



Объектно-ориентированное программирование

2018

Кто я?

Старший преподаватель кафедры 806

Дзюба Дмитрий Владимирович

ddzuba@yandex.ru

Примеры https://github.com/DVDemon/mai_oop

канал телеграмм https://t.me/oop_mai



Базовые требования к слушателям

- 1. Знание языка программирования С**
при изложении материала будем считать, что слушатель знает основные конструкции языка С, типы данных и правила написания программ

- 2. Знание операционной системы Microsoft Windows 7/8/10**
практические занятия будут проходить на компьютерах, работающих под управлением Microsoft Windows 7

- 3. Знание среды разработки Microsoft Visual Studio 2013/2015/2017 или Visual Studio Code + GCC**
лабораторные работы должны делаться в Microsoft Visual Studio/VS Code, мы будем создавать консольные приложения с unmanaged кодом

Отчетность по курсу рейтинг

5-балльная система	Рейтинговая система	Европейская система
5 - Отлично	90-100	A
4 – Хорошо	82-89	B
	75-81	C
3 - Удовлетворительно	67-74	D
	60-66	E
2 - Неудовлетворительно	Менее 60	F

Балы даются:

1. Вовремя сделанная и сданная Лабораторная работа (8 шт) – от 5 до 15 баллов.
2. Лабораторная работа сданная с задержкой в две недели оценивается не более 5 баллов.
3. Зачет (два задания) по 15 баллов за задание (итого до 30).



Расписание занятий

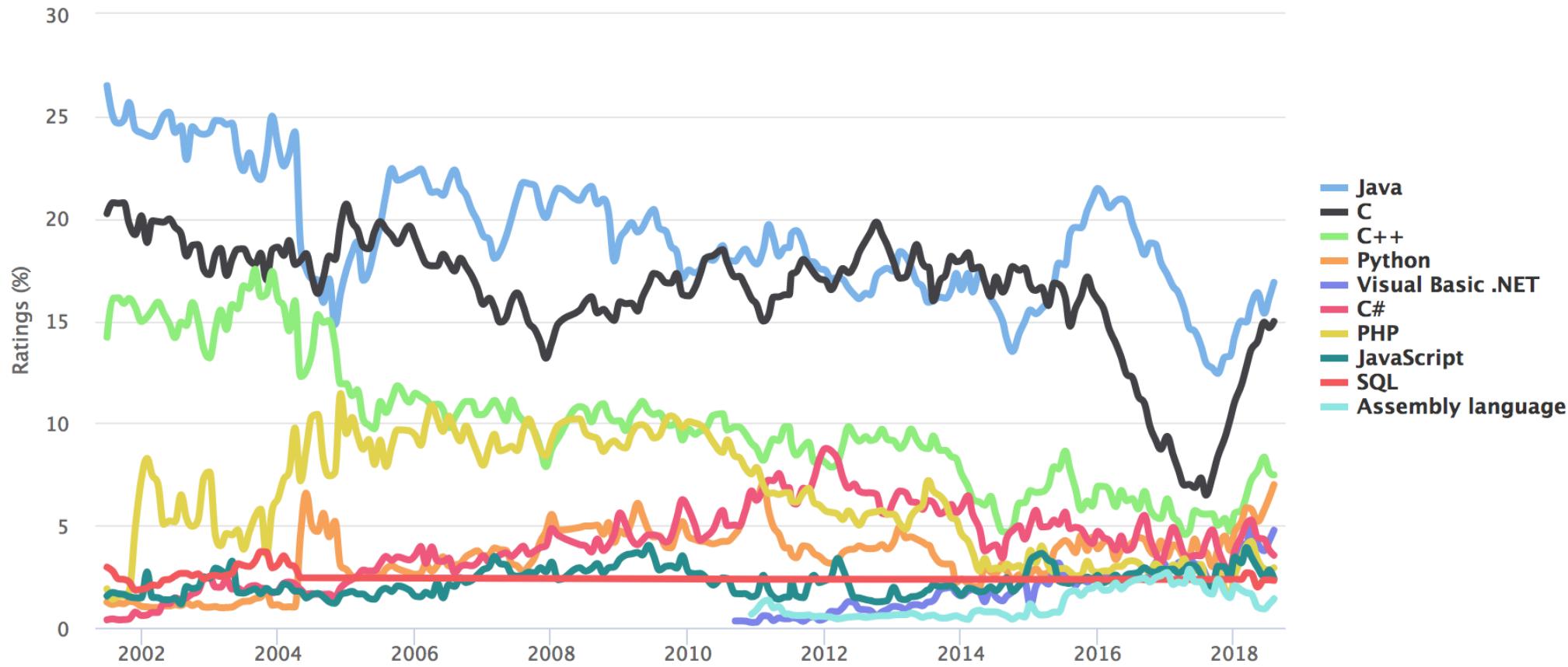
1. Лекции проходят каждую неделю по пятницам с 10.45. В 460 ГУК.
2. Лабораторные работы:
 - 08-204Б-17 понедельник 16.30-19.45
 - 08-206Б-15 пятница 13.00-16.15
 - 08-207Б-17 вторник 13.00-16.15
 - 08-208Б-17 пятница 13.00-16.15



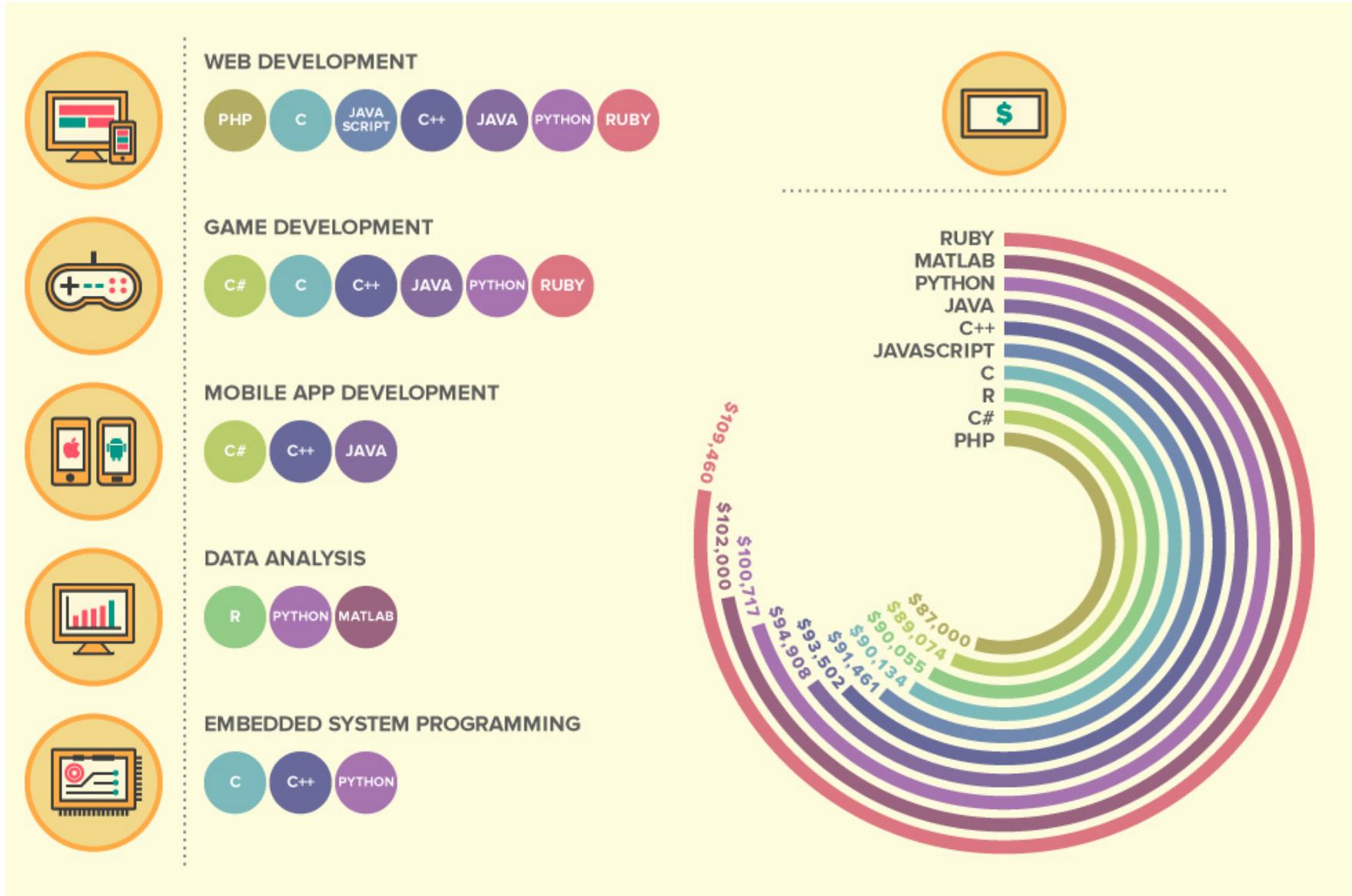
Online трансляция: <https://goo.gl/BvB3Uc>

