Московский авиационный институт
(технический исследовательский университет)

Кафедра 403
“Электронно-вычислительные средства и информатика”

Методическое пособие по теме «Отношения между классами C++»

Выполнил студент группы № 40-103С
Боровиков Дмитрий

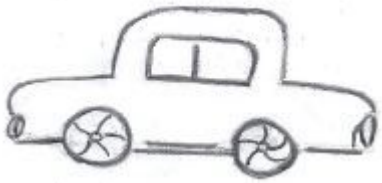
Научный руководитель:
старший преподаватель кафедры 403
Мальшаков Григорий Викторович

Москва 2013

Классы могут состоять из элементов других классов, либо могут быть разновидностью других, более общих классов. В силу этого классы могут вступать в отношения ***включения*** или ***наследования***.

Включение – это отношение между классами, при котором объект одного класса строится из объектов другого класса. Это отношения между классами “быть частью”. Примеры включения:

Колесо есть часть автомобиля



Лепесток есть часть растения

Задание. Приведи собственный пример включения :

Включение на языке С++ выполняется посредством полей:

```
class A {};
class B {                    // В объект класса B входит объект класса A
   A obj;
 };
```

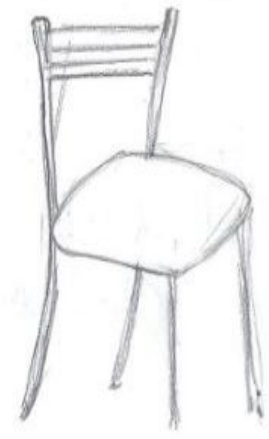
Включение может быть ***множественным*** или ***композиционным***

При множественном включении объект одновременно может входить в состав нескольких объектов.

Работник работает одновременно на нескольких предприятиях

При композиционном включении объект является неотъемлемой частью другого объекта

Стул имеет спинку и ножки

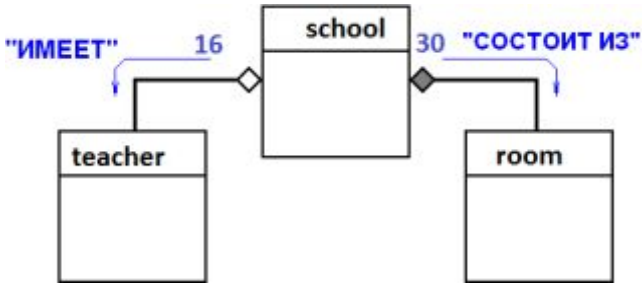


В комнате имеется пол, потолок, и стены

В программе при множественном включении, в отличие от композиционного в классе хранится не сам объект, а только его адрес. Поэтому объект может одновременно входить в несколькие объекты класса.

```
class  room { };          // класс “комнаты”
class  teacher { };       // класс “учителя”
class  school             // класс “школа”
{
   room      objR[30];  // школа состоит из 30 комнат
   teacher  *objT[16];  // в школе работают 16 учителей.
};
```

Диаграмма UML:



Количество объектов указывают в виде диапазона, например «0..*», «1..*», «*».

В примере в класс школа (**school**) входит 30 комнат (**room**) и 16 учителей (**teacher**). Причем комнаты входят по значению (т.е. являются неотъемлемой частью объекта школы) , а учителя входят по адресу (не являясь при этом неотъемлемой частью школы) : т.е. адрес одного и того же учителя может одновременно храниться в различных объектах «школа», уничтожение объекта «Школа» не приведет к уничтожению объекта «Учитель».

Задание. Придумай свое отношение включения. Определите его вид. Отобразите его в виде картинок, диаграммы UML и программного кода.

