

# Виртуальные функции и полиморфизм

# Виртуальные функции

- Функция-член, объявленная в базовом классе и переопределенная в производном
- То есть виртуальный означает видимый, но не существующий в реальности
- Программа, которая, казалось бы, вызывает функцию одного класса, может вызывать функцию совсем другого класса

# деструктры

- Могут объявляться как **virtual**.  
Используя виртуальные деструкторы, можно уничтожать объекты, не зная их тип — правильный деструктор для объекта вызывается с помощью механизма виртуальных функций. Обратите внимание, что для абстрактных классов деструкторы также могут объявляться как чисто виртуальные функции.

# Зачем нужны виртуальные функции

- Пусть имеется набор объектов разных классов
  - Например, есть разные геометрические фигуры: треугольник, шар и квадрат. В каждом классе есть метод `draw()`, который прорисовывает на экране соответствующие фигуры
- Задача: нарисовать картинку, сгруппировав эти элементы, без дополнительных сложностей
- Решение: создать массив указателей на все неповторяющиеся элементы картинки, т.е. указатель на объект шарик, указатель на квадрат и т.п.
- Обращаясь к разным элементам массива можно рисовать разные фигуры

# Причем здесь полиморфизм

- То есть имеем «один интерфейс (функции называются одинаково draw()) и несколько методов (реально вызываются разные функции, рисующие разные фигуры)»
- Это есть – **полиморфизм**
- Перегрузка функций – **статический полиморфизм**
- Наследование и виртуальные функции – **динамический полиморфизм**

# Как создаются виртуальные функции

- В базовом классе перед объявлением виртуальной функции указывается ключевое слово: **virtual**
- В производном классе функция переопределяется – то есть создается конкретная реализация функции
- Для примера рассмотренного выше:
  - Все классы (шар, треугольник, квадрат) должны быть наследниками одного и того же базового класса
  - Функция draw() должна быть объявлена как virtual

# Доступ к обычным методам через

указатели: базовый и производные классы содержат функции с одним и тем же именем, к ним обращаются с помощью указателей, но без виртуальных функций

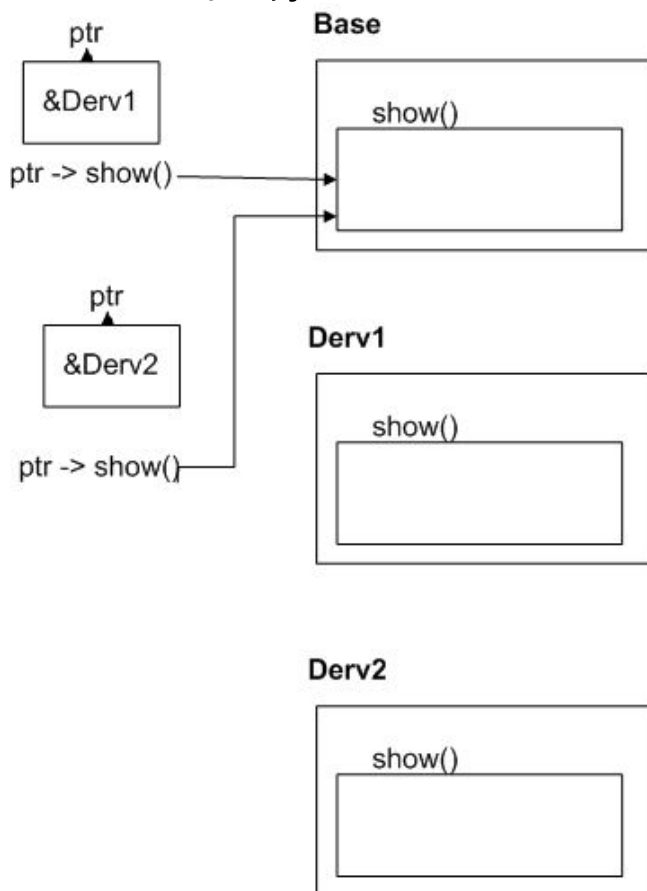
```
class base
{
public:
    void show() { cout << "Base \n"; }
};
Class Derv1:public Base
{
    void show() { cout << "Derv1\n"; }
};
Class Derv2:public Base
{
    void show() { cout << "Derv2\n"; }
};
```

```
void main ()
{
    Derv1 dv1;
    Derv2 dv2;
    Base* ptr;

    ptr = &dv1;
    ptr -> show();

    ptr = &dv2;
    ptr -> show();
}
```

Вывод: ?



