Класове в C++

Любомир Чорбаджиев[1]
lchorbadjiev@elsys-bg.org

[1]Технологическо училище "Електронни системи"
Технически университет, София

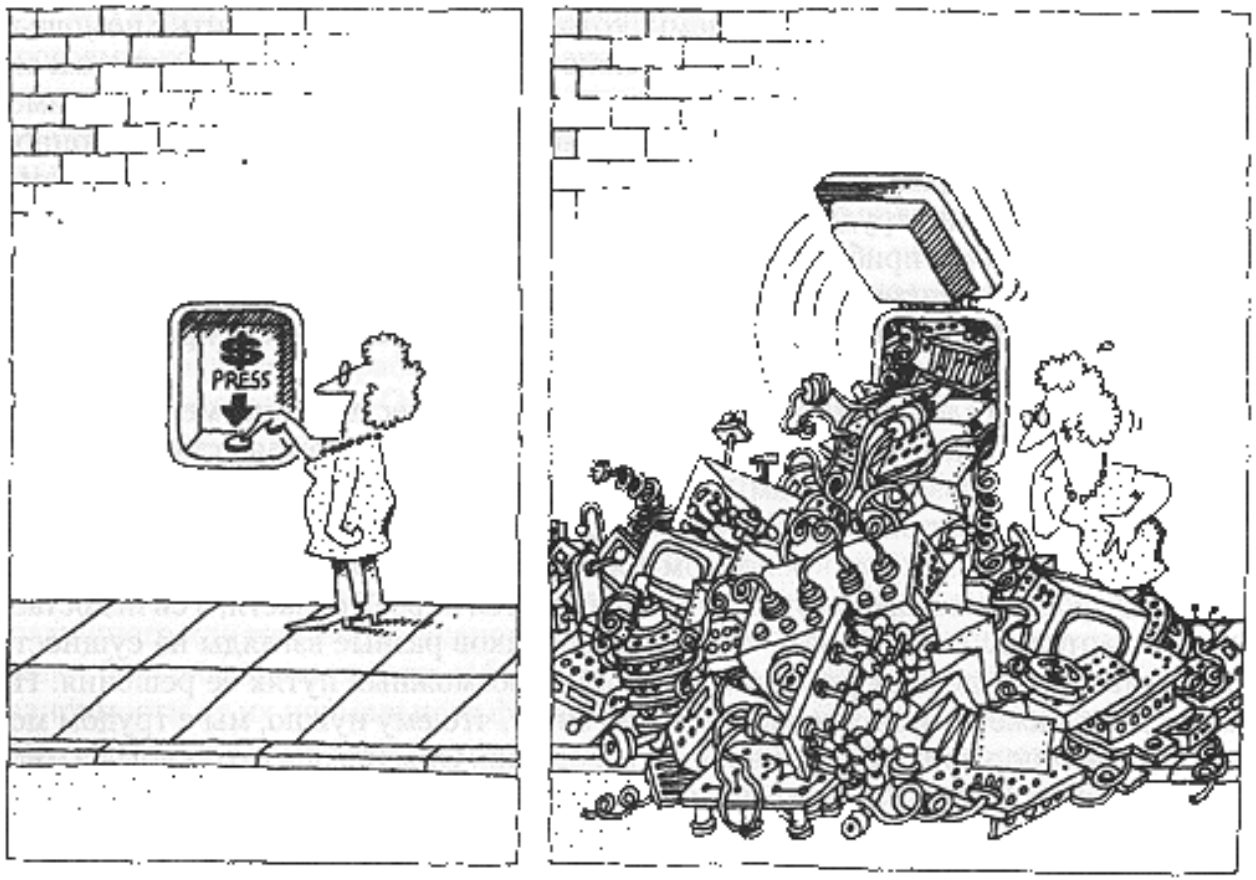
29 март 2009 г.

Съдържание

Сложност на програмното обезпечение

- Сложността е присъща на по-голямата част от програмните продукти, които се разработват в софтуерната индустрия.
- Сред основните причини за сложността на програмните системи са:
  - Сложността на предметната област, която програмните системи моделират.
  - Размерът на програмните продукти.
  - Гъвкавостта на програмните системи.
  - Нуждата от развитие на програмните системи.

Сложност на програмното обезпечение



[1] Grady Booch, *Object Oriented Analysis and Design with Applications, 1991.*

Декомпозиция

- "Начинът за управление на сложни системи е известен още от древността—разделяй и владей (divide et impera)"
  (Dijkstra, E. 1979. Programming Considered as a Human Activity.)
- Програмната система се разделя на части—модули, компоненти.
- Отделните модули трябва да могат да се развиват самостоятелно. Модулите трябва да са независими.
- Връзките между отделните модули трябва да са слаби. Те трябва да се осъществяват чрез специално дефинирани интерфейси.

