Севастопольский государственный университет
Кафедра информационных систем

Курс лекций по дисциплине

«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ"

(ООП)

Лектора: Пелипас Всеволод Олегович
Сметанина Татьяна Ивановна

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.

Множественное наследование позволяет одному классу обладать свойствами двух и более родительских классов.

Наследование

Наследование - это механизм создания нового класса на основе уже существующего. 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 { int i;                    //базовый класс
        public:  void f(){ i=10; cout<<i;}
        };
class B : public A {                //производный класс
         int i;
        public: void f(){ i=20; cout<<i*i;}  //переопределение метода
        };
main() {
 A ob_a;
 B ob_b;
  ob_a.f();
  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 клас-се **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;
- конструкторы копирования;
- деструкторы;
- операторы присваивания, определенные программистом;
- друзья класса.

Пример простого наследования

Простым называется наследование, при котором производный класс наследует свойства одного базового класса.

Пример 1

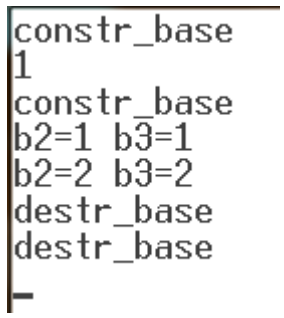
```
class figure{    //базовый класс
protected: int n;
public:
 int f(int _n=10){ n=_n; return n;}
};

class B : public figure {}; //производный класс

main()
{
 B ob_b;                  //объявление объекта производного класса
  cout<<ob_b.f();         // вызов метода из базового класса
}
```

Пример простого наследования

```
class base{  private: int b1;  int f1() {b1++; return b1;}  // базовый
             protected: int b2;
             public: int b3;
  base(){b1=b2=b3=1;cout<<"constr_base"<<endl;}
  ~base(){b1=b2=b3=0;cout<<"destr_base"<<endl;}
  void f3(){b1=222;cout<<"b1="<<b1;}
};
class derived : private base{                          // производный
 public:
void f_inc(){b2++; b3++;}   // к b1 нет доступа
 void f_print(){cout<<"b2="<<b2<<" b3="<<b3<<endl;}
};
main()
{  base ob;
// cout<<ob.b1; // cout<<ob.b2; // нет доступа
  cout<<ob.b3<<endl;
```

Пример простого наследования

```
derived ob1;
//   cout<<ob1.b1<<ob1.b2<<ob1.b3; // нет доступа
 ob1.f_print();              //печать полей b2, b3
 ob1.f_inc();               //изменение полей b2, b3
 ob1.f_print();              //печать полей b2, b3
//ob1.f3(); // нет доступа
}
```



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::f3;        //!!!
void f_inc(){b2++;b3++;}
 void f_print(){cout<<"b2="<<b2<<" b3="<<b3<<endl;}
};

// тогда можно обратиться к методу f3()
main()
{  base ob;
cout<<ob.b3<<endl;
 derived ob1;
          ob1.f_print();
          ob1.f_inc();
          ob1.f_print();
 ob1.f3();
}
```

```
constr_base
1
constr_base
b2=1 b3=1
b2=2 b3=2
b1=222destr_base
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

```
class Coord                          //базовый
{   int x, y;
  public:   Coord(int _x, int _y){ x=_x; y=_y;}  // конструктор
            Coord(){x=0;y=0;}          // конструктор по умолчанию
     int get_x(){return x;}
     int get_y(){return y;}
};

class Out_Coord: public Coord                //производный
{
 public:
   Out_Coord(int _x, int _y) : Coord( _x, _y){}   //конструктор
   void show_x(){cout<<get_x()<<" ";}
   void show_y(){cout<<get_y()<<" ";}
};
```

Пример вызовов конструкторов при наследовании

```
main()
{  Out_Coord* ptr;
   ptr=new OutCoord(10, 20);
  ptr->show_x();
     ptr->show_y();
   delete ptr;
}
```

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. Рассмотрим на примере:

Пример вызовов конструкторов и деструкторов при простом наследовании

```
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
```

Доступ к переопределяемым методам

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>

Пример:

```
class Coord{
  protected: int x, y;
  public:
   Coord(int _x, int _y){x=_x; y=_y;}
   Coord(){x=0; y=0;}
   int get_x() {return x;}
   int get_y() {return y;}
};
```

