Объектно-ориентированное программирование

2017

Объектно-ориентированный язык

Язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

1. Поддерживаются объекты, то есть абстракции данных, имеющие интерфейс в виде именованных операций и собственные данные, с ограничением доступа к ним.
2. Объекты относятся к соответствующим типам (классам).
3. Типы (классы) могут наследовать атрибуты супертипов (суперклассов)

Cardelli, L. and Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. December 1985. ACM Computing Surveys vol.17(4). p.481.

Объектно-ориентированная модель

Объектно-ориентированное программирование

Объектно-ориентированное программирование - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Объектно-ориентированная декомпозиция.

1. С точки зрения программирования класс – это сущность имеющая отображение в реальном мире (или полезная для реализации алгоритма) и важная с точки зрения разрабатываемого приложения.
2. В сценарии (алгоритме) любое существительное может являться бизнес-объектом или его атрибутом.
3. Если факт существования объекта важнее чем его значение, то скорее всего это существительное – объект, в противном случае – атрибут.

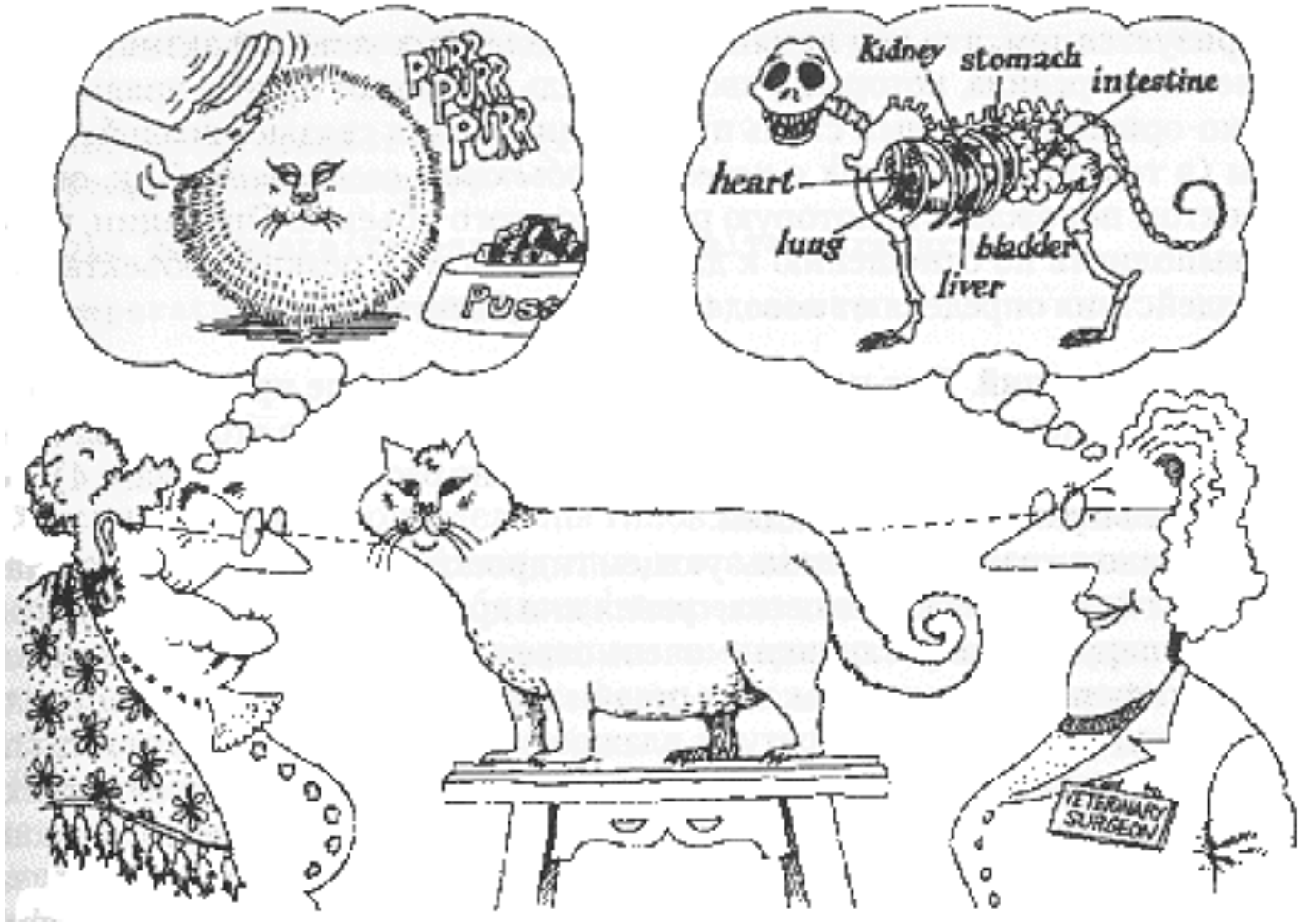
Объектная модель

1. абстрагирование;
2. инкапсуляция;
3. модульность;
4. иерархия;
5. типизация;
6. параллелизм;
7. сохраняемость.

1. абстрагирование

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

