**Лабораторная работа № 2. “НАСЛЕДОВАНИЕ И**  
**ВИРТУАЛЬНЫЕ ФУ НКЦИИ”**

**Цель.** Получить практические навыки создания иерархии классов и  
использования статических компонентов класса.  
**Основное содержание работы.**  
Написать программу, в которой создается иерархия классов. Включить полиморфные объекты в связанный список, используя статические  
компоненты класса. Показать использование виртуальных функций. Краткие теоретические сведения.

**Статические члены класса.**  
Такие компоненты должны быть определены в классе, как **статические** (**static**). Статические данные классов не дублируются при создании  
объектов, т.е. каждый статический компонент существует в единственном  
экземпляре. Доступ к статическому компоненту возможен только после  
его инициализации. Для инициализации используется конструкция  
*тип имя\_класса : : имя\_данного инициализатор;*  
Например, int complex : : count = 0;  
Это предложение должно быть размещено в глобальной области после определения класса. Только при инициализации статическое данное  
класса получает память и становится доступным. Обращаться к статическому данному класса можно обычным образом через имя объекта  
*имя\_объекта.имя\_компонента*  
Но к статическим компонентам можно обращаться и тогда, когда  
объект класса еще не существует. Доступ к статическим компонентам возможен не только через имя объекта, но и через имя класса  
*имя\_класса : : имя\_компонента*  
Однако так можно обращаться только к *public* компонентам.  
Для обращения к *private* статической компоненте извне можно с помощью **статических компонентов-функций**. Эти функции можно вызвать через имя класса.  
*имя\_класса : : имя\_статической\_функции*

Пример (С++):

#include <iostream.h>  
class **TPoint**  
{  
double x,y;  
static int N; // статический компонент − данное : количество точек  
public:

TPoint(double x1 = 0.0,double y1 = 0.0){N++; x = x1; y = y1;}  
static int& count(){return N;} // статический компонент-функция  
};  
int TPoint : : N = 0; //инициализация статического компонента-данного  
void **main**(void)  
{

TPoint A(1.0,2.0);  
TPoint B(4.0,5.0);  
TPoint C(7.0,8.0);  
cout<< \nОпределены ”<<TPoint : : count()<<“точки”;

}

Пример (С#):

class TPoint

{

private double x, y;

private static int N = 0;

public TPoint(double x1, double y1)

{

x = x1;

y = y1;

N++;

}

public static int count()

{

return N;

}

}

class Program

{

static void Main(string[] args)

{

TPoint A = new TPoint(1.2, 1.4);

TPoint B = new TPoint(2.6, 5.7);

TPoint C = new TPoint(4.2, 7.1);

Console.WriteLine("Количесво объектов TPoint = {0}", TPoint.count());

Console.ReadKey();

}

}

**Указатель this.**  
Когда функция-член класса вызывается для обработки данных кон-  
кретного объекта, этой функции автоматически и неявно передается указа-  
тель на тот объект, для которого функция вызвана. Этот указатель имеет  
имя **this** и неявно определен в каждой функции класса следующим образом:  
*имя\_класса \*const this = адрес\_объекта*  
Указатель this является дополнительным скрытым параметром каждой нестатической компонентной функции. При входе в тело принадлежащей классу функции this инициализируется значением адреса того объекта,  
для которого вызвана функция. В результате этого объект становится дос-  
тупным внутри этой функции.  
В большинстве случаев использование **this** является неявным. В частности, каждое обращение к нестатической функции-члену класса неявно  
использует this для доступа к члену соответствующего объекта.  
Примером широко распространенного явного использования this являются операции со связанными списками.  
**Наследование.**  
Наследование − это механизм получения нового класса на основе  
уже существующего. Существующий класс может быть дополнен или изменен для создания нового класса.  
Существующие классы называются **базовыми**, а новые – **производными**. Производный класс наследует описание базового класса; затем он  
может быть изменен добавлением новых членов, изменением существующих функций-членов и изменением прав доступа. С помощью наследования может быть создана иерархия классов, которые совместно используют  
код и интерфейсы.

