Основы ООП

Классы и объекты

Основные понятия ООП: абстракция, инкапсуляция, наследование, полиморфизм

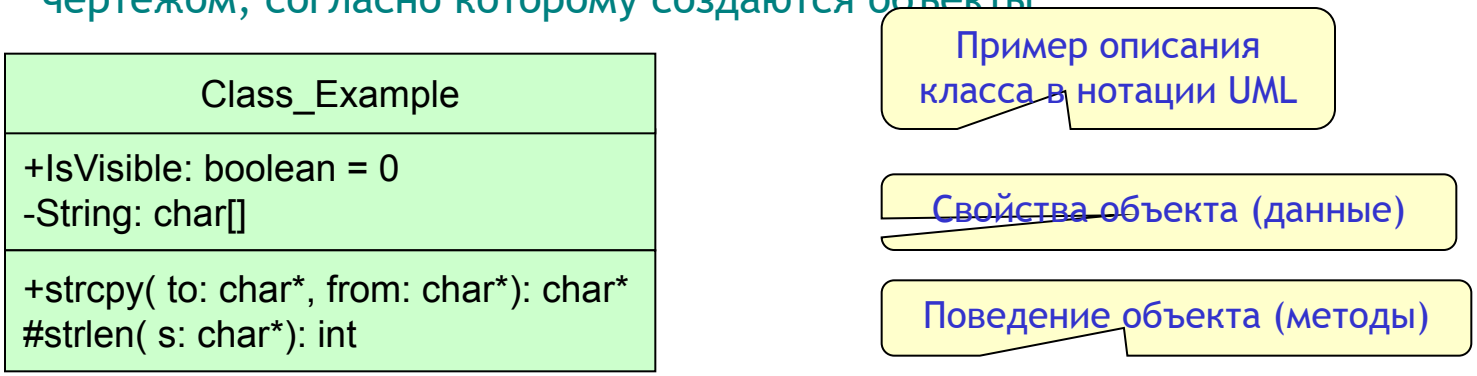
Объе́ктно-ориенти́рованное программи́рование (ООП) – это технология проектирования и разработки программного обеспечения (ПО), в которой основными концепциями являются понятия класса и объекта.

Объект есть некая обособленная сущность в памяти компьютера, которая обладает свойствами и поведением. Свойства объекта задаются данными объекта. Поведение объекта задается методами объекта.

Обычно объект объединяет в себе данные и код для их корректной обработки. Часто объект представляет собой упрощенную, идеализированную модель некоторой реальной или абстрактной сущности предметной области.

Для начала, объект можно понимать просто как поименованную область памяти компьютера, которая обладает структурой, свойствами и поведением.

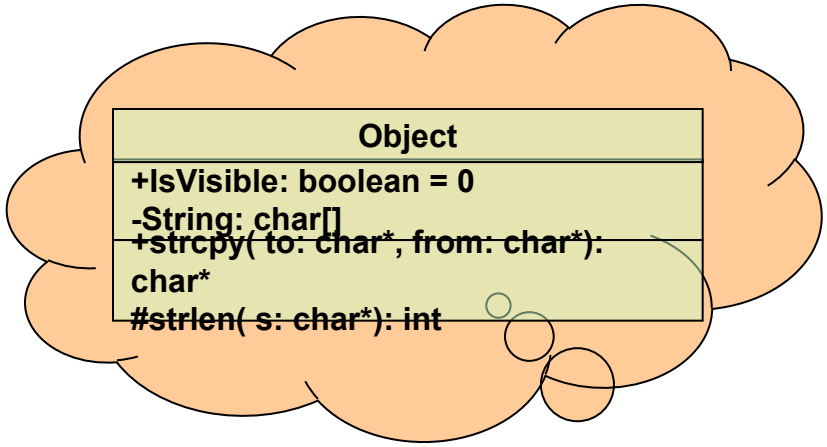
Класс есть обобщение понятия типа. Класс - это тип, который описывает устройство объектов, их данные и методы. Класс можно сравнить с чертежом, согласно которому создаются объекты.



Классы и объекты

В свою очередь объект - это экземпляр класса. Обычно классы разрабатывают таким образом, чтобы объекты классов соответствовали некоторым объектам предметной области.

По мере эволюции компьютеров, с 60-х годов XX века, развивались различные подходы (парадигмы) проектирования и разработки ПО. Устоявшейся классификации нет. Наиболее распространены:



Структурное проектирование ПО (с 70-х годов XX века);

Информационное моделирование предметной области (развитие баз данных, с 80-х годов XX века);

Объектно-ориентированная парадигма (распространяется с 90-х годов XX века).