TIOBE Programming Community Index

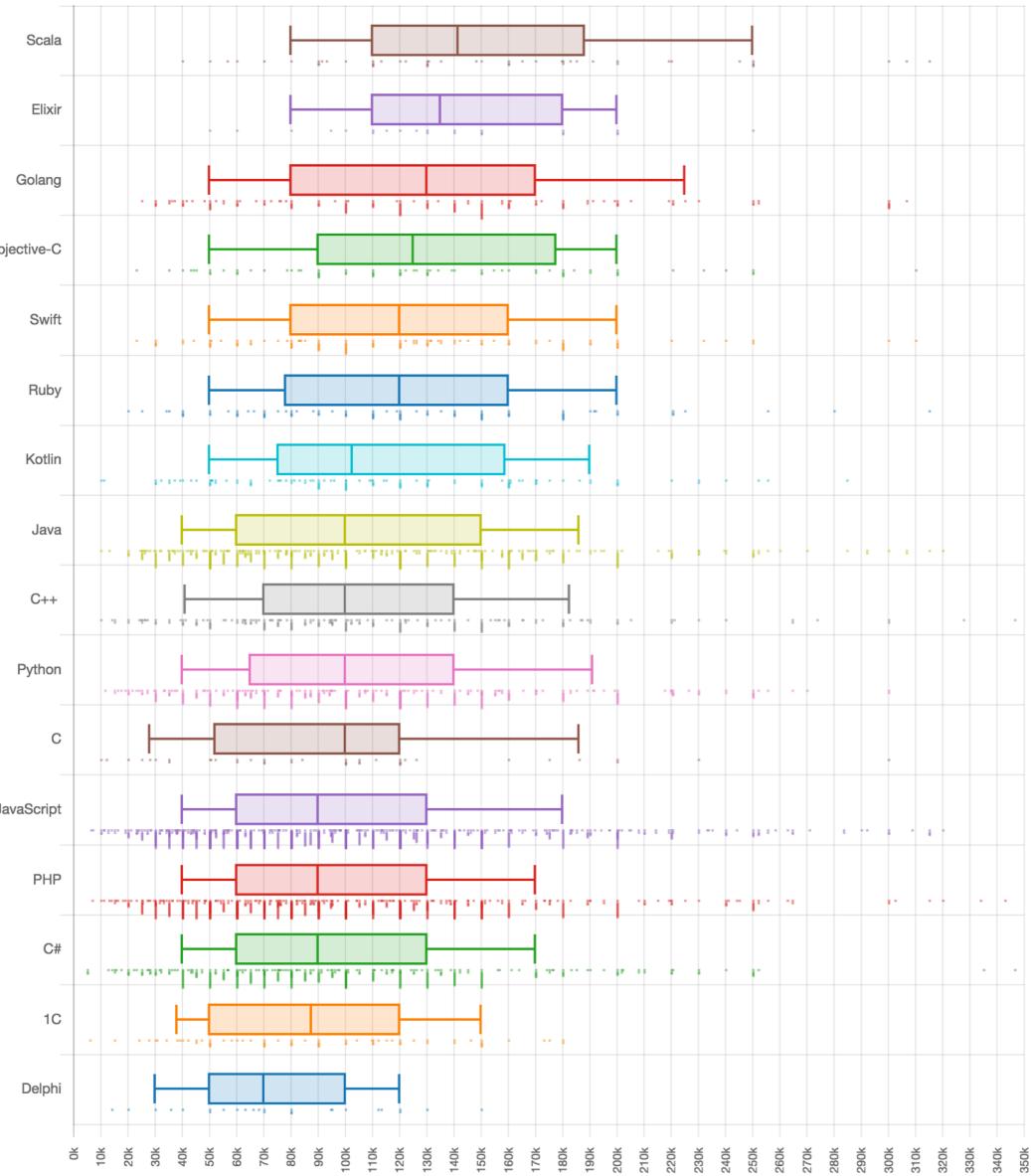
Source: www.tiobe.com



В курсе мы будем изучать язык программирования C++ ориентируясь на 11/14 стандарт.



Зарплаты в зависимости от языка программирования



<https://habr.com/company/moikrug/blog/420391/>

Лабораторные работы

№	Цель
1	<ul style="list-style-type: none">• Изучение базовых понятий ООП.• Знакомство с классами в C++.• Знакомство с операциями ввода-вывода из стандартных библиотек.
2	<ul style="list-style-type: none">• Закрепление навыков работы с классами.• Знакомство с перегрузкой операторов.• Знакомство с дружественными функциями.• Создание простых динамических структур данных.• Работа с объектами, передаваемыми «по значению».
3	<ul style="list-style-type: none">• Закрепление навыков работы с классами.• Знакомство с умными указателями.
4	<ul style="list-style-type: none">• Знакомство с шаблонами классов.• Построение шаблонов динамических структур данных.
5	<ul style="list-style-type: none">• Закрепление навыков работы с шаблонами классов.• Построение итераторов для динамических структур данных.
6	<ul style="list-style-type: none">• Закрепление навыков по работе с памятью в C++.• Создание аллокаторов памяти для динамических структур данных.
7	<ul style="list-style-type: none">• Создание сложных динамических структур данных.• Закрепление принципа ОСР.
8	<ul style="list-style-type: none">• Знакомство с параллельным программированием в C++.
9	<ul style="list-style-type: none">• Знакомство с лямбда-выражениями.