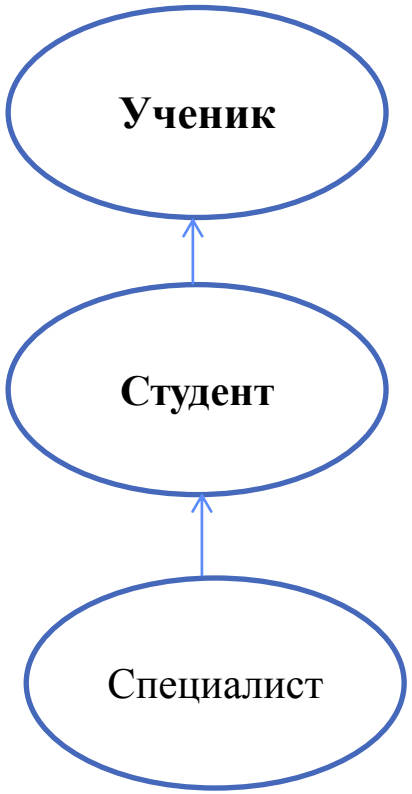
Доступ к переопределяемым методам

```
class Out_Coord:public Coord{
 public:
   Out_Coord(int _x, int _y): Coord( _x, _y){}
   int get_x(){return ++x;}
   int get_y(){return ++y;}
      void show_x(){cout<<get_x()<<" ";}
             void show_x_(){cout<<Coord::get_x()<<" ";}
      void show_y(){cout<<get_y()<<" ";}
};
main()
{ Out_Coord* ptr;
   ptr=new Out_Coord(10, 20);
           ptr->show_x_();
            ptr->show_x();
  cout<<ptr->Coord::get_y()<<endl;
  ptr->show_y();
  delete ptr; }
```

```
10 11 20
21
```

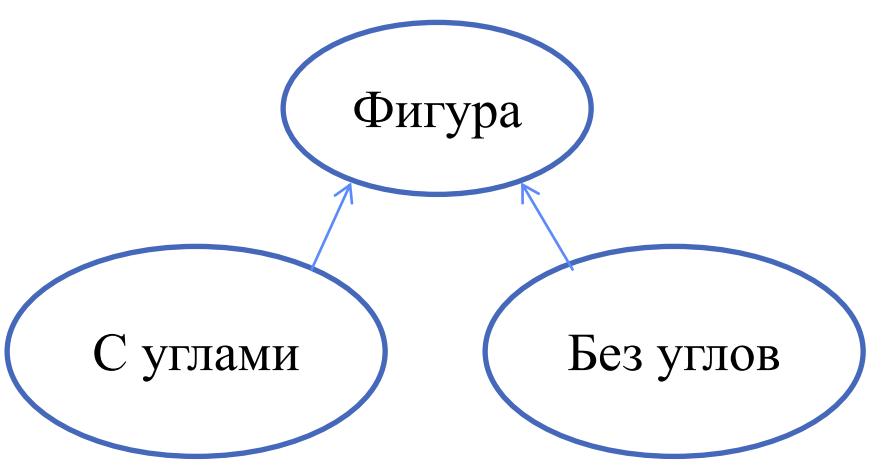
Косвенный базовый класс

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

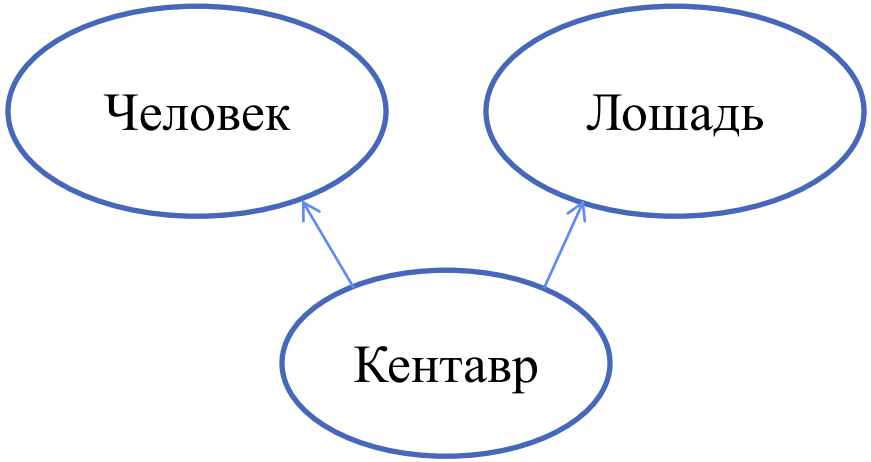


Наследование

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.