В основе структурного проектирования лежит идея последовательной декомпозиции предметной области и целенаправленное структурирование кода. Типичными методами структурного проектирования являются:

- нисходящее проектирование (сверху вниз, HIPO-технология);
- модульное программирование, в том числе снизу вверх;

Информационное моделирование предметной области – «код обслуживает данные», СУБД.

ООП - современная технология создания надежного ПО при умеренной его стоимости. ООП есть структурное программирование, доведенное до логического конца. Преимущества ООП проявляются в крупных проектах.

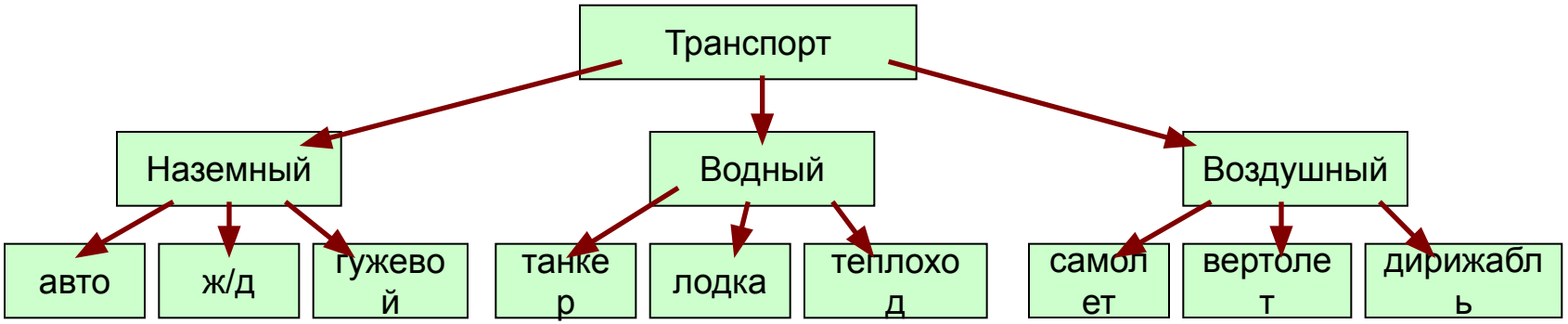
Классы и объекты

ООП основано на четырех базовых понятиях (идеях, принципах): абстракции, инкапсуляции, наследовании и полиморфизме. Рассмотрим их.

Абстракция.

Люди пользуются абстракциями в повседневной практике и в языке.

В интеллектуальной деятельности человечество для своего удобства использует такие приёмы, как классификация, обобщение и абстрагирование (в философии – анализ и синтез). Рассмотрим простой пример из техники:



Компьютеру же требуется каждый раз до деталей расписывать, как устроен каждый новый объект. Это утомительно и не продуктивно. Компьютер «мыслит» всегда конкретно. Нельзя ли «научить» его «мыслить» абстрактными категориями? Это делается в ООП при помощи абстрактных классов объектов.

Сначала это может показаться странным, но особенно полезны те классы, которые являются моделями абстрактных, а не реальных объектов. Идея абстракции – первая фундаментальная идея, лежащая в основе ООП.

Классы и объекты

Инкапсуляция.

ООП исходит из того, что всем объектам присущи некоторые свойства и некоторое поведение. Свойства объекта задаются данными объекта. Поведение объекта задается методами объекта. Процесс объединения в одно целое данных объекта и методов объекта называется инкапсуляций. При этом часть данных и методов может быть защищена (скрыта) внутри объекта.

Пример: автомобиль. Свойства: технические характеристики, текущие координаты, запас бензина. Поведение: движение в разных направлениях с разной скоростью. В языке C++ инкапсуляция достигается путем использования классов, например так:

```
class date {              // класс «дата»
    int day, month, year;   // день, месяц и год (данные скрыты)
public:                   // открытая (интерфейсная) часть класса
    void SetDate ( int, int, int );        // установить дату
    void GetDate ( int&, int&, int& );  // получить дату
    void NextDate ( int&, int&, int& );// получить следующую дату
    void PrintDate ( );              // напечатать дату
} ;
```

Данные класса здесь заданы с помощью трех элементов данных типа int.

Методы класса здесь заданы с помощью четырех функций-членов, которые обеспечивают доступ к данным и манипуляции с этими данными.

Итак, класс описывает данные и код для их корректной обработки.