Среда разработки

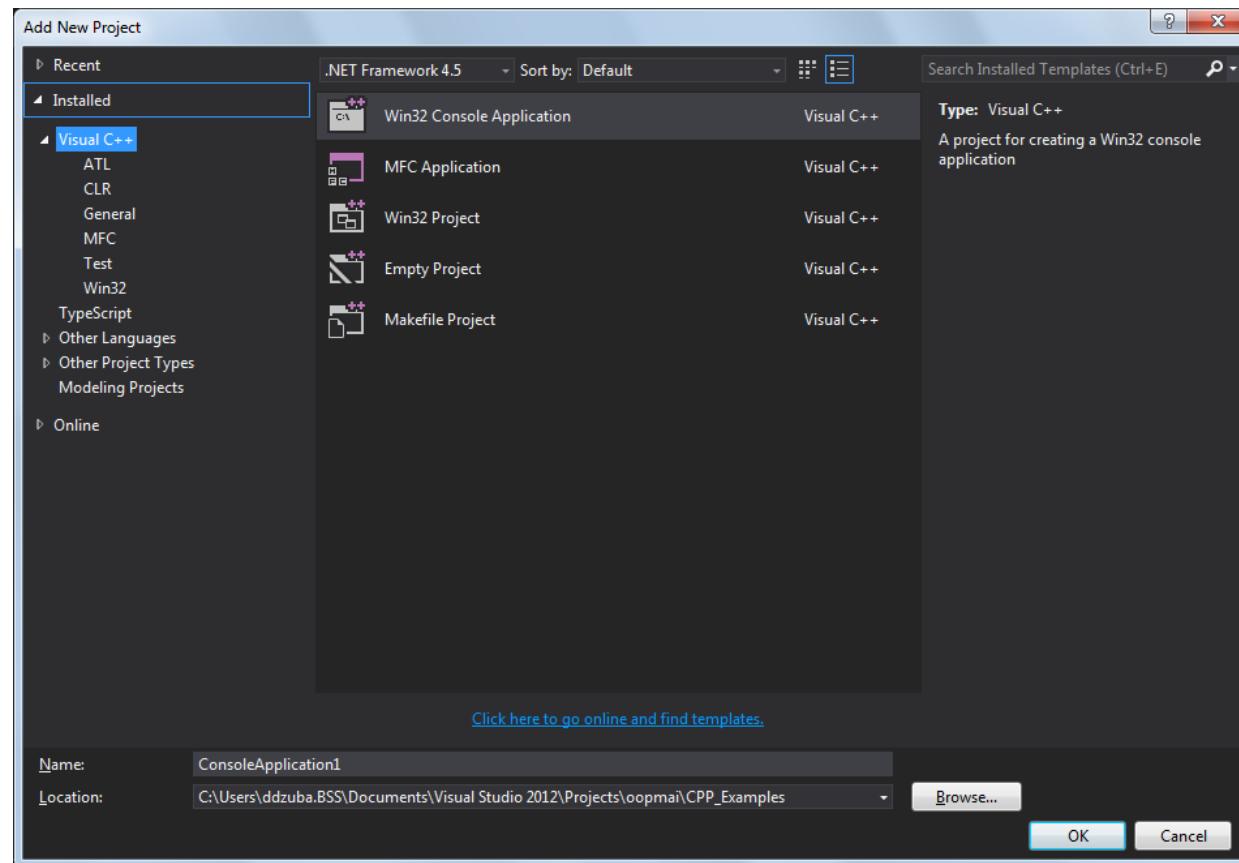
Допускается использование следующих сред разработки/компиляторов:

- Microsoft Visual Studio 2013/15/17 для MS Windows 7/8.1/10
- X-Code (clang) для MacOS X 10.x
- gcc для Linux (например, Ubuntu).

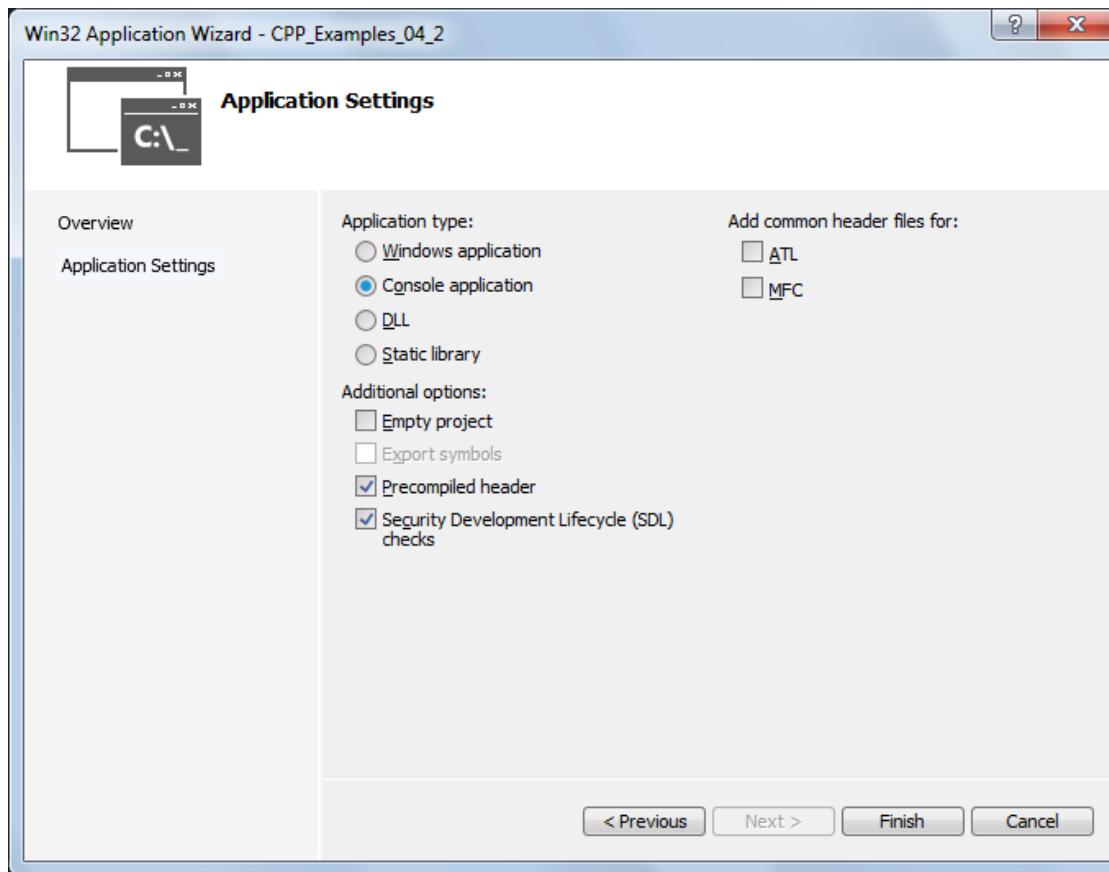
Допускается использование других компиляторов C++ поддерживающих стандарт C++ 11 и выше.



Создание проекта на C++ в Microsoft Visual Studio 2013 [1/2]



Создание проекта на C++ в Microsoft Visual Studio 2013 [2/2]



Общие понятия

ЛЕКЦИЯ №1



Семейство языков С

C

1972 , Dennis Ritchie @ Bell Labs.

