

## Лекция 4

НАСЛЕДОВАНИЕ.  
БАЗОВЫЕ И ПРОИЗВОДНЫЕ КЛАССЫ.  
ПРОСТОЕ И МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ.

# Наследование

При большом количестве никак не связанных классов управлять ими становится невозможным. Наследование позволяет справиться с этой проблемой путем объединения общих для нескольких классов свойств в одном классе и использования его в качестве базового.

Механизм наследования классов позволяет строить иерархии, в которых производные классы получают элементы родительских, или базовых, классов и могут дополнять их или изменять их свойства.

Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных черт.

# Наследование

**Наследование** — это механизм создания нового класса на основе уже существующего.

Наследование бывает двух видов:

- простое;
- множественное.

При **простом наследовании** один класс наследует свойства от **одного** класса.

При **множественном наследовании** класс наследует свойства от **двух и более** классов.

Родительский класс – это **базовый** класс,  
**производный** класс – это класс-наследник.

# Наследование

При описании класса в его заголовке перечисляются все классы, являющиеся для него **базовыми**. Возможность обращения к элементам этих классов регулируется с помощью спецификаторов доступа **private**, **protected** и **public**:

```
class <имя> : [private|protected|public] <базовый_класс>
            { <тело класса> };
```

Если базовых классов несколько, они перечисляются через запятую. Ключ доступа может стоять перед каждым классом, например:

```
class A { ... };
class B { ... };
class C { ... };
class D: A, protected B, public C { ... }; //D наследует от A, B, C
```

По умолчанию для классов используется ключ доступа **private**, а для структур – **public**.

# Наследование

Если в определении класса присутствует список базовых классов, то такой класс называется **производным (derived)**, а классы в базовом списке - **базовыми (base) классами**.

Производные классы получают доступ к элементам своих базовых классов, и, кроме того, могут пополняться собственными.

Наследуемые элементы **не перемещаются** в производный класс, а остаются в базовых классах. Если для обработки нужны данные, отсутствующие в производном классе, то их пытаются отыскать автоматически в базовом классе.

При наследовании некоторые имена методов базового класса могут быть по-новому определены в производном классе. В этом случае соответствующие элементы базового класса становятся недоступными из производного класса. Для доступа из производного класса к элементам базового класса, имена которых повторно определены в производном, используется операция **::**.

# Пример простого наследования

```
class A { //базовый класс
    int i;
    public: void f(){ i=10; cout<<i;}
};

class B : public A { //производный класс
    int i;
    public: //переопределение метода
        void f(){ i=20; cout<<i*i;}
};

int main() {
    A ob_a;
    ob_a.f();
    B ob_b;
    ob_b.f();
    ob_b.A::f();
}
```

Результат

10 400 10

А если закомментировать

```
//public: void f(){ i=20; cout<<i*i;}
```

то результат:

10 10 10

# Доступ при наследовании

Производному классу доступны все компоненты базовых классов, как если бы это были его собственные компоненты. Исключение составляют компоненты базового класса с спецификаторами доступа **private**. При определении производного класса можно также влиять на доступ к элементам базовых классов.

Доступ в базовом классе	Спецификатор наследуемого доступа	Доступ в производном классе
public protected private	public	public protected нет доступа
public protected private	protected	protected protected нет доступа
public protected private	private	private private нет доступа

# Доступ при наследовании

Спецификатор наследуемого доступа устанавливает тот уровень доступа, до которого понижается уровень доступа к элементам базового класса.

**Private** элементы базового класса в производном классе недоступны вне зависимости от ключа. Обращение к ним может осуществляться только через методы базового класса. Элементы **protected** при наследовании с ключом **private** становятся в производном классе **private**, в остальных случаях права доступа к ним не изменяются.

Доступ к элементам **public** при наследовании становится соответствующим ключу доступа.

Следует отметить, что **не все** элементы класса будут наследоваться.

**Не подлежат наследованию следующие элементы класса:**

- конструкторы (в том числе и конструкторы копирования);
- деструкторы;
- операторы присваивания, определенные программистом;
- друзья класса.



