МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ГРАЖДАНСКОЙ АВИАЦИИ

Л.А. Надейкина

ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие
по выполнению практических работ ч. III

для студентов I-II курсов
направления 09.03.01
очной формы обучения

Москва
2019

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ (МГТУ ГА)»

Кафедра вычислительных машин, комплексов, систем и сетей

Л.А. Надейкина

ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие
по выполнению практических работ ч. III

для студентов I-II курсов
направления 09.03.01
очной формы обучения

Москва
2019

ББК 6Ф7.3
Н-17

Рецензент:
Черкасова Н.И. – канд. физ.-мат. наук

Надейкина Л.А.
Н-17 Программирование: учебно-методическое пособие по выполнению практических работ. / Л.А. Надейкина. – Воронеж: ООО «МИР», 2019. – 32 с.

Данное учебно-методическое пособие издается в соответствии с рабочей программой учебной дисциплины «Программирование» по учебному плану для студентов I-II курсов направления 09.03.01 очной формы обучения.

Рассмотрено и одобрено на заседании кафедры 24.09.2019 г. и методического совета 24.09.2019 г.

В авторской редакции.

Подписано в печать 07.10.2019 г.
Формат 60x84/16 Печ.л. 2 Усл. печ. л.
1,86 Заказ 526/2124 Тираж 80 экз.

Московский государственный технический университет ГА
125993 Москва, Кронштадтский бульвар, д.20

Отпечатано ООО «МИР»
394033, г. Воронеж, Ленинский пр-т 119А, лит. Я, оф. 215
Тел.: 8 (958) 649-53-31 Email: 89586495331@mail.ru

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ ЗАНЯТИЙ

1. КЛАССЫ. РАЗРАБОТКА ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ МЕХАНИЗМОВ ПЕРЕГРУЗКИ СТАНДАРТНЫХ ОПЕРАЦИЙ

1.1. Цель занятия

Формирование компетенции (ОПК-2.1.6), в виде получения навыков программирования, используя современные системы программирования, включая объектно-ориентированные. Формирование компетенций (ОПК-2.2.3) – в форме получения навыков использования основных концепций объектно-ориентированного программирования, понятия класса, объекта, поля, метода, конструкторов, деструкторов; (ОПК-2.3.1), в виде получения навыков владения языками процедурного и объектно-ориентированного программирования, навыками разработки и отладки программ не менее чем на одном из алгоритмических процедурных языков программирования высокого уровня; (5.3.4), в виде опыта работы с библиотеками функций, с библиотеками классов, с динамическими объектами. Формирование компетенции (ПК 1.1.2) в виде навыка использования современных систем программирования, включая объектно-ориентированные. Формирование компетенции (ПК-2.2.3) в виде опыта работы с многомерными статическими и динамическими массивами;

1.2. Методические указания по теме

Объектно-ориентированное программирование (ООП) представляет собой идеологию разработки программы, суть которой состоит в том, что объединяются некоторые совокупности данных и функций, связанных с этими данными в субстанции, называемые ***классами.*** Далее на основе разработанных классов можно создавать новые ***производные*** классы, в которые будут входить все средства базовых классов плюс дополнительные - свои. ООП основано на представлении программы в виде совокупности объектов, каждый из которых является экземпляром некоторого класса, а классы образуют иерархию наследования. Идея объединения кода и данных в объектах программы, взаимодействующие друг с другом, основывается на том, что объекты в программе отражают сущности реального мира, ведь каждый предмет или процесс обладает набором характеристик или отличительных черт, иными словами, свойствами и поведением.

Существенная черта современной программы — ее *сложность,* вытекающая из сложности предметной области. Способ управления сложными системами – это *декомпозиция* системы на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. В рамках структурного подхода декомпозиция понимается как разбиение алгоритма, когда каждый из модулей системы выполняет один из этапов общего процесса, ООП предлагает совершенно другой подход. В качестве *критерия декомпозиции* принимается принадлежность ее элементов к различным *абстракциям проблемной области.* Анализируя предметную область, вычленяются из нее отдельные объекты. Для каждого из этих объектов

определяются свойства, существенные для решения задачи. Затем каждому реальному объекту предметной области ставится в соответствие программный объект. И *объектно-ориентированная декомпозиция* оказалась более эффективным средством борьбы со сложностью процессов проектирования программных систем, чем *функциональная* декомпозиция в соответствие с критериями качества проекта.

Класс является абстрактным типом данных, определяемым пользователем, и представляет собой модель реального объекта в виде данных и функций для работы с ними.

Данные класса называются ***полями*** (по аналогии с полями структуры), а функции класса — ***методам***. Поля и методы называются *элементами класса (членами, компонентами класса).* Описание класса в первом приближении выглядит так:

class <имя>{
[private:]
<описание скрытых элементов>
public:
<описание доступных элементов>
} ;

Спецификаторы доступа ***private*** **и** ***public*** управляют видимостью элементов класса. Элементы, описанные после служебного слова *private*, видимы только внутри класса. Этот вид доступа принят в классе по умолчанию. Интерфейс класса описывается после спецификатора *public*. Действие любого спецификатора распространяется до следующего спецификатора или до конца класса. Можно задавать несколько секций *private* и *public*, порядок их следования значения не имеет.

Поля класса (компонентные данные):

- могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);
- могут быть описаны с модификатором const, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;
- могут использовать in-place initiliazer, который позволяет указывать умалчиваемое значение переменной для всех существующих конструкторов, включая пользовательские, непосредственно в контексте определения класса;
- могут быть описаны с модификатором static.

Определим класс ***book***:

```
class book  {
char title [256],  author[40] ;
float price;
public:
void  show_title (void) {cout<<title<<"\n ";};
float get_price (void) {return (price); };
```

void set_data();
};
void book::set_data() { /*тело метода*/ }

При объявлении в классе компонентных функций (методов) имеется выбор: можно определить функцию вне класса или непосредственно в теле класса. В приведенном классе содержится два определения методов и одно объявление (метод ***set_data***). Если тело метода определено внутри класса, он является ***встроенным*** (***inline***). Как правило, встроенными делают короткие методы. Если внутри класса записано только объявление (заголовок) метода, сам метод должен быть определен в другом месте программы с помощью операции доступа к области видимости (::).

Класс – это тип, который служит для определения переменных класса ***- объектов класса***.

Конструктор – это компонентная функция (метод класса), вызываемая автоматически при создании объекта класса и выполняющая необходимые инициализирующие действия. Формат определения конструктора в теле класса:

имя_ класса (список_параметров) инициализатор_конструктора
{операторы_тела_конструктора}

Особенности конструкторов:

1) Имя конструктора должно совпадать с именем класса.

2) Конструктор не может возвращать результат, даже тип ***void.***

3) Конструктор вызывается при определении объекта, или при размещении объекта в памяти с помощью операции ***new***.

4) Существуют два способа инициализации полей данных создаваемых объектов. Во – первых, можно в теле конструктора присваивать значения полям данных объектов. Эти значения обычно предоставляют параметры конструктора. Инициализатор, помещенный между списком параметров и телом конструктора, при этом опускается.

Второй способ предусматривает применение ***инициализатора*** (***"ctor-initializer"***). Инициализатор представляет собой список инициализаторов полей данных объекта, расположенный после списка параметров и отделенный от него двоеточием. Каждый инициализатор относится к не статическому полю данных и имеет вид:

имя_поля_данных (список выражений).