1. **Абстракция сущности**
   Объект представляет собой полезную модель некой сущности в предметной области
2. **Абстракция поведения**
   Объект состоит из обобщенного множества операций
3. **Абстракция виртуальной машины**
   Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня
4. **Произвольная абстракция**
   Объект включает в себя набор операций, не имеющих друг с другом ничего общего


PURR PURR PURR
Kidney
stomach
intestine
heart
lung
bladder
liver
VETERINARY SURGEON

Пример – абстракция сущности

Решение квадратного уравнения вида $\mathbf{ax^2+bx+c=0}$ находится по формуле:

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Объектом будет являться «Квадратное уравнение» - SquareEquation.

Методом – нахождение решения.

Атрибутами – коэффициенты уравнения;

Класс на C++

Example06_SecondClass

```
class SquareEquation {
public: SquareEquation(double,double,double);
        double FindX1();
        double FindX2();
private:
        double a;
        double b;
        double c;
};
```

```
#include "SquareEquation.h"
#include <math.h>

double SquareEquation::FindX1(){
return (-b-sqrt(b*b-4*a*c))/(2*a);
};

double SquareEquation::FindX2(){
return (-b-sqrt(b*b-4*a*c))/(2*a);
};
```

Конструктор

Если у класса есть конструктор, он вызывается всякий раз при создании объекта этого класса. Если у класса есть деструктор, он вызывается всякий раз, когда уничтожается объект этого класса.

Объект может создаваться как:

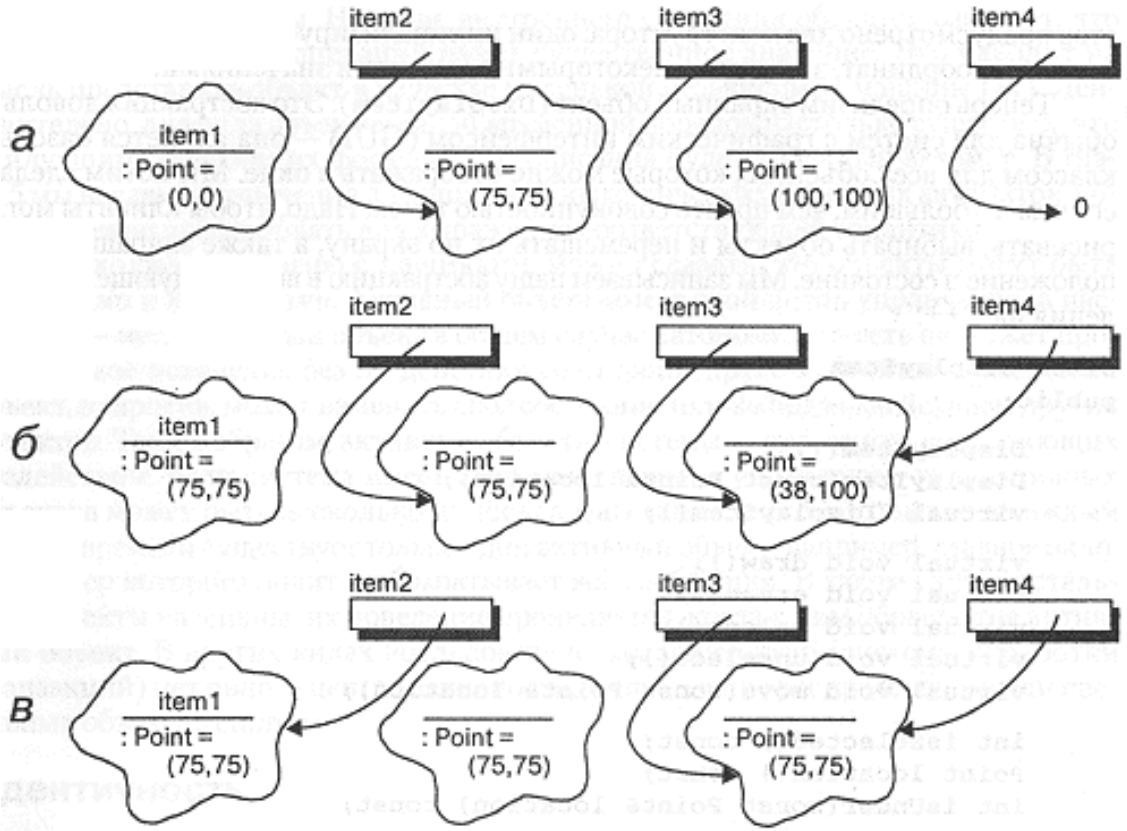
1. автоматический, который создается каждый раз, когда его описание встречается при выполнении программы, и уничтожается по выходе из блока, в котором он описан;
2. статический, который создается один раз при запуске программы и уничтожается при ее завершении;
3. объект в свободной памяти, который создается операцией new и уничтожается операцией delete;
4. объект-член, который создается в процессе создания другого класса или при создании массива, элементом которого он является.

Жизненый цикл данных

Example07_Life

```
class Life{
public:

    // Конструкторы
    Life() { std::cout << "I'm alive" << std::endl;}
    Life(const char* n) : Life() { name=n; std::cout<< "My name is " << name << std::endl;};

    // Деструкторы
    ~Life() { std::cout << "Oh no! I'm dead!" << std::endl;}
private:
    std::string name;
};
```

Мы работаем с указателями на объекты



Сравнение объектов и указателей

Example08_Reference

```
1.int main(int argc, char** argv) {
2.     Life a("Ivan"),b("Ivan");
3.     // result - false
4.     std::cout << "Is Equal pointers:" << (&a==&b) << std::endl;
5.
6.     Life *ptr = &a;

7.     // result true
8.     std::cout << "Is Equal pointers:" << (ptr==&a) << std::endl;
9.     ptr= &b;

10.     //result again false
11.     std::cout << "Is Equal pointers:" << (ptr==&a) << std::endl;
12.     // Error no operator == defined!
13.     //std::cout << "Is Equal:" << (a==b) << std::endl;
14.    return 0;
15.}
```

Lvalue & Rvalue переменные

С каждой **обычной** переменной связаны две вещи – **адрес** и **значение**.

int I; // создать переменную по адресу, например 0x10000

I = 17; // изменить значение по адресу 0x10000 на 17

А что будет если у меня есть только значение? Могу ли я сделать так: 20=10; ?

Итого: с любым выражением связаны либо адрес и значение, либо только значение.

Для того, чтобы отличать выражения, обозначающие объекты, от выражений, обозначающих только значения, ввели понятия **lvalue** и **rvalue**. Изначально слово lvalue использовалось для обозначения выражений, которые могли стоять слева от знака присваивания (*left-value*); им противопоставлялись выражения, которые могли находиться только справа от знака присваивания (*right-value*).

С каждой функцией компилятор также связывает две вещи: ее адрес и ее тело («значение»).