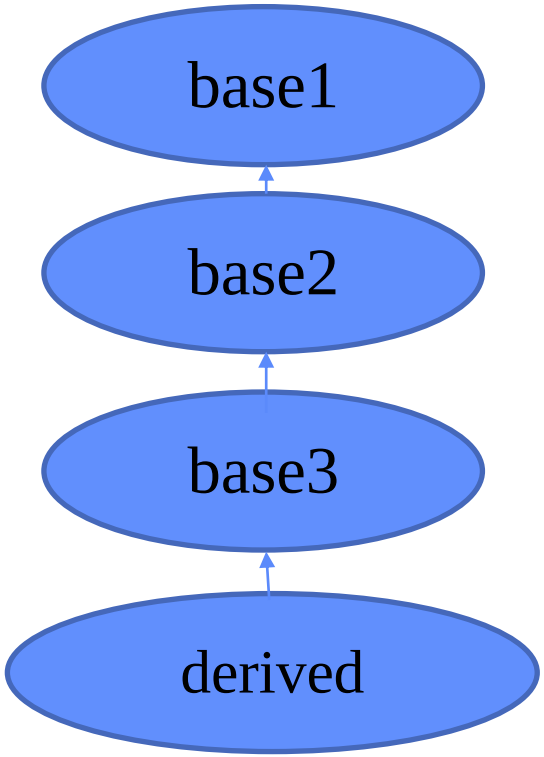
- В отличие от конструкторов, при написании деструктора производного класса в нем не требуется явно вызывать деструкторы базовых классов, поскольку это будет сделано автоматически.

- 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

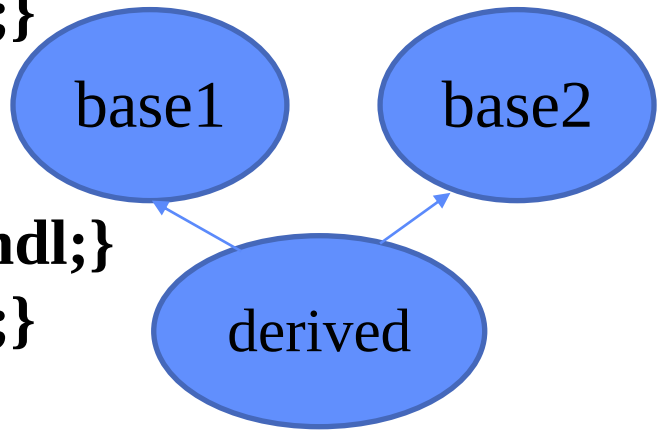
```
class base1{
  public: base1(){cout<<"constr_base1"<<endl;}
        ~base1(){cout<<"destr_base1"<<endl;}
};
class base2: public base1{
  public: base2(){cout<<"constr_base2"<<endl;}
        ~base2(){cout<<"destr_base2"<<endl;}
};
class base3: public base2{
  public: base3(){cout<<"constr_base3"<<endl;}
        ~base3(){cout<<"destr_base3"<<endl;}
};
class derived: public base3{
 public:  derived (){cout<<"constr_derived"<<endl;}
        ~derived(){cout<<"destr_derived"<<endl;}
};
main()
{ derived ob;}
```




```
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;}
};

main()
{ derived ob; }
```




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

Пример множественного наследования

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

```
class orel                                          // класс Орел
{  public: float razmax, speed;
    orel (float _razmax, float _speed)                  //конструктор
   {  razmax=_razmax; speed=_speed;}

    void show_orel()                                // функция печати
   {    cout<<" razmax="<< razmax << endl
          <<" speed=" << speed <<endl; }
};

class lev                                           // класс Лев
{public:  float massa; char name[15];
    lev (char *_name, float _massa)             // конструктор
   { massa=_massa;    strcpy(name,_name);
   }
```

