top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

ОБЪЕКТНО -ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

Урок №12

Виртуальные функции

Содержание

Виртуальные функции

В объектно-ориентированном программировании виртуальной функцией называется функция-член класса, которая может быть переопределена в классах-наследниках так, что конкретная реализация функции для вызова будет определяться во время исполнения. Таким образом, программисту необязательно знать точный тип объекта для работы с ним через виртуальные функции: достаточно лишь знать, что объект принадлежит наследнику класса, в котором функция объявлена.

Виртуальные функции — один из важнейших приёмов реализации полиморфизма. Они позволяют создавать общий код, который может работать как с объектами базового класса, так и с объектами любого его класса-наследника. При этом базовый класс определяет способ работы с объектами и любые его наследники могут предоставлять конкретную реализацию этого способа.

Говоря иными словами — с помощью виртуальных функций объект сам определяет свое поведение (собственные действия). Объект в вашей программе в действительности может представлять не один класс, а множество различных классов, если они связаны механизмом наследования с общим базовым классом. Ну и поведение объектов этих классов в иерархии, конечно же будет разным.

А теперь, момент истины: согласно правилам C++, указатель на базовый класс может ссылаться на объект этого класса, а также на объект любого другого класса,

производного от базового. Понимание этого правила очень важно. Давайте рассмотрим простую иерархию неких классов A, B и C. A будет у нас базовым классом, B — наследуется от класса A, ну а C — наследуется от B. В программе объекты этих классов могут быть объявлены, например, таким образом.

```
A object_A; //объявление объекта типа A
B object_B; //объявление объекта типа B
C object_C; //объявление объекта типа C
```

Согласно данному правилу указатель типа A может ссылаться на любой из этих трех объектов. То есть, возможна следующая форма записи:

```
A *point_to_Object;          //объявим указатель
                             //на базовый класс
point_to_Object=&object_C; //присвоим указателю
                             //адрес объекта C
point_to_Object=&object_B; //присвоим указателю
                             //адрес объекта B
```

Несмотря на то, что указатель point_to_Object имеет тип A*, а не C* (или B*), он может ссылаться на объекты типа C (или B). А теперь рассмотрим вариант неправильной записи:

```
//объявим указатель на производный класс
B *point_to_Object;

//!ВНИМАНИЕ! нельзя присвоить указателю адрес
//базового объекта
point_to_Object=&object_A;
```

Примечание: Может быть правило будет более понятным, если вы будете думать об объекте C, как особом виде объекта A. Ну, например, пингвин — это особая разновидность птиц, и он все-таки остается птицей, хоть и не летает. Конечно, эта взаимосвязь объектов и указателей работает только в одном направлении. Объект типа C — особый вид объекта A, но вот объект A не является особым видом объекта C. Возвращаясь к пингвинам смело можно сказать, что если бы все птицы были особым видом пингвинов — они бы просто не умели летать!

Этот принцип становится особенно важным, когда в классах, связанных наследованием определяются виртуальные функции. Виртуальные функции имеют точно такой же вид и программируются так же, как и самые обычные функции. Только их объявление производится с ключевым словом virtual. Например, наш базовый класс A может объявить виртуальную функцию v_function().

```
class A
{
    public:
    virtual void v_function(); //функция описывает
                               //некое поведение
                               //класса A
};
```

Виртуальная функция может объявляться с параметрами, она может возвращать значение, как и любая другая функция. В классе может объявляться столько виртуальных функций, сколько вам потребуется. И на-

ходиться они могут в любой части класса — закрытой, открытой или защищенной. Если в классе B, порожденном от класса A нужно описать какое-то другое поведение, то можно объявить виртуальную функцию, названную опять-таки v_function().

```
class B: public A
{
    public:

    //замещающая функция описывает некое
    //новое поведение класса B
    virtual void v_function(void);

};
```

Когда в классе, подобном B, определяется виртуальная функция, имеющая одинаковое имя с виртуальной функцией класса-предка, такая функция называется замещающей. Виртуальная функция v_function() в B замещает виртуальную функцию с тем же именем в классе A.