Пример

```
char a [10];
i      —   lvalue
++i    —   lvalue
*&i    —   lvalue
a[5]   —   lvalue
a[i]   —   lvalue

однако:
10     —   rvalue
i + 1  —   rvalue
i++    —   rvalue
```

Rvalue сылки

Example09_Reference

```
1.#include <cstdlib>
2.#include <iostream>

3.void swap(int &a, int &b){
4.     a = a+b;
5.     b = a-b;
6.     a = a-b;
7.}

8.int main(int argc, char** argv) {
9.      int a=10,b=20;
10.     swap(a,b);
11.     std::cout << "a=" << a << " ,b=" << b << std::endl;
12.     return 0;
13.}
```

Lvalue ссылки

Example10_Reference

```
1.void over(A &a) {
2.    std::cout << "LValue:" << a.value << std::endl;
3.}

4.void over(A &&a) {
5.   std::cout << "RValue:" << a.value << std::endl;
6.}

7.void cross(A a) {
8.   std::cout << "Copy:" << a.value << std::endl;
9.}
```

Пустой указатель **nullptr**

Раньше, для обнуления указателей использовался макрос NULL, являющийся нулем — целым типом, что, естественно, вызывало проблемы (например, при перегрузке функций).

Ключевое слово **nullptr** имеет свой собственный тип **std::nullptr_t**, что избавляет нас от бывших проблем.

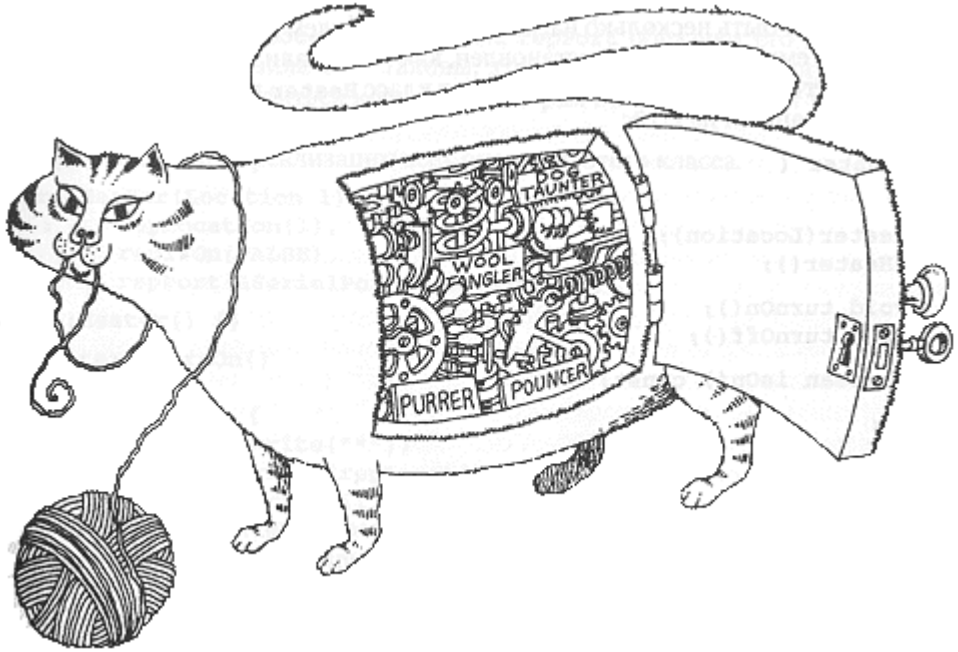
Существуют неявные преобразования nullptr к нулевому указателю любого типа и к Boolean.

Пример

Example11_nullptr

```
1.#include <cstdlib>
2.#include <iostream>

3.void foo(int* p) { std::cout << "nullptr" << std::endl; };
4.void foo(int a) { std::cout << "0?" << std::endl; };

5.int main(int argc, char** argv) {
6.      int* p1 = 0;
7.      int* p2 = nullptr;
8.      void *p3 = nullptr;
9.      if (p1 == p2) std::cout << "1:Equal!" << std::endl;
10.     if (p3 == p2) std::cout << "2:Equal!" << std::endl;
11.     delete p3; // errror? no way!
12.     foo(p1);
13.     foo(p2);
14.     return 0;
15.}
```

2. инкапсуляция



Инкапсуляция - это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Инкапсуляция, пример: контроль доступа в C++

Член класса может быть **частным** (**private**), **защищенным** (**protected**) или **общим** (**public**):

1. Частный член класса X могут использовать только функции-члены и друзья класса X.
2. Защищенный член класса X могут использовать только функции-члены и друзья класса X, а так же функции-члены и друзья всех производных от X классов (рассмотрим далее).
3. Общий член класса можно использовать в любой функции.