# Пример простого наследования

```
class A{
    int a1; f1(){cout<<"f1\n";}
protected: int a2; f2(){cout<<"f2\n";}
public: int a3; f3(){cout<<"f3\n";}
    A(){a1=a2=a3=10;cout<<"constr_A\n";}
    ~A(){a1=a2=a3=0;cout<<"destr_A\n";}
};

class B:public A{
    int b1;
    void f4(){cout<<"a2="<<a2<<endl; cout<<"f4\n";}
protected: int b2; f5(){cout<<"f5\n";}
public: int b3; f6(){cout<<"f6\n";}
    B(){b1=b2=b3=10;cout<<"constr_B\n";}
    ~B(){b1=b2=b3=0; cout<<"destr_B\n";}
};
```

```
int main(){
    B ob;
    cout<<"b3="<<ob.b3<<endl; cout<<"a3="<<ob.a3<<endl;
    //ob.f1(); //ob.f2();
    ob.f3();
    //ob.f4(); //ob.f5();
    ob.f6();
}
```

```
constr_A
constr_B
b3=10
a3=10
f3
f6
destr_B
destr_A
```

# Пример простого наследования

// базовый

```
class base{  private: int b1;
              int f1() {b1++; return b1;}
  protected: int b2;
  public: int b3;
              base() {b1=b2=b3=1;}  //...
              void f2() {b1=2; cout<<"b1="<<b1;}
};

class derived : private base{           // производный
  public: void f() {b2++; b3++;}         // к b1 нет доступа
          void f_print() {cout<<"b2="<<b2<<" b3="<<b3<<endl;}
};

int main()
{  base ob;
   //cout<<ob.b1; // cout<<ob.b2; // нет доступа
   cout<<ob.b3<<endl;
   derived ob1;
   // cout<<ob1.b1<<ob1.b2<<ob1.b3; // нет доступа
   ob1.f(); ob1.f_print(); //изменение и печать полей b2, b3
   // ob1.f2();           // нет доступа
}
```

# Пример простого наследования

При любом способе наследования в производном классе доступны только открытые (**public**) и защищенные (**protected**) элементы базового класса; закрытые элементы базового класса остаются закрытыми, независимо от того, как этот класс наследуется.

Однако можно сделать некоторые элементы базового класса открытыми в производном классе, объявив из секции **public** производного класса. Рассмотрим на примере: переопределим класс **derived**.

# Пример простого наследования

```
class derived : private base{
public:  base::f2;           //!!!
void f() {b2++;b3++;}
void f_print() {cout<<"b2="<<b2<<" b3="<<b3<<endl; }
//...
};

// тогда можно обратиться к методу f2()

int main()
{derived ob1;
  // ...
  ob1.f2();
}
```

b1=2

# Пример вызовов конструкторов и деструкторов при простом наследовании

```
#include<iostream>
using namespace std;
class base{
    public: base() {cout<<"constr_base"<<endl;}
           ~base() {cout<<"destr_base"<<endl;}
};

class derived: public base{
    public: derived () {cout<<"constr_derived"<<endl;}
           ~derived() {cout<<"destr_derived"<<endl;}
};

int main(){
    derived ob;
    return 0;
}
```

```
constr_base
constr_derived
destr_derived
destr_base
```

```
// derived *ob=new derived(); delete ob;
```

# Пример вызовов конструкторов и деструкторов при простом наследовании

```
#include<iostream>
using namespace std;
class base{
    public: base() {cout<<"constr_base"<<endl;}
           ~base() {cout<<"destr_base"<<endl;}
};

class derived: public base{
    public: derived(): base() {cout<<"constr_derived"<<endl;}
           ~derived() {cout<<"destr_derived"<<endl;}
};

int main(){
    derived ob;
    return 0;
}

// derived *ob=new derived(); delete ob;
```

ЯВНЫЙ ВЫЗОВ КОНСТРУКТОРА БАЗОВОГО КЛАССА

constr\_base  
constr\_derived  
destr\_derived  
destr\_base

# Инициализация при наследовании

Конструкторы не наследуются, поэтому производный класс либо должен объявить свой конструктор, либо предоставить возможность компилятору сгенерировать конструктор по умолчанию.