5) Класс может иметь *несколько конструкторов* с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки). Виды конструкторов: ***конструктор общего вида***, ***конструктор без параметров*** называется ***конструктором по-умолчанию*** (единственный, необязательный), ***конструктор копирования*** (единственный).

6) Если в классе программист не определил ни одного конструктора, то по умолчанию формируются конструктор без параметров и конструктор

копирования с прототипами соответственно (их автоматически добавляет компилятор):

T::T(); T::T(const T&);

7) Параметром конструктора не может быть его собственный объект, но может быть указатель или ссылка на него. Нельзя получить адрес конструктора.

8) Конструктор нельзя явно вызывать как обычный метод класса. Конструктор неявно вызывается при создании именованного объекта класса или безымянного, например, в следующих конструкциях:

имя_класса имя_объекта (аргументы конструктора);
имя_класса (аргументы конструктора);

Последний случай, например, используется при создании объекта в динамической памяти с использованием операции ***new***:

имя_класса*имя_указателя = new имя_класса (аргументы конструктора);

9) Конструкторы не наследуются.

10) Конструкторы нельзя описывать с модификаторами ***const, virtual и static.***

11) Конструкторы глобальных объектов вызываются до вызова функции ***main***. Локальные объекты создаются, как только становится активной область их действия. Конструктор вызывается и при создании временного объекта (например, при передаче объекта из функции).

Создание объектов в общем случае:

имя_класса имя_объекта (аргументы_конструктора);
указатель_на_объект = new имя_класса (аргументы_конструктора);

При отсутствии параметров в конструкторе или при использовании умалчиваемых значений параметров конструктора:

имя_класса имя_объекта;
указатель_на_объект = new имя_класса;

Обращение к полям и методам объекта:

1) с помощью квалифицированного имени:

имя_объекта. имя_ класса :: имя_компонента
имя_объекта. имя_класса :: имя_компонентной_функции(аргументы)

2) с помощью уточненного имени:

имя_объекта. имя_компонента
имя_объекта. имя_компонентной_функции(аргументы)

Не редко возникает необходимость использования каких-либо внешних по отношению к классу функций общего назначения для обработки закрытых данных класса, при этом эти функции не могут быть методами класса. Для этих целей в языке С++ предусмотрено объявление в классе дружественных функций**.**

Дружественной функцией класса называют функцию, которая не является компонентной функцией класса, но имеет доступ к защищенным и собственным компонентам класса.

Свойства функции:

1) должна быть описана в теле класса со спецификатором ***friend;***

2) не является компонентной функцией (методом) класса, в котором она определена как дружественная;

3) может быть глобальной:

class A { friend void f (…); … } ;
void f (…) {…};

4) может быть компонентной функцией (методом) другого ранее определенного класса; и тогда при описании в классе надо использовать полное имя функции, включающее имя класса, которому она принадлежит:

class A {… void f1 (…); …};
class B {… friend void A :: f1(…); …} ;

5) может быть дружественной по отношению к нескольким классам:

class A; ***// неполное определение класса A***
class B {…friend void f2 (A, B); …}; *//полное определение класса* ***B***
class A {…friend void f2 (A, B) ; …}; *//полное определение класса* ***A***
void f2 (A tip1, B tip2) {тело функции} *// определение функции* ***f2***

Перегрузка операций

Перегрузка операций — это *распространение действий стандартных операций на операнды, для которых эти операции не предполагались или предание стандартным операциям другого назначения.*

Если операнды операции (или хотя бы один) – объекты некоторого класса, то есть введенного пользователем типа, можно использовать специальную функцию, называемую ***"операция – функция" (operator function***), определяющую новое поведение операции.

Так стандартные операции *"***>>***"* и *"***<<***"* — это битовые сдвиги, однако при использовании их с объектами потоковых классов эти операции приобретают смысл "извлечения из потока и вставки данных в поток".

Формат определения операции-функции:

тип_возвращаемого_значения operator знак_операции
(спецификация_формальных_параметров)
{тело_операции_функции}

Перегрузку можно проводить следующими способами:

1) *операция - функция* является компонентной функцией класса;

2) *операция – функция* является глобальной функцией:

а) *операция - функция* является дружественной функцией класса;

б) *операция - функция* является недружественной функцией класса, но хотя бы один параметр функции (недружественной) был бы объектом или ссылкой на объект некоторого класса.

Особенности перегрузки стандартных операций С++:

1) С++ запрещает вводить операции с новым обозначением.

2) Нельзя изменить приоритет стандартной операции, перегрузив ее.
3) Нельзя изменять арность операции.
4) Перегрузка бинарной операции определяется либо как метод класса с одним параметром, либо как внешняя функция, возможно дружественная, с двумя параметрами. Выражение ***X <операция>Y*** означает вызовы**:**
X. operator<операция> (Y);// если операция-функция *- метод класса*
operator <операция> (X, Y) // если операция-функция *–внешняя*.
5) Перегрузка унарной операции определяется либо как компонентная функция без параметра, либо как внешняя функция, возможно дружественная, с одним параметром. Выражение ***<операция>X*** означает вызовы:
X. operator<операция> ( ) // если операция-функция *-метод класса*
operator<операция> (X) // если операция-функция *–внешняя*
6) В соответствие с семантикой бинарных операций, операции =, [ ], -> операции-функции с названием *operator=, operator [ ], operator ->* не могут быть внешними функциями, а должны быть нестатическими методами того класса, для которого они определены.
7) Унарные операции, имеющие префиксную **@*obj*** и постфиксную ***obj*@** формы (здесь знак @ обозначает знак операции, ***obj*** – объект некоторого класса), в соответствии со Стандартом языка выполняются по-разному. Префиксная операция выполняется в соответствие с общими правилами (пункт 5), а постфиксная операция-функция должна иметь дополнительный параметр, который компилятор обнуляет при выполнении операции.

Подробно описанный выше материал представлен в лекциях «Понятие класса, объекта», «Конструкторы и деструкторы», «Перегрузка операций».

Задача. Определить класс *"Комплексное число"* .

Закрытые члены класса: поля класса - два вещественных числа (действительная и мнимая части комплексного числа).

Открытые члены класса:

- конструктор класса с параметрами (двумя) с умалчиваемыми нулевыми значениями параметров;

- методы класса - перегрузка бинарной операции ‘ * ’

Дружественные функции:

Перегрузки операций включения в поток *" <<"* и извлечения данных из потока *" >>"* для объектов данного класса.

Создать несколько экземпляров класса и дать примеры использования перегруженных операций с объектами данного класса.

Листинг программы:

```
#include <iostream>
using namespace std;
class comp {
double re, im;
public
comp (double, double);      //конструктор с параметрами
comp operator * (comp&); // перегрузка умножения
```

```
friend ostream& operator << ( ostream &, comp&); //перегрузка операции вывода
friend istream& operator >> ( istream &, comp&); //перегрузка ввода
};
//----------определение конструктора с параметрами---------------
comp :: comp (double _re = 0, double _im =0 ) { re = _re;  im=_im; }
//-----------определение перегрузки операции умножения-----------
comp comp :: operator * (comp& z) {
return comp ( re * z.re – im * z.im, re * z.im – im * z.re) }
//----определение глобальной перегрузки операции вывода------------
ostream& operator << ( ostream out, comp&z){
out<<z.re;
if(z.im>0) out<<'+';
out << z.im << 'i' <<endl; return out;}
//----определение глобальной перегрузки операции ввода------------
istream& operator >> ( istream &in, comp&z) {
return ( in >> z.re >> z.im; ) }
int  main ( ) {
comp A(3.14, 4), B(5.2, 6.3), D=A*B, C=A*1.5;
cout << A << B << D << C;
return 0;}
```

Задача. Определить пользовательский класс ***"вектор".***

Закрытые члены класса

Поля класса: указатель на вещественный тип - адрес одномерного массива координат вектора, и целое число – количество координат.