Пример множественного наследования

```
void show_lev()                                          // функция печати
   {  cout<<"\n name "<<name <<"\n massa="<<massa<<endl;}
};

class grifon:public orel, public lev              // класс Грифон
{public:
   grifon(float _razmax, float _speed, char *_name, float _massa):
    orel (_razmax, _speed), lev (_name, _massa) {}
   show();
};

grifon::show()                                           // функция печати
{ cout<<"**********************";
   show_lev();
   show_orel();
   getch();
}
```

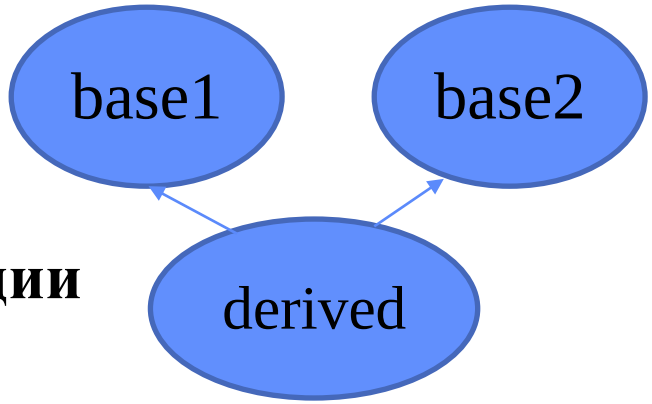
Пример множественного наследования

```
int main()
{    grifon *gr;          // указатель на объект
                          // инициализация
   gr= new grifon(25, 40, "Грифончик Гриша", 200);
   gr->show();}          // вывод
```

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

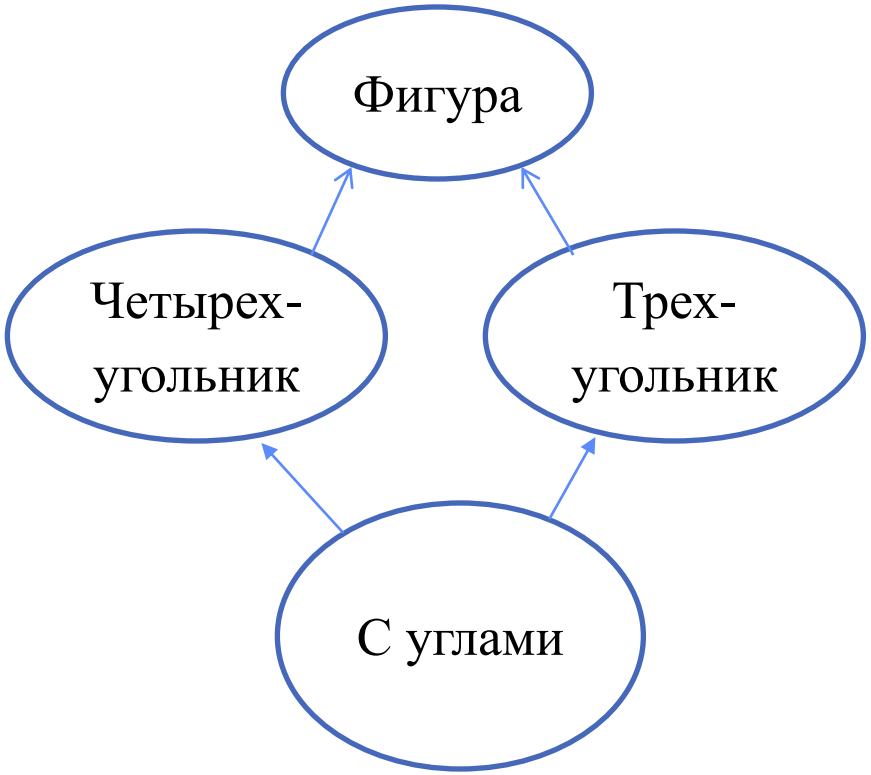
Пример:

```
class base1{ public: f(){cout<<"base1"<<endl;}    };
class base2{ public: f(){cout<<"base2"<<endl;}    };
class derived: public base1, public base2{ };
main()
{ derived ob;
   // при вызове ob.f(); - ошибка компиляции
  ob.base1::f();  ob.base2::f();  }
```



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 scuare: virtual public figure{…};
class triangle: virtual public figure{…};
class derived: public scuare, 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.