# Доступ к виртуальным методам через указатели

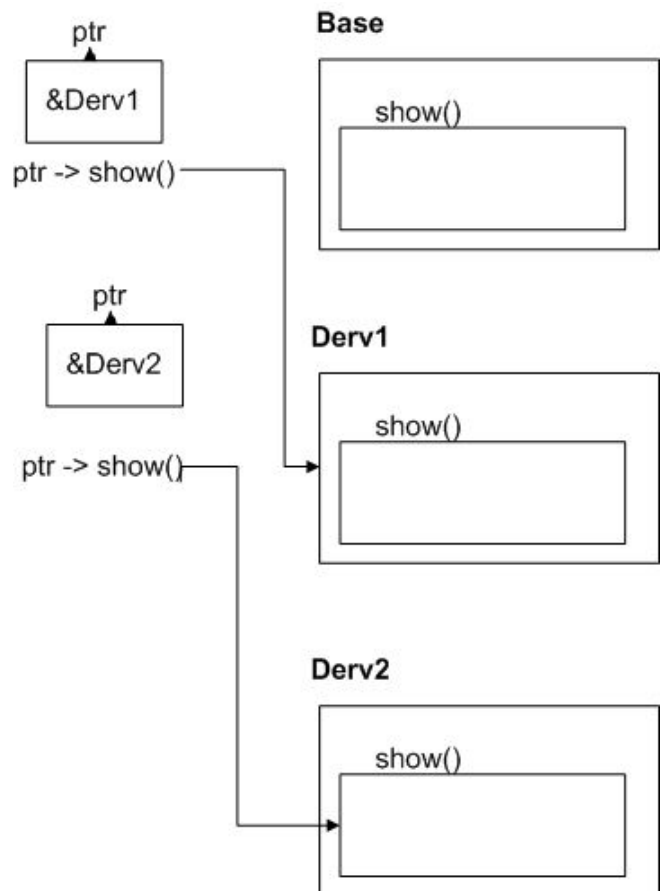
```
class base
{
public:
    virtual void show() { cout << "Base \n"; }
};
Class Derv1:public Base
{
    void show() { cout << "Derv1\n"; }
};
Class Derv2:public Base
{
    void show() { cout << "Derv2\n"; }
};
```

```
void main ()
{
    Derv1 dv1;
    Derv2 dv2;
    Base* ptr;

    ptr = &dv1;
    ptr -> show();

    ptr = &dv2;
    ptr -> show();
}
```

Вывод: ?





# Позднее или динамическое

## СВЯЗЫВАНИЕ

Какая функция компилируется при  
компиляции выражения:

`ptr ->show(); ?`

- Всегда компилируется функция из базового класса
- Однако в последней программе компилятор не знает к какому классу относится содержимое `ptr`.
- Когда программа поставлена на выполнение, когда известно, на что указывает `ptr`, тогда запускается соответствующая версия `show()`.
- Выбор функции во время компиляции называется ранним или статическим связыванием
- Позднее связывание требует больше ресурсов, но дает выигрыш в возможностях и гибкости

# Пример

```
class B{
int a;
public:
B() { };
B(int x) { a=x; }
void show_B() { cout<<"B= " << a <<"\n"; }
virtual void showV_B() { cout<<"virt B= " << a <<"\n"; }
};
```