Открытые члены класса

- конструктор класса с одним параметром – целое число (число координат) для создания неинициализированного вектора данной размерности;

- конструктор с двумя параметрами – указатель на вещественный тип для передачи массива координат вектора и целое значение-размерность вектора.

- конструктор копирования;

- деструктор для освобождения памяти, выделяемого на массив координат вектора;

- методы класса:

Перегрузки унарной операции ' - ', бинарной операции [ ].

Дружественные функции – перегрузка операции '*' объектов класса на целое число и операции включения в выходной поток '<<' и извлечения данных из потока *" >> "* для объектов данного класса .

Создать экземпляры класса, используя все три конструктора и дать примеры использования перегруженных операций. Листинг:

```
#include <iostream>
using namespace std;
// Определение класса
class vect {
double * p;
size_t  size;
public
```

```
vect  (size_t);                 //конструктор с одним параметром
vect  (double *, size_t ); //конструктор с двумя параметрами
vect  (const vect &);      //конструктор копирования
~vect  () { delete [ ] p; } // деструктор
vect operator – ( );        //перегрузка унарной операции ‘ - ‘
double& operator [ ] (size_t i )  { return p[i]; }//перегрузка операции квадратные //скобки
friend vect operator + (vect &, int );  //дружественная перегрузка сложения вектора
//с целым числом
friend ostream& operator << ( ostream &, vect&); //перегрузка операции вывода
friend istream& operator >> ( istream &, vect &); //перегрузка ввода
};
//----------определение конструкторов с параметрами----------
vect::vect (size_t n ) { size=n;
p=new double [size]; }

vect::vect (double* pm, size_t  n ) { size=n;
p=new double [size];
for (size_t  i=0; i < size; i++) p[i]=pm[i];
}
vect::vect (const vect& z ) { size=z.size; // конструктор копирования
p=new double [size];
for (size_t  i=0; i < size; i++) p[i]=z[i];
}
//-------------------------перегрузка унарного минуса------------------------------
vect vect :: operator – ( ){
vect  z (size);
 for (size_t  i=0; i < size; i++) z[i]=-p[i];
return z;}

//-----дружественные перегрузки операций бинарного плюса, ввода, вывода----

vect operator + (vect &a, int z ){
vect b ( a.size);
for (size_t  i=0; i <b.size; i++) b [i]= a [i]+z;
return b;}
ostream& operator << ( ostream &out, vect&z ) {
for (size_t  i=0; i < z.size; i++)
out<<z[i]<<” “;
out<<endl;
return out; }
istream& operator >> ( istream & in, vect &z) {
for (size_t  i=0; i < z.size; i++) in>> z[i];
return in; }

int  main ( ) {

double z[ ]={3,8,6,9,25,0,44,1,2};

size_t  n=sizeof(z)/sizeof (z[0]);

vect A(10), B(z, n), D=B+5;

for (size_t  i=0; i < A.size; i++) cin>>A[i];

vect C= -A;

cout << A << B <<D << C<<endl;

return 0; }
```

1.3. Задания для самостоятельного решения

1) Определить класс ***“матрица”.***

Закрытые члены класса: *Поля класса:* двойной указатель на вещественный тип - адрес массива указателей на строки матрицы (для преставления массива значений матрицы), и два целых числа – количество строк и столбцов матрицы.

Открытые члены класса:

- конструктор класса с двумя параметрами – два целых числа для создания неинициализированной матрицы данного размера;
- конструктор с тремя параметрами – двойной указатель на вещественный тип для передачи массива значений матрице-члену класса и два целых значения, определяющие в создаваемом объекте размеры матрицы.
- конструктор копирования;
- деструктор для освобождения памяти;
- методы класса:

Дружественные функции – перегрузка операции ‘*’ на целое число и операций включения в выходной поток ‘<<’ и ввода ‘>>’.

2) Определить пользовательский класс ***″комплексное число″,*** в котором кроме данных, конструкторов, деструктора, определить перегрузку арифметических операций (+, -, *) и операции извлечения >>и включения в поток << для пользовательского класса комплексного числа.

3) Определить пользовательский класс ***“строка”***. Определить конструкторы (умолчания, копирования, с параметрами), деструктор, перегрузку арифметической операции ‘+ ’, присваивания и операции включения в поток <<, преобразования типа для пользовательского класса ***“строка”.***

4) Написать класс для эффективной работы со строками, позволяющий форматировать и сравнивать строки, хранить в строках числовые значения и извлекать их. Для этого необходимо реализовать:

- перегруженные операции присваивания и конкатенации;
- операции сравнения и приведения типов;
- преобразование в число любого типа;
- форматный вывод строки.

5) Составить описание класса для объектов-векторов, задаваемых координатами концов в ***n-***мерном пространстве. Обеспечить операции сложения и вычитания векторов с получением нового вектора (суммы или разности), вычисления скалярного произведения двух векторов, длины вектора, косинуса угла между векторами.

6) Составить описание класса для определения одномерных массивов строк фиксированной длины. Предусмотреть возможность обращения к отдельным строкам массива по индексам, контроль выхода за пределы массива, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, вывод на экран элемента массива по заданному индексу и всего массива. Написать программу, демонстрирующую работу с этим классом.

1.4. Контрольные вопросы

1) Объектно-ориентированное программирование
2) Определение класса. Что может быть элементом класса. Поля и методы класса.
3) Доступность элементов класса.
4) Конструкторы и деструкторы. Свойства. Виды конструкторов.
5) Создание объектов класса статических и динамических
6) Статические элементы класса.
7) Друзья класса.
8) Перегрузка операция. Особенности механизма перегрузки стандартных операций С++.
9) Перегрузка операций присваивания, инкремента, декремента.
10) Перегрузка операций приведения типа, сравнения.
11) Перегрузка операции индексирования.

2. РАЗРАБОТКА ПРОГРАММ С ПРОСТЫМ И МНОЖЕСТВЕННЫМ НАСЛЕДОВАНИЕМ. РЕАЛИЗАЦИЯ ВИРТУАЛЬНЫХ БАЗОВЫХ КЛАССОВ. РАССМОТРЕНИЕ ЗАДАЧ С ПЕРЕГРУЗКОЙ ОПЕРАЦИЙ ПРИ НАСЛЕДОВАНИИ.

2.1. Цель занятия

Формирование компетенции (ОПК-5.3.4), получая навыки работы с библиотеками функций и классов. Формирование компетенций (ОПК 2.1.6, ПК-1.1.5), проводя разработки и отладки программ, используя современные системы программирования. Формирование компетенции (ПК-2.3.3), получая навыки владения принципами наследования, инкапсуляции, полиморфизма.

2.2. Методические указания по теме

Наследование – это процесс создания новых производных классов–наследников на основе уже существующих классов, называемых базовыми.

Допускается ***множественное наследование*** - возможность для некоторого класса наследовать компоненты нескольких базовых классов, несвязанных между собой.