Рассмотрим как строится конструктор производного класса, предоставляемый программистом. Поскольку производный класс должен унаследовать все элементы базового класса, при построении своего класса он должен обеспечить инициализацию унаследованных полей, причем она должна быть выполнена до инициализации полей производного класса, так как последние могут использовать значения первых. Для построения конструктора производного класса применяется следующая конструкция:

```
<констр_произ_класса>(<список параметров>) :  
    <констр_базового_класса>(<список_арг>)  
    {<тело_конструктора>}
```

Часть параметров, переданных конструктору производного класса, обычно используется в качестве аргументов конструктора базового класса.

# Пример вызовов конструкторов при наследовании

```
#include<iostream>
using namespace std;

class Coord {          //базовый
    int x, y;
public:
    Coord(int _x, int _y) { // конструктор
        x=_x;
        y=_y;
        cout<<"constr_Coord"<<endl;
    }
    Coord() { // конструктор по умолчанию
        x=0;   y=0;
        cout<<"constr_Coord()"<<endl;
    }
    ~Coord() {cout<<"destr_Coord"<<endl;}
    void show();
};
```



# Пример вызовов конструкторов при наследовании

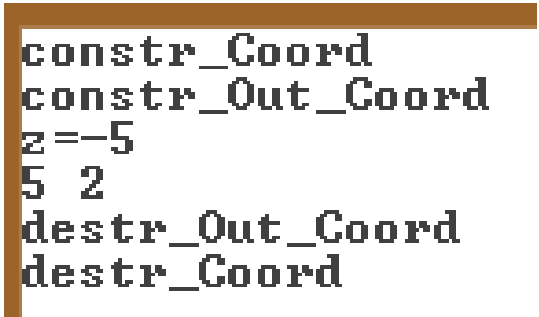
```
class Out_Coord: public Coord {    //производный
    int z;
public:
    //конструктор
    Out_Coord(int _x, int _y, int _z) : Coord(_x, _y) {
        z=_z;  cout<<"constr_Out_Coord"<<endl;    }
    ~Out_Coord(){ cout<<"destr_Out_Coord"<<endl;}
    void show();
};

void Coord::show() {
    cout<<x<<" "<<y<<endl;}

void Out_Coord::show() {
    cout<<"z="<<z<<endl;}

int main() {
    Out_Coord ob(5, 2, -5);
    ob.show();
    ob.Coord::show();

    /*Out_Coord* ptr;
    ptr=new Out_Coord(10, 20, 30);
    ptr->show();
    ptr->Coord::show();
    delete ptr;*/
}
```



The diagram illustrates the sequence of constructor and destructor calls during program execution:

- constr\_Coord
- constr\_Out\_Coord
- z=-5
- 5 2
- destr\_Out\_Coord
- destr\_Coord

# Пример вызовов конструкторов при наследовании

Если предоставляемый программистом конструктор не имеет параметров, т.е. является конструктором по умолчанию, то при создании экземпляра производного класса автоматически вызывается конструктор базового класса. После того как объект создан, конструктор базового класса становится недоступным.

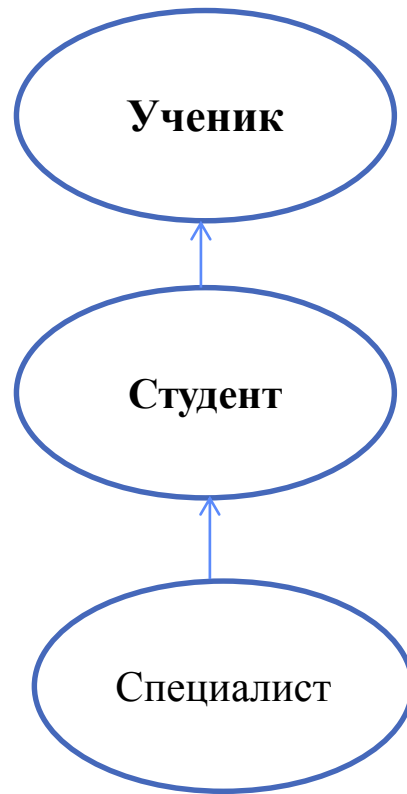
Деструктор производного класса должен выполняться раньше деструктора базового класса. Вся работа по организации соответствующего вызова возлагается на компилятор, программист не должен заботиться об этом.