Капсулация

- Капсулация се нарича скриването на всички детайли на модула, които нямат принос към неговите съществени характеристики.
- Капсулацията е принцип, който се използва при създаването на общата структура на програмата, съгласно който всеки компонент на програмата трябва да капсулира (или скрива) вътрешното си устройство — *encapsulation, information hiding*.
- Интерфейсът на всеки модул трябва да се дефинира така, че да разкрива колкото може по-малко от вътрешната структура на модула.

Капсулация

- Да се разкриват само детайлите, които са съществени за взаимодействието на модула с външния свят. Ударението се поставя върху въпроса “Какво?” прави модула, а не “Как?” го прави.
- Скриват се детайлите на реализацията на поведението на модула.
- Запазва модула от външно вмешателство в тяхното вътрешно състояние.
- Намалява зависимостите между различни части на програмата.
- Модулите взаимодействат един с друг само посредством дефинираните интерфейси.

Обектно-ориентирано програмиране

- Първият обектно-ориентиран език за програмиране е SIMULA I, разработен в периода 1962–1965 от Ole-Johan Dahl и Kristen Nygaard. Малко по-късно се появява и SIMULA 67 (1967).
- В езикът SIMULA 67 са въведени повечето от ключовите концепции на обектно-ориентираното програмиране — обекти, класове, наследяване, полиморфизъм.
- През 90 години на XX век обектно-ориентираното програмиране се превръща в доминираща софтуерна технология.

Обектно-ориентирана терминология

- *Обект* — обединение между данните и функциите, предназначени за обработването на тези данни.
- *Метод* — предоставя услуга, характерна за даден обект.
- *Съобщение* — заявка за изпълнение на даден метод.
- *Клас* — шаблон, по който се създават обекти.
- *Инстанция* — обект, който принадлежи на даден клас.