Вернемся к указателю point_to_Object типа A*, который ссылается на объект object_B типа B*. Давайте внимательно посмотрим на оператор, который вызывает виртуальную функцию v_function() для объекта, на который указывает point_to_Object.

```
A *point_to_Object; //объявим указатель на базовый класс
point_to_Object=&object_B; //присвоим указателю
                           //адрес объекта B
point_to_Object->v_function(); //вызовем функцию
```

Указатель point_to_Object может хранить адрес объекта типа A или B. Значит во время выполнения этот оператор

point_to_Object-gt; v_function(); вызывает виртуальную функцию класса на объект которого он в данный момент ссылается. Если point_to_Object ссылается на объект типа A, вызывается функция, принадлежащая классу A. Если point_to_Object ссылается на объект типа B, вызывается функция, принадлежащая классу B. Итак, один и тот же оператор вызывает функцию класса адресуемого объекта. Это и есть действие, определяемое во время выполнения программы. Иначе говоря, реализация полиморфизма.

Применение. Предположим на минуту, что мы собираемся написать компьютерную игру. Пусть это будет игра, где используется оружие. Естественно, что его необходимо будет в игре создать. Причем, в нашей игре будет множество разновидностей оружия. Поэтому, вполне логичным решением будет — завести базовый класс. Скажем — так:

```
class Weapon
{
    public:

    ...//тут будут данные-члены, которыми может
    //описываться, например, как
    //толщина дубины, так и количество гранат
    //в гранатомете
    //эта часть для нас не важна

    virtual void Use1(void);//обычно — левая кнопка мыши
    virtual void Use2(void);//обычно — правая кнопка мыши

    ...//тут будут еще какие-то данные-члены и методы

};
```

Не вдаваясь в подробности этого класса, можно сказать, что самыми важными, пожалуй, будут функции Use1() и Use2(), которые описывают поведение (или применение) этого оружия. От этого класса можно порождать любые виды вооружения. Будут добавляться новые данные-члены (типа количества патронов, скорострельности, уровня энергии, длины лезвия и т.п.) и новые функции. А переопределяя функции Use1() и Use2(), мы будем описывать различие в применении оружия (для ножа это может быть удар и метание, для автомата — стрельба одиночными и очередями). Коллекцию вооружения надо где-то хранить. Видимо, проще всего организовать для этого массив указателей типа Weapon*. Для простоты предположим, что это глобальный массив Arms, на 10 видов оружия, и все указатели для начала инициализированы нулем.

```
Weapon *Arms[10]; //массив указателей на объекты
                  //типа Weapon
```

Создавая в начале программы динамические объекты-виды оружия, будем добавлять указатели на них в массив. Для того чтобы указать, какое оружие находится в пользовании, заведем переменную-индекс массива, значение которой будем изменять в зависимости от выбранного вида оружия.

```
int TypeOfWeapon;
```

В результате этих усилий, код, описывающий применение оружия в игре может выглядеть, например, так:

```
if("нажата левая кнопка мыши") Arms[TypeOfWeapon]->Use1();
else Arms[TypeOfWeapon]->Use2();
```

Вот и всё. Мы создали код, который описывает оружие еще до того, как решили, какие его типы будут использоваться. Более того. У нас вообще еще нет ни одного реального типа вооружения! Дополнительная (иногда очень важная) выгода — этот код можно будет скомпилировать отдельно и хранить в библиотеке. В дальнейшем вы (или другой программист) можете вывести новые классы из Weapon, сохранить их в массиве Arms[] и использовать. При этом не потребуется перекомпиляции вашего кода. Особо заметьте, что этот код не требует от вас точного задания типов данных объектов на которые ссылаются указатели Arms[], требуется только, чтобы они были производными от Weapon. Объекты определяют во время выполнения, какую функцию Use() им следует вызвать.