Простейший синтаксис определения (спецификации) производного класса:

ключ_класса имя_производного_класса:
список_спецификаторов_базовых_классов
{поля_данных_и_методы_производного_класса};

где ***ключ_класса*** – одно из служебных слов ***struct, class.***

Спецификаторы базовых классов в списке разделены запятыми и могут быть представлены одним из следующих конструкций:

1) ***спецификатор_доступа имя_класса***
2) ***virtual спецификатор_доступа имя_класса***
3) ***спецификатор_доступа virtual имя_класса***

Производный класс, получая в наследство поля и методы базового класса, не перемещает к себе наследуемые компоненты, они остаются в базовом классе. Однако в каждый объект производного класса входит безымянный объект базового класса со всеми своими полями данных и методами.

При наследовании классов важную роль играет статус доступа компонентов базового класса и спецификатор доступа в определении производного класса.

Принято следующее соглашение:

1) собственные (***private***) методы и данные доступны только внутри класса, где они определены.

2) защищенные (***protected)*** компоненты доступны внутри класса, где они определены, и также доступны во всех производных классах.

3) общедоступные (***public***) компоненты класса – глобальны, т.е. доступны из любой точки программы.

На доступность унаследованного компонента влияют: 1) статус доступа компонента в базовом классе, 2) каким ключом класса вводится производный класс (***class*** или ***struct***) и 3) - какой спецификатор доступа стоит перед базовым классом в определении производного класса.

Рассмотрим пример наследования:

```
class A{                                    //базовый класс
int x;                                       // закрытое данное класса A
public:
A( int xx =0)  {  x=xx; cout<< " A ! ";}      // конструктор класса A
int GetX( ) {return x;}                     // функция-метод класса A
~A( ) {cout<<endl<< " DA "; }                  // деструктор класса A
};
class B :public A{                         // класс B, производный от A
int y;                                       // закрытое данное класса B
public :
B ( int yy=0 ) { y = yy ; cout<< " B ! "; }    // конструктор класса B
int GetY{ return y ;}                      // функция-метод класса B
~B( ) { cout<< " DB "; }                   // деструктор класса B
};
int main ( )   {
B  b (5) ;
cout<<endl<< " b = " << b.GetY ( );
cout<<endl<< " a = " << b.GetX ( );
return 0;}
```

В производный класс ***B*** включаются все данные и функции родителя ***A***, при этом данное ***x*** **-** недоступно для прямого обращения из объектов класса ***B***, но к нему можно обращаться из доступных компонентных функций класса ***A*****,** которые стали полноправными членами класса ***B*** и, следовательно, допустим вызов**: *b.GetX().***

Открытая функция ***GetY()*** производного класса ***B*** предоставила доступ к закрытому данному ***y*** производного класса.

В ***main*** создается объект производного класса ***B*,** в данное которого передается значение ***5***.

Данное базового класса (***x***) инициируется умалчиваемым значением ***0***.

При создании объекта производного класса сначала автоматически вызывается конструктор базового класса, который участвует в создании объекта базового класса, после этого вызывается конструктор производного класса, который проводит инициализацию полей данных производного класса.

В нашем случае сначала вызывается конструктор ***A()***, который по умолчанию инициирует ***x*** значением ***0***, а затем вызывается конструктор ***B(5)***, инициирующий ***y*** значением *5*.

Деструкторы автоматически вызываются в обратном порядке в соответствии с порядком уничтожения объекта. Сначала уничтожается то, что добавилось в производном классе, а затем и базовая часть.

Результат программы:

A! B!
b= 5
a= 0
DBDA

Если ***A***– базовый класс, а ***B***– производный класс, то их отношения можно представить графически (рис.1).

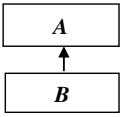


Рисунок 1. Отношения наследования классов

Итак, при наследовании:

- производный класс беспрепятственно обращается к доступным для него полям данных и методам базового класса;
- базовый класс не имеет доступа к полям данных и методам производного класса;
- в объект производного класса включаются поля данных и методы базового класса, то есть в объект производного класса входит экземпляр объекта базового класса
- в производном классе у наследуемой функции базового класса может быть "разная судьба", возможны три варианта:

1) Производный класс получает некоторую функцию ***A::f()*** без каких либо изменений и соответственно вызывает ее со своим экземпляром наряду со своими родными функциями.
2) Производный класс может заменить унаследованную функцию***A::f()*** своей ***B::f().*** У функции производного класса ***B*** та же спецификация

параметров, в этом случае говорят, что метод производного класса "экранирует" одноименный метод базового класса.

3) Производный класс может определить функцию с тем же именем, но с другой спецификацией параметров – имеет место перегрузка.

В случаях 2 и 3 алгоритм функции ***B::f()*** может быть абсолютно независим от алгоритма функции ***A::f()*** базового класса. Однако функция ***B::f()*** может дополнить действия функции ***A::f()***. Для этого в теле функции вызывается функция ***A::f()***, указывая область видимости: ***имя_класса::имя_компонента.***

Передача параметров в базовый класс.

Обычно при создании объекта производного класса требуется инициализировать данные не только производного, но и базового класса. Для этого в конструкторе производного класса надо явно вызвать конструктор базового класса. Рассмотрим этот вопрос на примере:

```
class A  {
int x1, x2 ;
public:
A ( int ax1, int ax2 ) { x1 = ax1 ; x2 = ax2; }
};
class B: public A {
int y;
public :
B ( int  _x1,    int _x2 ,      int  _ y): A ( _x1 , _x2) { y = _ y ; }
};
```

В конструкторе производного класса перечисляются в качестве параметров все переменные как производного, так и базового класса, которые надо инициализировать (с написанием их типов).

После знака ***':'*** проводится вызов конструктора базового класса с перечисленными выше параметрами для базового.

Таким образом, при вызове конструктора производного класса ***B()*,** ему необходимо передать три параметра, два из которых будут переданы конструктору базового класса ***A ().***

При следующем определении объекта производного класса: ***B b (2, 3, 4);*** значения ***2*** и ***3*** будут переданы полям данных базового класс ***x1*** и ***x2,*** а значение ***4*** – переменной производного класса ***y*.**

Для правильного построения конструктора производного класса необходимо иметь описание конструктора базового класса!

Задача. Определить базовый класс ***"точка на плоскости"****:*

Данные: координаты точки;

Методы: 1) конструктор с параметрами с умалчиваемыми значениями, 2) метод переместить точку в новое место (изменить координаты, два параметра с умалчиваемыми значениями), 3) метод вывода координат точки в стандартный поток.

Определить класс ***эллипс,*** производный от "точки":

Данные: полуоси ***эллипс*** вдоль осей координат.
Методы*:* 1) конструктор с параметрами, 2) метод вывода осей эллипса и как дополнение метода – вызов метода вывода центра эллипса, 3) метод возвращающий площадь эллипса.

Продемонстрировать в main возможности созданных классов.

