БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ РАДИОФИЗИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ
КАФЕДРА ИНФОРМАТИКИ И КОМПЬЮТЕРНЫХ СИСТЕМ

Н. В. Серикова

ПРАКТИЧЕСКОЕ РУКОВОДСТВО
к лабораторному практикуму

КЛАССЫ. НАСЛЕДОВАНИЕ. ПОЛИМОРФИЗМ

по дисциплине

«ПРОГРАММИРОВАНИЕ НА C++»

2024
МИНСК

Практическое руководство к лабораторному практикуму «КЛАССЫ. НАСЛЕДОВАНИЕ. ПОЛИМОРФИЗМ» по дисциплине «ПРОГРАММИРОВАНИЕ НА С++» предназначено для студентов, изучающих базовый курс программирования на языке С++, специальностей «Компьютерная безопасность», «Прикладная информатика», «Радиофизика».

Руководство содержит некоторый справочный материал, примеры решения типовых задач с комментариями.

Автор будет признателен всем, кто поделится своими соображениями по совершенствованию данного пособия.

Возможные предложения и замечания можно присылать по адресу:
E-mail: *Serikova@bsu.by*

ОГЛАВЛЕНИЕ

НАСЛЕДОВАНИЕ

Наследование – принцип ООП, который позволяет на основе одного класса объявить другой класс. Наследование используется для поддержки ООП.

При наследовании:

- **класс-предок** (родитель, порождающий класс);
- **класс-наследник** (потомок, порожденный класс).

В C++ порождающий класс называется **базовым,** а порожденный – **производным**.

Каждый производный класс может выступать базовым классом.

Отношения между родительским классом и его потомками называются **иерархией наследования**. Глубина наследования не ограничена.

Простым называется наследование, при котором производный класс имеет только одного родителя.

Множественное наследование – способность класса наследовать характеристики более чем одного базового класса.

Объекту базового класса можно присвоить **значение другого объекта** этого же класса, а также **любого объекта-потомка**.

Указатель и ссылка на базовый класс совместимы по типам соответственно с указателем и ссылкой на производный класс.

СИНТАКСИС ОБЪЯВЛЕНИЯ ПРОИЗВОДНОГО КЛАССА

Простое наследование:

```
class имя_производного_Класса:
            [спецификатор наследования] имя_базового класса
            {
                тело класса
            };
```

Множественное наследование:

```
class имя_производного_Класса:
      [спецификатор наследования1] имя_базового класса1,
      [спецификатор наследования2] имя_базового класса2,
                                   ...
   {   тело класса  };
```

Спецификатор наследования определяет доступность элементов базового класса в производном классе.

Если спецификатор наследования не указан, то по умолчанию базовый класс наследуется как закрытый.

Имеются 3 спецификатора наследования:

- **Public** (открытый)
- **Private** (закрытый)

- **Protected** (защищенный)

Если спецификатор наследования ***public,*** все открытые компоненты и функции-члены интерфейса базового класса открыты и в производном классе, а закрытые – закрыты и недоступны.

Если спецификатор наследования ***private,*** все открытые компоненты и функции-члены интерфейса базового класса закрыты в производном классе, также как и закрытые – закрыты и недоступны. Личные (private) члены базового класса доступны в производном классе только через вызов соответствующих функций-членов базового класса.

Спецификатор наследования ***protected*** эквивалентен спецификатору ***private*** с единственным исключением: защищенные члены базового класса доступны для членов всех производных классов этого базового класса, вне базового или производных классов защищенные члены недоступны.

ДОСТУП К ЭЛЕМЕНТАМ БАЗОВОГО КЛАССА В КЛАССЕ-НАСЛЕДНИКЕ

Доступность элементов базового класса из классов-наследников изменяется в зависимости от спецификаторов доступа в базовом классе и спецификатора наследования.

Спецификатор доступа в базовом классе	**Спецификатор наследования**	**Доступ в производном классе**
public	отсутствует	private
public	private	private
public	public	public
public	protected	protected
protected	отсутствует	private
protected	private	private
protected	public	protected
protected	protected	protected
private	отсутствует	недоступны
private	private	недоступны
private	public	недоступны
private	protected	недоступны

МЕХАНИЗМ НАСЛЕДОВАНИЯ

Класс-потомок наследует структуру (элементы данных) и поведение (все методы) базового класса.

Возможности, предоставляемые механизмом наследования:

- Добавлять в производном классе данные, которые представляет базовый класс
- Дополнять в производном классе функциональные возможности базового класса
- Модифицировать в производном классе методы базового класса

Возможности, которых нет:

- Модифицировать в производном классе данные, представленные базовым классом (сохранив их идентификаторы)

Что происходит в порожденном классе:

- Поля данных и методы– **члены класса наследуются от базового класса.** Можно считать, что они описаны в порожденном классе. Однако, возможность доступа к ним из методов производного класса и извне этого класса определяется спецификатором доступа (private, protected, public) к членам в базовом классе и спецификатором доступа к базовому классу, задаваемому при описании производного класса.
- В производном классе **можно добавлять свои поля – члены класса**.
- В производном классе **можно добавлять свои методы – члены класса.**
- В производном классе **можно переопределять методы** базового класса (сохраняя точное совпадение с исходным прототипом, то есть количество и типы аргументов и возвращаемый тип). Исключение: если возвращаемый тип является указателем или ссылкой на базовый класс, он может быть заменен указателем или ссылкой на порождаемый класс.
- Если Вы в производном классе переопределили метод, **доступ** из него **к родительскому методу** можно получить, используя оператор ::
- Если в классе-наследнике имя метода и его прототип совпадают с именем метода базового класса, то метод производного класса **скрывает** метод базового класса.
- **Статические поля** наследуются. Все потомки разделяют единственную копию статического поля. **Статические методы** наследуются.
- Ограничений в наследовании вложенных классов нет: внешний класс может наследовать от вложенного и наоборот.
- **Указатель базового класса** может указывать на объект любого класса, производного от этого базового. Обратное неверно.

КОНСТРУКТОР И ДЕСТРУКТОР ПРОИЗВОДНОГО КЛАССА

Конструкторы и деструктор базового класса в производном классе **не наследуются.**

Конструкторы.

