Информатика: семестр 3

Программирование на C++

Конспекты лекций

Лектор: Хирьянов Т.Ф.

2 января 2018 г.

Оглавление

Лекция 10

Инкапсуляция

10.1 Понятие инкапсуляции

В Питоне инкапсуляции не было. Когда мы делаем структуру, мы полагаем, что пользователь будет взаимодействовать с ней через те методы, что мы пропишем. Однако это может нарушиться. Для того, чтобы предотвратить такую ситуацию, существует *инкапсуляция*. Инкапсуляция создает ограничения для программиста, тем самым организуя *логику* нашего кода.

Код 10.1: Инкапсуляция на примере класса Student

```cpp
#include<iostream>

class Student
{
private:
    std::string name;
    int age;
public:
    Student(std::string x, int a)
    {
        age = a;
        name = x;
    }
    // Setter
    void set_age(int a)
    {
        if(a > 0 && a > age) age = a;
    }

    void aging()
    {
        age++;
        std::cout << "BIRTHDAY!" << std::endl;
```

```
24      }
25
26      // Getter
27      int get_age() const // See description
28      {
29          return age;
30      }
31  };
32
33  int main()
34  {
35      Student a("Vasya", 17);
36      // a.age = 17   // Won't work, in private section
37      const Student& c = a;
38      std::cout << c.get_age();    // Call const method
39  }
```

В коде выше класс делится на две секции: **private** и **public**. То, что находится в секции **private**, не будет видно снаружи класса. Эти секции могут следовать друг за другом в любом порядке и любом количестве(т.е., мы можем сначала описать публичную, потом приватную, затем опять публичную), но обычно их пишут один раз, объявляя сразу все, что нужно.

Общий принцип инкапсуляции заключается в следующем: все атрибуты класса надо "прятать".

При принятии решения, в какую из секций пихнуть свой метод, следует помнить, что *конструкторы* класса **должны** быть публичными.

Доп. инфа от ТФ. Иногда может возникнуть необходимость разрешить создавать экземпляры класса, но не копировать их. В таких случаях конструктор класса следует сделать публичным, а конструктор клонирования положить в приватную секцию.

10.2 Геттеры и сеттеры

При сокрытии полей стандартом обращения к ним служат *геттеры* (getters) и *сеттеры* (setters) – методы, соответственно возвращающие и изменяющие поле. В нашем примере 10.1 они объявлены на **13** и **18** строках.

Геттеры и сеттеры имеют интересную фишку: они не обязаны работать с существующим полем класса. Вполне возможно, что они будут задавать/возвращать другие величины, но пользователь будет думать, что они содержатся в классе. Так можно создавать удобные абстракции.

10.3 Модификатор const после функции

В примере 10.1 на строчке **25** мы видим модификатор **const** *после* объявления метода.

Это говорит компилятору, что метод не будет менять никаких полей класса в ходе своей работы. Стоит учесть, что если мы будем пользоваться константной ссылкой на экземпляр класса, то в ней мы сможем вызывать только константные методы (чтоб компилятор был уверен, что мы не изменим константный объект).

10.4 Модификатор static

Статические поля

Иногда бывает необходимым сделать поле класса универсальным (т.е. единым) для всех его экземпляров. В таком случае используется модификатор **static**. Рассмотрим пример:

Код 10.2: Простой пример статического поля

```
#include<iostream>

class SelfCounter
{
private:
    static int counter;
public:
    SelfCounter()
    { counter++;}

    ~SelfCounter()
    { counter--;}

    static int get_counter()
    {
        return counter;
    }
};

void foo()
{
    SelfCounter a;
    for(int i = 0; i < 3; i++)
    {
        SelfCounter b;
    }
    std::cout << b.get_counter();
}

int main()
{
    std::cout << SelfCounter::get_counter();     // See description
```

```
33      SelfCounter x;
34      foo();
35  }
```

Здесь мы при создании нового экземпляра класса увеличиваем **count** на 1, а при удалении – уменьшаем. Важно отметить, что переменная **count** существует всегда, даже если не было создано ни одного объекта данного класса.

Статические методы

Однако у такого подхода есть проблема: чтобы получить это поле через какой-либо метод класса, нам необходимо будет создать объект. Решает это, как можно догадаться, *статический метод* (строка **12**). Такой метод привязан к классу, но никак не к объекту класса. Это несет свои минусы: статический метод не может в своем коде обращаться к нестатическим методам и полям (они ведь уже зависят от объекта, а статический метод от этого избавляется). Более того, в статическом методе не определен **this** – указатель на экземпляр класса.

Пространство имен

В коде мы использовали (строка **32**) **SelfCounter** как *пространство имен*. Каждый класс создает свое пространство имен, в котором находятся его поля и методы (разумеется, те, что в публичной секции). Таким способом удобно обращаться к статическим полям и методам.

10.5 Дружественные функции

Бывает, что определенному классу **2** требуется добраться до атрибутов другого класса (класса **1**) (как это могут делать его экземпляры), но при этом сами классы никак не связаны. Для этого придется немного ослабить *инкапсуляцию* и объявить в классе **1** (откуда хотим брать) требуемые дружественные функции или даже классы. Затем дружественные функции или классы достаточно объявить где-нибудь снаружи – и они уже будут иметь доступ внутрь **1**. Более того, можно сделать функцию дружественной даже в том случае, если она не была еще объявлена (предполагая, что это будет сделано позднее). При этом дружественные функции **не** являются методами класса **1** (того, куда мы открываем доступ).

Иллюстрацией этому может служить следующий пример:

Код 10.3: Пример дружественных функций

```
1  class CarMaster;     // For 8 line
2  class Car
3  {
4  private:
5      int engine;
6      friend class CarMaster; // Give class all access
7      // But how to give less permssions?
8      friend void CarMaster::repair(Car& x);  // Told full signature
9      // Won't work, CarMaster is not defined!
10     // We define it in 1st line, yeah!
11 };
12
13 class CarMaster
14 {
15 public:
16     void repair(Car& x)
17     {
18         x.engine++; // Can access private field. CarMaster is a friend!
19     }
20 }
```

В строке **1** мы объявляем класс **CarMaster**, но описываем его позже. Зачем?

Если не объявить заранее CarMaster, то строка **8** будет вызывать ошибку, так как мы попытаемся обратиться к необъявленному классу. А объявить класс **CarMaster** перед **Car** мы не можем, так как в этом случае он будет обращаться к полю еще не объявленного класса **Car**. Решение этого – описать сигнатуру класса **CarMaster** перед классом **Car**, но *описать* ее позднее. Такой трюк мы будем часто использовать в будущем.