Листинг программы:

```
class point {                  // Определение базового  класса "точка"
double x, y;                          //координаты точки
public:
point (double x1=0.0, double y1=0.0):     //конструктор с параметрами
x(x1), y(y1) { }
void move (double dx, double dy) {x+=dx; y+=dy;}    //перемещение точки
void display()
{cout<<"x= "<<x<<"\ty= "<<y<<endl;}          //вывод значения координат
};
class ellipse : public point { //Определение производного класса "эллипс":
double dmin, dmax;           //полуоси эллипса
public:
ellipse  (double dmin1, double dmax1, double x1, double y1 ):// конструктор
point(x1, y1), dmin( dmin1), dmax(dmax1) { }
void display()           //вывод координат центра и полуосей эллипса
{cout<<"Centre: ";
point:: display ();
cout<<"dmin = " <<dmin<<"\tdmax= "<<dmax<<endl;}
};
double square () {return 3.14159*dmin*dmax;}/     /вычисление площади эллипса
};
int main (){
ellipse elli (50, 120, 100, 200);
elli.display();                //метод класса ellipse
elli.move(-5.0, 5.0);       //наследуемый метод класса point
elli.display();
cout<<" square = "<<elli.square();
return 0;}
```

Класс является ***прямым базовым классом***, если он входит в список базовых классов при определении производного класса.

А если сам базовый класс является производным от некоторого родителя, причем этот родитель не входит в список базовых классов, то этот родитель является ***непрямым (косвенным) базовым классом*.**

Иерархию производных классов принято отображать в виде ***направленного ациклического графа (НАГ)***, где стрелкой изображают связь "производный от" (рис.2).

Последовательность объявления классов должна быть такой:

class A {…};
class B: public A {…};
class C: public B {…};

A- базовый класс, прямая база для класса ***B***

B **-** производный класс от ***A*** **–** прямая база для ***C***

C - производный класс с прямой базой ***B*** и косвенной - ***A***

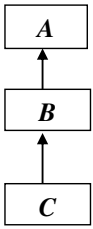


Рисунок 2. Иерархия наследования

Множественное наследование. Виртуальные базовые классы

Наличие в определении производного класса несколько прямых базовых классов называют ***множественным наследованием*****.** Иерархия множественного наследования представлена на рис. 3.

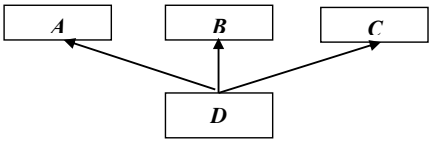


Рисунок 3. Схема множественного наследования

Определение классов:

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

Родители в определении производного класса перечисляются через запятую.

Как и в случае одиночного наследования, при создании объекта производного класса сначала конструируются объекты базовых классов (в том порядке, в котором базовые классы перечислены в объявлении производного), и лишь после этого составляется объект производного класса.

Деструкторы выполняются в обратном порядке.

При множественном наследовании никакой класс более одного раза не может быть прямым базовым классом. Однако класс более одного раза может быть непрямым базовым. Пример дублирования базового класса приведен на рис. 4.

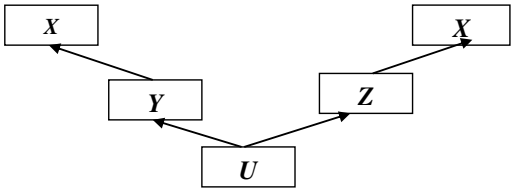


Рисунок 4. Дублирование непрямого базового класса

И соответственно объявление классов:

class X {long double x;};
class Y: public X {double y;};
class Z : public X {int z;};
class U:public Y, public Z {…};

Размеры объектов при дублировании базового класса:

sizeof(X)= 12 (long double)
sizeof(Y)= 20 (long double+ double)
sizeof(Z)= 16 (long double+ int)
sizeof(U)= 36 (long double+ double + long double+ int)

При наследовании, особенно множественном могут возникать неоднозначности при доступе к одноименным компонентам разных базовых классов. Способ устранения неоднозначностей – использование квалифицированных имен компонентов (включающих имена классов и операцию принадлежности ***"::").*** Для данного примера: ***U::Y::X::x*** или ***U::Z::X::x.***

Чтобы устранить дублирование объектов непрямого базового класса при множественном наследовании, этот базовый класс наследуется как ***виртуальный.*** Слово ***virtual*** помещается в спецификатор базового класса. Причем это делается не в объявлении самого базового класса **(*X*),** а в классах, производных от него. Пример графа приведен на рис. 5.

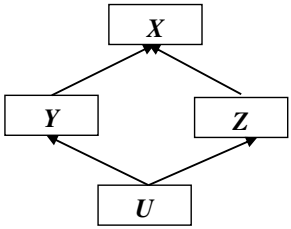


Рисунок 5. Виртуальное наследование классов

И соответствующее определение классов:

class X {long double x;};
class Y: virtual public X {double y;};
class Z: virtual public X {int z;};
class U :public Y, public Z {…};

При реализации виртуального наследования компилятор добавляет в производный класс в качестве данного указатель на виртуальный базовый класс. Это можно обнаружить, если определить размеры объектов классов с виртуальным базовым.

Размеры объектов без дублирования базового класса:

sizeof(X)= 12 (long double)

sizeof(Y)= 24 (long double+ double+ type*)
sizeof(Z)= 20 (long double+ int + type*)
sizeof(U)= 32 (long double+ double + int+ type*+ type*)

Итак, класс производный от виртуального включает:

1) объект (данные) базового класса;
2) указатель на объект базового класса:
3) данные производного класса.

Один и тот же класс при множественном наследовании может, включен в производный класс при непрямом наследовании и как виртуальный и как не виртуальный.

Задача. Определить классы и размеры объектов в байтах, если иерархия наследования представлена на рис. 6:

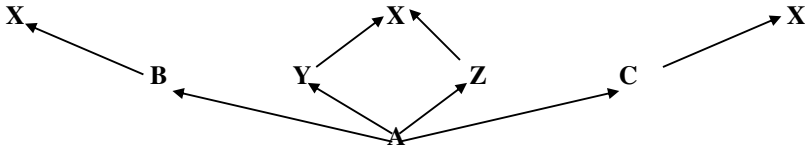


Рисунок 6. Виртуальное и не виртуальное наследование

Определение классов:

```
class X { long double} ;
class  Y : virtual public X { double y}  ;
class  Z : virtual public X { int z }  ;
class  B :  public X { int b }  ;
class  C :  public X { int c }  ;
class A : public B , public Y , public Z , public C {  . . . } ;
```

Размеры объектов:

```
sizeof(X)= 12 (long double)
sizeof(Y)= 24 (long double+ double+ type*)
sizeof(Z)= 20 (long double+ int + type*)
sizeof(B)= 16 (long double+ int)
sizeof(c)= 16 (long double+ int)
sizeof(A)= 64 (long double+ long double + long double +int + int + double + int+type*+ type*)
```

Объект класса **A** включает три экземпляра класса **X**: один виртуальный, общий для классов **Y** и **Z**, и два не виртуальных, относящихся к классам B и **C.**

Виртуальный класс может быть прямым родителем.

Задача. Определить классы, если иерархия наследования представлена на рис. 7:

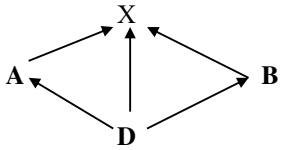


Рисунок 7. Пример иерархии наследования

Определение классов:

```
class X  { . . . } ;
class A : virtual public X { . . . } ;
class B : virtual public X { . . . } ;
class D : public A, public B , virtual public X  { . . . } ;
```

Производные классы располагаются ниже базовых. В том же порядке они должны располагаться в программе и так их объявления рассматривает компилятор.

Отметим, что виртуальность класса – это не свойство класса, а результат особенностей процедуры наследования.