Императивный язык программирования

C++

1979 , Bjarne Stroustrup @ Bell Labs.

Императивный, объектно-ориентированный язык программирования



Язык С

1. Компилируемый.
2. Императивный.
3. Ручное управление памятью.
4. Используется когда нужно написать программу, которая:
 - Эффективна по скорости работы.
 - Эффективна по потребляемой памяти.



Вспоминаем С



Вспоминаем С локальные и глобальные переменные

```
int x;  
  
int y, z;  
  
x= 1;  
  
/* У функции могут быть локальные переменные */  
  
void foo() {  
  
    int x;  
  
    x= 2;  
  
}  
  
/* Параметры имеют локальную область видимости */  
  
void bar(int x) {  
  
    x= 3;  
  
}
```



Вспоминаем С УСЛОВИЯ

```
int foo( int x) {  
    /* Используются обычные булевые операторы . */  
    if (3 ==x) {  
        return 0;  
    } }  
    /* Условия используют целый тип, 1 - это истина. */  
int bar()  
{  
    if (1) {return 0; }  
}
```



Вспоминаем С ЦИКЛЫ

```
void foo() /* Цикл в стиле for */  
  
int i;  
  
for (i=1; i < 10; ++i) {  
  
    printf ("%d\n", i); } }  
  
  
void bar() /* Цикл в стиле while */  
  
int lcv =0;  
  
while (lcv < 10) {  
  
    printf ("%d\n", lcv++);  
  
} }
```



Вспоминаем С вызовы

```
/* Declaration . */
void print sum(int , int );
/* Each executable needs to have a main function with type int . */
int main() {
    print_sum(3, 4);
    return 0;
}
/* Definition . */
void print_sum( int arg1 , int arg2)
{ /* Body defined here . */ }
```



Вспоминаем С МОДУЛИ

1. Файлы определений Header (обычно имеют расширение .h).
Говорят компилятору, что функции где-то определены.
2. Файлы с описанием функций (обычно имеют расширение .c).
Содержат текст описания алгоритмов функций.
3. Возможность «включить» Header в файл с описанием функций:
 - `#include <stdio.h>` //подключаем файл из стандартной библиотеки
 - `#include "myfile.h"` //подключаем локальный файл



Итого: С

1. Мы можем писать большие алгоритмы.
2. Мы можем организовывать программный код в
большое количество библиотек и модулей.

О работе с памятью в С



Ручное управление памятью в С

Цели процесса

Позволить программе помечать области памяти как «занятые» полезной информацией.

Позволить программы помечать области памяти как «не занятые» по окончании работы.
Что бы эти области могли использовать другие алгоритмы и программы.

Что есть в С

В библиотеке stdlib.h есть функции malloc и free.



Управление памятью: Куча (heap)

1. Куча – это область памяти, который может использовать программа.
2. Кучу можно сравнить с гигантским массивом.
3. Для работы с кучей используется специальный синтаксис указателей.
4. Вся программа может получить доступ к куче.

Addr.	Contents
:	:
0xbeef	0xbeef
0xbf4	0xfeed
:	:

Пример работы с кучей в С

Example01_MemoryC

```
#include "stdlib.h" // работа с памятью
#include "stdio.h" // работа с вводом и выводом
#include "string.h" // работа со строками

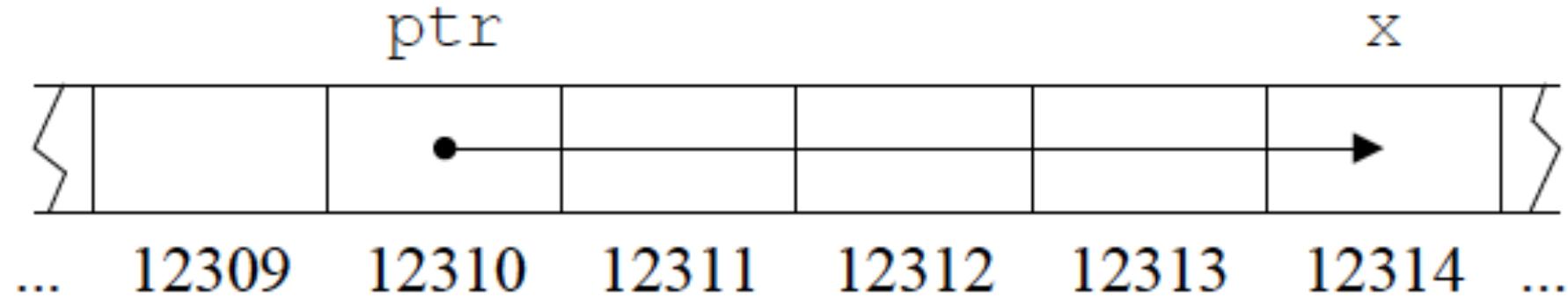
int main()
{
    // выделяем память
    char *pointer = (char*) malloc(sizeof(char)*100);
    // копируем в память данные
    strcpy(pointer, "Hello World!");
    // получаем данные из памяти
    printf("%s", pointer);
    // освобождаем указатель
    free(pointer);
    // ждем нажатия любой клавиши
    getchar();
    return 0;
}
```



Указатель – это число

Указатель хранит адрес переменной!

```
int x = 5;  
int * ptr = &x;
```



Что считает функция?

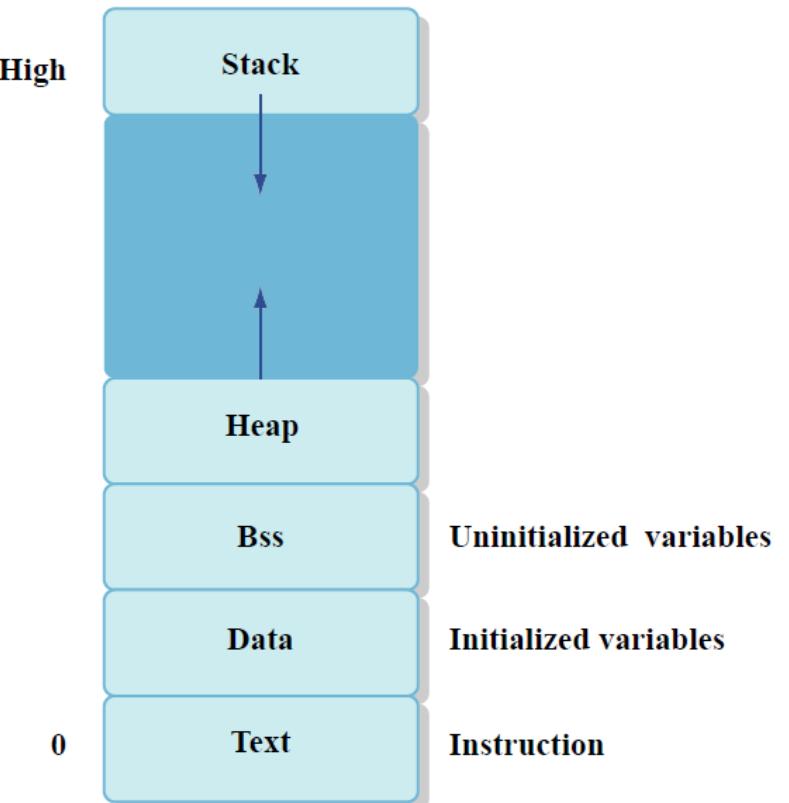
```
void squareByPtr ( int * numPtr ) {  
    *numPtr = *numPtr * *numPtr ;  
}  
  
int main () {  
    int x = 5;  
  
    squareByPtr (& x);  
  
    cout << x; // Prints 25  
}
```



Выделение памяти в стеке (stack)

Функции С размещаются в стеке:

1. Функции помещаются в стек, в момент вызова.
2. Функции удаляются из стека в момент когда вызывается return.
3. Функция может использовать любую память в пределах стека.