1. Если в базовом классе нет конструкторов или есть конструктор без аргументов (или аргументы присваиваются по умолчанию), то в производном классе конструктор можно не писать – будет создан конструктор копирования и конструктор по умолчанию.
2. Если в базовом классе все конструкторы с аргументами, производный класс обязан иметь конструктор, в котором явно должен быть вызван конструктор базового класса.
3. При создании объекта производного класса сначала вызывается конструктор базового класса – потом производного.

Деструкторы.

1. При отсутствии деструктора в производном классе система создает деструктор по умолчанию.
2. Деструктор базового класса вызывается в деструкторе производного класса автоматически.
3. Деструкторы вызываются в порядке, обратном вызову конструкторов.

Расширенный синтаксис объявления конструктора производного класса

```
Конструктор_производного_сласса(список_фргументов):
                        базовый_класс(список_аргументов)
{ тело конструктора производного класса}
```

ПОЛИМОРФИЗМ

Полиморфизм (многообразие форм) – принцип ООП.

Полиморфизм – свойство различных объектов выполнять одно и то же действие (с одним и тем же названием) по-своему.

Полиморфизм – возможность для объектов разных классов, связанных с помощью наследования, реагировать различным образом при обращении к одной и той же функции-члену.

ВИРТУАЛЬНЫЕ ФУНКЦИИ

Связывание – сопоставление вызова функции с телом функции. Для обычных методов связывание выполняется на этапе трансляции до запуска программы. Такое связывание называется **«ранним»** или **статическим**.

При наследовании обычного метода его поведение не меняется в наследнике. Чтобы добиться разного поведения методов базового класса и классов–наследников необходимо объявить функцию-метод **виртуальной (virtual)**.

Виртуальной называется функция-член класса, вызов (и выполняемый код) которой зависит от типа объекта, для которого она вызывается.

Виртуальные функции обеспечивают механизм **«позднего»** или **динамического** связывания, которое работает во время выполнения программы.

Класс, в котором определены виртуальные функции (хотя бы одна), называется **полиморфным**.

Ключевое слово **virtual можно** писать только в базовом классе.

Описание виртуальной функции (обычно в базовом классе) имеет вид:

```
virtual тип имя-функции (список_формальных_параметров);
```

Модификатор **override** после объявления виртуальной функции приводит к тому, что компилятор проверит, что эта функция точно перегружает какую-то функцию родительского класса. То есть что в родительском классе существует точно такая же функция (с тем же именем, набором параметров и модификаторов, кроме разве что самого модификатора override) и более того она виртуальная.

ПРАВИЛА ОПИСАНИЯ И СОЗДАНИЯ ВИРТУАЛЬНЫХ ФУНКЦИЙ

Правила описания и создания виртуальных функций:

- Виртуальная функция может быть **только методом** класса.
- **Любую перегружаемую операцию-метод** класса можно сделать виртуальной.
- Виртуальная функция **наследуется**.
- Виртуальная функция **может быть константной**.
- Если в базовом классе определена виртуальная функция, то метод производного класса с таким же именем и прототипом **автоматически** является виртуальным и замещает функцию-метод базового класса.
- **Статические методы не могут** быть виртуальными.
- **Конструкторы не могут** быть виртуальными.
- **Деструкторы могут** (чаще - должны) быть виртуальными.

Внутри конструкторов и деструкторов динамическое связывание не работает, хотя вызов виртуальных функций не запрещен. В конструкторах и деструкторах всегда вызывается «родная» функция класса.

Виртуальные методы **можно перегружать и переопределять** в наследниках с другим списком аргументов. Если виртуальная функция переопределена, она замещает (скрывает) родительские методы.

АБСТРАКТНЫЕ И КОНКРЕТНЫЕ КЛАССЫ

Чистой виртуальной функцией является виртуальная функция, у которой в ее объявлении тело определено как 0 (инициализатор =0).

```
virtual тип имя-функции (список_формальных_параметров) = 0;
```

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

Они применяются **в качестве базовых** в процессе наследования.

Их назначение – возможность другим классам унаследовать от них интерфейс и реализацию.

Классы, объекты которых могут быть реализованы, называются **конкретными**.

Попытка создать объект абстрактного класса даст синтаксическую ошибку.

ПРИМЕРЫ

ПРИМЕР 1. Синтаксис объявления производного класса

Простейшее объявление открытого класса наследника без новых полей и методов.

```
#include <iostream>  // for cin cout
using namespace std;
class  Base         // объявление базового класса
{   double x, y;
public:
     void set_xy(double bx = 0, double by = 0)
     {x = bx; y = by; }
     void Print()
     {cout <<" x="<<x<<"  y="<<y<<endl;}
};
         //   объявление класса наследника
         //   без новых полей и методов
class  Derived : public Base {};

void main()
{    Base P1;
     P1.set_xy();
     P1.Print();           //  0 0

     P1.set_xy(1,2);
     P1.Print();           // 1 2

     Derived P2;           //  объект-наследник
     P2.set_xy();          // доступен метод базового класса
     P2.Print();           // 0 0

     P2.set_xy(1,2);      // доступен метод базового класса
     P2.Print();           // 1 2
}
```

ПРИМЕР 2. Спецификатор наследования public

Базовый класс наследуется в производном классе как открытый. В программе доступны методы базового класса.

```
#include <iostream>
using namespace std;
class Base
{   int x;
  public:
    void setx(int n) { x = n; }
    void showx()     { cout << x << endl; }
};
                  // Класс наследуется как открытый
class Derived : public Base
{    int y;
public:
```

```
    void sety(int n) { y = n; }
    void showy()     { cout << y << endl; }
                   // ОШИБКА - поле x недоступно
    // void showxy(){ cout << x << y << endl; }
                   // но можно так
    void showxy()    { showx(); showy();}
};
int main()
{   Derived ob;
       // доступ к членам базового класса через методы
    ob.setx(10);
    ob.showx();
             // доступ к членам производного класса через методы
    ob.sety(20);
    ob.showy();
    ob.showxy();
    return 0;
}
```