Контроль доступа применяется единообразно ко всем именам. На контроль доступа не влияет, какую именно сущность обозначает имя.

Друзья класса объявляются с помощью ключевого слова **friend**. Объявление указывается в описании того класса, к частным свойствам и методам которого нужно подучать доступ.

Пример

Example12_PublicPrivate

```
1. #include <iostream>
2. class B;
3. class A {
4.     friend B;
5. private:
6.     int value;
7. public:
8.     A(int v) : value(v) {};};
9. class B {
10.public:
11.    B(A a[], int size) {
12.        int total = 0;
13.        for (int i = 0; i < size; i++) total += a[i].value;
14.        std::cout << "Total:" << total << std::endl;    }
15.};
16.int main(int argc, char** argv) {
17.    A array[3] = {A(1), A(2), A(3)};
18.    B sum(array, 3);
19.    return 0;
20.}
```

3. модульность

Модульность - это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули.

Модули в C++ [1/2]

1. Программа C++ почти всегда состоит из нескольких раздельно транслируемых "модулей".
2. Каждый "модуль" обычно называется исходным файлом, но иногда - единицей трансляции. Он состоит из последовательности описаний типов, функций, переменных и констант.
3. Описание **extern** позволяет из одного исходного файла ссылаться на функцию или объект, определенные в другом исходном файле.

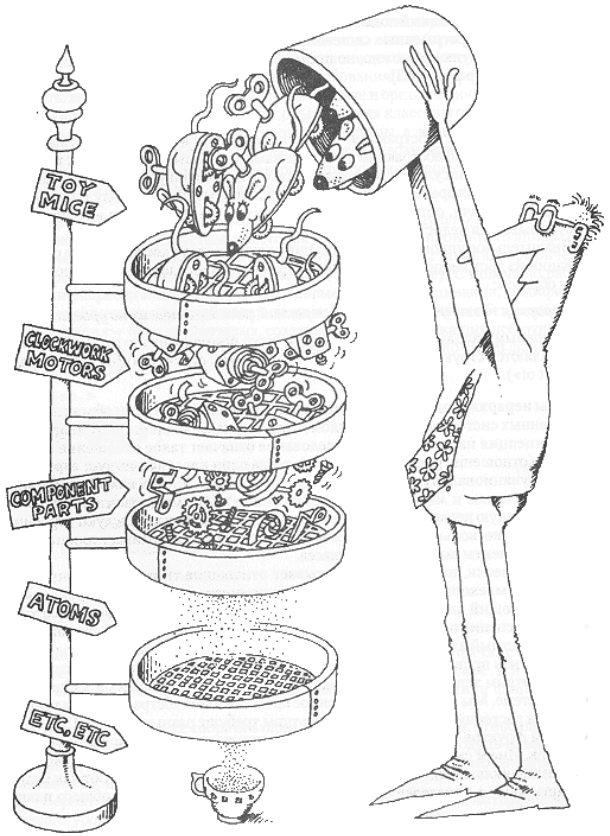
```
extern "C" double sqrt ( double );

extern ostream cout;
```

Модули в C++ [2/2]

Заголовочные файлы можно включать во все исходные файлы, в которых требуются описания внешних. Например, описание функции sqrt хранится в заголовочном файле стандартных математических функций с именем math.h.

```
#include <math.h>
//...
x = sqrt ( 4 );
```

4. иерархия



Иерархия - это упорядочение абстракций, расположение их по уровням.