Переменные «на стеке»

Example02_StackFault

```
1. #include "stdlib.h" // работа с памятью
2. #include "stdio.h" // работа с вводом и выводом
3. #include "string.h" // работа со строками
4. int a[10];
5. int *array_func(int val) {
6. //    int a[10];
7. for (int i = 0; i < 10; i++) a[i] = val;
8. return a;
9. }
10.int main(int argc, char** argv) {
11.int *array = array_func(3);
12.for (int i = 0; i < 10; i++)      printf("Array is %d\n", *(array+i));
13.return 0;
14.}
```



Основные понятия ООП

Объект

«Объект представляет собой конкретный опознаваемый предмет, единицу или сущность (реальную или абстрактную), имеющую четко определенное функциональное назначение в данной предметной области»

Smith, M. and Tockey, S. 1988. An Integrated Approach to Software Requirements Definition Using Objects. Seattle, WA: Boeing Commercial Airplane Support Division, p.132.



Свойства объекта

1. Состояние

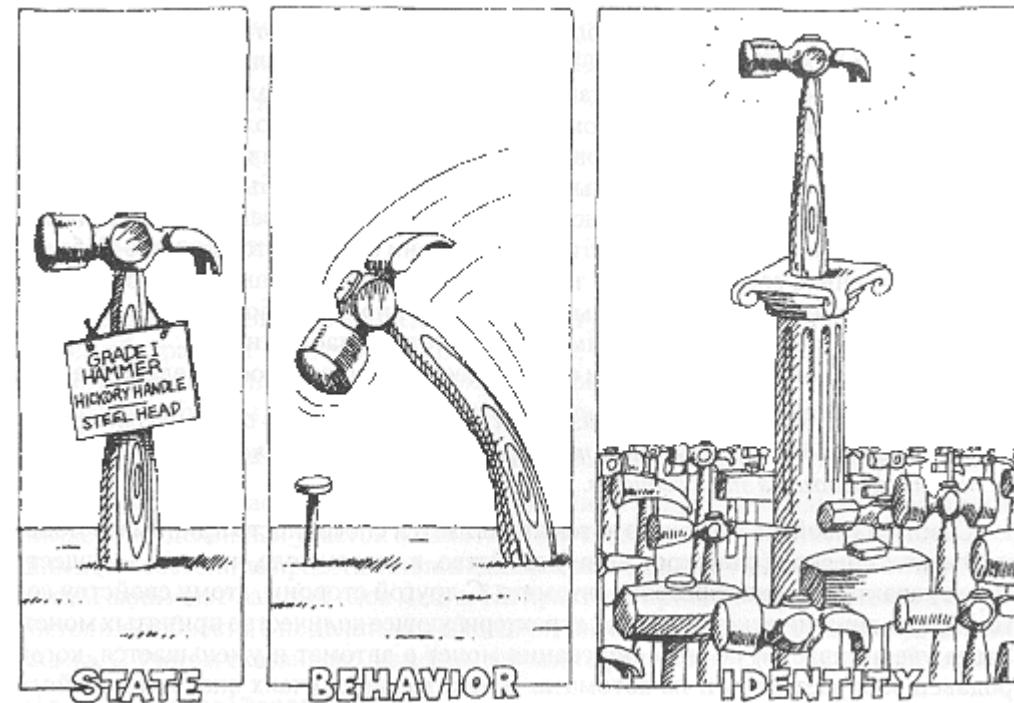
в любой момент времени объект находится в каком-либо состоянии, которое можно измерить / сравнить / скопировать

2. Поведение

объект может реагировать на внешние события либо меняя свое состояние, либо создавая новые события

3. Идентификация

объект всегда можно отличить от другого объекта



Класс

1. Определение.

Классом будем называть группу объектов, с общей структурой и поведением.

2. Смысл программы на C++ это описание классов!

3. Даже если нужен всего один объект – мы будем описывать класс.



Очень простой класс объектов

Example03_FirstClass

```
class MyClass
{
    public:
        int Number;
        void doSomething();
};
```

class – ключевое слово

public – область видимости атрибутов и методов класса

int Number – атрибут класса

void doSomething() - метод класса

Как работать с вводом/выводом в C++?

<http://www.cplusplus.com/reference/iostream/>

Механизм для ввода-вывода в Си++ называется потоком . Название произошло от того, что информация вводится и выводится в виде потока байтов – символ за символом.

- Класс **istream** реализует поток ввода,
- Класс **ostream** – поток вывода.

Библиотека потоков ввода-вывода определяет три глобальных объекта: cout, cin и cerr.

- **cout** называется стандартным выводом,
- **cin** – стандартным вводом,
- **cerr** – стандартным потоком сообщений об ошибках.

cout и **cerr** выводят на терминал и принадлежат к классу **ostream**, **cin** имеет тип **istream** и вводит с терминала. Разница между **cout** и **cerr** существенна в **Unix** – они используют разные дескрипторы для вывода. В других системах они существуют больше для совместимости.

Вывод осуществляется с помощью операции <<, ввод с помощью операции >>.

```
int x; cin >> x; // ввод числа X
```



Пример по работе с потоками

Example04_Stream

```
1. #include <cstdlib>
2. #include <iostream>
3. #include <string>
4. #include <fstream>
5. int main(int argc, char** argv) {
6.     std::string file_name;
7.     std::string file_text;
8.     std::cout << "Please enter file name:";
9.     std::cin >> file_name;
10.    std::ofstream out_file(file_name, std::ofstream::out);
11.    std::cout << "Please enter file text:";
12.    while (std::cin >> file_text) out_file << file_text << std::endl;
13.    out_file.close();
14.    std::cout << "Result:" << std::endl;
15.    std::ifstream in_file(file_name);
16.    while (in_file >> file_text) std::cout << file_text << std::endl;
17.    in_file.close();
18.    return 0;
19.}
```



Перегрузка операций

Почему операция `std::cin >> file_text` имеет смысл?

В C++ существуют механизмы, которые позволяют сопоставлять арифметический и другие операции, такие как побитовый сдвиг обычным функциям!

Это позволяет лучше описывать типы. Мы можем описать не просто класс, но и операции с объектами этого класса.

Как это работает мы узнаем немного позже.