Методы при наследовании классов

Методы производного класса могут быть

- унаследованы;
- явно определены программистом;
- созданы компилятором, независимо от действий программиста.

В зависимости от того какие методы явно определил программист в производном классе, компилятор может добавить или не добавить так называемые ***специальные методы***:

- конструктор умолчания
- конструктор копирования;
- деструктор;
- операцию - функцию присваивания,

которые сильно влияют на создание, копирование и уничтожение объектов.

Нередко указанные функции создаются, наследуются и вызываются неявно. Но при наследовании в производном классе в ряде случаев необходимо обращаться явно, даже к неявно определенным специальным базовым методам.

Конструкторы и операция присваивания не наследуются!

То есть если в производном классе не определена операция присваивания, компилятор создаст ее автоматически по своему усмотрению.

Если в производном классе нет конструкторов, то конструктор умолчания и конструктор копирования будут в нем определены компилятором.

Деструктор базового класса не может быть закрытым!

Он может быть только открытым или защищенным. Деструктор производного класса вызывается раньше, чем деструктор базового, он автоматически вызывает деструктор базового класса. При множественном наследовании деструкторы базовых классов вызываются в обратном порядке по отношению к перечислению базовых классов.

Перегрузка операций при наследовании

Нужный метод базового класса, а именно перегрузку операции присваивании (даже при отсутствии его явного определения), необходимо явно вызывать из соответствующего метода производного класса.

Рассмотрим перегрузку операции присваивания при наследовании.

Задача. Рассмотреть простевший случай наследования классов. Явно определить в производном классе операцию-функцию перегрузки присваивания.

Определение может быть таким:

```
class Bas{ protected: int b;};
class Dir: public Bas
{ double d;
Dir& operator= (const Dir&x);
};
Dir& Dir::operator= (const Dir&x)
{
If (this==&x) return *this;
//вызов перегрузки для базовой части
Bas::operator= (dynamic_cast<const Bas&>(x));
//приведение ссылки на объект производного класса к значению
// ссылки на объект базового класса
this ->d=x.d;
return *this;
};
```

Перегрузка операций ввода/вывода при наследовании.

Перегрузку операций ввода/вывода нельзя выполнить с помощью методов класса, чаще эти операции вводятся как дружественные.

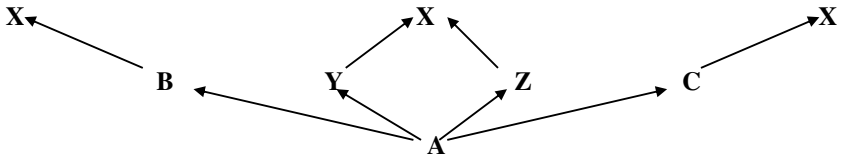
Задача. Показать в примере наследования классов обращение из дружественных функций производного класса к дружественным функциям базового класса.

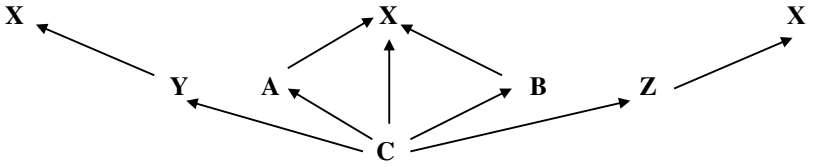
```
//определение базового класса
class Bas {
protected: int k;
friend istream& operator >>( istream&in, Bas&b);
friend ostream& operator <<( ostream&out, constBas&b);
};
istream& operator>>( istream&in, Bas&b)
{cout<<"k= ";
in>>b.k;
return in; }
ostream& operator<<( ostream&out, constBas&b)
{out<<"k= "<<b.k<<endl;
return out;
}
//определение производного класса
class Dir : Bas {
double z;
friend istream& operator>>( istream&in, Dir&d);
friend ostream& operator<<( ostream&out, const Dir&b);
};
istream& operator>>( istream&in, Dir&d)
{cout<< "Bas: ";
```

```
operator >> (in, dynamic_cast< Bas&> (d));
cout<< "Dir::z= ";
in>>d.z;
return in;
}
ostream& operator<<( ostream&out, const Dir&d)
{ out<< "Bas: ";
out <<dynamic_cast<const Bas&>(d);
out << "Dir::z= " <<d.z<<endl;
return out;
}
```

2.3. Задания для самостоятельного решения

1) Определить классы, если иерархии наследования представлены на рисунке:

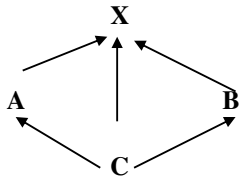




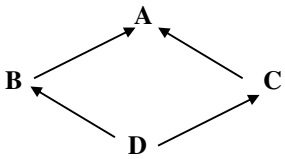
2) Определить иерархию классов: автомобиль, поезд, транспортное средство, экспресс и определить соответствующие классы. Перегрузить операции ввода/вывода в иерархии классов.

3) Определить классы, если иерархии наследования представлены на рисунке:

а)



б)



2.4. Контрольные вопросы

1) Наследование. Суть метода. Определение производного класса. Влияния формата определения производного классов и спецификаторов доступа на доступ наследуемых элементов.
2) Передача параметров конструктора в базовый класс. Конструкторы с инициализацией по умолчанию в иерархии классов.
3) Множественное наследование. Порядок вызовов конструкторов и деструкторов базовых классов при множественном наследовании.
4) Множественное наследование. Прямое и косвенное наследование.
5) Иерархия производных классов в виде графа (НАГ).
6) Дублирование объектов базового класса, косвенно наследуемого при множественном наследовании.
7) Виртуальные базовые классы. Примеры иерархии классов с участием виртуальных базовых классов.
8) Перегрузка операций при наследовании классов.

3. РАЗРАБОТКА АБСТРАКТНЫХ БАЗОВЫХ КЛАССОВ И РЕАЛИЗАЦИЯ МЕХАНИЗМА ВИРТУАЛЬНЫХ ФУНКЦИЙ ПРИ НАСЛЕДОВАНИИ. ИСПОЛЬЗОВАНИЕ СТАТИЧЕСКОГО КОМПОНЕНТА ДЛЯ СОЗДАНИЯ СПИСКА ОБЪЕКТОВ ПРОИЗВОДНЫХ КЛАССОВ.

3.1. Цель занятия

Формирование компетенции (ОПК-5.3.4), получая навыки работы с библиотеками функций и классов. Формирование компетенций (ОПК 2.1.6, ПК-1.1.5), проводя разработки и отладки программ, используя современные системы программирования. Формирование компетенций (ОПК-5.1.5), получая навыки работы с типовыми решениями, библиотеками программных модулей, шаблонов, классов, объектов, используемые при разработке программного обеспечения. Формирование компетенции (ПК-2.3.3), получая навыки владения принципами наследования, инкапсуляции, полиморфизма.

3.2. Методические указания по теме

Виртуальные функции включены в ***C++*** для обеспечения одного из видов ***полиморфизма***. Термин *полиморфизм* дословно означает "множество форм".

Механизм виртуальных функций позволяет с помощью *указателя с типом базового класса* обращаться к *переопределенным методам в производных классах*. Доступ к методам, переопределенным в производных классах, через указатель на базовый класс решается в C++ посредством использования *виртуальных функций.*

Чтобы сделать некоторый нестатический метод виртуальным, надо в базовом классе предварить его заголовок спецификатором ***virtual***, метод становится ***виртуальной функцией***.