Некоторые особенности применения

А, теперь, давайте вернемся к началу — к классам A, B и C.

Класс C на данный момент стоит у нас в самом низу иерархии, в конце линии наследования. В классе C точно также можно определить замещающую виртуальную функцию. Причем применять ключевое слово virtual совсем необязательно, поскольку это конечный класс в линии наследования. Функция и так будет работать и выбираться как виртуальная. Но, если вам понадобится вывести некий класс D из класса C, да еще и изменить поведение функции v_function(), то тут как раз ничего

и не выйдет. Для этого в классе C функция v_function() должна быть объявлена, как virtual. Отсюда правило:

Ключевое слово virtual лучше не отбрасывать — вдруг пригодится?

В производном классе нельзя определять функцию с тем же именем и с тем же набором параметров, но с другим типом возвращаемого значения, чем у виртуальной функции базового класса. В этом случае произойдет ошибка на этапе компиляции программы.

Если в производном классе ввести функцию с тем же именем и типом возвращаемого значения, что и виртуальная функция базового класса, но с другим набором параметров, то эта функция производного класса уже не будет виртуальной. Даже если вы сопроводите ее ключевым словом virtual, она таковой не будет. В этом случае с помощью указателя на базовый класс при любом значении этого указателя будет выполняться обращение к функции базового класса. Вспомните правило о перегрузке функций! Это просто разные функции. У вас получится совсем другая виртуальная функция. Отсюда еще одно правило.

При замещении виртуальных функций требуется полное совпадение типов параметров, имен функций и типов возвращаемых значений в базовом и производном классах.

В конце нашего повествования добавим, что виртуальной функцией может быть только нестатическая компонентная функция класса. Виртуальной не может быть глобальная функция. Виртуальная функция может быть объявлена дружественной (friend) в другом классе.

Таблица виртуальных функций

Для каждого класса, имеющего хотя бы одну виртуальную функцию, создаётся таблица виртуальных функций. Каждый объект хранит указатель на таблицу своего класса. Для вызова виртуальной функции используется такой механизм: из объекта берётся указатель на соответствующую таблицу виртуальных функций, а из неё, по фиксированному смещению, — указатель на реализацию функции, используемую для данного класса. При использовании множественного наследования ситуация несколько усложняется за счёт того, что таблица виртуальных функций становится нелинейной.

Раннее и позднее связывание. Статический и динамический полиморфизм

Мы только что познакомились с виртуальными функциями и, теперь, нам необходимо рассмотреть такие понятия как раннее и позднее связывание. Приступим.

Сравним два подхода к покупке, к примеру, килограмма апельсинов. В первом случае мы заранее знаем, что нам надо купить 1 кг. апельсинов. Поэтому мы берем небольшой пакет, чётко определённое количество денег, чтобы хватило на этот килограмм. Во втором случае, мы, выходя из дома, не знаем что и как много нам надо купить. Поэтому мы берем машину (а вдруг будет много всего), запасаемся пакетами больших и малых размеров и берем как можно больше денег. Едем на рынок и выясняется, что надо купить только 1 кг. апельсинов.

Приведенный пример в определенной мере отражает смысл применения раннего и позднего связывания, соответственно. Очевидно, что для данного примера первый вариант оптимален. Во втором случае мы слишком много всего предусмотрели, но нам это не понадобилось. С другой стороны, если по дороге на рынок мы решим, что апельсины нам не нужны и решим купить 10 кг. яблок, то в первом случае мы уже не сможем этого сделать. Во втором же случае — легко.

А, теперь, рассмотрим этот пример с точки зрения программирования. При применении раннего связыва-

ния, мы как бы говорим компилятору: "Я точно знаю, чего я хочу. Поэтому статически связывай все вызовы функций". При применении механизма позднего связывания мы как бы говорим компилятору: "Я пока не знаю чего я хочу. Когда придет время, я сообщу что и как я хочу".

Таким образом, во время раннего связывания вызывающий и вызываемый методы связываются при первом удобном случае, обычно при компиляции.