Namespace

Example05_Namespace

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

3. namespace MyNameSpace{
4.     int value = 0;
5. }

6. int value = 0;
7. int main(int argc, char** argv) {
8.     MyNameSpace::value = 7;
9.     std::cout << value << std::endl;
10.    std::cout << MyNameSpace::value << std::endl;
11.
12.    using namespace MyNameSpace;
13.    //std::cout << value << std::endl;
14.    return 0;
15.}
```



Объектно-ориентированный язык

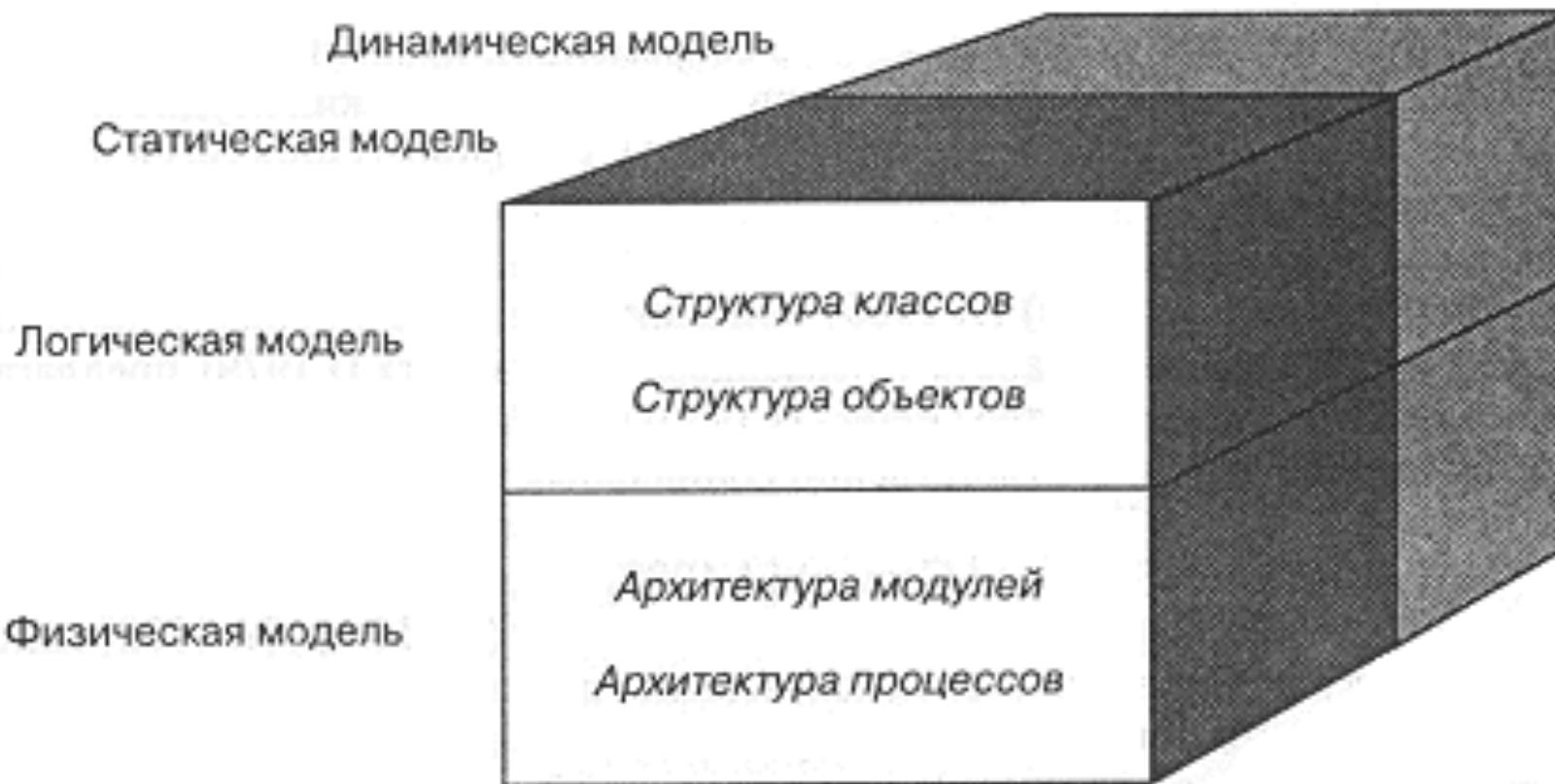
Язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

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

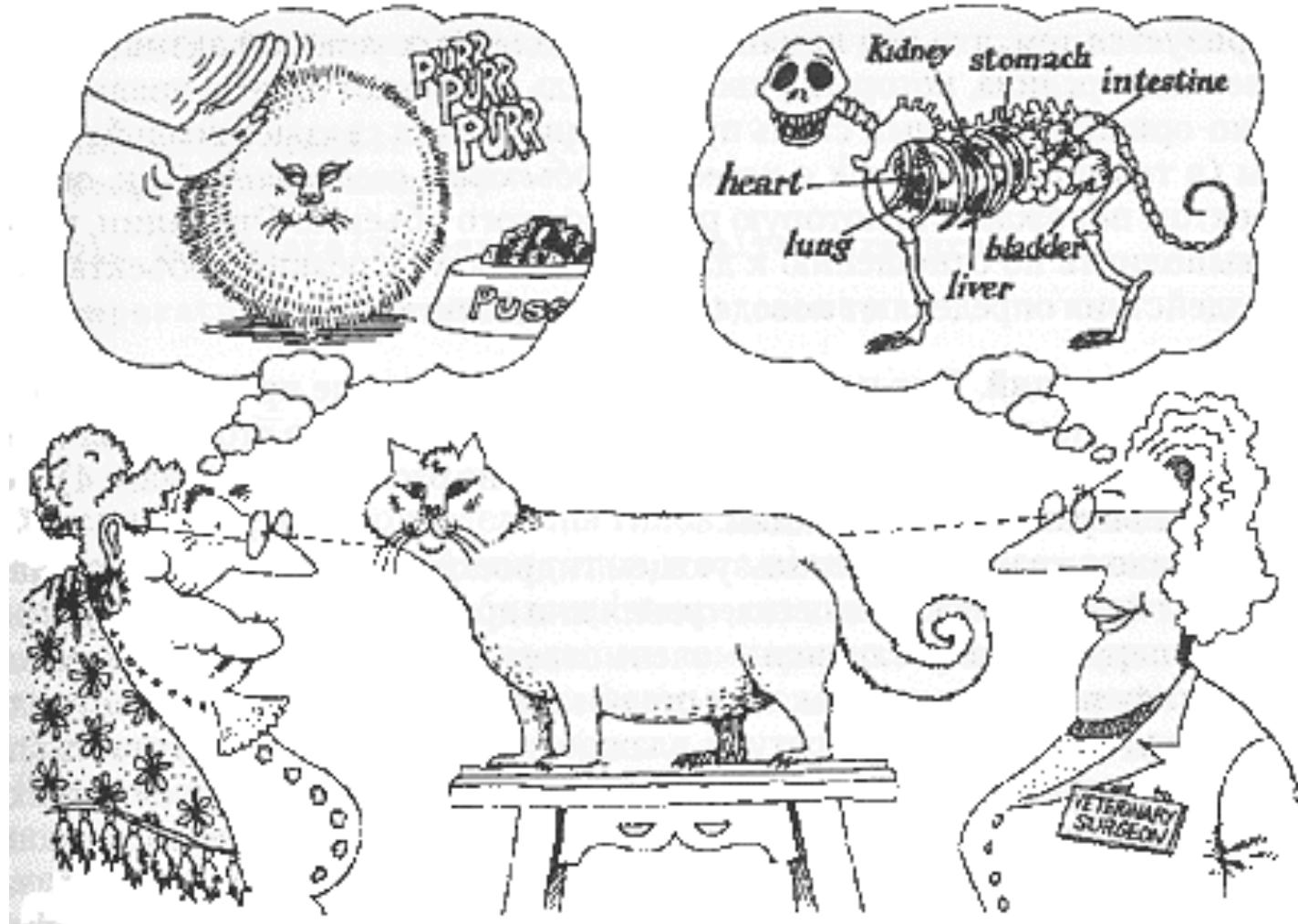
Объект состоит из обобщенного множества операций

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



Пример – абстракция сущности

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

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

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.

Жизненный цикл данных

Example07_Life