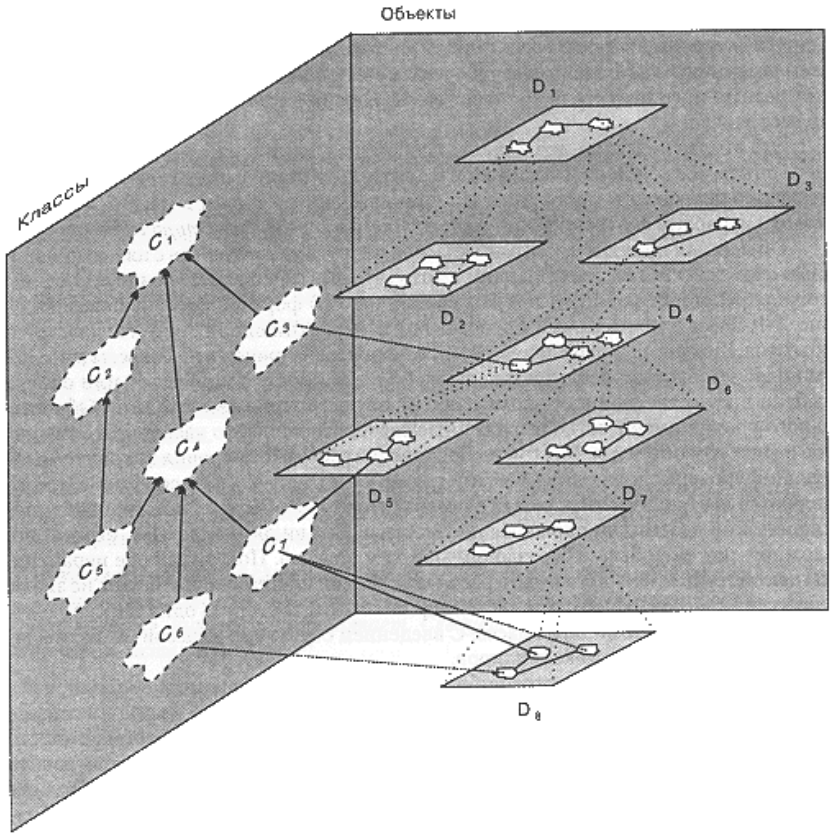
Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия "is-a") и структура объектов (иерархия "part of")

Примеры:

1. аггрегация

2. наследование

Классы и объекты



Классы связанны между собой иерархией наследования.

Объекты связанны между собой динамическими ссылками (можно думать о них как о указателях в С).

Пример иерархии: агрегация в C++

```
class A
{
 public:
     B * link_to_b;
     C array_of_c[100];
};
```

Пример иерархии: Наследование

В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общем суперклассе. По этой причине говорят о наследовании, как об иерархии *обобщение-специализация.* Суперклассы при этом отражают наиболее общие, а подклассы - более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы и даже скрыты.

Наследование в C++

Example13_Inheritance

```
class BaseItem
{
public:
virtual const char * GetMyName();
const char * GetMyOriginalName();
};
```

```
#include "BaseItem.h"
class ChildItem  : public BaseItem
{
public:
virtual const char * GetMyName();
const char * GetMyOriginalName();
};
```

О виртуальных функциях

1. Виртуальную функцию можно использовать, даже если нет производных классов от ее класса.
2. В производном же классе не обязательно переопределять виртуальную функцию, если она там не нужна.
3. При построении производного класса надо определять только те функции, которые в нем действительно нужны.

Абстрактные классы

```
class Item
{
public:
        virtual const char * GetMyName() = 0;
};
```

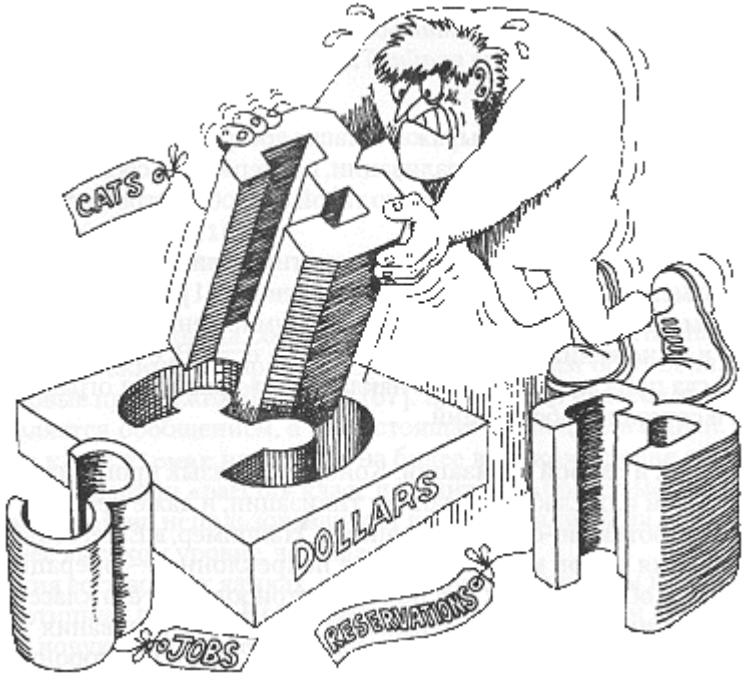
Абстрактный класс нельзя создать!