При позднем связывании вызываемого метода и вызывающего метода они не могут быть связаны во время компиляции. Поэтому реализован специальный механизм, который определяет как будет происходить связывание вызываемого и вызывающего методов, когда вызов будет сделан фактически. Именно этот механизм и реализуют виртуальные функции.

Очевидно, что скорость и эффективность при раннем связывании выше, чем при использовании позднего связывания. В то же время, позднее связывание обеспечивает некоторую универсальность связывания.

И, напоследок, давайте чётко сформулируем определение того свойства ООП, реализацией которого является связывание:

Полиморфизм — переопределение наследником функций-членов базового класса. Полиморфизм бывает динамическим, когда вызываемая функция определяется во время выполнения (позднее связывание) и статическим (раннее связывание).

Абстрактные классы

Давайте продолжим рассмотрение использования виртуальных функций. На этот раз мы с вами разберем простой пример. Но, для начала, еще немного теории.

Чисто виртуальные функции

Слово "чисто" — в данном случае используется в контексте "пусто". Иными словами, чисто виртуальная функция — функция пустая. Синтаксис создания её таков:

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

Как видите, все отличие только в том, что появилась конструкция «=0», которая называется «чистый спецификатор». Чисто 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 чисто виртуальную функцию, называется абстрактным классом. Это связано с тем, 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.

> **Примечание:** Приведем пример. Все животные в своем поведении имеют такие функции, как «есть», «пить», «спать», «издавать звук». Имеет смысл определить базовый класс, в котором сразу объявить все эти функции и сделать их чисто виртуальными. А потом из этого класса выводить классы, описывающие конкретных животных (или виды), со своим специфичным поведением. А базовый класс при этом действительно получается абстрактным. Ведь он не описывает никакое более-менее конкретное животное (даже вид животных). Это может быть и рыба и птица....

По сравнению с обычными классами, абстрактные классы пользуются «ограниченными правами».

- Как и всякий класс, абстрактный класс может иметь явно определенный конструктор. Из конструктора можно вызывать методы класса. Но обращение из конструктора к чистым виртуальным функциям приведут к ошибкам во время выполнения программы.
- Как уже говорилось, невозможно создать объект абстрактного класса.
- Абстрактный класс нельзя применять для задания типа параметра функции, или в качестве типа возвращаемого значения.
- Его нельзя использовать при явном приведении типов. Зато можно определять ссылки и указатели на абстрактные классы.

Пример

Рассмотрим пример иерархии классов, описывающих неких животных. Для упрощения примера ограничимся в описании каждого животного его кличкой и издаваемым животным звуком. Ну, а основной возможностью программы будет вывод на экран списка кличек животных и представления издаваемых ими звуков.

```
#include <iostream>
#include <string.h>
using namespace std;

//абстрактный базовый класс
class Animal
{
    public:
        //кличка животного
        char Title[20];
        //простой конструктор
        Animal(char *t){
            strcpy(Title,t);
        }
        //чисто виртуальная функция
        virtual void speak()=0;
};

//класс лягушка
class Frog: public Animal
{
    public:
        Frog(char *Title): Animal(Title){};
        virtual void speak(){
            cout<<Title<<" say "<<"\'kwa-kwa\'\n";
        }
};
```

```
//класс собака
class Dog: public Animal
{
    public:
        Dog(char *Title): Animal(Title){};
        virtual void speak(){
            cout<<Title<<" say "<<"\'gav-gav\'\n";
        }
};

//класс кошка
class Cat: public Animal
{
    public:
        Cat(char *Title): Animal(Title){};
        virtual void speak(){
            cout<<Title<<" say "<<"\'myau-myau\'\n";
        }
};

//класс лев
class Lion: public Cat
{
    public:
        Lion(char *Title): Cat(Title) {};
        /*virtual void speak(){
            cout<<Title<<" say "<<"\'rrr-rrr\'\n";
        }*/

        /*virtual int speak(){
            cout<<Title<<" say "<<"\'rrr-rrr\'\n";
            return 0;
        }*/

        virtual void speak(int When){
            cout<<Title<<" say "<<"\'rrr-rrr\'\n";
        }
};
```

```
void main ()
{
    //объявим массив указателей на базовый класс Animal
    //и сразу его заполним указателями, создавая объекты
    //список животных
    Animal *animals[4] = {new Dog("Bob"),
                          new Cat("Murka"),
                          new Frog("Vasya"),
                          new Lion("King")};

    for(int k=0; k<4; k++)
        animals[k]->speak();
}
```