```
class Life{
public:

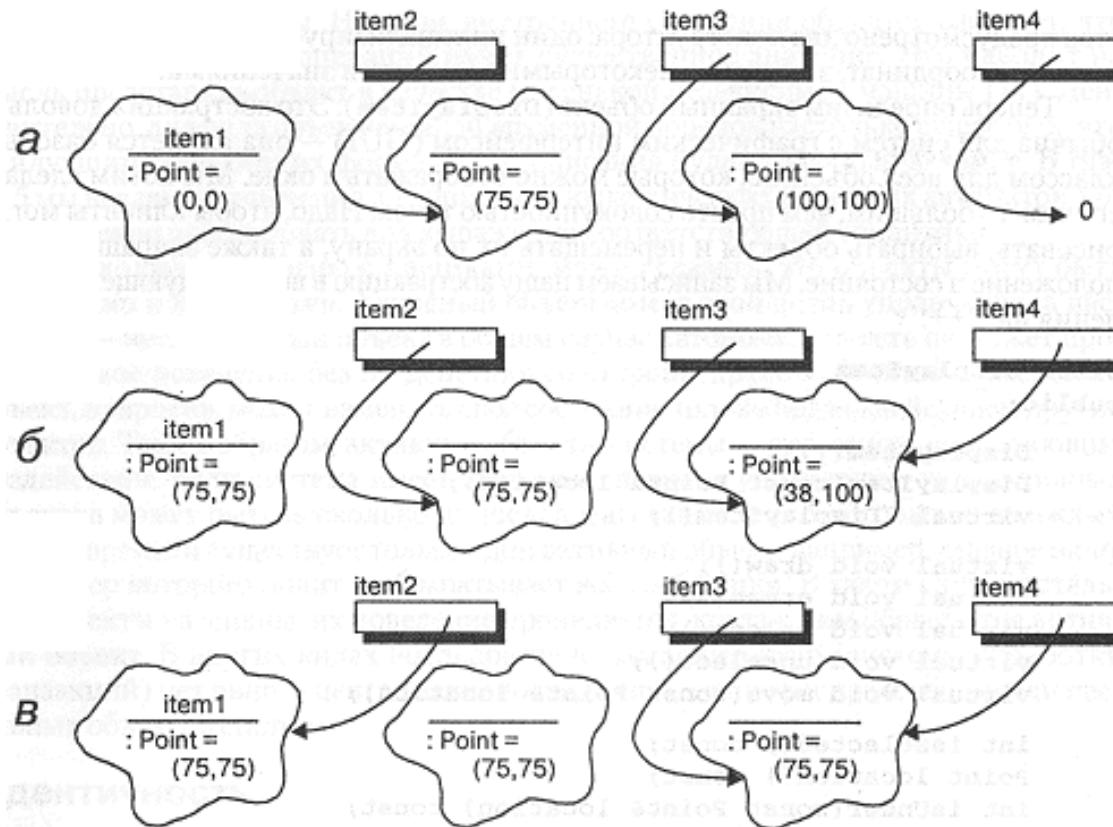
    // Конструкторы
    Life() { std::cout << "I'm alive" << std::endl; }
    Life(const char* n) : Life() { name=n; std::cout<< "My name is " << name << std::endl; }

    // 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.}
```



Сколько раз вызовется конструктор?

```
class Integer {  
public:  
int val;  
Integer() {  
    val = 0;  
    cout << "default constructor" << endl;  
}  
};  
  
int main() {  
    Integer arr[3];  
}
```



Lvalue & Rvalue переменные

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

```
int l; // создать переменную по адресу, например 0x10000
```

```
l = 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 ссылки

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

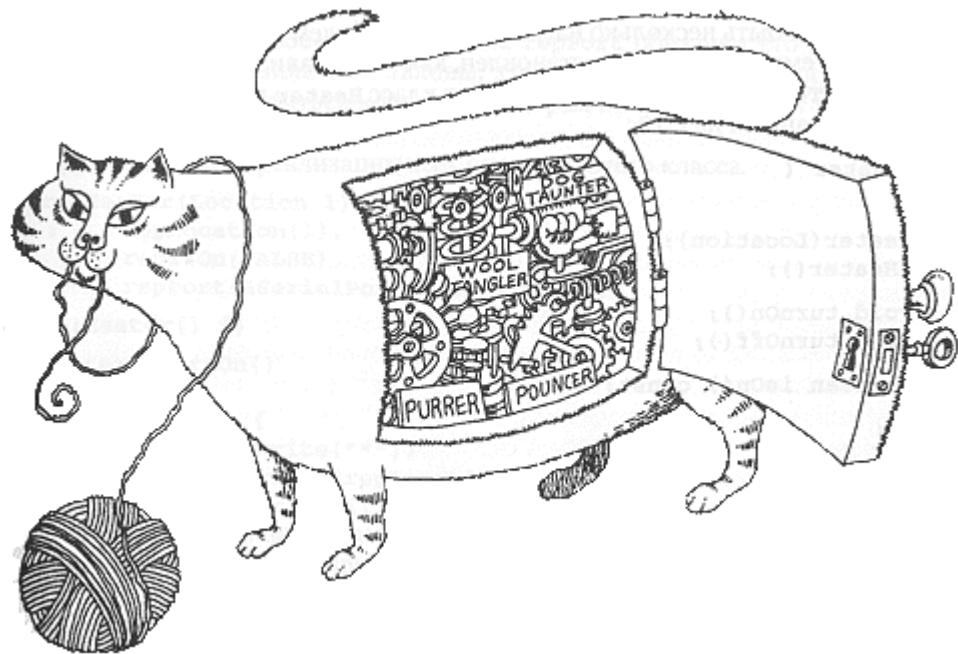
```
1.#include <cstdlib>
2.#include <iostream>

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; // error? 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;
```