Наследуемые компоненты не перемещаются в производный класс, а  
остаются в базовых классах.  
В иерархии производный объект наследует разрешенные для наследования компоненты всех базовых объектов (*public, protected*).  
Допускается множественное наследование – возможность для некоторого класса наследовать компоненты нескольких никак не связанных  
между собой базовых классов. В иерархии классов соглашение относительно доступности компонентов класса следующее:  
***private*** *–* член класса может использоваться только функциями –  
членами данного класса и функциями – “друзьями” своего класса. В производном классе он недоступен.  
***protected*** – то же, что и ***private,*** но дополнительно член класса с данным атрибутом доступа может использоваться функциями-членами и  
функциями – “друзьями” классов, производных от данного.  
***public*** – член класса может использоваться любой функцией, которая  
является членом данного или производного класса, а также к ***public*** - членам возможен доступ извне через имя объекта.  
Следует иметь в виду, что объявление friend не является атрибутом  
доступа и не наследуется.  
Синтаксис определения производного класса:  
***class имя\_класса : список\_базовых\_классов***  
***{список\_компонентов\_класса};***  
В производном классе унаследованные компоненты получают статус  
доступа **private**, если новый класс определен с помощью ключевого слова  
**class**, и статус **public**, если с помощью **struct**.  
Явно изменить умалчиваемый статус доступа при наследовании  
можно с помощью атрибутов доступа – *private, protected и public*, которые  
указываются непосредственно перед именами базовых классов.  
**Конструкторы и деструкторы производных классов.**  
Поскольку конструкторы не наследуются, при создании производного класса наследуемые им данные-члены должны инициализироваться  
конструктором базового класса. Конструктор базового класса вызывается  
автоматически и выполняется до конструктора производного класса. Параметры конструктора базового класса указываются в определении конструктора производного класса. Таким образом происходит передача аргументов от конструктора производного класса конструктору базового  
класса.  
Например (C++).  
class Basis  
{

int a,b;  
public:

Basis(int x,int y){a=x;b=y;}  
};  
class Inherit:public Basis  
{

int sum;  
public:  
Inherit(int x,int y, int s):Basis(x,y){sum=s;}  
};

Например (C#).

class Basis

{

private int a, b;

public Basis(int a1, int b1)

{

a = a1;

b = b1;

}

}

class Inherit : Basis

{

private int sum;

public Inherit(int x, int y, int s) : base(x, y) {sum = s;}

}

Объекты класса конструируются снизу вверх: сначала базовый, потом компоненты-объекты (если они имеются), а потом сам производный  
класс. Таким образом, объект производного класса содержит в качестве  
подобъекта объект базового класса.  
Уничтожаются объекты в обратном порядке: сначала производный,  
потом его компоненты-объекты, а потом базовый объект.  
Таким образом, порядок уничтожения объекта противоположен по  
отношению к порядку его конструирования.

**Виртуальные функции.**  
К механизму виртуальных функций обращаются в тех случаях, когда  
в каждом производном классе требуется свой вариант некоторой компонентной функции. Классы, включающие такие функции, называются **полиморфными** и играют особую роль в ООП.  
Виртуальные функции предоставляют механизм **позднего** (**отложенного**) или **динамического связывания.** Любая нестатическая функция  
базового класса может быть сделана виртуальной, для чего используется  
ключевое слово **virtual**.  
Пример (C++).  
class base  
{  
public:  
**virtual** void print(){cout<<“\nbase”;}  
. . .  
};  
class **dir** : public base  
{  
public:  
void print(){cout<<“\ndir”;}  
};  
void **main**()  
{  
base B,\*bp = &B;  
dir D,\*dp = &D;  
base \*p = &D;  
bp –>print(); // base  
dp –>print(); // dir  
p –>print(); // dir  
}

Пример (C#).

class Basis

{

public virtual void print()