В качестве базового класса создан абстрактный класс Animal. Он имеет единственный член Title, описывающий кличку животного. В нем есть явно определенный конструктор, который присваивает животному его «имя». И, единственная чисто виртуальная функция speak(), которая описывает, какие звуки издает животное.

От этого класса отнаследованны все остальные. Кроме одного. Класс «лев» порожден от класса «кошка» (львы это тоже кошки). Это сделано для демонстрации тонкостей применения виртуальных функций. Но об этом классе немного позже.

Во всех производных классах также описана собственная замещающая виртуальная функция speak(), которая печатает на экран, звуки, которые издает конкретное животное.

В основном теле программы объявлен массив animals[4] указателей типа Animal*. И сразу же созданы динамические объекты классов и заполнен массив указателей. А в

цикле for() по указателю просто вызывается виртуальная функция speak().

Результат работы программы таков:

```
Bob say 'gav-gav'
Murka say 'myau-myau'
Vasya say 'kwa-kwa'
King say 'rrr-rrr'
```

Теперь обратимся к описанию класса Lion (лев). В нем вместо одной виртуальной функции speak() содержится сразу три. Две из них закомментированы.

Если вы закомментируете первую функцию, а раскомментируете вторую, то сможете проверить вариант, когда производится попытка соорудить виртуальную замещающую функцию с другим типом возвращаемого значения. В данном случае вторая (неправильная) функция возвращает тип int вместо типа void, который был у функции speak() в базовом классе. Попробуйте скомпилировать программу и произойдет ошибка на этапе компиляции.

Теперь, попробуйте раскомментировать третью функцию, а первые две закомментируйте. Компилятор на сей раз просто выдаст только предупреждение. Это тот самый случай, когда объявляется замещающая виртуальная функция с тем же самым типом возвращаемого значения, но с другим набором параметров. Посмотрим, что получилось:

```
Bob say 'gav-gav'
Murka say 'myau-myau'
Vasya say 'kwa-kwa'
King say 'myau-myau'
```

Лев у нас уже не рычит, а мяукает. Это потому, что работает уже совсем другая функция! То есть, раз в данном классе нет правильно определенной виртуальной функции, то по указателю вызывается виртуальная функция speak() из базового класса. А в нашем случае базовым для класса Lion является класс Cat. Вот лев и замяукал.

Виртуальный базовый класс

Иногда при множественном наследовании возникают ситуации, когда нужен определенный контроль над тем, как наследуются базовые классы. Рассмотрим пример.

```
 class A {

    public:
    int val;
};

class B : public A {...};
class C : public A {...};
class D : public B, public C{

    public:
    int Get_Val(){
        return val; //ошибка!
    }
};
```

В вышеописанном примере доступ к члену val неоднозначен. Компилятор не поймет на какую копию val ссылаться и поэтому просигнализирует ошибку. Для разрешения неоднозначности следует либо использовать оператор разрешения видимости, например, так:

```
int Get_Val(){

    return B::val;

 }
```

...либо использовать ***виртуальный базовый класс.*** Разберем на примере, как это можно сделать. Определим дерево иерархии следующим образом:

```
class A {
    public:
        int val;
};

class B : public virtual A {...};

class C : public virtual A {...};

class D : public B, public C {

    public:
        int Get_Val() {
            return val; //все работает корректно
        }
};
```

Объявление базового класса виртуальным заставляет компилятор принимать только одну копию базового класса в объявлении производного. Поэтому только одна копия члена val присутствует в классе D и оператора разрешения области видимости, для уточнения, не требуется. Виртуальные базовые классы используются только при множественном наследовании.