ПРИМЕР 3. Спецификатор наследования private

Базовый класс наследуется в производном классе как закрытый. В программе недоступны методы базового класса.

```
#include <iostream>
using namespace std;
class Base
{   int x;
public:
    void setx(int n) { x = n; }
    void showx()     { cout << x << endl; }
};
                // Класс наследуется как закрытый
class Derived : private Base
 {  int y;
public:
    void sety(int n) { y = n; }
    void showy()     { cout << y << endl; }
};
int main()
{   Derived ob;
// ОШИБКА - закрыто для производного класса
//  ob.setx(10);
// правильный доступ к члену производного класса
     ob.sety(20);
// ОШИБКА -  закрыто для производного класса
//  ob.showx();
// правильный доступ к члену производного класса
    ob.showy();
    return 0;
}
```

ПРИМЕР 4. Спецификатор наследования private

Исправленная версия программы. Базовый класс наследуется в производном классе как закрытый. В программе недоступны методы базового класса. В производном классе доступны методы базового класса.

```
#include <iostream>
using namespace std;
class Base
{   int x;
public:
    void setx(int n) { x = n; }
    void showx()     { cout << x << endl; }
};
                // Класс наследуется как закрытый
class Derived : private Base
{    int y;
public:
    // метод setx доступен внутри класса Derived
    void setxy(int n, int m)
    {   setx(n);
        y = m;
    }
    // метод showx доступен внутри класса Derived
    void showxy()
    {   showx();
        cout << y << endl;
    }
};
int main()
{   Derived ob;
    ob.setxy(10, 20);
    ob.showxy();
    return 0;
}
```

ПРИМЕР 5. Спецификатор доступа protected

В программе недоступно поле класса со спецификатором *protected*.

```
#include <iostream>
using namespace std;
class samp
{                // члены класса, закрытые по умолчанию
    int a;
  protected:      // тоже закрытые члены класса samp
    int b;
  public:
    int c;
    samp(int n, int m) { a = n; b = m; }
    int geta() { return a; }
    int getb() { return b; }
};
```

```
int main()
{    samp ob(10, 20);
      // Ошибка! Переменная b защищена и поэтому закрыта
      //    ob.b = 99;
       // Правильно! Переменная c является открытым членом
     ob.c = 30;
     cout << ob.geta() << ' ';
     cout << ob.getb() << ' ' << ob.c << endl;
     return 0;
}
```

ПРИМЕР 6. Спецификатор доступа protected

Базовый класс наследуется в производном классе как открытый. В программе недоступны методы базового класса. В производном классе доступны методы базового класса.

```
#include <iostream>
using namespace std;
class Base
{ protected:      // закрытые члены класса Base,
    int a,b;    // но для производного класса они доступны
public:
    void setab(int n, int m)
       { a = n; b = m; }
};
class Derived : public Base
 {   int c;
   public:
    void setc(int n) { c = n; }
// эта функция имеет доступ к переменным a и b класса Base
    void showabc()
    {
        cout << a << ' ' << b << ' ' << c << '\n';
    }
};
int main()
{   Derived ob;
 /* Переменные a и b здесь недоступны, поскольку являются
    закрытыми членами классов Base и Derived    */
    ob.setab(1, 2);
    ob.setc(3);
    ob.showabc();
    return 0;
}
```

ПРИМЕР 7. Спецификатор наследования protected

Базовый класс наследуется в производном классе как защищенный. В производном классе доступны методы базового класса. В программе недоступен метод базового класса.

```
#include <iostream>
using namespace std;
class Base
{ protected:      // закрытые члены класса Base,
    int a,b;    // но для производного класса они доступны
public:
    void setab(int n, int m) { a = n; b = m; }
};
      // класс Base наследуется как защищенный
class Derived : protected Base
{   int c;
   public:
    void setc(int n) { c = n; }
// эта функция имеет доступ к переменным a и b класса Base
    void showabc()
    {
        cout << a << ' ' << b << ' ' << c << '\n';
    }
};
int main()
{   Derived ob;
    // ОШИБКА: теперь функция setab()
    // является защищенным членом класса Base
    // ob.setab(1, 2);
     // функция setab() здесь недоступна
    ob.setc(3);
    ob.showabc();
    return 0;
}
```

ПРИМЕР 8. Простое наследование. Конструктор, деструктор в базовом классе

Объявление класса наследника без конструктора.

```
#include <iostream>  // for cin cout
using namespace std;

class  Base                    //   объявление класса
{  double x, y;
public:
     Base(double bx = 0, double by = 0)      //  конструктор
     {x = bx; y = by; }
     ~Base()                                 //  деструктор
     {    cout << "destructor Base class  " << endl;   }
     void Print()
     {    cout << " x=" << x << "  y=" << y << endl;   }
};
         //   объявление класса наследника без конструктора
class  Derived: public Base{};
void main()
{    Base P1;
     P1.Print();                       //  0 0
     Base P2(1,2);
```

```
     P2.Print();                    // 1 2
     Derived P3;
     P3.Print();                    // 0 0
     //Point2 P4(1,2);              //   !!! невозможно
     //P4.Print ();
}                                  // вызов деструкторов
```

ПРИМЕР 9. Конструктор и деструктор при наследовании

Порядок вызова конструкторов и деструкторов при наследовании. Конструкторы вызываются в порядке наследования. Деструкторы – в обратном порядке.

```
#include <iostream>
using namespace std;
class Base
{   public:
    Base()
         { cout << "Constructor Base class \n"; }
    ~Base()
         { cout << "Destructor Base class\n"; }
};
class Derived : public Base
{  public:
    Derived()
    { cout << "Constructor Derived class\n"; }
    ~Derived()
    { cout << "Destructor Derived class \n"; }
};
int main()
{   Derived o;
    return 0;
}
```

ПРИМЕР 10. Передача аргумента конструктору производного класса

Передача аргумента конструктору производного класса. Конструкторы вызываются в порядке наследования. Деструкторы – в обратном порядке.

```
#include <iostream>
using namespace std;

class Base
{  public:
    Base()
    { cout << "Constructor Base class\n"; }
    ~Base()
    { cout << "Destructor Base class\n"; }
};
class Derived : public Base
{   int j;
public:
    Derived(int n)
```

```
    {   cout << "Constructor Derived class\n";
        j = n;
    }
    ~Derived()
    {  cout << "Destructor Derived class\n";
    }
    void showj()
    {  cout << j << '\n';
    }
};
int main()
{   Derived o(10);
    o.showj();
    return 0;
}
```