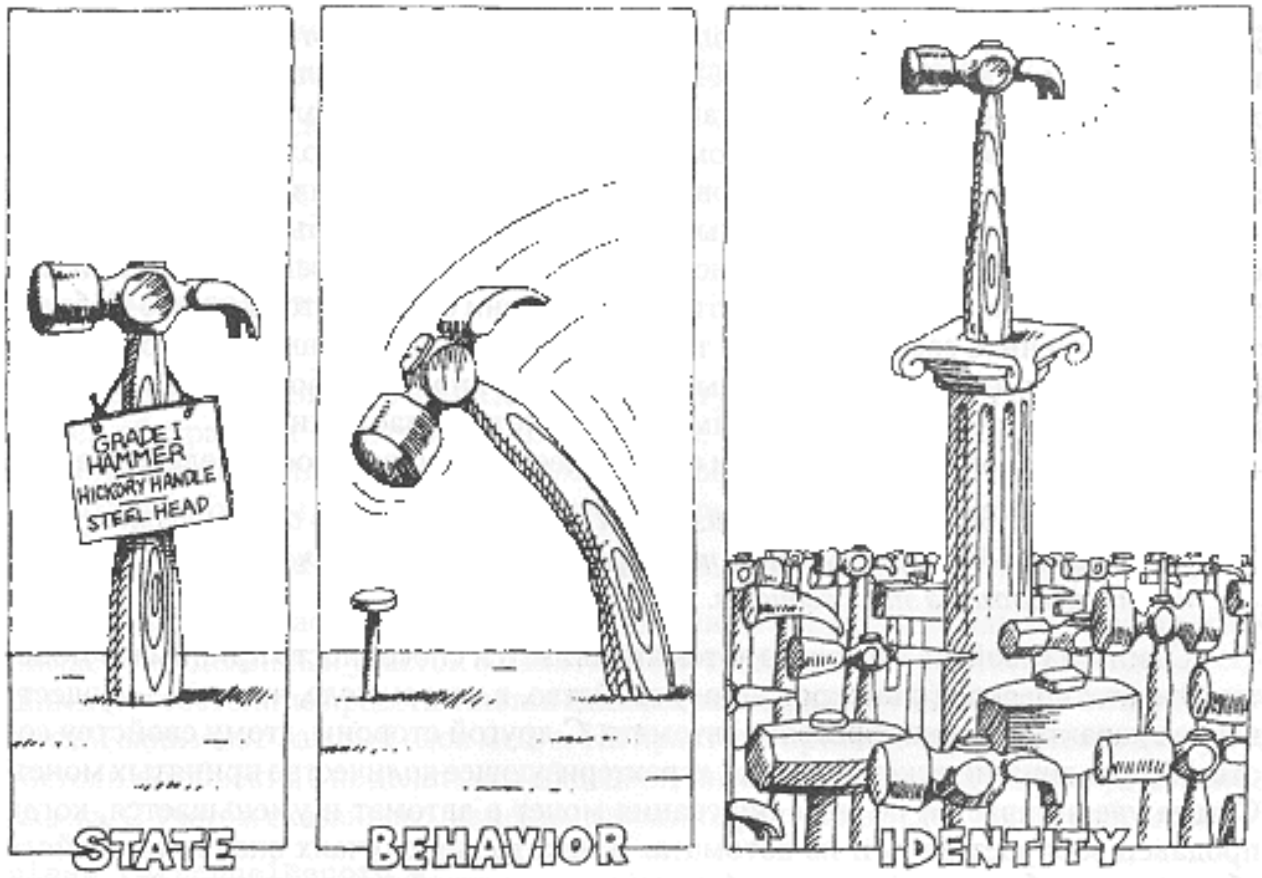
Обектно-ориентирана терминология

- *Капсулация* — скриване на вътрешното устройство на обектите.
- *Наследяване* — механизъм, който позволява повторно използване на спецификациите на класовете.
- *Йерархия от класове* — дървовидна структура, която отразява наследствените взаимоотношения между класовете.
- *Полиморфизъм* — възможността на различни класове да отговарят на едни и същи съобщения по различен начин.

Обектно-ориентиран подход

- Програмната система е организирана около обекти.
- Обектите изпращат съобщения един на друг.
- Данните и функциите, които ги обработват, са обединени в обекти.
- Предметната област на програмната система се моделира в термините на обекти.

Обекти



Grady Booch, *Object Oriented Analysis and Design with Applications, 1991.*

Обекти

- Обектът е инстанция на даден клас. Обектът притежава идентичност (уникален екземпляр).
- Класът дефинира както интерфейса, така и реализацията на множество обекти. Класът определя поведението на всички негови обекти.
- След като обектът се създаде, той не може да променя класа към който принадлежи.

Обекти

- Обектите притежават състояние, поведение и идентичност. Структурата и поведението на подобни обекти се дефинира от техния общ клас. Термините обект и инстанция са взаимозаменяеми.
- Обектите представят реални или абстрактни неща.
- Обикновено не са много сложни или големи.
- Трябва да имат представа за малък брой от другите обекти.
- Трябва да са колкото може по-малко обвързани с интерфейсите на другите обекти.

Класове

- Класът е множество от обекти, които имат общи атрибути и поведение.
- Класификацията на дадено множество от обекти зависи от гледната точка.
- "Клас" е общ термин, който се използва при различни видове класификация.
- "Група еднородни предмети, които по своите особености заемат определено място сред други подобни предмети" [Български Тълковен Речник, Издателство на БАН, 1993г].

Класове

- Група, множество или вид, притежаващи общи атрибути.
- В контекста на обектно-ориентираното програмиране, класът е множество от обекти, които притежават обща структура и общо поведение.
- Класът представлява спецификация на структурата, поведението (методите) и наследствеността на обектите.
- Абстрактен клас се нарича клас, който няма инстанции.

Класове в C++

- Класовете в C++ предоставят на програмистите механизъм за създаване на нови типове.
- Класът е тип, дефиниран от потребителя. Добре подбраният набор от типове, дефинирани от потребителя, прави програмата по-кратка и по-изразителна.
- Основният смисъл на дефинирането на нови типове се състои в разделянето на несъществените детайли от свойствата, които имат определящо значение за правилното функциониране на програмата.

Дефиниция на клас

- Дефиницията на класа се състои от две части:
  - *Заглавна част*, която се състои от ключовата дума **class** и идентификатор, обозначаващ името на класа.
  - *Тяло на класа*, което е затворено във фигурни скоби.
- Дефиницията на класа трябва да бъде последвана от точка и запетая или от списък от дефиниции на променливи.

```
class Point {/*...*/};
class Rectangle {/*...*/} r1,r2;
```

Тяло на класа

- В тялото на класа се дефинира списъкът от членове на класа и нивото на достъп до тях.
- Тялото на класа дефинира област на действие на класа. Декларирането на членове на класа в тялото на класа въвежда имената им в областта на действие на класа.
- Напълно възможно е два класа да имат членове с еднакви имена, тъй като те се отнасят до различни области на действие.