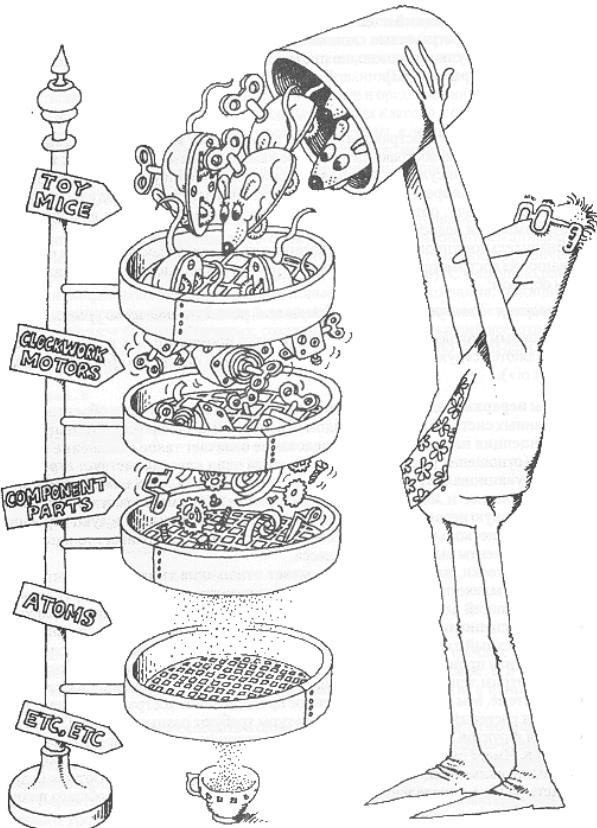
Модули в C++ [2/2]

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



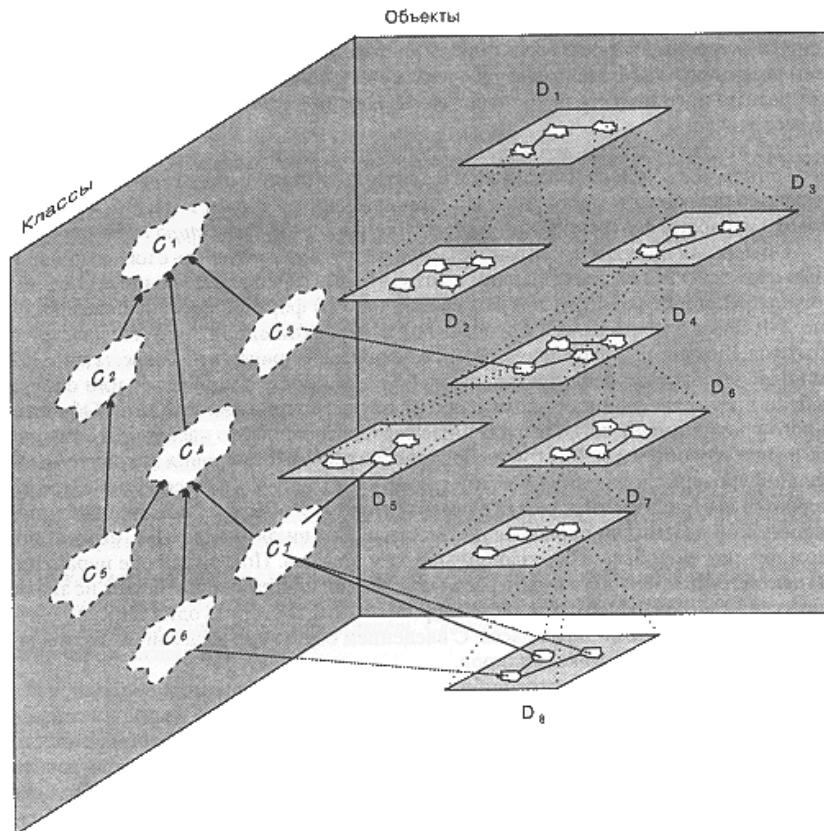
Иерархия - это упорядочение абстракций, расположение их по уровням.

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

Примеры:

1. агрегация
2. наследование

Классы и объекты



Классы связаны между собой
иерархией наследования.

Объекты связаны 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;
};
```

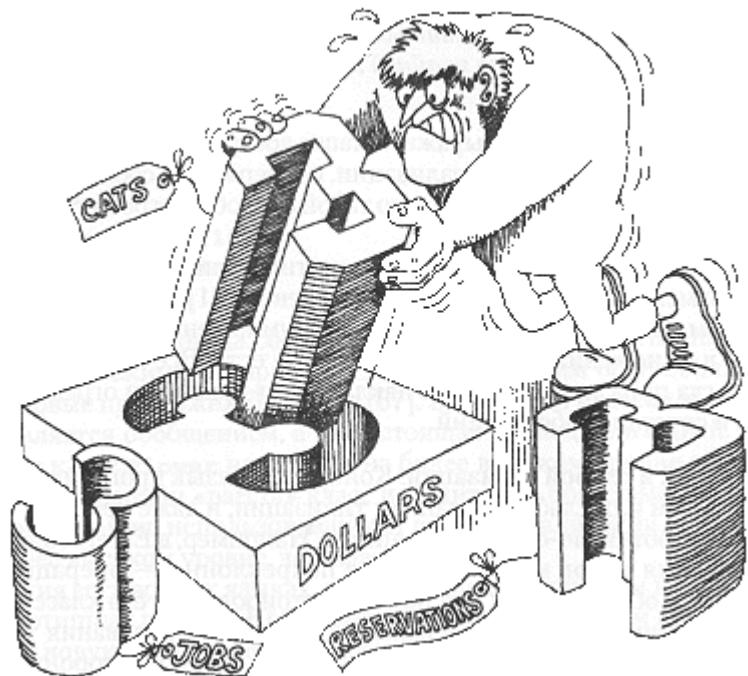
Абстрактный класс нельзя создать!

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:

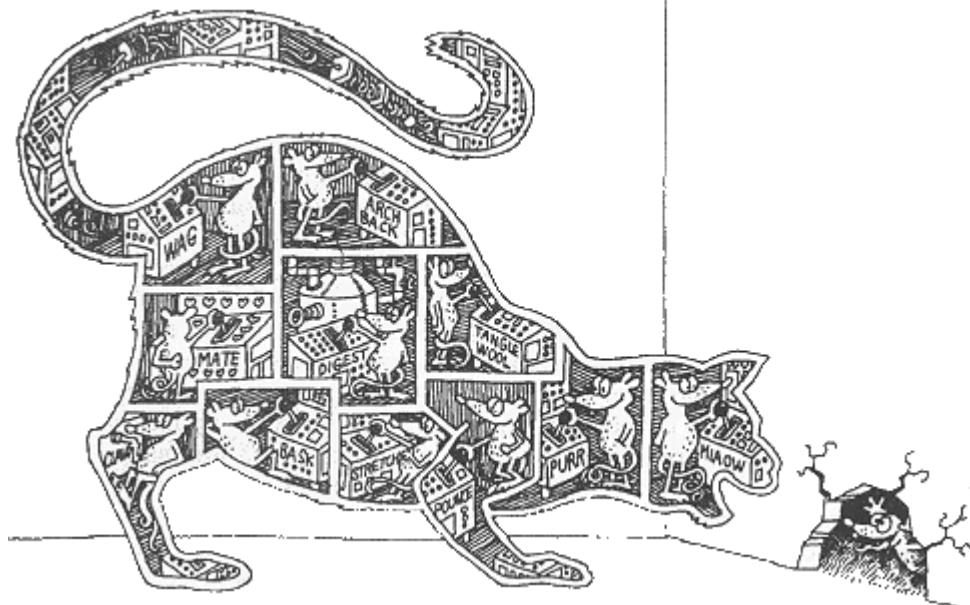
```
s1 x;  
s2 y = x; // ошибка: несоответствие типов
```

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
