtual ~StackImplementation() {};
};
```

```
template <class ValueType>
class VectorStack : public MyVector<ValueType>,
virtual public StackImplementation<ValueType>
{
public:
    //конструктор
    VectorStack();
    //конструктор копированием
    VectorStack(const VectorStack& copyVec);
    VectorStack& operator=(const VectorStack& copyVec);
    // Конструктор копирования присваиванием
    VectorStack(VectorStack&& moveVec) noexcept;
    VectorStack& operator=(VectorStack&& moveVec) noexcept;
    // добавление в конец
    void push(const ValueType& value) override;
    // удаление с хвоста
    void pop() override;
    // посмотреть элемент в хвосте
    const ValueType& top() const override;
    // проверка на пустоту
    bool isEmpty() const override;
    // размер
    size_t size() const override;
    // деструктор
    ~VectorStack();

};
```

1.5 Типы отношений объектов

Определение 18 *Отношения - способ организации взаимодействия.*

1.6 Классификация:

- По виду: одно- и дву- направленные;
- По характеру: содержит (**has-a**), является (**is-a**), использует (**uses-a**);
- По кратности: один на один, один объект на много объектов, много на много...

Определение 19 ***Интерфейс*** *- это контракт с системой, гарантирующий определённое поведение и свойства объекта.*

1.6.1 Типы отношений:

Зависимость	Она же может считаться конкретизацией поведения. Рассматривая на примере: один объект обращается к функции другого. При этом объекты **не являются полями друг друга** - отношение ($friend$). Является однонаправленным отношением.
Часть-целое	**Определение 20** ***Агрегация*** *- целое не управляет жизненным циклом частей.* Пример: работник переживает кампанию. Иными словами, если используем агрегацию, то в деструкторе агрегированный объект не удаляем. *has-a*
	Определение 21 ***Композиция*** *- часть принадлжеит только одному объекту, который за неё 'отвечает'.* Пример: часть удаляется в деструкторе объекта-хозяина, а также находится в приватной части. *has-a*
Ассоциация	Объекты используют друг друга для своих нужд. Отношения двунаправленные, никто никому не принадлежит. Прекрасный пример: отношения врача и пациента - это явно не отношения часть-целое! У врача спокойно может быть огромное количество пациентов в день, а у пациента своя жизнь с блэкджеком и программированием в задачах радиолокации. *uses-a*
Наследование	Комментарии тут излишни, всё сказали раньше. *is-a*

Замечание 8 *Примеры на C++ для приведённых выше отношений объектов Вы без проблем можете придумать и сами. Поясним только следующее: пример* ***зависимости*** *мы можем найти, вспомнив класс $MyString$, где нам приходилось делать friend-ом класс 'ostream', чтобы работал корректный вывод;* ***часть-целое*** *- односвязный список - в идеале это композиция, ибо в деструкторе мы разрушаем все node в списке; наследование - очевидный пример со стеком;* ***агрегация*** *- возъмём какой-нибудь класс, который имеет полем std :: string, получим, что класс не несёт ответственности за жизнь этого поля после себя, при этом это поле может быть передано и другим объектам;* ***ассоциация*** *- один класс запрашивает возвращаемые значения методов другого, чтобы конкретизировать свою работу - на пальцах сказать сложно, но если Вам попадётся код с такими отношениями, Вы сразу поймёте.*