{

Console.WriteLine("Base class");

}

}

class Inherit : Basis

{

public override void print()

{

Console.WriteLine("Inherit class");

}

}

class Program

{

static void Main(string[] args)

{

Basis A = new Basis();

A.print();

Inherit B = new Inherit();

B.print();

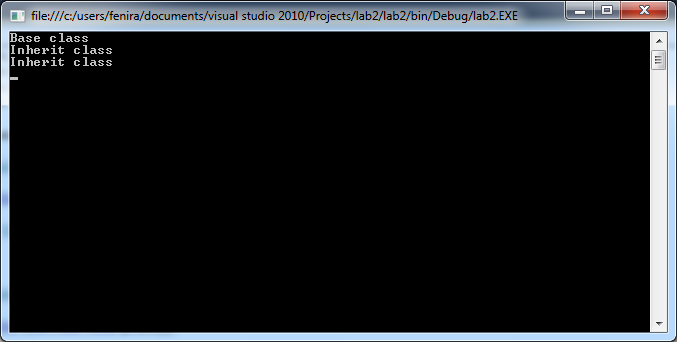
Basis C = new Inherit();

C.print();

Console.ReadKey();

}

}



Таким образом, интерпретация каждого вызова виртуальной функции через указатель на базовый класс зависит от значения этого указателя,  
т.е. от типа объекта, для которого выполняется вызов.  
Выбор того, какую виртуальную функцию вызвать, будет зависеть от  
типа объекта, на который фактически (в момент выполнения программы)  
направлен указатель, а не от типа указателя.  
Виртуальными могут быть только нестатические функции-члены.  
Виртуальность наследуется. После того как функция определена как  
виртуальная, ее повторное определение в производном классе (с тем же  
самым прототипом) создает в этом классе новую виртуальную функцию,  
причем спецификатор virtual может не использоваться.  
Конструкторы не могут быть виртуальными, в отличие от деструкторов. Практически каждый класс, имеющий виртуальную функцию, должен  
иметь виртуальный деструктор.  
**Абстрактные классы.**  
Абстрактным называется класс, в котором есть хотя бы одна чистая  
(пустая) виртуальная функция.  
Чистой виртуальной функцией называется компонентная функция,  
которая имеет следующее определение:  
***virtual тип имя\_функции (список\_формальных\_параметров) = 0;***  
Чистая виртуальная функция ничего не делает и недоступна для вызовов. Ее назначение – служить основой для подменяющих ее функций в  
производных классах. Абстрактный класс может использоваться только в  
качестве базового для производных классов.  
Механизм абстрактных классов разработан для представления общих  
понятий, которые в дальнейшем предполагается конкретизировать. При  
этом построение иерархии классов выполняется по следующей схеме. Во  
главе иерархии стоит абстрактный базовый класс. Он используется для наследования интерфейса. Производные классы будут конкретизировать и  
реализовать этот интерфейс. В абстрактном классе объявлены чистые виртуальные функции, которые по сути есть **абстрактные методы**.

Пример (C++).  
class Base{  
public:  
Base(); // конструктор по умолчанию  
Base(const Base&); // конструктор копирования  
virtual ~Base(); // виртуальный деструктор  
virtual void Show()=0; // чистая виртуальная функция  
// другие чистые виртуальные функции  
protected: // защищенные члены класса  
private:  
// часто остается пустым, иначе будет мешать будущим разработкам  
};  
class Derived: virtual public Base{  
public:  
Derived(); // конструктор по умолчанию  
Derived(const Derived&); // конструктор копирования  
Derived(параметры); // конструктор с параметрами  
virtual ~Derived(); // виртуальный деструктор  
void Show(); // переопределенная виртуальная функция  
// другие переопределенные виртуальные функции  
// другие перегруженные операции  
protected:  
// используется вместо private, если ожидается наследование  
private:  
// используется для деталей реализации  
};