В производном классе обычно добавляются новые элементы к элементам базового класса. Однако можно переопределять элементы базового класса. Если необходимо вызвать метод базового класса, а не переопределенный вариант, то можно это сделать с помощью операции ::, применяемой в форме:

**<имя\_класса>::<имя\_элемента>**

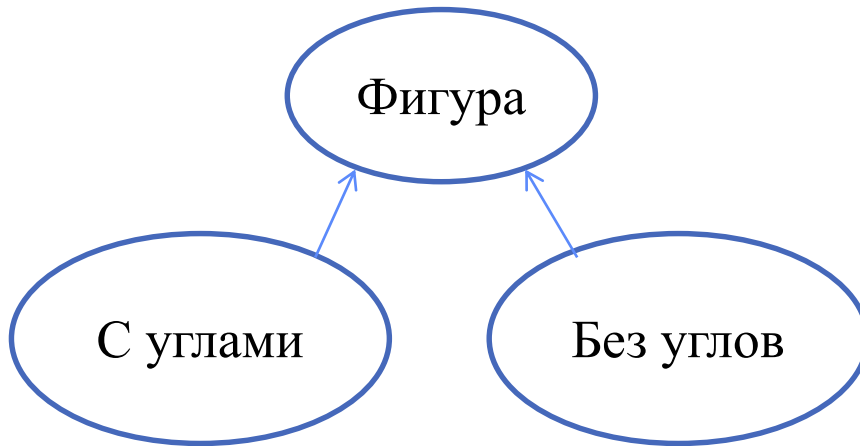
# Косвенный базовый класс

Производный класс сам может являться базовым классом для некоторого другого класса. При этом исходный базовый класс называется *косвенным* базовым классом для производного класса.

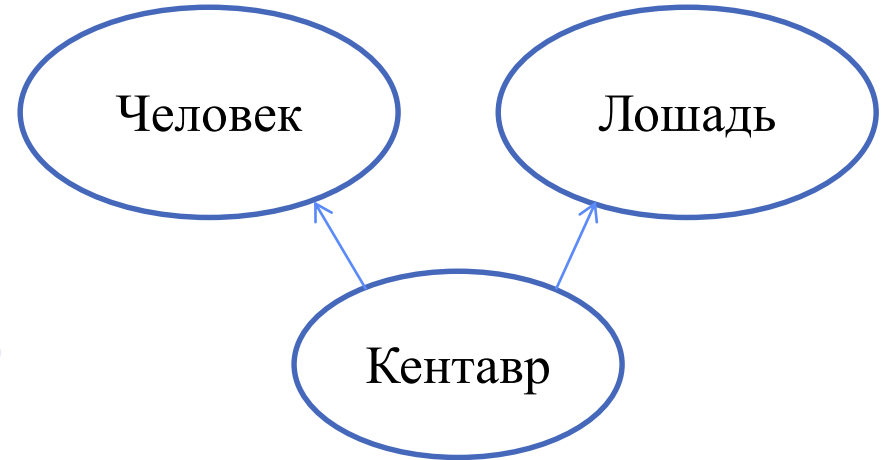


# Наследование

Если производный класс наследует свойства более чем от одного базового класса, то такое наследование называется *множественным*.



простое  
наследование



множественное  
наследование

# Правила наследования различных методов

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными ниже правилами.

- Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса по умолчанию (то есть тот, который можно вызвать без параметров), а для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.

- В случае нескольких базовых классов их конструкторы вызываются в порядке объявления.

# Правила наследования деструкторов

- Деструкторы не наследуются, и если программист не описал в производном классе деструктор, он формируется по умолчанию и вызывает деструкторы всех базовых классов.
- В отличие от конструкторов, при написании деструктора производного класса в нем не требуется явно вызывать деструкторы базовых классов, поскольку это будет сделано автоматически.
- Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор класса, затем — деструкторы элементов класса, а потом деструктор базового класса.