Классы и объекты

Наследование.

Абстрактные классы полезны тем, что из них можно порождать более конкретные производные классы. Это удобно, так как избавляет разработчика ПО от необходимости каждый раз повторяться в деталях.

Производные классы могут наследовать данные и методы от ранее определенных базовых классов. При этом возможно перекрытие и добавление новых методов и данных базовых классов.

Возникает естественная иерархия (схема наследования) классов. Наверху схемы находятся максимально абстрактные классы, внизу – конкретные классы.

Итак, наследованием называется процесс порождения одного класса от другого класса с сохранением и/или перекрытием свойств и методов класса-предка и добавлением, при необходимости, новых свойств и методов в классы-потомки.

Набор классов, связанных отношением наследования, называют иерархией классов.

Наследование отображает такое свойство реального мира, как иерархичность.

Удобный набор абстракций позволяет с легкостью порождать конкретные практически полезные классы, а не создавать классы каждый раз заново.

Этим достигается надежность и эффективность при проектировании ПО, ускоряется и удешевляется процесс разработка ПО.

В качестве примера схемы наследования см. библиотеку классов MFC.

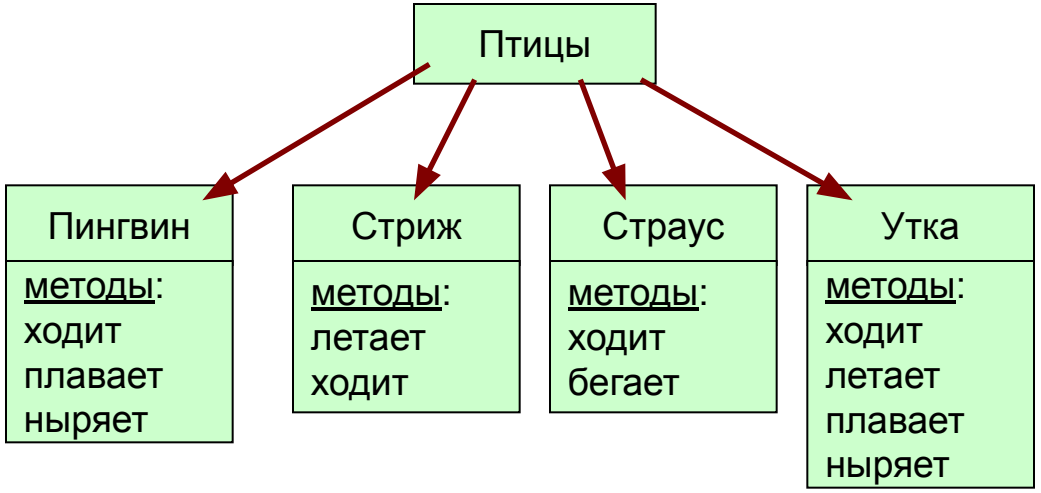
Классы и объекты

Полиморфизм.

Понятие полиморфизма в биологии восходит к Дарвину и означает изменчивость видов, в том числе за счет естественного отбора и мутаций. Полиморфизм в природе обеспечивает многообразие животного мира и растений. В ООП полиморфизмом называют явление, при котором один и тот же код может выполняться по-разному в зависимости от того, для какого объекта он выполняется. Производные классы (потомки) видоизменяют и дополняют свойства и методы базовых классов (предков). При этом сигнатура методов предков не должна изменяться. Интерфейс класса-предка сохраняется и может только дополняться у потомков. Поведение потомков (конкретных объектов) обычно богаче, чем у абстрактных предков. Более того, один и тот же объект может иметь много предков и много методов с одним и тем же именем (перегруженные функции).

Это дает большую гибкость в программировании и обеспечивает эффективность кода.

В C++ полиморфизм реализуется при помощи виртуальных функций и механизма позднего связывания.