Конструкторы при наследовании

```
class employee {
// ...
public:
// ...
employee(char* n, int d);
};
class manager : public employee {
// ...
public:
// ...
manager(char* n, int i, int d);
};
```

```
manager::manager(char* n, int l, int d)
: employee(n,d), // вызов родителя
level(l), // аналогично level=l
group(0)
{
}
```

5. типизация



Тип- точная характеристика свойств, включая структуру и поведение, относящаяся к некоторой совокупности объектов.

Zilles, S. 1984. Types, Algebras, and Modeling, in On Conceptual Modeling: Perspectives from Artificial Intelligence. Databases, and Programming Languages. New York, NY: Springer-Verlag, p.442.

Типизация - это способ защититься от использования объектов одного класса вместо другого, или по крайней мере управлять таким использованием.

Эквивалентность типов

Два структурных типа считаются различными даже тогда, когда они имеют одни и те же члены.

Например, ниже определены различные типы:

```
class s1 { int a; };
class s2 { int a; };
```

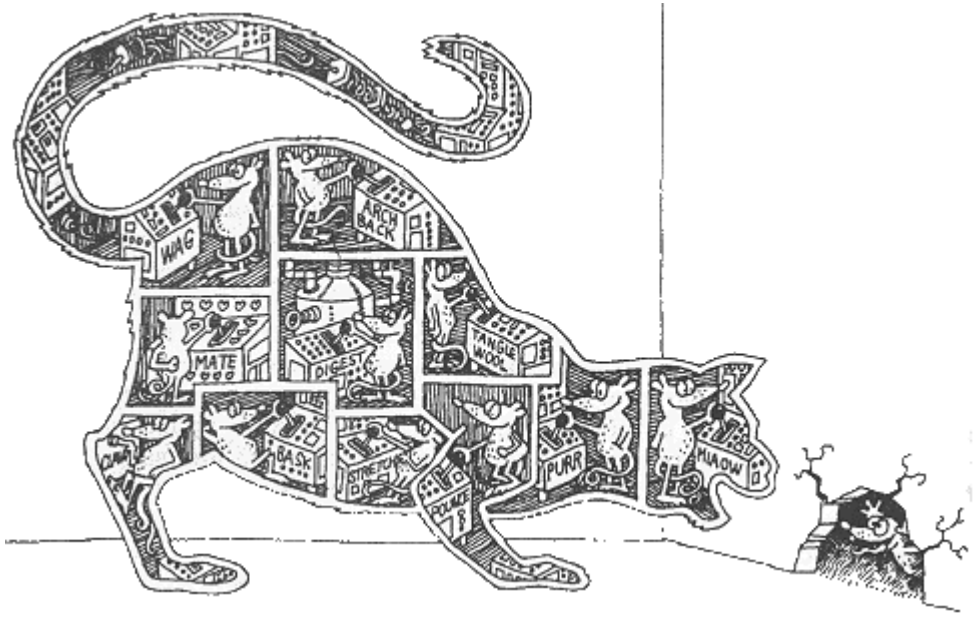
В результате имеем:

```
s1 x;
s2 y = x; // ошибка: несоответствие типов
```

Кроме того, структурные типы отличаются от основных типов, поэтому получим:

```
s1 x;
int i = x; // ошибка: несоответствие типов
```

6. параллелизм



Параллелизм позволяет различным объектам действовать одновременно

Будет подробно рассмотрено в следующих лекциях.

Спасибо!

НА СЕГОДНЯ ВСЕ