Обекти

- Дефиницията на класа може да се разглежда като шаблон, по който се създават обекти.
- Дефинирането на клас създава нов тип в областта на видимост, в която е направена дефиницията.
- За да се дефинира обект от даден клас, трябва да се дефинира променлива от съответния тип.

```
Point p1;
Point p2;
```

Член-променливи

- Дефинирането на член-променливите на класа е аналогично на дефинирането на променливи.
- Класовете могат да имат *нестатични* и *статични* член-променливи.

```
1 class Point {
2   double x_;
3   double y_;
4 };
```

```
1 class Rectangle {
2   double x_, y_, width_, height_;
3 };
```

Член-променливи

- Нестатичните член-променливи не могат да бъдат инициализирани при тяхното дефиниране.

```
1 class Foo {
2   int bar_=42; // error!
3 };
```

- При дефинирането на член-променлива не се заделя памет. Заделянето на памет и инициализирането на член-променливите се извършва едва при създаването на обект от дадения клас.
- Член-променливите на класа се инициализират от *конструктора* на класа.

Член-функции

- *Член-функциите* реализират множеството от операции, които могат да се извършват върху обектите от даден клас.
- За да стане една функция член на класа, тя трябва да бъде декларирана в тялото на класа.
- Член-функциите могат да се дефинират в тялото на класа.

```
1 class Point {
2   ...
3   void set_x(double x);
4   int get_x() {return x_;}
5 };
```

Член-функции

- Член-функциите са дефинирани в областта на действие на класа.
- Имената на член-функциите не се виждат извън областта на действие на класа.
- Извикването на член функция става като се използва оператора . (точка) или оператора -> (стрелка).

```
1 Point p1;
2 p1.set_x(4.2);
3 Point* ptr=&p1;
4 ptr->get_x();
```

- Член-функциите имат пълен достъп до всички член-променливи и член-функции на класа.

Модификатори за достъп

- *Капсулирането (скриването на информацията)* е механизъм който предпазва вътрешното представяне на данните.
- Класовете в C++ имат силно развит механизъм за скриване на информацията. В основата му са спецификаторите за достъп — **public**, **private** и **protected**.
- *Публичните членове* на класа са достъпни от всички точки на програмата.
- *Скритите членове* на класа са достъпни само в член-функциите на класа и в *приятелите* на класа.
- *Защитените членове* се държат като публични за членовете на производните класове и като скрити за всички останали точки на програмата.

Достъп до членовете на класа

```
1  class Point {
2    double x_;
3    double y_;
4  public:
5    void set_x(double x) {x_=x;}
6    void set_y(double y) {y_=y;}
7    double get_x(void) {return x_;}
8    double get_y(void) {return y_;}
9  };
10 ...
11 Point p1, p2;
12 p1.set_x(10.0);
13 p2.x_=10.0;// грешка
```

Достъп до членовете на класа

```
1 #include <iostream>
2 using namespace std;
3
4 class Point {
5   double x_;
6   double y_;
7 public:
8   ...
9 };
10
11 void print(Point& p) {
12   cout<<p.x_<<"␣"<<p.y_<<endl;// грешка!!
13 }
```

Приятели на клас

- Механизмът на *приятелите* позволява да се даде достъп на определени функции до скритата част на класа.
- Приятелските декларации съдържат ключовата дума **friend**. Няма значение в коя секция на тялото на класа е направена приятелската декларация.

```
1 class Point {
2   friend void print(Point& p);
3   double x_, y_;
4 public:
5   ...
6 };
7 void print(Point& p) {
8   cout << p.x_ << "␣" << p.y_ << endl;
9 }
```

Обекти

- Дефиницията на класа може да се разглежда като шаблон, по който се създават обекти.
- Дефинирането на клас създава нов тип в областта на видимост, в която е направена дефиницията.
- За да се дефинира обект от даден клас, трябва да се дефинира променлива от съответния тип.
- При дефиниране на променлива от типа на даден клас се създава обект (екземпляр, инстанция) от класа. Всеки обект притежава собствено копие на член-променливите на класа.

Обекти

- Всеки обект притежава собствено копие на нестатичните член-променливи на класа.
- Всички обекти от даден клас си поделят само едно копие на член-функциите на класа. С други думи в програмата има само едно копие на член-функциите на класа.

```
Point p1,p2;
p1.get_x();
p2.get_x();
```

Когато методът `get_x()` се извиква чрез обекта `p1`, то използваната в метода член-променливата `x_` принадлежи на обекта `p1`. Аналогично за `p2`.