Вывод

Итак, виртуальный базовый класс нужен тогда, когда производный класс наследует два (или более) класса, каждый из которых сам унаследовал один и тот же базо-

вый класс. Без виртуального базового класса в последнем производном классе существовало бы две (или более) копии общего базового класса. Однако, благодаря тому, что исходный базовый класс делается виртуальным, в последнем производном классе представлена только одна копия базового.

Виртуальный деструктор

Этой примечательной темой мы продолжим рассмотрение использования виртуальных функций. Мы надеемся, что вы помните, как создаются и уничтожаются объекты классов и что такое конструкторы и деструкторы). Поэтому, давайте начнем изучение вопроса с рассмотрения простого примера.

Создадим некий класс, который может запоминать строковое значение. И пусть он у нас будет базовым классом (правда не абстрактным, так как это не важно в данном случае), из него мы будем выводить другие.

```
class Base
{
    private:
        char *sp1;
        int size;

    public:
        //конструктор
        Base(const char *S, int s){
            size=s;
            sp1=new char[size];
        }

        //деструктор
        ~Base(){
            cout<<"Base";
            delete[]sp1;
        }
};
```

Итак. Конструктор класса выделяет память для строки путем обращения к конструкции new и сохраняет адрес новой строки в указателе sp1. Деструктор класса освобождает эту память, когда объект класса Base выходит из области видимости. Далее, из базового класса выведем новый класс. Вот такой:

```
class Derived: public Base
{
    private:
        char *sp2;
        int size2;

    public:
        //конструктор
        Derived(const char *S1,int s1,
               const char *S2, int s2): Base(S1,s1){

            size2=s2;
            sp2=new char[size2];
        }

        //деструктор
        ~Derived(){

            cout<<"Derived";
            delete[]sp2;
        }
};
```

Этот класс сохраняет вторую строку, на которую ссылается его указатель sp2. Новый конструктор вызывает конструктор базового класса, передавая строку в базовый класс, а также выделяет память под вторую строку и со-

храняет адрес новой строки в указателе sp2. Деструктор этого класса освобождает эту память.

Теперь где-то в программе мы можем создать объект такого класса:

```
Derived MyStrings("string 1",9,"string 2",9);
```

Когда этот объект выйдет из области видимости, сначала вызовется деструктор класса Derived, а затем деструктор базового класса Base. Вся память будет аккуратно освобождена. Все по теории, все красиво.

Рассмотрим другой вариант. Предположим, что мы объявили указатель на базовый класс Base, но присвоили ему адрес объекта класса Derived. Это вполне допустимо, мы уже обсуждали этот вопрос ранее. То есть, это будет выглядеть в программе так:

```
Base *pBase; //указатель на базовый класс
pBase=new Derived("string 1",9,"string 2",9);
```

Что же произойдет, когда в программе будет удален объект, на который ссылается указатель pBase?

```
delete pBase;
```

Компилятор "видит", что указатель pBase должен ссылаться на объекты класса Base (откуда бы ему узнать, что именно присвоено этому указателю?). И вполне естественно программа вызовет только деструктор базового класса, и он удалит одну строку, но оставит в памяти другую. Ведь деструктор класса Derived не вызывался. Получается классическая утечка памяти). И, вот здесь, появляется виртуальный деструктор.

Все, что нужно сделать для исправления этой ситуации — это объявить в классах деструкторы с ключевым словом virtual. Таким образом, деструкторы будут выглядеть так:

```
virtual ~Base(){
    cout<<"Base";
    delete[]sp1;
}

virtual ~Derived(){
    cout<<"Derived";
    delete[]sp2;
}
```

Смысл таков. Поскольку деструкторы объявлены виртуальными, то их вызовы будут компоноваться уже во время выполнения программы. То есть, объекты сами будут определять, какой деструктор нужно вызвать. Поскольку наш указатель pBase на самом деле ссылается на объект класса Derived, то деструктор этого класса будет вызван, так же как и деструктор базового класса. Деструктор базового класса автоматически выполняется после деструктора производного класса.