Альтернативное определение ООП

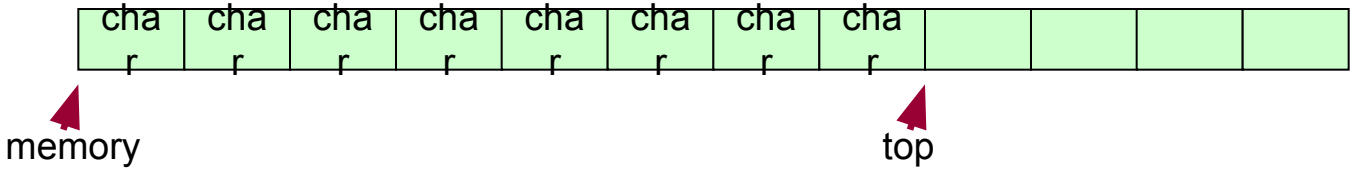
По мнению Алана Кея, создателя языка Smalltalk, одного из основателей ООП, объектно-ориентированный подход основан на следующих шести принципах:

1. Всё является объектом.
2. Объекты взаимодействуют, посылая и получая сообщения. Сообщение - это запрос на выполнение действия, дополненный некоторым набором аргументов. В большинстве языков программирования «отправка сообщения» объекту - это просто вызов метода данного объекта.
3. Каждый объект имеет независимую память, которая состоит из других объектов.
4. Каждый объект является представителем (экземпляром) некоторого класса. Класс задаёт общие свойства объектов (данные класса).
5. Класс задаёт поведение объекта (методы класса). Все объекты, которые являются экземплярами одного класса, могут выполнять заданные действия, т.е. обладают одинаковым поведением.
6. Классы организованы в древовидную структуру, называемую иерархией наследования. Данные и методы класса доступны любому другому классу, расположенному ниже в иерархическом дереве.

Пример класса: стек для символов

Пример:

```
class stack {
    unsigned SIZE;          // размер стека
    char *memory;           // память под стек
    char *top;          // вершина стека
public:
    stack ( int size ) { top = memory = new char [SIZE = size]; }    // конструктор
    ~stack() { delete memory; }         // деструктор
    void push ( char c ) { *top++ = c; }       // размещение символа в стеке
    char pop() { return *--top; }          // извлечение символа из стека
} ;
void main (void) {
    stack s1 ( 128 ); s1.push ( 'a' );    // s1 - первый стек
    stack s2 ( 256 ); s2.push( s1.pop() );   // s2 – второй стек
    stack *ps = new stack( 512 );      // стек в динамической памяти
    cout << s2.pop();
}
```



Объекты классов: статические, автоматические, динамические

Пусть имеется такое описание:

```
class  X  {        // описан класс X ( это похоже на описание структуры)
public:            // доступ извне к элементам класса разрешен
     char *data; // элемент данных класса X
     void f( void );    // функция-член класса X
};

X a;          // описан a – объект класса X
static X v[10];  // описан v – статический массив объектов класса X
X *px = new X; // описан px – указатель на объект класса X
```

Доступ извне описания класса к элементам данных и членам класса осуществляется (если он разрешён) с помощью уточненных имен:

- `X::data` - уточненное имя элемента данных data класса X
- `X::f()` - уточненное имя функции-члена f() класса X

Если имеется объект a класса X и указатель px на объект класса X, тогда доступ извне класса осуществляется (если он разрешён) следующим образом:

- `a.data` или `px->data` - доступ к элементу данных извне класса
- `a.f()` или `px->f()` - вызов функции-члена извне класса

Классы и объекты

Доступ изнутри класса к элементам данных и функциям-членам класса возможен по их простым именам. Кроме того, у любой функции-члена класса всегда имеется неявный аргумент с именем this.

Неявный аргумент this есть указатель на тот объект класса, для которого эта функция-член вызывается.

Например, пусть имеется объект a класса X.

Тогда this ⇔ &a, и внутри любой функции-члена класса возможны следующие корректные обращения:

this -> data - доступ к элементу данных из функции-члена класса

this -> f() - вызов функции-члена из другой функции-члена класса

Пример 1. Статические данные и методы классов

```
class S {
    static int a;          // статический элемент данных класса
public:
    static void h ( S* ) { }   // статический метод класса
} ;
void main ( void ) {
    S obj, &objref = obj, *pobj = &obj, v[9]; // объект, ссылка, указатель, массив
    h ( &obj );            // это ошибка!
    obj.h(pobj);           // это правильно
    pobj -> h( &obj );         // это тоже правильно
};
```