Порядок вызовов конструкторов и деструкторов рассмотрим на примере:

# Вызов конструкторов и деструкторов

```
class base1{  
    public: base1() {cout<<"constr_base1"<<endl;}  
           ~base1() {cout<<"destr_base1"<<endl;}  
};
```

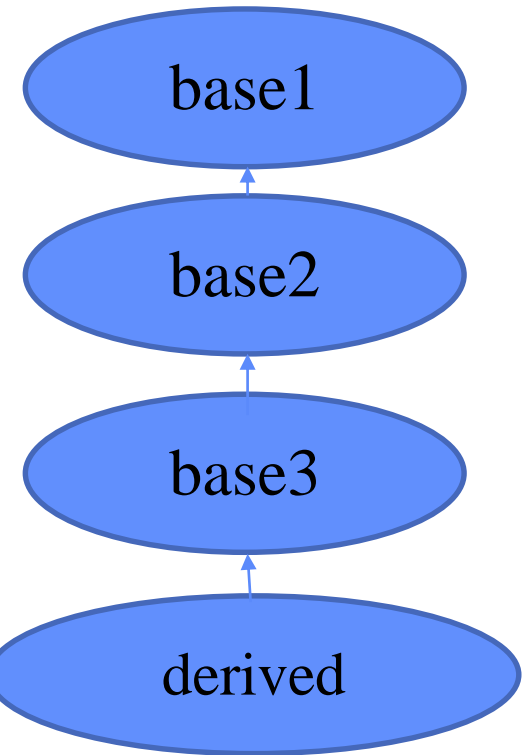
```
class base2: public base1{  
    public: base2() {cout<<"constr_base2"<<endl;}  
           ~base2() {cout<<"destr_base2"<<endl;}  
};
```

```
class base3: public base2{  
    public: base3() {cout<<"constr_base3"<<endl;}  
           ~base3() {cout<<"destr_base3"<<endl;}  
};
```

```
class derived: public base3{  
    public: derived () {cout<<"constr_derived"<<endl;}  
           ~derived() {cout<<"destr_derived"<<endl;}  
};
```

```
int main() {    derived ob;
```

```
    // derived *ob2=new derived;    delete ob2;
```



constr\_base1  
constr\_base2  
constr\_base3  
constr\_derived  
destr\_derived  
destr\_base3  
destr\_base2  
destr\_base1

# Вызов конструкторов и деструкторов при множественном наследовании

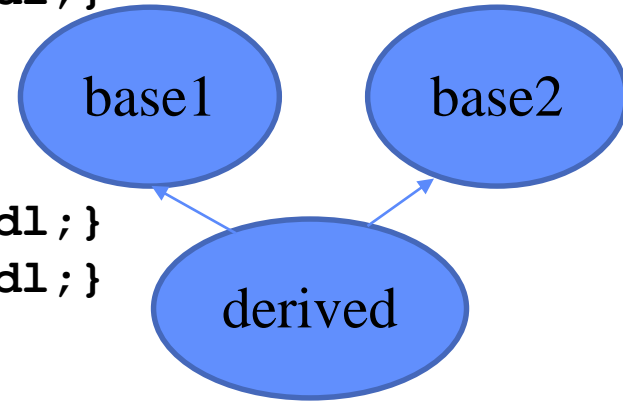
```
class base1{
    public: base1() {cout<<"constr_base1"<<endl;}
           ~base1() {cout<<"destr_base1"<<endl;}
};

class base2{
    public: base2() {cout<<"constr_base2"<<endl;}
           ~base2() {cout<<"destr_base2"<<endl;}
};

class derived: public base1, public base2{
    public: derived() {cout<<"constr_derived"<<endl;}
           ~derived() {cout<<"destr_derived"<<endl;}
};

int main() {
    derived ob1;

    // derived *ob2=new derived;  delete ob2;
```



```
constr_base1
constr_base2
constr_derived
destr_derived
destr_base2
destr_base1
```