Замечание 9 *Если у Вас всё хорошо с английским и Вы не поняли того, что написано выше, предлагаю прочитать статью по ссылке:*

`https://www.learncpp.com/cpp-tutorial/10-1-object-relationships/`

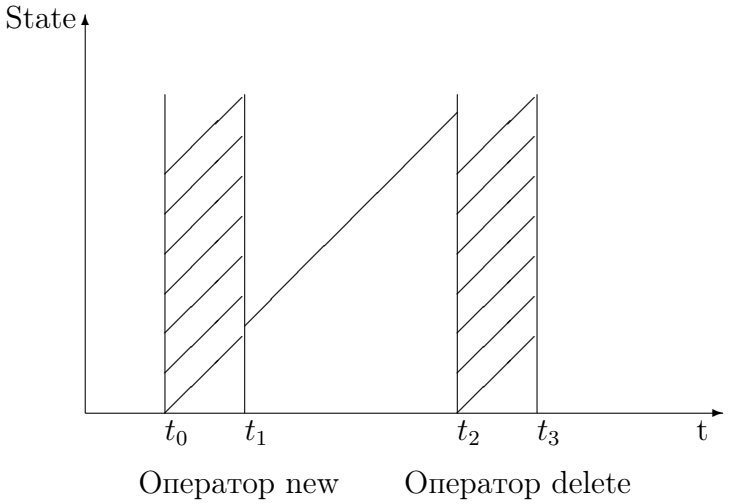
1.7 Связь системы и подсистемы

Замечание 10 *Я в душе не чаю, что имел в виду гений Милонов, когда говорил об этих системах и подсистемах. Тут два варианта: либо он хочет услышать много интересного о теории систем, которой у нас нет, ибо мы не системный анализ и управление; либо он хочет услышать про что-то попроще. Если попроще, предлагаю познакомиться с 5 принципами программирования* ***SOLID****. Ниже прикрепляю ссылку.*

`https://habr.com/ru/company/mailru/blog/412699/`

2 Классы и объекты

2.1 Жизненный цикл объекта



На схеме выше показан примерный 'график' жизни некоторого объекта, сейчас поясним, что на нём есть что:

1. Отрезок времени до момента t_0: объекта ещё нет.

2. Отрезок $t_0 - t_1$: с помощью оператора *new* мы создаём объект. Здесь стоит понимать, что *new* - это абстракция над выделением памяти и прочими служебными командами, которые бы нам пришлось делать вручную, реши мы работать на более низком уровне. На этапе работы оператора *new*, в частности, отрабатывает некоторый *конструктор* объекта. О конструкторах подробнее речь пойдёт ниже.

3. Отрезок $t_1 - t_2$: объект живёт, развивается.

4. Отрезок $t_2 - t_3$: использован оператор *delete*. Здесь также необходимо понимать, что *delete* - абстракция над рядом служебных команд и вызовов, таких, например, как вызов *деструктора*, а затем высвобождение памяти и возвращение памяти на кучу. В данный отрезок времени происходит вызов деструктора и служебных команд, отвечающих за высвобождение памяти(последнее C++ берёт, как сказано выше, на себя).

   Замечание 11 *В C++ необходимо, чтобы вся память, выделенная с помощью оператора new, была освобождена оператором delete. Иначе в памяти так и останется висеть объект, даже после того, как программа отработала и процесс завершился.*

2.2 Конструктор и деструктор

2.2.1 Конструктор

Объявление конструктора не сильно отличается от объявления обычного метода класса. Первое отличие: название конструктора как метода совпадает с названием самого класса. Второе отличие: конструктор не имеет возвращаемого значения! Обычно конструкторы размещаются в public области класса, но если есть необходимость запретить создание экземпляров класса, то достаточно просто объявить конструктор в private или protected областях, а не в public.

Замечание 12 *Старайтесь избегать в конструкторах и деструкторах вызова виртуальных функций, поскольку вызывая виртуальную функцию, можно забыть, что при обращении к this мы вызываем последний перегруженный метод (то есть, если в нашем потомке он перегружен, то вызываем его; если не перегружен, то идём вверх по иерархии наследования до первой попавшейся инициализации этого метода).*

Виды конструкторов: конструктор по умолчанию, конструктор с параметрами, конструктор копирования, конструктор перемещения.

Название конструктора	Сигнатура	Описание и назначение
Конструктор по умолчанию	className()	Вызывается, если при инициализации объекту не передаётся никаких параметров. Инициализация происходит заранее описанным способом по умолчанию.
Конструктор с параметрами	className(type1 param1, type2 param2, ...)	Инициализация объекта происходит с учётом переданных внешних параметров.