Классы и объекты

Пример 2. Объекты, которые самоорганизуются в линейный список

```
class Link {      // разбирая этот пример, сделайте поясняющий рисунок!
    Link *next;   // это для организации связей между объектами
public:
    void insert ( Link*& );          // встраивание объекта в список
    void iterator ( void ( * ) ( void* ) );       // итерирование списка объектов
} ;                          // конец описания класса
// определение методов класса вне класса
void Link :: insert( Link*&first ) { next = first; first = this; }
void Link::iterator( void (*f)(void*) )
       {  Link *p = this; while ( p ) { (*f) (p); p = p -> next;  } }
// вспомогательная функция выдает адреса объектов (она не член класса!)
void address ( void *adr) { printf ( "%p\n", adr ); }

// поработаем с объектами класса Link
void main ( void ) {
Link a, b;                    // создадим пару объектов
Link *list = NULL, *ptr, *q = &a;    // несколько указателей на объекты класса
a.insert ( list );  b.insert ( list );      // сколько и каких списков здесь создано?
b.Iterator ( address );                  // посмотрим на созданный список
for ( int n = 0; n < 5; n++ ) { ptr = new Link; ptr -> insert ( q ); }   // и ещё так …
ptr -> iterator ( address );             // опять посмотрим на список
}
```

Управление доступом к элементам классов

Управление доступом к элементам классов по существу позволяет управлять степенью инкапсуляции в классе. Сокрытие данных и методов защищает объекты от несанкционированного доступа и от непреднамеренной порчи.

Рассмотрим значения следующих ключевых слов:

Private – элементы класса доступны функциям-членам этого класса и дружественным функциям. Private есть значение по умолчанию.

Protected – элементы класса доступны методам этого класса, методам производных от него классов и дружественным функциям.

Public – элементы класса доступны любым функциям текущего проекта.

Friend – открывает доступ к элементам класса для функций, не являющихся методами данного класса (т.е. объявляет дружественные функции), а также используется для объявления дружественных классов.

```
class Point {                    // это пример сокрытия данных
protected:
     int X, Y;                   // координаты точки
public:
     Point ( int a, int b ) { X = a; Y = b; }     // конструктор объекта
     int GetX ( )  { return X; }                  // получить координату X
     int GetY ( )  { return Y; }                  // получить координату Y
} ;
void main ( void ) { Point Point ( 10, 20 ); }
```

Классы и объекты

Каким образом регулируются права доступа к элементам классов при наследовании классов? Рассмотрим следующую синтаксическую конструкцию:

class Derived : <модификатор-доступа> Base { …… };

Здесь Base – базовый класс (предок), а Derived – производный класс (потомок).

Таблица определения прав доступа при наследовании

Base	<модификатор-доступа>	Derived
private	private	не доступны
protected	private	private
public	private	private
private	public	не доступны
protected	public	protected
public	public	public

Шаблоны функций и шаблоны классов

Шаблоны функций и шаблоны классов есть дальнейшее развитие идеи макросов и перегруженных функций. Рассмотрим их использование на примерах.

Пример.

Шаблон функции для обмена значений объектов произвольных классов

```
#include <stdio.h>
// опишем шаблон функции swap_
template <class T> void swap_ ( T &a, T &b ) { T temp = a; a = b; b = temp; }

// протестируем работу шаблона функции swap_
void main ( void ) {
        int n = 0, m = 1;
        float x = 0., y = 1.;
        char *s1 = "one", *s2 = "two";
        swap_( n, m );  // обменяем значения объектов типа int
        swap_( x, y );   // обменяем значения объектов типа float
        swap_( s1, s2 );       // обменяем значения объектов типа char*
        printf("%i %f %s\n",n,x,s1);      // посмотрим результаты работы
}                               // шаблона функции swap_
```

Классы и объекты

Пример. Шаблон стека для указателей на объекты произвольных классов

```
#include <stdio.h>   // разбирая этот пример, сделайте поясняющий рисунок!
// опишем шаблон класса stack_
template <class T> class stack_ {
        unsigned SIZE;        // размер стека
        T **base, **top;      // указатели на начало и на вершину стека
public:
        stack_(unsigned n) { base=top=new T*[SIZE=n]; }  // конструктор
        ~stack_() { delete[] base; }       // деструктор
        void push(T*p) { *top++ = p; }  // размещение указателя в стеке
        T pop(void) { return **--top; }    // извлечение объекта из стека
};
// протестируем работу шаблона класса stack_
void main ( void ) {
      stack_<int> intstack(128);          // стек для объектов типа int
      stack_<char*> stringstack(256);// стек для объектов типа char*
      int n=0, m=1;                       // пара объектов типа int
      intstack.push(&n); intstack.push(&m);       // разместим их в стеке
      char *s1="one", *s2="two";          // пара объектов типа char*
      stringstack.push(&s1); stringstack.push(&s2);   // разместим их в стеке
      printf("%i %i \n",intstack.pop(),intstack.pop());   // посмотрим содержимое
      printf("%s %s \n",stringstack.pop(),stringstack.pop());       // стеков
}
```