# Обращение к элементам с одинаковыми именами

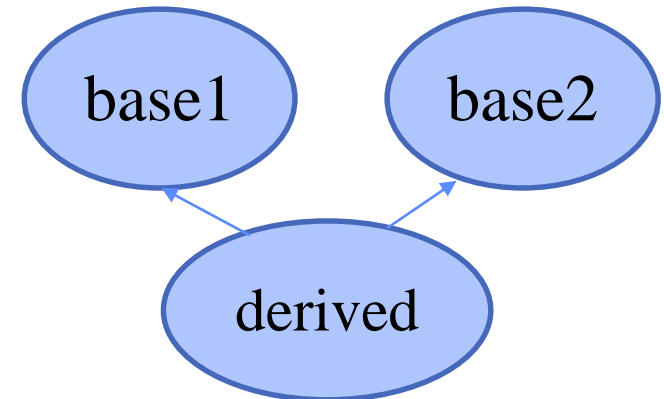
Если в базовых классах есть одноименные элементы, то обращение к ним в производном классе может привести к конфликту, который разрешается с помощью “::”.

Пример:

```
class base1{ public: void f(); };  
class base2{ public: void f(); };  
class derived: public base1, public base2{ };
```

```
void base1::f() {cout<<"base1"<<endl;}  
void base2::f() {cout<<"base2"<<endl;}
```

```
int main()  
{ derived ob;  
  // при вызове ob.f(); - ошибка компиляции  
  ob.base1::f();  
  ob.base2::f();  
}
```



# Пример множественного наследования

Описать иерархию классов: базовые классы: Орел (размах крыльев, скорость), Лев(масса, кличка); класс-наследник: Грифон(возврат).

```
class orel{
    float razmax, speed;
public:
    orel();
    orel(float _razmax, float _speed);
    ~orel();
    void show();
};

orel::orel() {
    razmax=1.5; speed=15;
    cout<<"constr_orel"<<endl;
}

orel::orel(float _razmax, float _speed)
{
    razmax=_razmax; speed=_speed;
    cout<<"constr_orel"<<endl;
}

orel::~~orel() {cout<<"destr_orel";}

void orel::show() {
    cout<<" razmax="<< razmax << endl;
    cout<<" speed=" << speed <<endl;
}
```

# Пример множественного наследования

```
class lev{
    float massa;
    char name[15];
public:
    lev();
    lev(char *_name, float _massa);
    ~lev();
    void show();
};

lev::lev() {
    massa=120; strcpy(name, "Лева");
    cout<<"constr_lev"<<endl; }

lev::lev(char *_name, float _massa){
    massa=_massa; strcpy(name, _name);
    cout<<"constr_lev"<<endl; }

lev::~~lev() {cout<<"destr_lev"<<endl; }
    void lev::show() {
        cout<<"name="<<name <<"\n massa="<<massa<<endl; }
```

# Пример множественного наследования

```
class grifon:public orel, public lev
{public: int g;
  grifon();
  grifon(float _razmax, float _speed, char *_name, float
    _massa,int _g);
  ~grifon ();
  show();
};

grifon::grifon(): orel (), lev () {
    g=15; cout<<"constr_grifon"<<endl;}

grifon::grifon(float _razmax, float _speed, char *_name,
    float _massa, int _g): orel (_razmax, _speed),
    lev (_name, _massa)
{g=_g; cout<<"constr_grifon"<<endl;}

grifon::~~grifon () {cout<<"destr_grifon"<<endl;}

grifon::show() {
//orel::show(); //lev::show();
    cout<<"g="<<g<<endl;}
```

# Пример множественного наследования

```
int main() {
    grifon ob1(25, 40, "грифончик Гриша", 200, 5);
    ob1.show();
    ob1.lev::show();
    ob1.orel::show();
        grifon ob2;
        ob2.show();
        ob2.lev::show();
        ob2.orel::show();
    /*grifon *gr;
    gr= new grifon(25, 40, "грифончик Гриша", 200,
    gr->show();
    //((orel*)gr)->orel::show();
    //((lev*)gr)->lev::show();
    delete gr;*/
}
```

```
constr_orel
constr_lev
constr_grifon
g=5
name=грифончик Гриша
massa=200
razmax=25
speed=40
constr_orel
constr_lev
constr_grifon
g=15
name=лева
massa=120
razmax=1.5
speed=15
destr_grifon
destr_lev
destr_orel
destr_grifon
destr_lev
destr_orel
```

# Ромбовидное наследование

При множественном наследовании в сложной иерархии может получиться так, что производный класс косвенно наследует два или более экземпляра одного и того же класса - **ромбовидное наследование** (т.е. у базовых классов есть общий предок).