Конструктор копирования	className(const ClassName& Object)	Инициализация объекта происходит путём копирования данных из другого объекта этого же класса согласно описанному в конструкторе алгоритму. При этом **копируемый объект не меняется**, поэтому и передаётся в конструктор по *константной ссылке*. При копировании данных из одного объекта в другой нельзя 'перевешивать' и нежелательно копировать указатели копируемого объекта в создаваемый, т.к. иначе при уничтожении одного объекта мы рискуем получить 'висячий' указатель (т.е. указатель, к памяти по которому мы уже доступа не имеем, но мы сами об этом не знаем) в другом объекте. Поэтому, если класс имеет поля с указателями, **нежелательно** использовать конструктор копирования по умолчанию.
Конструктор копирования по умолчанию(объявление в явном виде)	className(const ClassName& Object)=default	Просто копирует значение полей из копируемого объекта в инициализируемый.
Конструктор перемещения	className(ClassName && Object) noexcept	Инициализация объекта происходит путём **переноса** данных из другого объекта этого же класса. При этом **передаваемый в конструктор объект меняется**. При переносе данных происходят такие страшные вещи, как перевешивание указателей, перенос кусков памяти жуткими вещами вроде *memcpy* и прочие непотребства. Нужно это за тем, чтобы при инициализации нового объекта с помощью rvalue не приходилось делать двойную работу: сначала делать копию *временного* объекта, потом убивать её, а потом убивать временный объект. Ведь гораздо дешевле сразу убить временный объект и забрать все его ~~ценные вещи~~ данные! Подробнее об этом в разделе по семантике перемещения.

Замечание 13 *В сигнатуре конструктора копирования может стоять ключевое слово explicit. Это будет означать, что компилятору запрещается использовать конструктор копирования неявно (что произойдёт, например, если не перегрузить оператор присваивания и вызвать выражение типа className obj1 = obj2).*

Замечание 14 *В сигнатуре конструктора перемещения используется параметр noexcept, указывающий, что этот конструктор не будет генерировать исключений. Это необходимо вот с какой целью: если исключение генерируется конструктором копирования, то на момент исключения компилятор знает, что копируемый объект не изменился и, по сути, не произошло ничего непоправимого. В слу-*

чае с конструктором перемещения всё иначе: если произойдёт ошибка в конструкторе перемещения, то компилятор не может понять, на каком шаге программа достигла трагедии: было ли уже что-то перенесено или ещё нет. Это проблема. Поэтому мы насильно говорим, что всё, что было в конструкторе перемещения, остаётся в конструкторе перемещения.

2.2.2 Деструктор

В **деструкторе**, как понятно из названия, происходит очистка памяти, которую забрал объект для своей жизни. Если объект использует отношение композиции и в нём присутствуют поля с данными на куче, в деструкторе необходимо вызвать оператор *delete* для каждого такого поля, иначе выделенная для них память так и останется 'висеть' после смерти объекта. **Деструктор** не имеет входных параметров, объявляется аналогично конструктору по умолчанию, но перед ним идёт символ '$\sim$'. Т.е. объявление деструктора некоторого класса $className$ будет выглядеть следующим образом:

$$\sim className();$$

Деструктор может быть также по умолчанию(просто используем '$= default$' аналогично тому, как делали в конструкторе копирования).

2.3 Атрибуты

Замечание 15 *Атрибуты появились в C++ начиная лишь с 11 стандарта. Естественно, на лекциях мы их не проходили, и как они оказались в списке тем для экзамена, одному Богу понятно.*

Определение 22 *Атрибуты позволяют задавать дополнительную информацию для различных конструкций языка, таких как типы, переменные, имена, блоки и единицы трансляции. Данная информация в частности может быть использована компилятором для генерации более эффективного кода и предоставления (или наоборот, подавления) предупреждающих сообщений пользователю на уровне конкретных участков кода, а не целой программы или компилируемого файла, как это обеспечивается ключами компиляции.*

Атрибуты указываются перед типом возвращаемого значения функции в виде:

$$[[atribute\_name]]$$

Список стандартных атрибутов:

- noreturn - функция всегда выдаёт исключение.