ПРИМЕР 11. Синтаксис вызова конструктора базового класса из конструктора производного класса

```
#include <iostream>
using namespace std;
class Base
{    int i;
  public:
    Base(int n)
    {   cout << "Constructor Base class\n";
        i = n;
    }
    ~Base()
    { cout << "Destructor Base class\n"; }
    void showi()
    { cout << i << '\n'; }
};
class Derived : public Base
{   int j;
  public:
    Derived(int n) : Base(n)
    {          // передача аргумента в базовый класс
       cout << "Constructor Derived class\n";
       j = n;
    }
    ~Derived()
    { cout << "Destructor Derived class\n"; }
    void showj()
    { cout << j << '\n'; }
};
int main()
{   Derived o(10);
    o.showi();
    o.showj();
    return 0;
}
```

ПРИМЕР 12. Передача аргументов в конструктор базового класса из конструктора производного класса

```
#include <iostream>
using namespace std;
class Base //   объявление базового класса
{   int i;
  public:
    Base(int n) //  конструктор
    {   cout << "Constructor Base class\n";
        i = n;
    }
    ~Base()//  деструктор
    { cout << "Destructor Base class\n"; }
    void showi()
    { cout << i << '\n'; }
};
class Derived : public Base //объявление класса наследника
{    int j;
public:
     Derived(int n, int m) : Base(m) //  конструктор
     {       // передача аргумента в базовый класс
       cout << "Constructor Derived class\n";
       j = n;
    }
    ~Derived()//  деструктор
    { cout << "Destructor Derived class\n"; }
    void showj()
    { cout << j << '\n'; }
};
int main()
{   Derived o(10, 20);
    o.showi();
    o.showj();
    return 0;
}
```

ПРИМЕР 13. Множественное наследование

Пример иерархии вида:

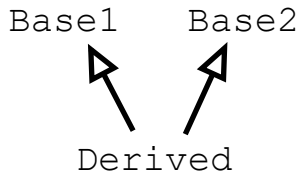
Конструкторы вызываются в порядке наследования. Деструкторы – в обратном порядке.

```
#include <iostream>
using namespace std;
class B1 //   объявление базового класса
{   int a;
  public:
    B1(int x)  { a = x; }//  конструктор
```

```
    int geta() { return a; }
};
            // Прямое наследование базового класса
class D1 : public B1
{    int b;
  public:
                // передача переменной у классу B1
    D1(int x, int y) : B1(y) //  конструктор
    {
        b = x;
    }
    int getb()   { return b; }
};
// Прямое наследование производного класса
// и косвенное наследование базового класса
class D2 : public D1
 {  int c;
  public:           // передача аргументов классу D1
    D2(int x, int y, int z) : D1(y, z) //  конструктор
    {   c = x;
    }
/* Поскольку базовые классы наследуются как открытые, класс D2 имеет
доступ к открытым элементам классов B1 и D1 */
    void show()
    {   cout << geta() << ' ' << getb() << ' ';
        cout << c << '\n';
    }
};
int main()
{   D2 ob(1, 2, 3);
    ob.show();
       // функции geta() и getb() здесь тоже открыты
    cout << ob.geta() << ' ' << ob.getb() << endl;
    return 0;
}
```

ПРИМЕР 14. Множественное наследование

Пример множественного наследования вида:

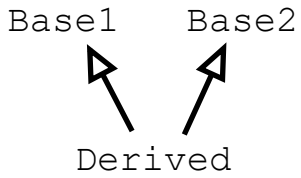



```
#include <iostream>
using namespace std;
              // Создание первого базового класса
class B1
 {  int a;
  public:
    B1(int x)  { a = x; }//  конструктор
    int geta() { return a; }
};
```

```
class B2 // Создание второго базового класса
{   int b;
public:
    B2(int x) //  конструктор
    {  b = x;
    }
    int getb() { return b; }
};
     // Прямое наследование двух базовых классов
class D : public B1, public B2
{   int c;
  public:
    // здесь переменные z и y
    // напрямую передаются классам B1 и B2
    D(int x, int y, int z) : B1(z), B2(y) //  конструктор
    { c = x;
    }
/* Поскольку базовые классы наследуются как открытые, класс D имеет
доступ к открытым элементам классов B1 и B2 */
    void show()
    {   cout << geta() << ' ' << getb() << ' ';
        cout << c << '\n';
     }
};
int main()
{   D ob(1, 2, 3);
    ob.show();
    return 0;
}
```

ПРИМЕР 15. Конструктор и деструктор при множественном наследовании

Порядок вызова конструкторов и деструкторов при множественном наследовании:

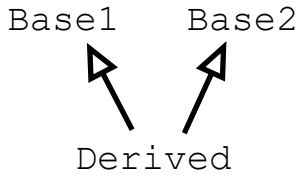



```
#include <iostream>
using namespace std;
class B1
{  public:
     B1() { cout << "Constructor classa B1\n"; }
    ~B1() { cout << "Destructor classa B1\n"; }
};
class B2
{    int b;
 public:
     B2() { cout << "Constructor classa B2\n"; }
    ~B2() { cout << "Destructor classa B2\n"; }
};
```

```
            // Наследование двух базовых классов
class D : public B1, public B2
{  public:
     D() { cout << "Constructor classa D\n"; }
    ~D() { cout << "Destructor classa D\n"; }
};
int main()
{   D ob;
    return 0;
}
```

ПРИМЕР 16. Множественное наследование. Неопределенность

Пример множественного наследования вида:



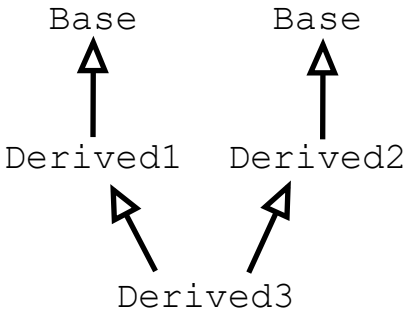
В каждом базовом классе есть свой метод Print. В производном классе такой метод отсутствует. При вызове метода Print из производного класса возникает неопределенность!

```
#include <iostream>  // for cin cout
using namespace std;
class  B1          //   объявление 1 базового класса
{  protected:
     double x;
public:
     B1(double bx = 0)      //  конструктор
     {x = bx; }
     ~B1 ()               //  деструктор
     { cout << "destructor Point1 done"<<endl;}
     void Print ()
     {cout <<" x="<<x<<endl;}
};
class B2    //   объявление 2 базового класса
{ protected:
    double y;
public:
     B2(double by)         //  конструктор
     {y = by;}
     ~B2 ()               //  деструктор
     { cout << "destructor Point2 done"<<endl;}
     void Print ()
     {cout <<"  y="<<y<<endl;}
};

class D: public B1, public B2
{    double z;
public:
```

```
  //  конструктор вызывает конструкторы 2 базовых классов
      D(double bx, double by, double bz) : B1(bx), B2(by)
      {     z = bz;   }
      ~D ()                          //  деструктор
      {     cout << "destructor Point3 done" << endl;    }
      void PrintD ()
      {     cout<<" z="<<z<<endl;     }
};
void main()
{     D P(1,2,3);
//    P.Print();              //     !!! неопределенность
      P.B1::Print();          //  1
      P.B2::Print();          //  2
      P.PrintD();             //  3
}
```

ПРИМЕР 17. Множественное наследование. Неопределенность

Пример множественного наследования вида:



В этом случае класс *Base* фактически наследуется классом *Derived3* дважды – через класс *Derived1* и *Derived2*. Если в классе Derived3 необходимо использовать член класса *Base*, это вызовет неоднозначность, поскольку в классе *Derived3* имеется две копии класса *Base*.

```
#include <iostream>
using namespace std;
class Base
{  public:
    int i;
};
     // Наследование класса Base как виртуального
class Derived1 : public Base
{  public:
    int j;
};
   // Здесь класс Base тоже наследуется как виртуальный
class Derived2 : public Base
{  public:
    int k;
};
/* Здесь класс Derived3 наследует как класс Derived1, так и класс
Derived2 */
class Derived3 : public Derived1, public Derived2
```

```
{  public:
    // Неоднозначность:  две копии класса Base
    int product() { return i * j * k; }
};
int main()
{   Derived3 ob;
    ob.i = 10;
    ob.j = 3;
    ob.k = 5;
    cout << "Rezult =  " << ob.product() << '\n';
    return 0;
}
```

ПРИМЕР 18. Виртуальный базовый класс

Решение неопределенности в предыдущим примере – объявление базового класса виртуальным.

```
#include <iostream>
using namespace std;
class Base
{ public:
    int i;
};
     // Наследование класса Base как виртуального
class Derived1 : virtual public Base
{  public:
    int j;
};
   // Здесь класс Base тоже наследуется как виртуальный
class Derived2 : virtual public Base
{  public:
    int k;
};
/* Здесь класс Derived3 наследует как класс Derived1, так и класс
Derived2. Однако в классе Derived3 создается только одна копия класса
Base  */
class Derived3 : public Derived1, public Derived2
{ public:
    int product() { return i * j * k; }
};
int main()
{   Derived3 ob;
    // Здесь нет неоднозначности, поскольку
    // представлена только одна копия класса Base
    ob.i = 10;
    ob.j = 3;
    ob.k = 5;
    cout << "Rezult =  " << ob.product() << '\n';
    return 0;
}
```

ПРИМЕР 19. Полиморфизм. Переопределение метода

Базовый класс имеет метод get. Производный класс имеет свой метод get.

```
#include <iostream>
using namespace std;
class Base
{   int x;
  public:
    void setx(int i) { x = i; }
    int get() { return x; }
};
class Derived : public Base
{   int y;
  public:
    void sety(int i) { y = i; }
    int get() {return y; }
};
int main()
{   Base    b_ob;   // объект базового класса
    Derived d_ob;   // объект производного класса
    b_ob.setx(10);  // доступ к объекту базового класса
    cout << "Object of Base class x: " << b_ob.get() <<endl;
    d_ob.setx(99); // доступ к объекту производного класса
    d_ob.sety(88);
    cout << "Object of Derived class y: "<<d_ob.get()<<endl;
    cout << "Object of Derived class x: " <<
                                    d_ob.Base::get()<<endl;
    return 0;
}
```

ПРИМЕР 20. Указатель на объект производного класса

Демонстрация указателя на объект производного класса. Указатель, объявленный в качестве указателя на базовый класс, может использоваться как указатель на любой класс, производный от этого базового класса.

```
#include <iostream>
using namespace std;
class Base
{   int x;
  public:
    void setx(int i) { x = i; }
    int get () { return x; }
};
class Derived : public Base
{   int y;
  public:
    void sety(int i) { y = i; }
    int get() {return y; }
};
int main()
{   Base *p;       // указатель базового класса
```

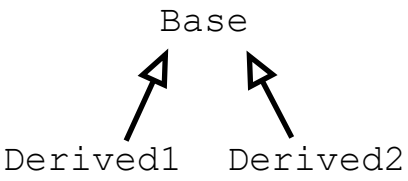
```
    Base b_ob;    // объект базового класса
    Derived d_ob; // объект производного класса. использование
           // указателя p для доступа к объекту базового класса
    p = &b_ob;
    p->setx(10); // доступ к объекту базового класса
    cout << "Object of Base class x: " << p->get() << endl;
                 // использование указателя p для доступа к
                 // объекту производного класса
    p = &d_ob;   // указывает на объект производного класса
    p->setx(99); // доступ к объекту производного класса
                 // т.к. p нельзя использовать для
                 // установки y, делаем это напрямую
    d_ob.sety(88);
    cout << "Object of Derived class x: " << p->get()<<endl;
    cout << "Object of Derived class y: "<<d_ob.get()<<endl;
    return 0;
}
```