# Ромбовидное наследование

При таком наследовании производный класс наследует от базовых классов два экземпляра полей, что чаще всего является нежелательным. Чтобы избежать такой ситуации, требуется при наследовании общего предка определить его как виртуальный класс.

```
class figure{...};  
class square: virtual public figure{...};  
class triangle: virtual public figure{...};  
class derived: public square, public triangle{...};
```

Множественное наследование применяется для того, чтобы обеспечить производный класс свойствами двух или более базовых.

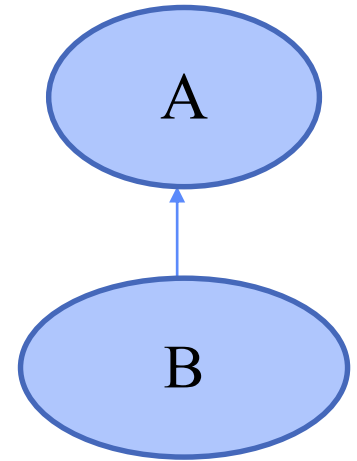
Чаще всего один из этих классов является основным, а другие обеспечивают некоторые дополнительные свойства, поэтому они называются *классами подмешивания*. По возможности классы подмешивания должны быть виртуальными и создаваться с помощью конструкторов без параметров, что позволяет избежать многих проблем, возникающих при ромбовидном наследовании.

```
// (men*)p ->men::show();
```

# ИТОГ: пример простого наследования

```
class A{
int a;
public:
    A();
    A(int x);
    ~A();
    f();
};

A::A(){cin>>a; cout<<"constr_A\n";}
A::A(int x){a=x; cout<<"constr_A\n";}
A::~~A(){cout<<"destr_A\n";}
A::f(){cout<<"a="<<a<<endl;}
```





# Пример простого наследования

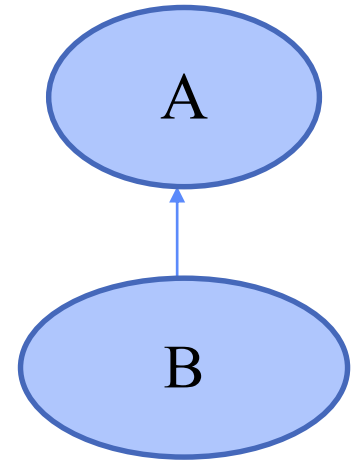
```
class B: public A{  
    int b;  
public:  
    B();  
    B(int x, int y);  
    ~B();  
    f();  
};
```

```
B::B() :A() {cin>>b;cout<<"constr_B\n";}
```

```
B::B(int x, int y) :A(x) {b=y;cout<<"constr_B\n";}
```

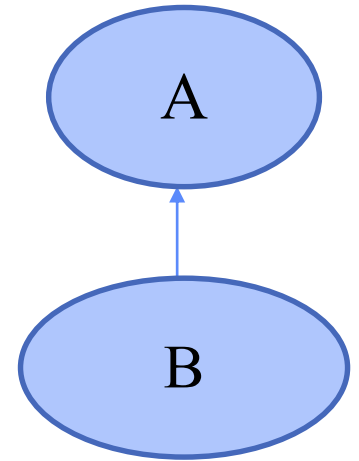
```
B::~~B() {cout<<"destr_B\n";}
```

```
B::f() {cout<<"b="<<b<<endl;}
```



# Пример простого наследования

```
int main() {  
    B ob(4,5);  
    ob.f();  
    ob.A::f();  
}
```



```
constr_A  
constr_B  
b=5  
a=4  
destr_B  
destr_A
```