Обекти

```
1 class Point {
2   double x_, y_;
3 public:
4   void set_x(double x) { x_=x;}
5   double get_x(void) {return x_;}
6 };
```

```
1 Point p1, p2;
2 p1.set_x(10);
3 p2.set_x(20);
4 p1.get_x();
5 p2.get_x();
```

Конструктори

- Член-променливите не могат да се инициализират при тяхната дефиниция. За инициализиране на обектите от даден клас се използва специализирана член-функция, която се нарича *конструктор*.
- Името на конструктора съвпада с името на самия клас.

```
1 class Point {
2   double x_,y_;
3 public:
4   Point(double x, double y); // конструктор
5   //...
6 };
```

Конструктори

- При създаването на всеки обект се извиква конструктор на класа, който инициализира обекта.
- Ако конструкторът има аргументи, то те трябва да се предадат при извикването му.

```
1 Point p1 = Point(1.0,1.0);
2 Point p2(2.0,2.0);
3 Point p3; // грешка
4 Point p4(4.0); // грешка
```

Конструктори

- Има възможност за един клас да се дефинират няколко конструктора.

```
1 class Point {
2 public:
3   Point(double x, double y);
4   Point(void);
5 };
```

- Кой конструктор ще се извика в даден конкретен случай се решава според аргументите, които се предадени при извикването на конструктора.

```
1 Point p1(1.0,1.0);
2 Point p2;
```

Конструктор по подразбиране

- *Конструктор по подразбиране* се нарича конструктор, който може да се извика без да му се предават аргументи.
- Ако програмистът не дефинира никакъв конструктор на класа, то компилаторът при необходимост ще генерира конструктор по подразбиране.
- Генерираният от компилатора конструктор по подразбиране извиква конструкторите по подразбиране на всички член-променливи на класа.

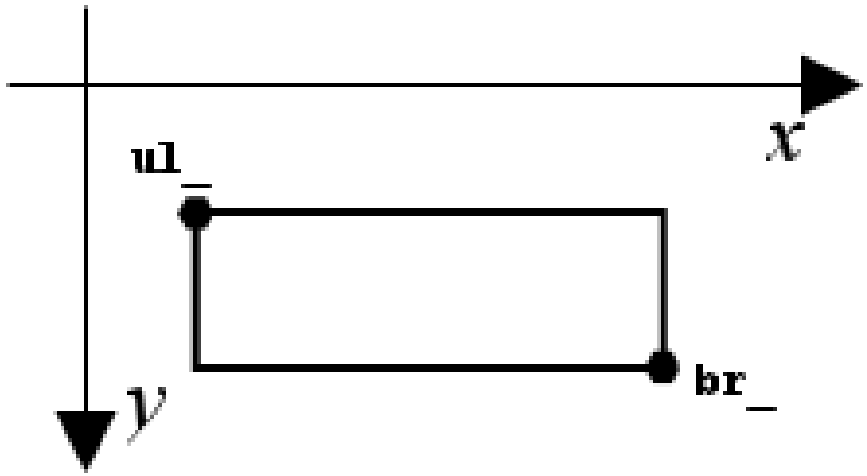
Конструктори: пример

```
1 class Point {
2   double x_, y_;
3 public:
4   Point(double x, double y) {
5     x_=x;
6     y_=y;
7   }
8 };
```

Конструктори: пример

```
1 class Point {
2   double x_, y_;
3 public:
4   Point(double x, double y)
5     :x_(x), y_(y)
6   {}
7 };
```

Конструктори: пример



Конструктори: пример

```
1 class Rectangle {
2   Point ul_, br_;
3 public:
4   Rectangle(double x,double y,
5             double w,double h){
6     ul_.set_x(x).set_y(y);
7     br_.set_x(x+w).set_y(y+h);
8   }
9 };
```

Конструктори: пример

```
1 class Rectangle {
2   Point ul_, br_;
3 public:
4   Rectangle(double x,double y,
5             double w,double h)
6     : ul_(x,y), br_(x+w,y+h)
7   {}
8 };
```

Дефиниране на член-функции: Point.hpp

```
1  #ifndef POINT_HPP__
2  #define POINT_HPP__
3
4  class Point {
5    double x_, y_;
6  public:
7    Point(double x, double y);
8    Point& set_x(double x);
9    ...
10 };
11 #endif
```

Дефиниране на член-функции: Point.cpp

```
1 #include "Point.hpp"
2
3 Point::Point(double x, double y)
4   : x_(x), y_(y)
5 {}
6 Point& Point::set_x(double x) {
7   x_=x;
8   return *this;
9 }
10 ...
```