- carries_dependency - используется, чтобы позволить зависимостям переноситься через вызовы функций. (я не знаю, что это означает, но пусть будет)
- deprecated - используется для пометки сущностей классов или библиотек, которые скоро будут удалены и об использовании которых уже стоит забыть. Генерируется warning.
- fallthrough ставится перед case-веткой в switch и указывает, что в данном месте намеренно пропущен break.
- nodiscard указывает, что возвращаемое функцией значение нельзя игнорировать и нужно сохранить в какую-либо переменную.
- maybe_unused заставляет компилятор погасить предупреждения о переменной, которая не используется в некоторых режимах компиляции.
- likely - разрешить компилятору применить оптимизацию таким образом, что, скорее всего, будет использован метод или функция, помеченная или помеченный этим атрибутом.
- no_unique_address - я пока не разобрался, что это. Если разобрались, напишите мне в личку.

2.4 Ссылки и ссылочный тип данных, связанные с этим ограничения

По ссылке ниже представлена краткая статья о том, что такое ссылка в C или C++, этот базис необходимо понимать, чтобы двигаться дальше.

`https://ravesli.com/urok-88-ssylki/`

Да-да, тот самый сайт, по заветам дядюшки Яна.

2.5 MOVE-семантика

Здесь я не хочу мудрствовать лукаво и просто прикрепляю ссылку на лекционный материал самого Милонова. Тут всё довольно легко и понятно написано, дополнительно есть ссылки на хабр, где можно понять то, что не поняли в самой лекции. Лично мне хватило этой лекции.

`http://vega.fcyb.mirea.ru/dist/docs/92i298t-rvalues_and_move_semantic.pdf`

2.6 Функция и сигнатура

Определение 23 *Сигнатура функции - это описание её заголовка, в которое обычно входят:*

1. *имя функции;*
2. *число, тип и порядок следоваения передаваемых в неё параметров (в т.ч. и то как именно они передаются, напр. по ссылке или по значению);*
3. *тип возвращаемого значения.*

Таким образом, сигнатура - это все что нужно знать (и не более того) о функции вызывающему её коду (т.е. для вызывающего кода важна только сигнатура, сама же реализация может быть черным ящиком).

Примеры сигнатур функций в ЯП C++:

```
/*Сигнатура функции, не возвращающей значение
 и принимающией указатель на тип int,ссылку
 на некоторый класс и
 аргумент типа int*/
void someFunc1(int*arg1, someClass& arg2, int arg3);

/*Сигнатура функции с неопределённым заранее числом
 входных переменных. Возвращает значение int*/
int horribleCheritage(...);
```

Замечание 16 *Думаю, этого более чем достаточно по данному пункту. Более того, если Вы хотя бы раз программировали, этот раздел был лишним для прочтения.*

2.7 Статические, не статические, константные методы

Определение 24 *Статическим методом называется метод, не привязанный ни к одному объекту типа. Перед такими методами в C++ пишут ключевое слово* ***static****.*

Такие методы не имеют доступа к указателю *this* и, следовательно, не могут влиять на состояние одного определённого объекта класса; внутри класса они могут изменять только **статические поля**. Однако это не мешает вызвать статический метод из какого-нибудь не статического метода класса и получить от него какое-то значение, а затем его использовать внутри этого самого не статического метода.

Определение 25 *Константным методом называется метод, который гарантирует, что не будет изменять объект или вызывать неконстантные методы класса. В C++ определяются с помощью идентификатора const.*

Пример константного метода-геттера:

$$int\ getSize()const\{return\ \_size;\}$$

Наблюдение 1 *Как видно из определений выше, статические методы не могут влиять на состояние конкретного объекта, т.е. статические методы на прямую не изменяют конкретный объект в любом случае. Поэтому статические методы спокойно могут быть константными (т.е. метод, изменяющий статик-поле, может быть константным).*

Замечание 17 *Поскольку мы находимся в теме со static значениями, напомню, что static-переменные или поля инициализируются не в заголовочных файлах, т.к. иначе при подключении их ко всем 'заинтересованным' файлам реализации мы получим переопределение одного и того же поля и, как результат, страшную ошибку.*