Если эту функцию переопределить в производном классе даже без спецификатора ***virtual,*** в производном классе также создается *виртуальная функция.* Виртуальность этих функций (или полиморфизм) проявляется в том, что выбор требуемой из множества определенных в иерархии классов виртуальных функций с одним именем осуществляется не во время компиляции программы, а динамически, по конкретному значению указателя базового типа, с помощью которого и вызывается функция. Какая функция будет вызываться зависит от типа указателя и типа того объекта, адрес которого присвоен указателю.

Виртуальность функций проявляется только в том случае, если она вызывается через указатель или ссылку на базовый класс!

Указатель на базовый класс может принимать конкретные значения.

Если значение указателя к моменту вызова функции есть адрес объекта базового класса, вызывается вариант функции из базового класса.

Если этот указатель имеет значение адреса объект производного класса (фактически указывает на данные базового класса в объекте производного класса), то вызывается вариант функции из производного класса.

Решение о том, какую функцию использовать, не может быть принято во время компиляции, поскольку компилятор не знает, с объектом какого типа собирается работать пользователь.

Компилятор должен генерировать код, который позволяет выбирать нужный виртуальный метод во время работы программы. Такой процесс называется динамическим (или поздним) связыванием.

Это означает, что на этапе компиляции компилятор не определяет, какой из методов должен быть вызван, а передает ответственность программе, которая принимает решение на этапе выполнения, когда уже точно известно, каков тип объекта, на который указывает наш указатель. Все сказанное относится также к вызову методов ***по ссылке*** на базовый класс.

Для реализации динамического связывания компилятор создает ***таблицу виртуальных функций***. Обычно таблица виртуальных функций – это массив или связанный список с указателями на виртуальные функции (***virtual table pointer – vptr)***.

Такой массив или список компилятор автоматически включает в реализацию каждого класса, в котором определены или унаследованы виртуальные функции, то есть каждый класс имеет собственную таблицу, элементы такой таблицы адресуют коды виртуальных функций именно этого класса.

Каждый объект такого класса включает дополнительный член - указатель (***pointer***) на таблицу виртуальных функций класса.

Механизм идентификации типа во время выполнения программы (**RTTI**) позволяет определять, на какой тип в текущий момент времени ссылается указатель.

Конструкторы не могут быть виртуальными. Производный класс не наследует конструкторы базового класса, и нет смысла делать их виртуальными.

Деструкторы должны быть виртуальными, за исключением тех классов, которые не используются в качестве базовых. Если базовый класс содержит хотя бы один виртуальный метод, то рекомендуется всегда снабжать этот класс виртуальным деструктором, даже если он ничего не делает. Наличие такого виртуального деструктора предотвратит некорректное удаление объектов производного класса, адресуемых через указатель на базовый класс, так как в противном случае деструктор производного класса вызван не будет.

Дружественные функции не могут быть виртуальными функциями: ведь они не являются членами класса, а виртуальными функциями могут быть только члены класса.

Пустая и чистая виртуальные функции. Абстрактный класс.

Реально в конкретных задачах вызываются лишь функции производных классов. "Исходная" виртуальная функция базового класса часто нужна только для того, чтобы в производных классах было, что замещать.

Можно объявить в базовом классе ***чистую виртуальную функцию***, которая вводится с помощью такого определения:

virtual тип имя (спецификация параметров) = 0;

Это абстрактная функция, предназначена для замещения в производных класса, в которых она и наполнится разумным содержанием. Объявление такой функции в базовом классе носит абстрактный характер для указания на виртуальность функций с данным именем.

Класс, в котором есть хотя бы одна чистая виртуальная функция, называется ***абстрактным классом.***

Основные свойства абстрактного класса:

- Невозможно создать самостоятельных объектов абстрактного класса.

- Используется только в качестве базового класса.

- Если в производном классе от абстрактного базового класса происходит замещение чистой виртуальной функции, то производный класс не является абстрактным. Если замещение не производится, то производный класс также является абстрактным.

- Абстрактные классы предназначены для представления общих понятий, которые предстоит конкретизировать в производных классах.

- Абстрактный класс может иметь поля данных, а также методы, отличные от виртуальных и явно определенный конструктор.

- Конструктор абстрактного класса не может использоваться для создания объектов, но может использоваться при наследовании. С его помощью инициализируются поля данных абстрактного базового класса, входящие в объект производного класса.

- Указатель на абстрактный класс может использоваться в качестве формального параметра. Соответствующий фактический параметр должен иметь тип указателя на объекты производного (уже не абстрактного) класса.

Задача. Определить абстрактного класса класс "фигура на плоскости", производный от не абстрактного класса "точка на плоскости". Определим производные класс "эллипс" от абстрактного класса класс "фигура на плоскости".

//Определение базового класса *point в файле point.h*:

```
class point {
protected:
double x, y;
public:
point (double x1=0.0, double y1=0.0): x(x1), y(y1) { }
void move (double x1=0.0, double y1=0.0) {x=x1; y=y1;}
};
```

// Определение в файле *figure.h* абстрактного производного класса *figure*:

```
#include "point.h"   // включаем определение базового класса
#include <string>
class figure :public point {
protected:
double dx, dy; // "габариты" фигуры
public:
//Определение конструктора
figure (double x1=0.0, double y1=0.0, double dx1=0.0, double dy1=0.0):
point (x1, y1), dx(dx1), dy(dy1) {}
// изменить на заданную величину габариты
void grow (double k) {dx +=k; dy +=k;
}
// вычислить площадь еще неизвестной фигуры
virtual double area() = 0; // чистая виртуальная функция
virtual string className() = 0; // чистая виртуальная функция
friend ostream& operator << (ostream& out, figure& );
};
//перегрузка операции вывода "фигуры"
ostream& operator << (ostream& out, figure &fig )
{out<<fig.className() << ":\t center: x= " <<fig.x<< ",\ty= "<<fig.y;
out<< ";\n\tdx= "<<fig.dx<< ",\tdy= "<<fig.dy;
<<";\tarea = "<<fig.area();
return out;
}
```

//Определение производного класса "эллипс":

```
//файл real_figures.h
#include "figure.h"
struct ellipse: public figure {
ellipse(double x1=0.0, double y1=0.0, double dx1=0.0, double dy1=0.0):
figure(x1, y1, dx1, dy1){ }
virtual double area ()
{
```

```
return ((dx/2)*(dy/2)*3.14159);
}
string className() {return string("ellipse")}
};
//      Программа:
#include <iostream>
using namespace std;
#include "real_figures.h"
int main() {
ellipse A(10.0, 8.0, 20.0, 20.0), B;
cout<<"Object A:\n "<<A<<endl;
A.move(5.0,5.0);
cout<<" A.move(5.0,5.0):\n" <<A<<endl;
A.grow(-18.0);
//cout<<" A.grow(-18.0):\n" <<A<<endl;
B=A;
cout<<"Object B:\n "<<B<<endl;
return 0;
}
```

Результат выполнения программы:

```
Object A:
ellipse:      centre: x=10,      y=8
dx=20      dy=20      area=314.159
A.move(5.0,5.0):
ellipse:      centre: x=5,      y=5
dx=20      dy=20      area=314.159
Object B:
ellipse:      centre: x=5,      y=5
dx=2      dy=2      area=3.14159
```

Использование статического компонента для создания списка объектов производных классов.

Задача. Определить иерархию классов: место, область, город, мегаполис.