ПРИМЕР 21. Виртуальная функция

Пример тот же. Метод get() объявим как виртуальный.

```
#include <iostream>
using namespace std;
class Base
{   int x;
  public:
    void setx(int i) { x = i; }
    virtual int get () { return x; }
};
class Derived : public Base
{   int y;
  public:
    void sety(int i) { y = i; }
    int get() {return y; }
};
int main()
{   Base *p;      // указатель базового класса
    Base b_ob;    // объект базового класса
    Derived d_ob; // объект производного класса. использование
            // указателя p для доступа к объекту базового класса
    p = &b_ob;
    p->setx(10); // доступ к объекту базового класса
    cout << "Object of Base class x: " << p->get() << endl;
                 // использование указателя p для доступа к
                 // объекту производного класса
    p = &d_ob;   // указывает на объект производного класса
    p->setx(99); // доступ к объекту производного класса
    // т.к. p нельзя использовать для установки y, делаем это напрямую
    d_ob.sety(88);
    cout << "Object of Derived class x: " << p->get()<<endl;
    cout << "Object of Derived class y: "<<d_ob.get()<<endl;
    return 0;
}
```

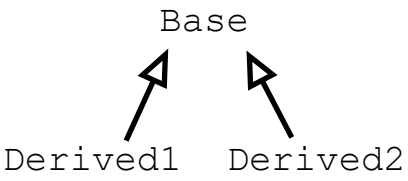
ПРИМЕР 22. Виртуальная функция



Простой пример использования виртуальной функции.

```
#include <iostream>
using namespace std;
class Base
{  public:
     int i;
     Base(int x) { i = x; }
     virtual void func()
     { cout << "func() of Base class: ";
       cout << i << '\n';
     }
};
class Derived1 : public Base
{  public:
     Derived1(int x) : Base(x) { }
     void func()
     { cout << "func() Derived1 class: ";
       cout << i * i << '\n';
     }
};
class Derived2 : public Base
{ public:
     Derived2(int x) : Base(x)
     {}
     void func()
     {    cout << "func() Derived2 class: ";
          cout << i + i << '\n';
     }
};
int main()
{    Base *p;
     Base ob(10);
     Derived1 d_ob1(10);
     Derived2 d_ob2(10);
     p = &ob;
     p->func();       // функция func() класса Base
     p = &d_ob1;
     p->func();
          // функция func() производного класса Derived1
     p = &d_ob2;
     p->func();
          // функция func() производного класса Derived2

     return 0;
}
```

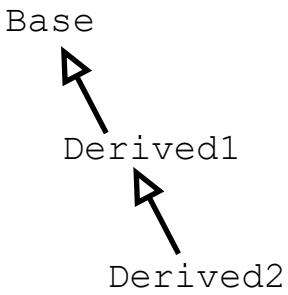
ПРИМЕР 23. Иерархический порядок наследования виртуальных функций



Виртуальные функции имеют иерархический порядок наследования. Пример отличается от предыдущего отсутствием виртуальной функции в класса *Derived*2. В этом случае для объекта класса *Derived*2 используется версия функии, определенная в базовом классе *Base*.

```
#include <iostream>
using namespace std;
class Base
{ int i;
public:
    Base(int x) { i = x; }
    virtual void func()
    {   cout << "func() Base class: ";
        cout << i << '\n';
    }
};
class Derived1 : public Base
{ public:
    Derived1(int x) : Base(x)
    {}
    void func()
    {   cout << "func() Derived1 class: ";
        cout << i * i << '\n';
    }
};
class Derived2 : public Base
{ public:
    Derived2(int x) : Base(x)
    {}// в классе Derived2 функция func() не подменяется
};
int main()
{   Base *p;
    Base ob(10);
    Derived1 d_ob1(10);
    Derived2 d_ob2(10);
    p = &ob;
    p->func();  // функция func() базового класса
    p = &d_ob1;
    p->func();
         // функция func() производного класса Derived1
    p = &d_ob2;
    p->func();     // функция func() базового класса

    return 0;
}
```

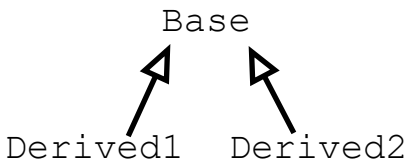
ПРИМЕР 24. Виртуальный базовый класс



Виртуальная функция при наследовании сохраняет свою виртуальную природу

```
#include <iostream>
using namespace std;
class Base
{  public:
    virtual void func()
    {  cout << "func() Base class \n";
    }
};
class Derived1 : public Base
{  public:
    void func()
    { cout << "func() Derived1 class \n";
    }
};
// Класс Derived1 наследуется классом Derived2
class Derived2 : public Derived1
{ public:
    void func()
    {
        cout << " func() Derived2 class \n";
    }
};
int main()
{   Base *p;
    Base ob;
    Derived1 d_ob1;
    Derived2 d_ob2;
    p = &ob;
    p->func(); // функция func() базового класса
    p = &d_ob1;
    p->func();
     // функция func() производного класса Derived1
    p = &d_ob2;
    p->func();
       // функция func() производного класса Derived2
    return 0;
}
```

ПРИМЕР 25. Позднее связывание



В этом примере показана работа виртуальной функции при наличии случайных событий во время выполнения программы.

```
#include <iostream>
#include <cstdlib>
using namespace std;
class Base
{public:
    int i;
    Base(int x)
    { i = x; }
    virtual void func()
    {   cout << "func() Base class: ";
        cout << i << '\n';
    }
};
class Derived1 : public Base
{public:
    Derived1(int x) : Base(x)  {}
    void func()
    {   cout << "func() Derived1 class: ";
        cout << i * i << '\n';
    }
};
class Derived2 : public Base
{public:
    Derived2(int x) : Base(x) {}
    void func()
    {   cout << "func() Derived2 class: ";
        cout << i + i << '\n';
    }
};
int main()
{   Base *p;
    Derived1 d_ob1(10);
    Derived2 d_ob2(10);
    for(int i = 0; i < 10; i++)
    {   int j = rand();
        if (j % 2)              // если число нечетное
            p = &d_ob1;         // использовать объект d_ob1
        else                    // если число четное
            p = &d_ob2;         // использовать объект d_ob2
        p->func();      // вызов подходящей версии функции
    }
    return 0;
}
```