Чисто виртуальный деструктор

Ну и, наконец, последняя порция информации о виртуальных функциях. Может так случиться, что в некоторых случаях будет очень удобно определить в классе чисто виртуальный деструктор.

Мы уже обсуждали сегодня чисто виртуальные функции. Они дают нам абстрактные классы, объект которых невозможно создать. Это основа для построения иерархии классов. Однако, иногда встречаются классы, которые имело бы смысл сделать абстрактными, но для этого в вашем распоряжении может не оказаться чисто виртуальных функций. Как быть? Решение не сложное. Объединим понятие чисто виртуальной функции и виртуального деструктора. Надо просто объявить в классе, который должен быть абстрактным, чисто виртуальный деструктор.

Приведем пример.

```
//абстрактный класс без виртуальных функций
class Something
{

    public:
        //а это чистый виртуальный деструктор
        virtual ~Something()=0;
};
```

Этот класс абстрактный, потому что включает в себя чисто виртуальную функцию (деструктор). Поскольку

деструктор виртуальный, то проблемы с вызовом деструктора в будущем возникнуть не должны. Все, что осталось сделать – это дать определение этого деструктора.

```
Something::~Something() {};
```

Это необходимо сделать, поскольку виртуальный деструктор работает таким образом, что вначале вызывается деструктор производного класса, а затем последовательно деструкторы классов, находящихся выше в цепи наследования, вплоть до базового абстрактного. Это означает, что компилятор будет генерировать вызов ~Something(), даже когда класс является абстрактным, поэтому тело функции надо определять обязательно. Если этого не сделать, компоновщик просто выдаст ошибку отсутствия символа. И сделать это все равно придется.

Несколько советов

Если у класса имеются виртуальные функции, имеет прямой смысл создать для него виртуальный деструктор. Даже если он и не требуется этому классу. Классы, которые будут потом произведены от него, может быть будут содержать деструкторы, которые должны вызываться соответствующим образом.

Если же класс не содержит виртуальных функций, то скорее всего он не предполагается к использованию в качестве базового. В таком случае определение в нем виртуального деструктора обычно неоправданно.

Примечание: Кстати! Конструкторы не могут быть виртуальными. Будьте бдительны!

Домашнее задание

1. Создать абстрактный базовый класс Employer (служащий) с чисто виртуальной функцией Print(). Создайте три производных класса: President, Manager, Worker. Переопределите функцию Print() для вывода информации, соответствующей каждому типу служащего.
2. Создать базовый класс список. Реализовать на базе списка стек и очередь с виртуальными функциями вставки и вытаскивания.
3. Создать абстрактный базовый класс с виртуальной функцией — площадь. Создать производные классы: прямоугольник, круг, прямоугольный треугольник, трапеция со своими функциями площади. Для проверки определить массив ссылок на абстрактный класс, которым присваиваются адреса различных объектов. Площадь трапеции: S=(a+b)h/2.
4. Создать класс живущих с местоположением. Определить наследуемые классы — лиса, кролик и трава. Лиса ест кролика. Кролик ест траву. Лиса может умереть — определен возраст. Кролик тоже может умереть. Кроме этого определен класс — отсутствие жизни. Если в окрестности имеется больше травы, чем кроликов, то трава остается, иначе трава съедена. Если лис слишком старый он может умереть. Если лис слишком много (больше 5 в окрестности), лисы

больше не появляются. Если кроликов меньше лис, то лис ест кролика.

5. Создать абстрактный базовый класс с виртуальной функцией — корни уравнения. Создать производные классы: класс линейных уравнений и класс квадратных уравнений. Определить функцию вычисления корней уравнений.