Рисунок

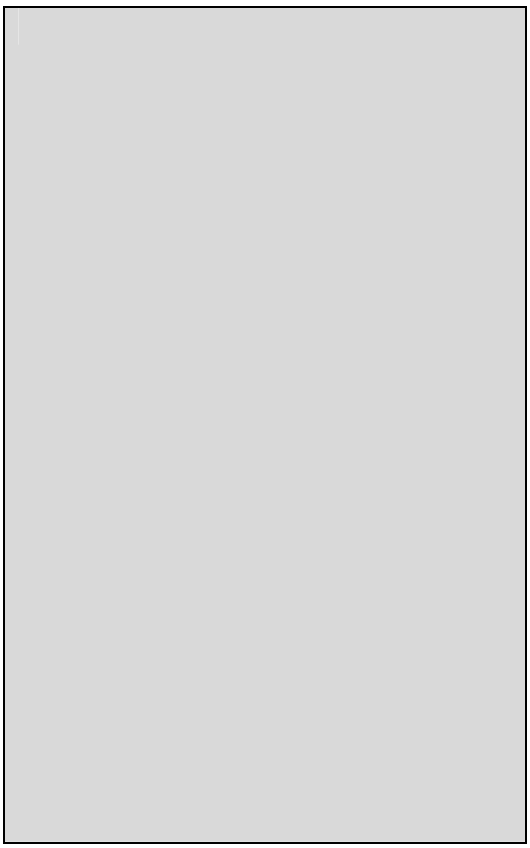
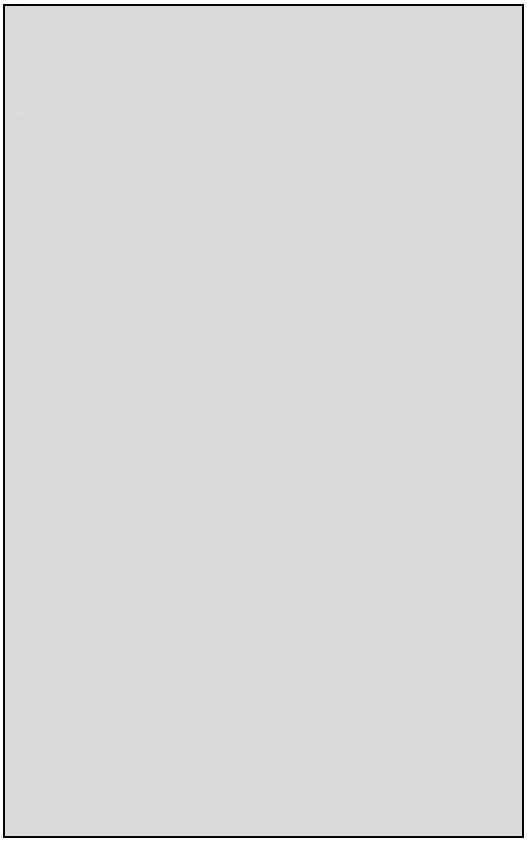
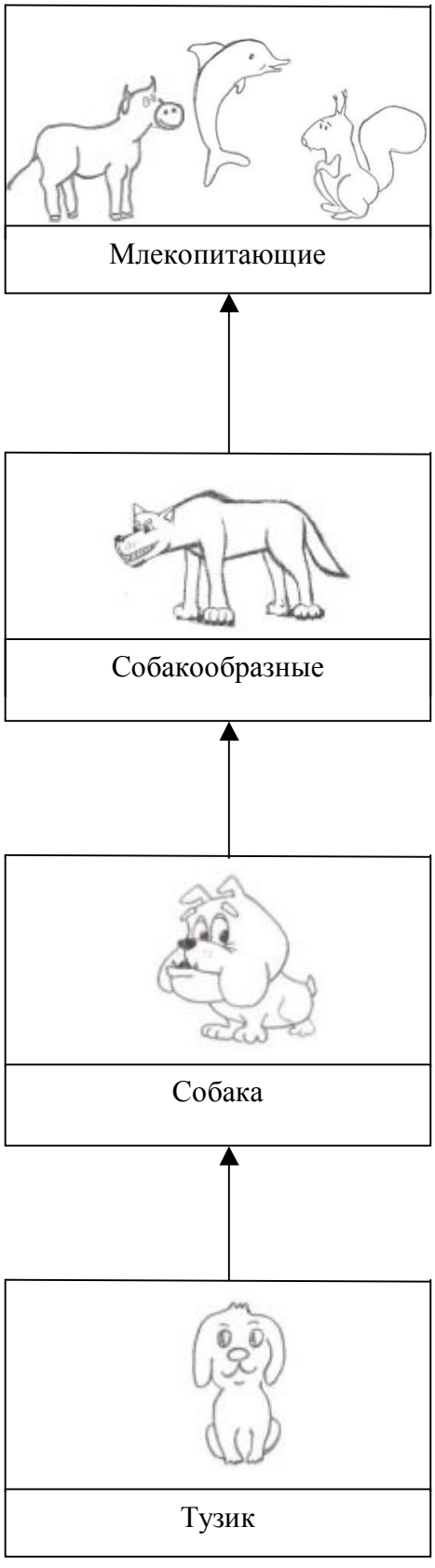


Диаграмма UML



Программный код

Наследование — механизм, позволяющий написать производный класс-потомок на основе уже существующего (родительского, базового) класса.

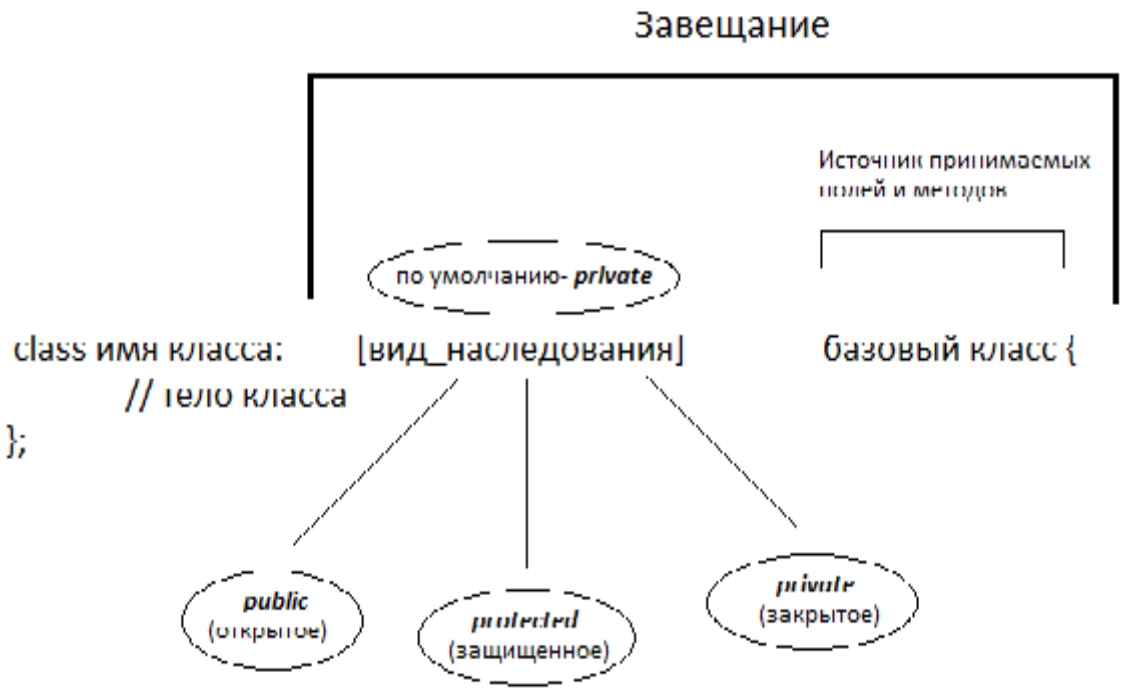


Связь понятий воедино с помощью наследования.