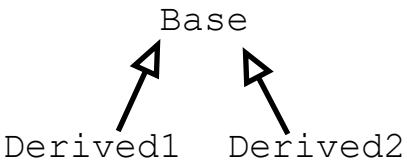
ПРИМЕР 26. Абстрактный класс

Базовый класс – абстрактный.

```
#include <iostream>  // for cin cout
using namespace std;
class  Base             //   объявление абстрактного класса
{  public:
     Base() {}                  //  конструктор
     ~Base () {}                //  деструктор
     virtual void get() =0;    // чистая виртуальная функция
};
class Derived1 : public Base//объявление производного класса
{protected:
       double x;
public:
     Derived1(double bx)        //  конструктор
     {x = bx;}
     ~Derived1()                //  деструктор
     { cout << "destructor Point1 done" << endl;}
     void get()
     { cout << "  x = "<< x << endl; }
};
                        //объявление производного класса
class Derived2: public Derived1
{    double y;
public:                 //  конструктор
     Derived2(double bx,double by) : Derived1(bx)
     {y = by; }
     ~Derived2()                       //  деструктор
     { cout << "destructor Point2 done"<<endl;}
     void get()
     { cout << " x = "<< x <<" y = "<< y << endl; }
};
void main()
{    Base *P;
 //  P = new Point();      //  !!! нельзя создать объект
                           // абстрактного класса
//   P -> get();
     P = new Derived1(1);
     P -> get();                          // 1
     P = new Derived2(2,3);
     P -> get();                          //  2  3
}
```

ПРИМЕР 27. Вызов метода производного класса из метода базового класса

Вызов виртуальной функции из метода базового класса. Два производных класса от одного базового.



В каждом классе определим свой метод *get*, который должен вызываться из метода *get_base* базового класса. При вызове метода *get* для объектов производных классов, происходит вызов метода *get* базового (!) класса.

```
#include <iostream>  // for cin cout
using namespace std;
class  Base                 //   объявление базового класса
{protected:
       double x,y;
public:
     Base(double bx = 0, double by = 0)      //  конструктор
     {x = bx; y = by; }
     ~Base ()                                //  деструктор
     { cout << "destructor Base done"<<endl;}
                 // вызов метода  Print из метода Print_B
     void get_base()
     {    cout <<"  Base class  " << endl;
          get();
     }
     void get ()
     { cout << " x = "<< x <<" y = "<< y <<endl;}
};
class Derived1: public Base // объявление класса наследника
{   double z;               // новое поле
public:                     //  конструктор
     Derived1(double bx, double by, double bz) : Base(bx,by)
     {z = bz; }
     ~Derived1()                             //  деструктор
     { cout << "destructor Derived1 done" << endl; }
     void get ()     // метод переопределен
     { cout<<" x = "<< x <<" y = "<< y <<" z = "<< z <<endl;}
};
class Derived2 : public Base  //объявление класса наследника
{
     int a,b;                                //  новые поля
public:                                      //  конструктор
     Derived2(double bx,double by, int ba,int bb)
                                             : Base(bx,by)
     { a = ba; b = bb; }
     ~Derived2 ()                            //  деструктор
     { cout << "destructor Derived2 done"<<endl;}
     void get ()     // метод переопределен
     { cout<<" x="<<x<<"  y="<<y<<" a="<<a<<"  b="<<b<<endl;}
};
```

```
void main()
{     Base     P1(1,2);
      Derived1 P2(3,4,5);
      Derived2 P3(6,7,8,9);

      P1.get_base();              // 1 2
      P2.get_base();              // 3 4              !!!
      P3.get_base();              // 6 7              !!!
}
```

ПРИМЕР 28. Вызов методов производных классов из метода базового класса. Виртуальные функции

Пример тот же. Добавляем слово **virtual** для метода *get* базового класса!

Для каждого объекта производных классов метод *get_base* вызывает соответствующий (!) метод *get*.

```
#include <iostream>  // for cin cout
using namespace std;
class Base                //   объявление базового класса
{protected:
      double x,y;
public:
      Base(double bx = 0, double by = 0)      //  конструктор
      { x = bx; y = by; }
      ~Base ()                                //  деструктор
      { cout << "destructor Base done"<<endl;}
                  // вызов метода  Print из метода Print_B
      void get_base()
      {     cout <<"  Base class  "<<endl;
            get();
      }
      virtual void get()
      { cout << " x = "<< x <<" y = "<< y <<endl;}
};
class Derived1: public Base //объявление класса наследника
{     double z;               // новое поле
public:                     //  конструктор
      Derived1(double bx, double by, double bz) : Base(bx,by)
      { z = bz; }
      ~Derived1()                             //  деструктор
      { cout << "destructor Derived1 done" << endl; }
                              //   метод переопределен
      virtual void get()
      { cout<<" x = "<< x <<" y = "<< y <<" z = "<< z <<endl;}
};
class Derived2: public Base  //объявление класса наследника
{
      int a,b;                            //  новые поля
public:                                   //  конструктор
      Derived2(double bx, double by, int ba, int bb)
                                          : Base(bx,by)
```

```
     { a = ba; b = bb; }
     ~Derived2 ()                          //  деструктор
     { cout << "destructor Derived2 done"<<endl;}
                                 //   метод переопределен
     virtual void get()
     { cout<<" x="<<x<<"  y="<<y<<" a="<<a<<"  b="<<b<<endl;}
};
void main()
{    Base     P1(1,2);
     Derived1 P2(3,4,5);
     Derived2 P3(6,7,8,9);

     P1.get_base();              // 1 2
     P2.get_base();              // 3 4 5           !!!
     P3.get_base();              // 6 7 8 9         !!!
}
```

ПРИМЕР 29. Перегрузка оператора присваивания в производных классах

Для объяснения перегрузки оператора присваивания в производных классах, в базовом классе определим поле-указатель.