Абстрактный метод может рассматриваться как обобщение *переопределения.* В обоих случаях поведение родительского класса изменяется  
для потомка. Для абстрактного метода, однако, поведение просто не определено. Любое поведение задается в производном классе.  
Одно из преимуществ абстрактного метода является чисто концептуальным: программист может мысленно наделить нужным действием абстракцию сколь угодно высокого уровня. Например, для геометрических фигур мы можем определить метод *Draw*, который их рисует: треугольник  
*TTriangle*, окружность *TCircle*, квадрат *TSquare.* Мы определим аналогичный метод и для абстрактного родительского класса *TGraphObject*. Однако  
такой метод не может выполнять полезную работу, поскольку в классе  
*TGraphObject* просто нет достаточной информации для рисования чего бы то ни было. Тем не менее, присутствие метода *Draw* позволяет связать  
функциональность (рисование) только один раз с классом *TGraphObject,* а  
не вводить три независимые концепции для подклассов *TTriangle*, *TCircle*,  
*TSquare.*  
Имеется и вторая, более актуальная причина использования абст-  
рактного метода. В объектно-ориентированных языках программирования  
со статическими типами данных, к которым относится и С++, программист  
может вызвать метод класса, только если компилятор может определить,  
что класс действительно имеет такой метод. Предположим, что программист хочет определить полиморфную переменную типа *TGraphObject,* которая будет в различные моменты времени содержать фигуры различного  
типа. Это допустимо для полиморфных объектов. Тем не менее компилятор разрешит использовать метод *Draw* для переменной, только если он  
сможет гарантировать, что в классе переменной имеется этот метод. Присоединение метода *Draw* к классу *TGraphObject* обеспечивает такую гарантию, даже если метод *Draw* для класса *TGraphObject* никогда не выполняется. Естественно, для того чтобы каждая фигура рисовалась посвоему, метод *Draw.*

**Порядок выполнения работы.**  
1. Определить иерархию классов (в соответствии с вариантом).  
2. Определить в классе статическую компоненту - указатель на начало связанного списка объектов и статическую функцию для просмотра  
списка.  
3. Реализовать классы.  
4. Написать демонстрационную программу, в которой создаются  
объекты различных классов и помещаются в список, после чего список  
просматривается.  
5. Сделать соответствующие методы не виртуальными и посмотреть,  
что будет.

6. Реализовать аналог на языке С# и Java.

**Содержание отчета.**  
1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.  
2. Постановка задачи. Следует дать конкретную постановку, т.е. указать, какие классы должны быть реализованы, какие должны быть в них  
конструкторы, компоненты-функции и т.д.  
3. Иерархия классов в виде графа.  
4. Определение пользовательских классов с комментариями.  
5. Реализация конструкторов с параметрами и деструктора.  
6. Реализация методов для добавления объектов в список.  
7. Реализация методов для просмотра списка.  
8. Листинг демонстрационной программы.  
9. Объяснение необходимости виртуальных функций. Следует показать, какие результаты будут в случае виртуальных и не виртуальных  
функций.  
**Приложение. Варианты заданий.**  
**Перечень классов:**  
**1) студент, преподаватель, персона, завкафедрой;**  
**2) служащий, персона, рабочий, инженер;**  
**3) рабочий, кадры, инженер, администрация;**  
**4) деталь, механизм, изделие, узел;**  
**5) организация, страховая компания, судостроительная компания, завод;**  
**6) журнал, книга, печатное издание, учебник;**  
**7) тест, экзамен, выпускной экзамен, испытание;**  
**8) место, область, город, мегаполис;**  
**9) игрушка, продукт, товар, молочный продукт;**  
**10) квитанция, накладная, документ, чек;**  
**11) автомобиль, поезд, транспортное средство, экспресс;**  
**12) двигатель, двигатель внутреннего сгорания, дизель, турбореактивный двигатель;**  
**13) республика, монархия, королевство, государство;**  
**14) млекопитающие, парнокопытные, птицы, животное;**  
**15) корабль, пароход, парусник, корвет.**