Тузик является частным случаем собаки, которая является частным случаем собакообразных, являющихся млекопитающих и т.д.

При наследовании в класс-потомок добавляются поля и методы базового класса.

В программе определение класса, наследуемого от некоторых, существующих классов, производится следующим образом:

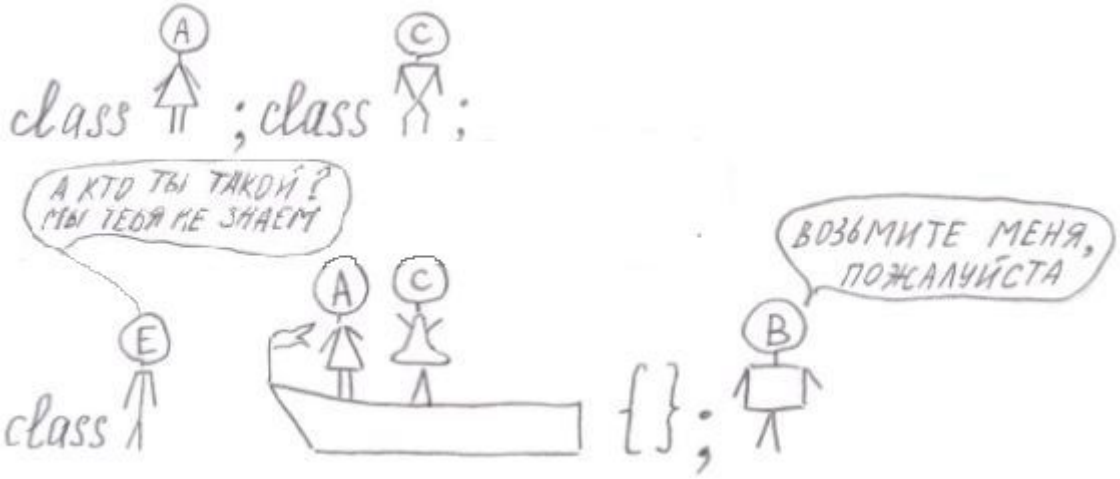


В теле класса располагают переопределение унаследованных элементов и определение собственных элементов производного класса.

Предъявляемые требования к базовым классам:

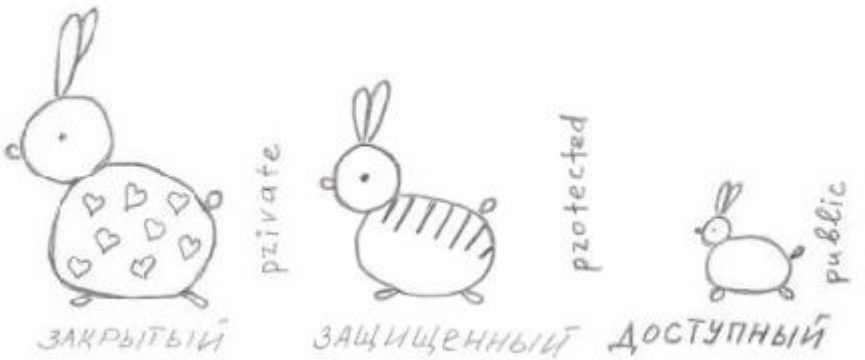
- количество базовых классов в списке наследования может быть любым

- один и тот же класс не может быть задан в списке наследования дважды



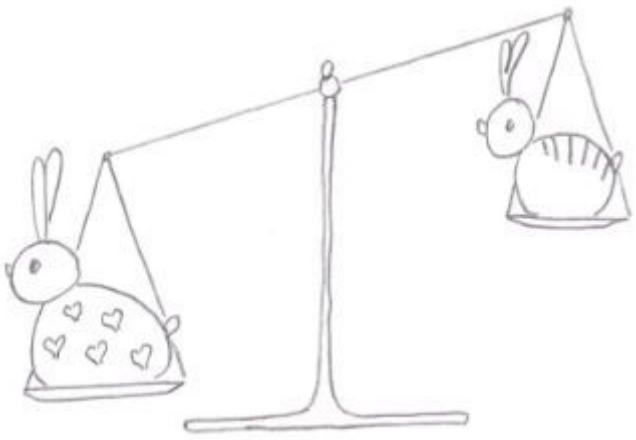
- базовый класс к моменту определения производного должен быть определен или описан

В качестве вида наследования (вида завещения) указывается одно из ключевых слов – ***private***(закрытый), ***protected***(защищенный), ***public***(общедоступный).