- Из перечисленных классов выбрать один, который будет стоять во главе иерархии. Это абстрактный класс. Имеет чистый виртуальный метод ***show***.

- Реализовать классы с определением виртуальных функций просмотра данных класса.

- Включать полиморфные объекты в связанный список, используя соответствующий метод базового класса ***add.***

- Определить в базовом классе указатель на начало связанного списка объектов (статический компонент) и статическую функцию для просмотра списка, а также указатель на следующий элемент списка.

- В базовом классе должен быть определен виртуальный деструктор.

- В программе создаются объекты производных классов и помещаются в список, после чего список просматривается.

Иерархия наследования классов представлена на рис. 9

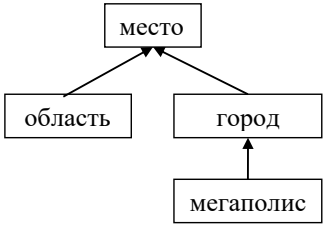


Рисунок 9. Иерархии наследования

Листинг программы:

```
#include <iostream>
using namespace std;
// Абстрактный класс "место", является базовым для других классов
class Place {
const char* name;   //название места, у любого места обязательно есть название.
// Указатель на следующий элемент в общем списке для текущего объекта
Place* next;
public:
/* У класса явно определен один конструктор для инициализации названия, все
конструкторы по-умолчанию не будут автоматически определяться
компилятором, то есть будут отсутствовать. Однако, операция присваивания
будет определена компилятором и ее следует явно запретить, скрыв в область
private, поскольку данный класс не предусматривает разумной возможности
 копирования экземпляров.*/
Place (const char* place_name) { //конструктор
name = place_name;
next = 0;
add_to_list();
}
const char* get_name()
{ return name;}
/* Отсутствие виртуального деструктора у базового класса
 может привести к серьезным проблемам (утечке памяти) при
вызове операции delete к указателю Place.
Достаточно объявить хотя бы пустой виртуальный деструктор. */
virtual ~Place() { }
//Чисто виртуальная функция show()
virtual void  show (int pos) = 0;
//Обход списка
static void print_list()  {
Place* now = list_begin;
int pos = 0;
while (now != 0) {
pos++;
now->show (pos);
// перемещаемся к следующему элементу списка
```

```
now = now->next;
}
}
// Статический метод - удаление всех элементов списка, освобождение памяти.
static void cleanup_list() {
Place* now = list_begin;
while (now != 0)
{Place* x = now;
now = now->next;
delete x; }
list_begin = 0;
}
// Статический (общий) указатель на начало однонаправленного списка
static Place* list_begin;
private:
//Функция добавления элемента в список.
void add_to_list() {
 if (list_begin == 0)
{ // Если в списке еще нет ни одного элемента.
list_begin = this;
}
else {
// Если в списке есть элементы, то добавим новый в конец.
 // Ищем указатель на конец списка
Place* last = list_begin;
while (last->next != 0) last = last->next;
// Добавляем новый элемент в конец
last->next = this;
}
}
// Присваивание для класса запрещено путем помещения в область private:
Place& operator= (const Place&);
};
// Класс "город", является производным от "место"
class City : public Place {
public:
City(const char* name) : Place(name) {}
virtual void show (int pos){
cout << pos << ": this is a city " << get_name() << endl;
}
};
// Класс "мегаполис", является производным от "город"
class Megapolis: public City {
public:
Megapolis (const char* name): City (name) { }
virtual void show (int pos) {
cout << pos << ": this is a megapolis " << get_name() << endl;
cout << "Any megapolis is also a city, but not every city is a megapolis" << endl;
}
};
```

```
// Класс "область", является производным от "место"
class State : public Place {
public:
State(const char* name) : Place(name) {}
virtual void show(int pos) {
cout << pos << ": this is a state " << get_name() << endl;
}
};
Place* Place::list_begin = 0;
int main() {
setlocale(LC_CTYPE, "Russian");
new State("Московская область");
new Megapolis("Москва");
new City("Чехов");
new City("Псков");
new Megapolis("Санкт-Петербург");
new State("Ленинградская область");
Place::print_list();
cout << "Очистка списка" << endl;
cout << endl;
Place::cleanup_list();
new State("Штат Нью-Йорк");
new Megapolis("Нью-Йорк");
new City("Санта-фе");
new City("Спрингфилд");
new Megapolis("Лос-Анджелес");
new State("Штат Калифорния");
new City("Сан-Хосе");
new City("Фресно");
Place::print_list();
cout << "Очистка списка" << endl;
Place::cleanup_list();
system("pause");
return 0;
}
```

3.3. Задания для самостоятельного решения

1) Определить класс **''товар'',** в котором кроме обычных характеристик товара (наименование, закупочная цена, …) имеется статический компонент – торговая наценка товара (в %), расположенный в закрытой области класса. Поэтому необходимо в открытой области класса объявить статическую функцию для изменения торговой наценки. Объявить методы класса, необходимые для создания объектов, копирования, для ввода/вывода данных. Продемонстрировать работу со статическими компонентами класса. Инициализация статического данного, использование его для расчета розничной цены товара, при создании объекта, изменение значения статического данного с помощью вызова статической компонентной функции. Создавайте статические и динамические объекты и массивы объектов.

2) Применить указатель ***this*** при создании односвязанного списка объектов класса, например, ***‘студент’***. Класс кроме данных о студенте должен включать указатель на следующий элемент списка, статический компонент – указатель на начало списка, статические функции для вывода списка целиком и деструктор для его уничтожения. Компонентные функции: добавление объекта в список в алфавитном порядке, извлечение из списка.

3) Определить иерархию классов: республика, монархия, королевство, государство. Из перечисленных классов выбрать один, который будет стоять во главе иерархии. Это абстрактный класс. Имеет чистый виртуальный метод ***show***. Реализовать классы с определением виртуальных функций просмотра данных класса. При создании включать полиморфные объекты в связанный список, используя соответствующий метод базового класса ***add.***

Определить в базовом классе указатель на начало связанного списка объектов (статический компонент) и статическую функцию для просмотра списка, а также указатель на следующий элемент списка. Базовый класс должен иметь виртуальный деструктор.

В программе создаются объекты производных классов и помещаются в список, после чего список просматривается.

3.4. Контрольные вопросы

1) Полиморфизм. Понятие виртуальной функции. Полиморфные классы.
2) Механизм виртуальных функций. Расширение интерфейса производного класса.
3) Режимы раннего и позднего связывания.
4) Конструкторы и деструкторы в иерархии наследования, реализуемого с помощью виртуальных функций. Дружественные функции.
5) Пустая и чистая виртуальные функции.
6) Абстрактный класс, назначение, свойства.
7) Преобразование типов указателей в иерархии классов.
8) Присваивание при наследовании с виртуальными функциями

СПИСОК ЛИТЕРАТУРЫ

1. Надейкина Л.А. Программирование. Часть 2: учебное пособие – М.:МГТУ ГА, 2017, 84с.

2. Страуструп Б. Программирование: принципы и практика использования C++ 2-е издание, ISBN-13: 978-0321992789, Бином, Невский диалект, 2013,1312 с.

3. Прата С. Язык программирования С. Лекции и упражнения. ISBN: 978-5-8459-1950-2 (рус.), Вильямс, 2015, 928 с.

4. Подбельский В.В. Стандартный Си++. М.: Финансы и статистика, 2008, 688с.