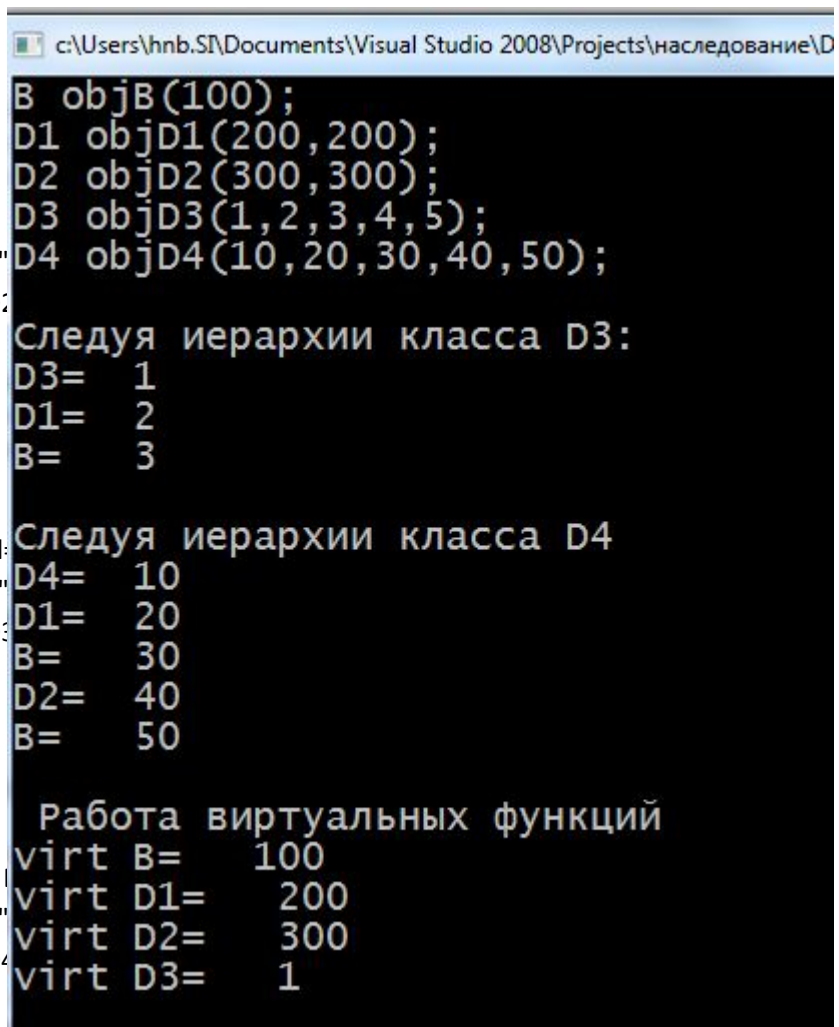
```
class D1: public B {
int b;
public:
D1(int x, int y) : B(y) { b=x;};
void show_D1() { cout<<"D1= " << b <<"\n"; show_B();}
void showV_B() { cout<<"virt D1= " << b <<"\n"; }
};
```

```
class D2: public B{
int c;
public:
D2(int x, int y) : B(y) { c=x;};
void show_D2() { cout<<"D2= " << c <<"\n"; }
void showV_B() { cout<<"virt D2= " << c <<"\n"; }
};
```

```
class D3: public D1 {
int d;
public:
D3(int x, int y, int z) : D1(y,z) { d=x;};
void show_D3() { cout<<"D3= " << d <<"\n"; show_D1();}
void showV_B() { cout<<"virt D3= " << d <<"\n"; }
};
```

```
class D4: public D2, public D1 {
int e;
public:
D4(int x, int y, int z, int i, int j) : D2(y,z), D1(i,j) { e=x;};
void show_D4() { cout<<"D4= " << e <<"\n"; show_D2(); show_D1();}
void showV_B() { cout<<"virt D4= " << e <<"\n"; }
};
```

```
int main() {
```



The screenshot shows a Visual Studio 2008 console window with the following output:

```
c:\Users\hnb.SA\Documents\Visual Studio 2008\Projects\наследование\D
B objB(100);
D1 objD1(200,200);
D2 objD2(300,300);
D3 objD3(1,2,3,4,5);
D4 objD4(10,20,30,40,50);

Следуя иерархии класса D3:
D3= 1
D1= 2
B= 3

Следуя иерархии класса D4
D4= 10
D1= 20
B= 30
D2= 40
B= 50

Работа виртуальных функций
virt B= 100
virt D1= 200
virt D2= 300
virt D3= 1
```

# Чисто виртуальные функции

# Задание

- Создайте программу, реализующую кошелек, используя виртуальные функции

## Помните:

- Виртуальные функции позволяют решать прямо в процессе выполнения программы, какую именно функцию вызывать
- Виртуальные функции дают большую гибкость при выполнении одинаковых действий над разнородными объектами

# Работа в малых группах

Создайте программу имитирующую работу калькулятора, выполняющего четыре арифметических действия над дробями. Например,

Сложение:  $a/b + c/d = (a*d + b*c)/(b*d)$

Вычитание:  $a/b - c/d = (a*d - b*c)/(b*d)$

Умножение:  $a/b * c/d = (a*c)/(b*d)$

Деление:  $a/b / c/d = (a*d)/(b*c)$

Пользователь должен сначала ввести первый операнд, затем знак операции и второй операнд. После вычисления результата программа должна отобразить его на экране и запросить пользователя о его желании произвести еще одну операцию.