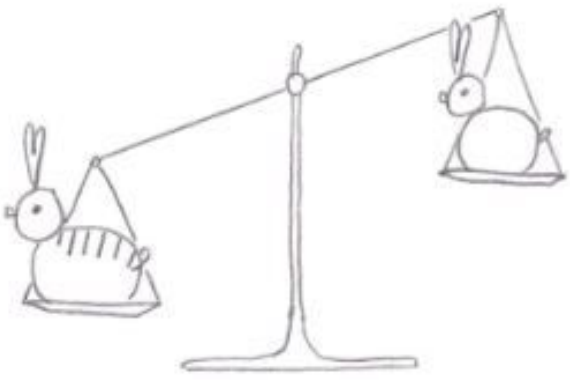


В зависимости от вида наследования (завещения) меняется видимость наследуемых элементов базового класса: в производном ***private*** превращает элементы ***protected*** и ***public*** в ***private***, ***protected*** превращает элементы ***public*** в ***protected.***

Это как в жизни сильнейший пожирает слабейшего, сильнейшая защита покрывает слабейшую.



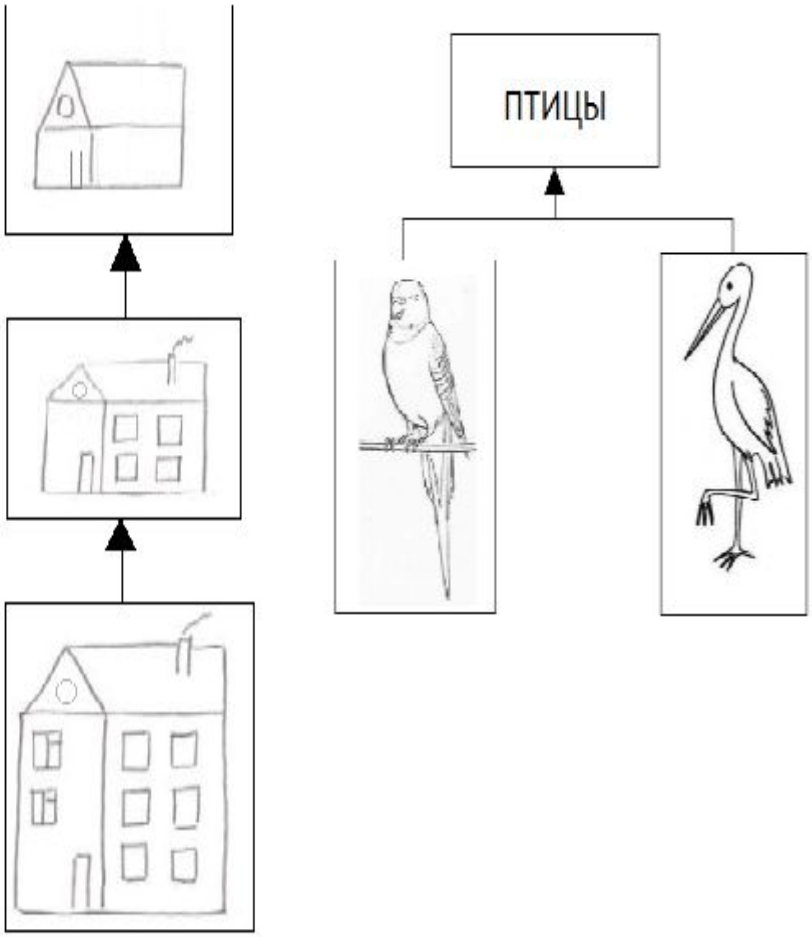
Закрытый сильнее защищенного



Защищенный сильнее доступного

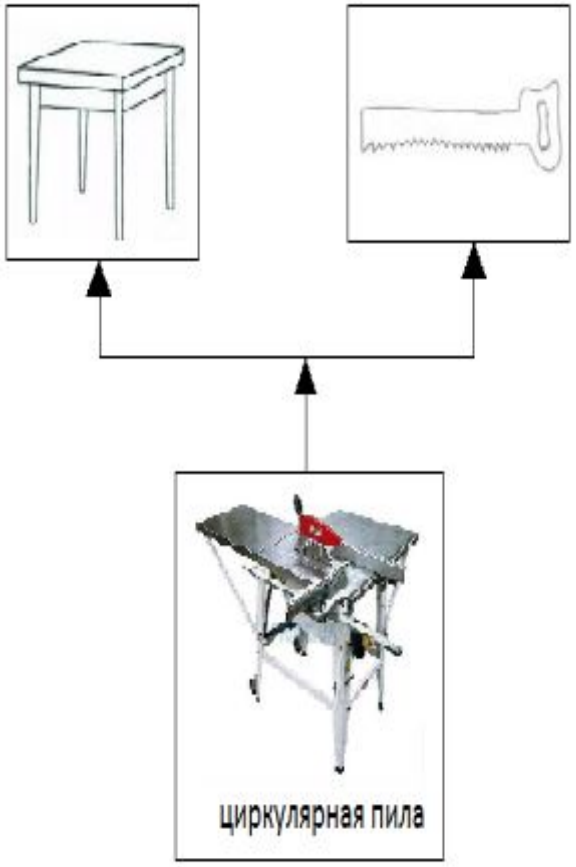
Наследование бывает одиночным и множественным

Одиночное наследование



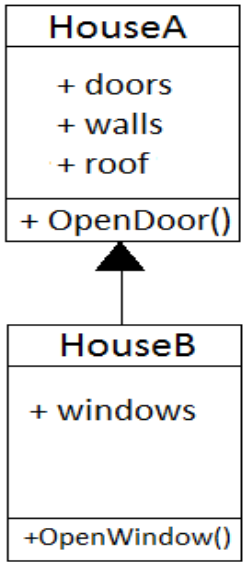
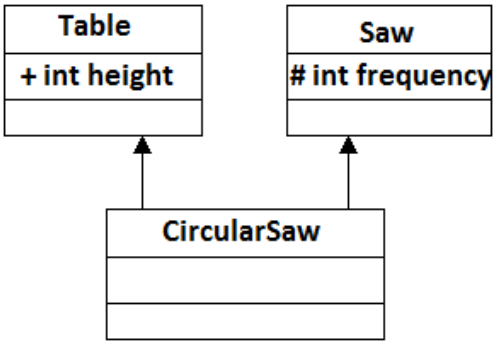
При одиночном наследовании один потомок наследуется от одного родителя

Множественное наследование



При множественном наследовании один потомок наследуется от нескольких родителей

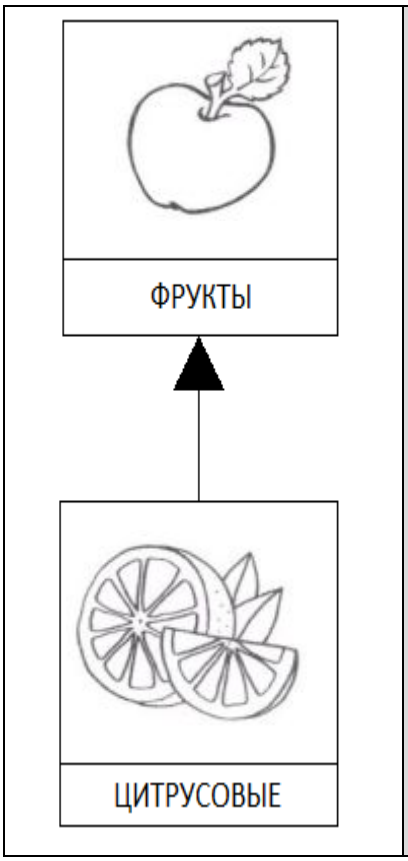
Примеры наследования:

Одиночное наследование	*Множественное* наследование
	
class HouseA { **public:** **int doors; *//двери*** **int walls; *//стены*** **int roof; *//крыша*** **OpenDoor(){};** **};** **class HouseB: public HouseA {** **public:** **int windows; *//окна*** **OpenWindows(){};** **};**	**class Table *//стол*** **{** **public:** **int height; *//высота*** **};** **class Saw** **{** **protected:** **int frequency; *//частота вращения*** **};** **class CircularSaw: public Table, protected Saw** **{** **…** **};**

Задание. Заполни таблицу «Доступ к элементам базового класса через производный класс»

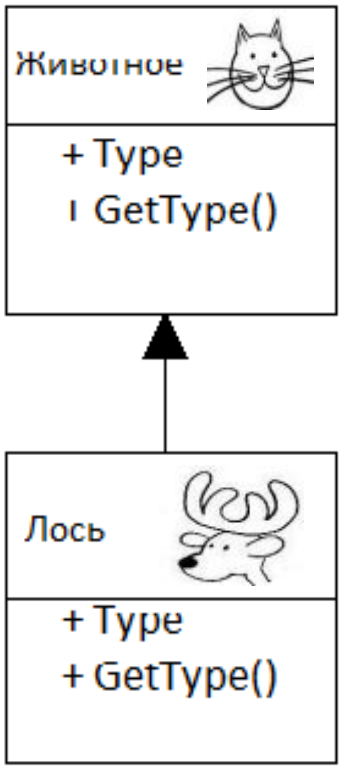
Объявление элементов в ***базовом*** классе	Вид наследования	Доступ к элементам базового класса через производный класс
private		
protected		
public		public
private		private
protected		protected
public		
private		private
protected		private
public		private

Задание. Напиши программный код, реализовав диаграмму наследования **Citrus** (цитрусовых) от **Fruits** (фруктов).



Программный код

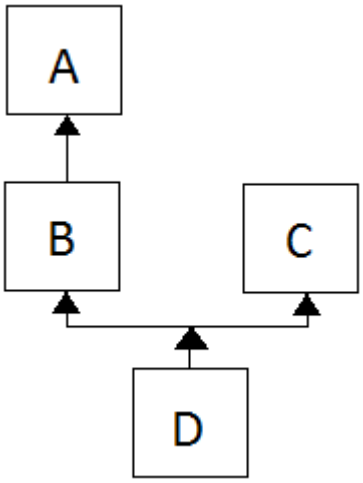
Множественное наследование — потенциальный источник ошибок, которые могут возникнуть из-за наличия одинаковых имен методов в предках. Для разрешения конфликтов использования наследованных методов с одинаковыми именами, возможно, например, применить операцию расширения видимости — «::» — для вызова метода конкретного родителя.




```
class Животное{
  public:
     char Type;
     char GetType(){return Type;};
}
class Лось: public Животное{
   public:
     char Type;
     char GetType(){return Type;};
}
main(){

   Лось Петя ;

   //Для полей
   Петя.Type="М";

   Петя.Животное::Type='Ж';

   //Для методов
   cout<<Петя.GetType();//М

   cout<<Петя.Животное::GetType();//Ж
}
```

При создании объекта производного класса: сначала вызываются конструкторы базовых классов, затем – конструкторы включаемых в класс объектов (объектов других классов, которые являются элементами данного класса) и в последнюю очередь - конструктор производного класса. Если производный класс наследуется от нескольких базовых классов то их конструкторы вызываются в порядке их следования в наследовании.

При вызове деструктора производного класса (в том числе при уничтожении объекта) вызываются деструкторы всех базовых классов, причем вызов производится в порядке, обратном вызову конструкторов.

Программный код:	Диаграмма UML:
class A{ **public:** **A() {printf("A()\n";}** **~A() {printf("~A()\n";}** **};** **class B: A{** **public:** **B() {printf("B()\n";}** **~B() {printf("~B()\n";}** **};** **class C: {** **public:** **C() {printf("C()\n";}** **~C() {printf("~C()\n";}** **};** **class D: protected B, private C{** **public:** **D() {printf("D()\n";}** **~D() {printf("~D()\n";}** **};** **main(){** **D obj_D;** **}**	 Выведется на экран: **A** **B** **C** **D** **~D** **~C** **~B** **~A**

При создании объекта производного класса базовые классы (их поля) проецируются на память в соответствии с порядком вызова конструкторов.
Для объекта D:

Чтобы при создании объекта производного класса вызывался необходимый конструктор базового класса с необходимыми нам параметрами, его вызов необходимо указать явно в описании конструктора производного класса

конструктор(список_форм_параметров) : конструктор_баз_кл_1 (список_факт_парам)
{ / тело_конструктора */ }*

На языке C++ это будет выглядеть следующим образом

```
class TPoint {                                  // черно – белая точка
    int x,y;
  public:
    TPoint() { x = y= 0;}
    TPoint (int aX, int aY) { x=aX; y=aY; }
};
class CPoint : public TPoint {                  // цветная точка
    int colour;
  public:
    CPoint() { colour = 0; }
    CPoint(int aX, int aY, int c): TPoint (aX, aY) {colour = c;}
};
main() {
  CPoint cp2;            //x=0  y=0  colour=0 – вызыв. констр. баз. класса по умолч.
  CPoint cp1(1, 2, 3);   //x=1  y=2  colour=3 – вызыв. констр. баз. класса с парам.
}
```

В примере явным образом вызывается конструктор **TPoint(aX,aY)** , при вызове конструктора **CPoint(int aX, int aY, int c)** параметры из **CPoint aX** и **aY** передаются в **TPoint.**

Задание. Придумайте пример из жизни множественного наследования. Отобразите его в виде рисунка, диаграммы UML и программного кода.

Рисунок

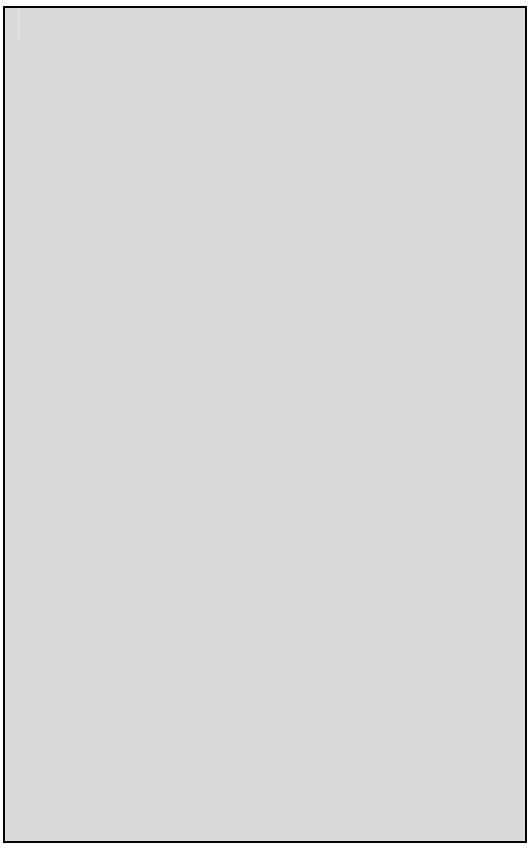
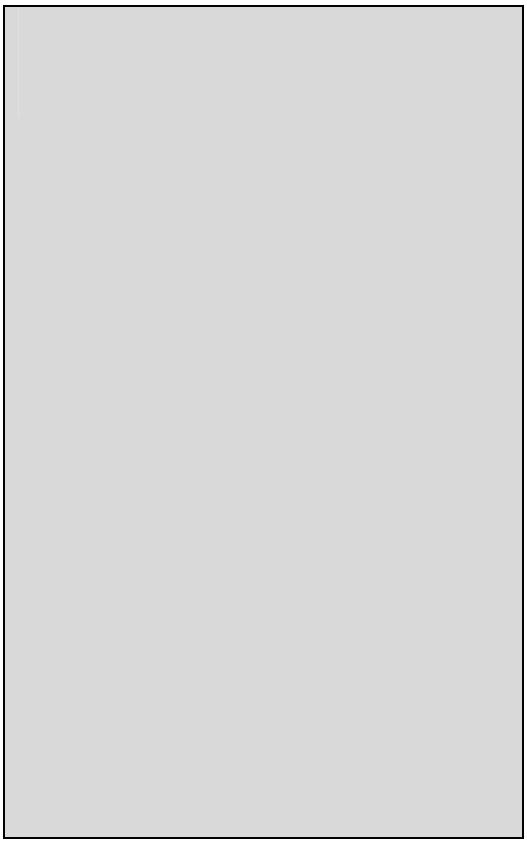


Диаграмма UML

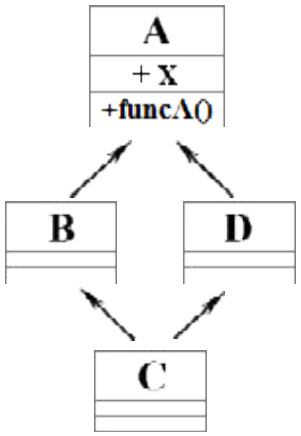


Программный код

Виртуальное наследование – наследование, объявленное с ключевым словом ***virtual.***

При множественном наследовании один и тот же класс не может быть дважды указан как прямой базовый в списке наследования, однако, существует возможность несколько раз косвенного наследования от одного базового класса.

```
class A {
    public:
        int x;
        void funcA();
};
class B: public A { };
class D: public A { };
class C: public B, public D { };
```



При создании объекта класса имеющего ромбообразное наследование, создаются несколько копий полей базового класса в объекте производного класса. В примере объект класса C будет содержать две копии поля X класса A.

Чтобы не было путаницы с дублированем косвенного базового класса, используют виртуальное наследование

```
class A {
    public:
        int x;
        void funcA();
};
class B: virtual public A { };
class C: virtual public A { };
class D: public A, public B { };
```

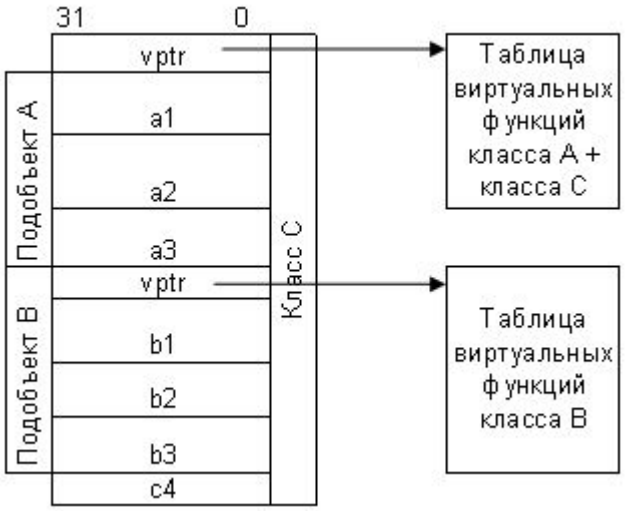
Виртуальная функция – это функция, объявленная со спецификатором ***virtual***.

```
class A
{
 public:
 virtual int Fun1(int);
};
```

Виртуальную функцию возможно подменить(в том числе и для базового) в производном классе. Подмена реализации виртуальной функции осуществляется динамически, в процессе выполнения программы. Объект каждого класса, имеющего виртуальные функции содержит скрытый от программиста указатель на так называемую ***таблицу виртуальных функций*** (virtual function table) своего класса, которая создаётся для каждого класса, имеющего хотя бы один виртуальный метод. Таблица виртуальных функций – это одномерный массив указателей на функцию, хранящий адреса конкретных реализаций функций для заданного класса в неё записываются в порядке объявления. Количество элементов в массиве равно количеству виртуальных функций в классе.

```
class A {
   public:
      A() { a1=1; a2=2; a3=3;};
      void a() { a1 = 0;};
      virtual void foo(){};
      int a1, a2, a3;
};

class B {
   public:
       B() { b1=1; b2=2; b3=3;};
       void b(){ b1 = 0;};
       virtual void bar(){};
       int b1, b2, b3;
};

class C : public A, public B {
   public:
        C() : A(), B(), c1(0xc4444444){};
        virtual void goo(){};
        int c1;
};
```

<u>В памяти объект класса С будет иметь вид</u>:



Если заранее неизвестно что будет делать виртуальная функция для экономии ресурсов ее можно определить без тела, как ***чисто виртуальную функцию***

```
class A
{
public:
virtual void v_function(void)=0;//чистая виртуальная функция
};
```

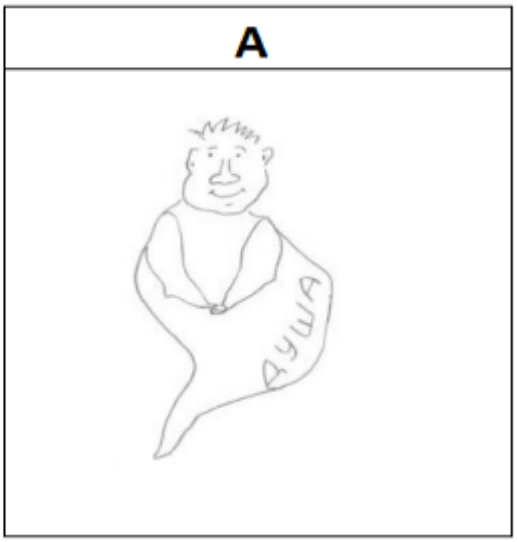
Чисто виртуальная функция – функция, не имеющая тела.

Как видите, все отличие только в том, что появилась конструкция «=0», которая называется «чистый спецификатор». Чистая виртуальная функция абсолютно ничего не делает и недоступна для вызовов. Ее назначение – служить основой (если хотите, шаблоном) для замещающих функций в производных классах.

Класс, который содержит хотя бы одну чистую виртуальную функцию, называется ***абстрактным*** классом.

Почему абстрактным? Потому, что создавать самостоятельные объекты такого класса нельзя. Это всего лишь заготовка для других классов. Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. Эти общие понятия обычно невозможно использовать непосредственно, но на их основе можно, как на базе, построить производные частные классы, пригодные для описания конкретных объектов.

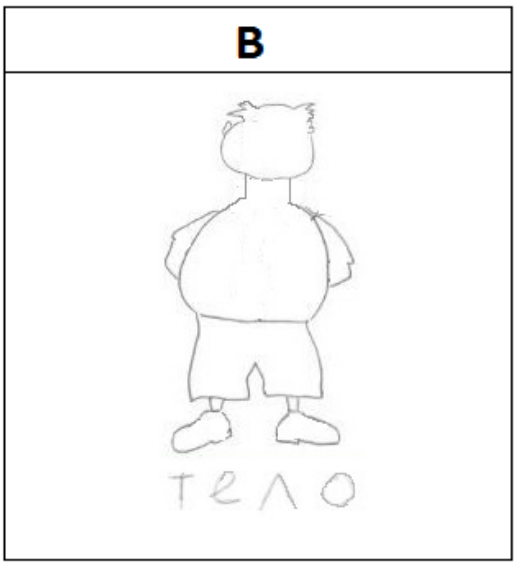
```
class A{
  public:
 virtual int v_function(void)=0;
};
```

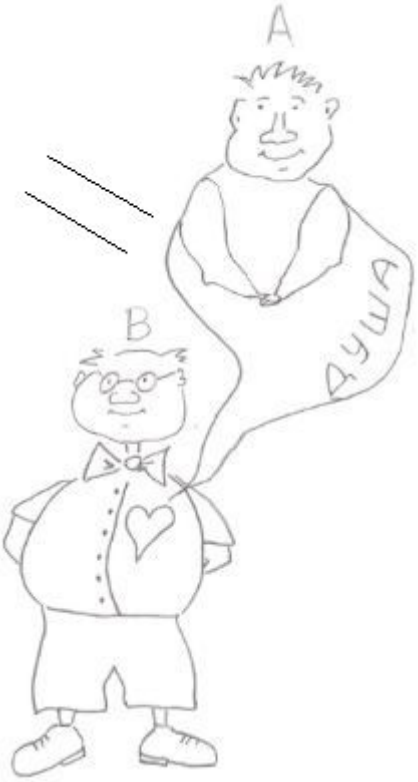


Абстрактный класс – класс, содержащий методы, неимеющие тел (аналог души А)

Наследование - вселение души А в тело В

```
class B: public A{
  public:
    int v_function(void){return S;
    }
};
```





Как вы думаете, можно ли использовать душу без тела? Нет. Так же на основе абстрактного класса невозможно создавать объекты, т.е. невозможно использовать по прямому назначению типа данных(класса). Ее возможно только вселить в другой класс в качестве идеи.

Пример программы на C++ с использованием абстрактных классов

```
#include <iostream.h>
// абстрактный базовый класс
class Animal{
public:
char *Title; //кличка животного
Animal(char *t) {Title=t;} //простой конструктор
virtual void speak(void)=0; //чисто виртуальная функция
};
//класс лягушка
class Frog: public Animal{
public:
Frog(char *Title): Animal(Title) { };
virtual void speak(void) { cout<<Title<<" говорит "<<"'ква"<<endl; };
};
//класс собака
class Dog: public Animal{
public:
Dog(char *Title): Animal(Title) { };
virtual void speak(void) { cout<<Title<<" говорит "<<"'гав"<<endl;};
};
//класс лев
class Lion: public  Animal {
public:
Lion(char *Title):  Animal (Title) { };
virtual void speak(void) { cout<<Title<<" говорит "<<"'ppp'"<<endl;};
};

int main (){
Animal *animals[3] = { new Dog("Бобик"),new Frog("Кермит"),
new Lion("Кинг")};
for(int k=0; k<3; k++) animals[k]->speak();
}
```

В примере созданы 3 производных класса **Frog**(лягушка), **Dog**(собака), **Lion**(лев) от абстрактного класса **Animal**. Они от абстрактного класса наследуют чисто виртуальную функцию **speak**(говорить). В каждом из производных классов эта чисто виртуальная функция перегружается, поэтому Бобик гавкает, Кермит квакает, а Кинг рычит.

Словарь терминов

Включение – это отношение между классами, при котором объект одного класса строится из объектов другого класса. Это отношения между классами “быть частью”.

Композиционное включение - вид включения, при котором объект является неотъемлемой частью другого объекта.

Множественное включение – вид включения, при котором объект одновременно может входить в состав нескольких объектов. При множественном включении, в отличие от композиционного в классе хранится не сам объект, а только его адрес. Поэтому объект может одновременно входить в нескольких объекты класса.

Наследование - механизм, позволяющий написать производный класс-потомок на основе уже существующего (родительского, базового) класса. В класс–потомок добавляются поля и методы и базового класса. Это отношение между классами “есть“

Одиночное наследование – вид наследования, при котором один потомок наследуется от одного родителя.

Множественное наследование – вид наследования, при котором один потомок наследуется от нескольких родителей.

Виртуальное наследование – вид наследования, объявляемый с ключевым словом ***virtual*** . Его используют чтобы исключить дублирование косвенного базового класса.

Виртуальная функция - это функция, объявленная со спецификатором ***virtual***. Ее возможно подменить для базового класса в производном классе.

Чисто виртуальная функция – это виртуальная функция, не имеющая тела. Ее нельзя вызывать и она лишь является основой для подмены методами производных классов

Абстрактный класс - класс, который содержит хотя бы одну чистую виртуальную функцию. Его можно использовать только в качестве базового класса. Объекты абстрактного класса создавать нельзя.