Если среди полей класса есть указатели, то действует "правило трёх": обязательно явно прописываем конструктор копирования, оператор присваивания и виртуальный деструктор.

```
class CBase          // родительский класс
{protected:
    char* m_pName;
public:         // конструктор
    CBase(const char* name) : m_pName(_strdup(name)) {}
                    // конструктор копирования
    CBase(const CBase& a) : m_pName(_strdup(a.m_pName)){}
    CBase& operator=(const CBase& a)  // оператор =
    {   delete[] m_pName;
        m_pName = _strdup(a.m_pName);
        return *this;
    }
    virtual ~CBase()  // деструктор
    {   delete m_pName;    }
};
class CChild : public CBase  // производный класс
{ protected:
   int m_Value;
public:
   CChild(const char* name, int value) :
           CBase(name),  m_Value(value)  {}  // конструктор
   CChild& operator=(const CChild& a)       // оператор =
   {   CBase::operator=(a);
      m_Value = a.m_Value;
      return *this;
   }
};
```

```
void main()
{  CChild a("AAA", 1);
   CChild b("BBB", 2);
   b = a;
// Оператор присвоения прописан явно, поэтому оператор присваивания
// класса CChild копирует свои поля, а для копирования родительских
// полей вызывает родительский оператор копирования
     CChild c(b);
// Явного конструктора копирования нет, поэтому он создаётся неявно,
// он копирует поля класса CChild,
// а для полей родительского класса вызывает родительский конструктор
// копирования, который создан явно.
}
```

ПРИМЕР 30. Иерархия классов. Создание массива указателей на объекты производных классов.

Разработать иерархию наследования классов, объединяющую людей на факультете, и позволяющую хранить необходимую информацию о них. Ввод значений полей и вывод значений полей на экран должен быть осуществлены через перегрузку операторов >> и <<.

Создать массив сотрудников (студентов и профессоров).

1. **Студент**
   - ФИО
   - средний балл обучения
2. **Профессор**
   - ФИО
   - Количество публикаций

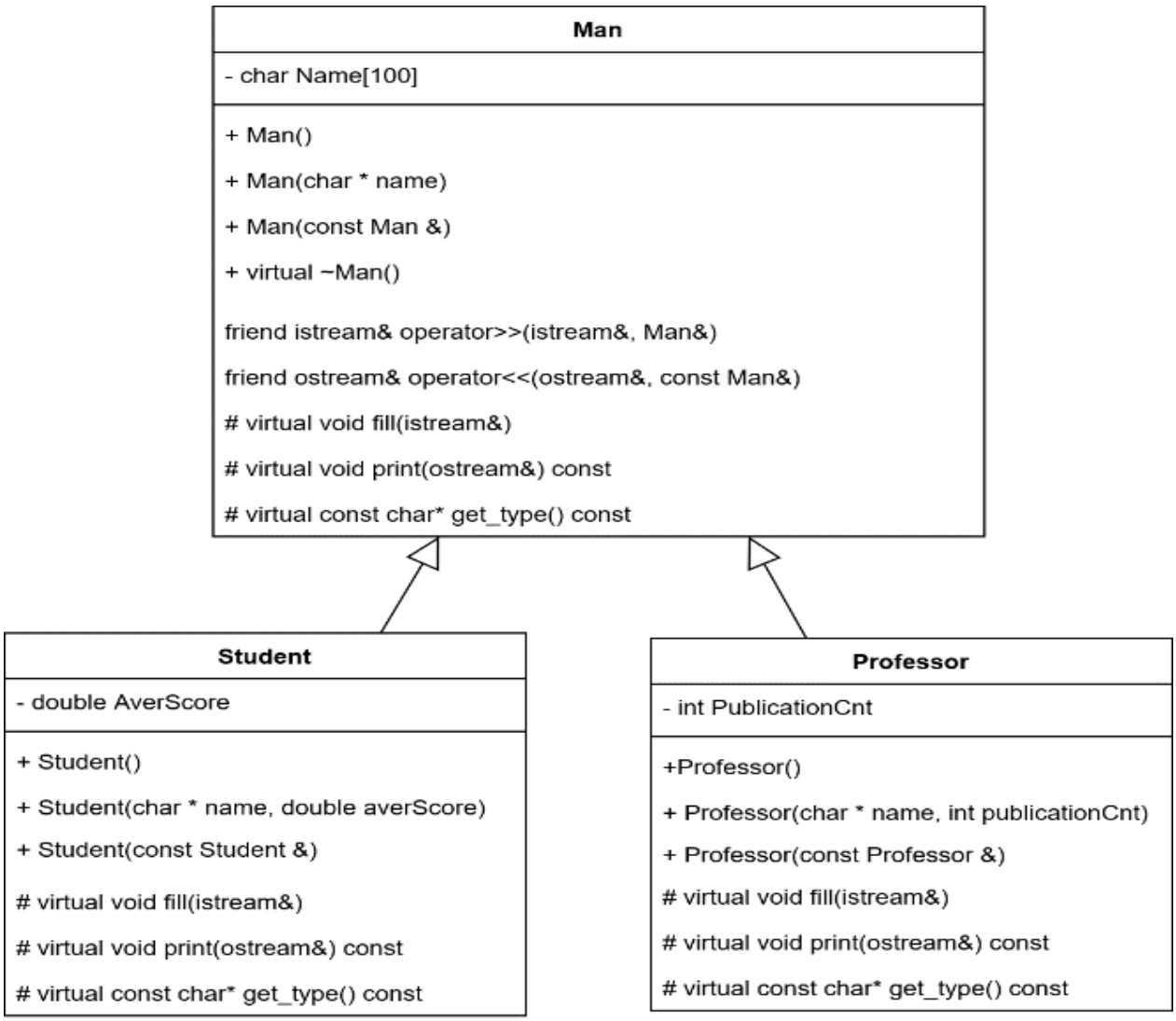
В каждом классе определяем:

- конструкторы,
- деструктор, где необходимо;
- заполнение полей (***fill***);
- вывод значения полей и имени типа объекта на экран (***print***).

Деструктор необходим, если:

- в классе есть указатели/ресурсы, которые необходимо удалить
- у самого первого родительского класса в иерархии наследования, чтобы можно было сделать его виртуальным.

Иерархия классов для задания следующая: два производных класса от одного базового:




```
#include <string>
#include <vector>
#include <iostream>  // for cin cout
using namespace std;

class Man         //   объявление базового класса
{ protected:
     char Name[100];     // ФИО
public:
     Man()            // конструктор без параметров
     { strcpy_s(Name, 100, "");
     }
     Man(const char* name)    // конструктор с параметром
     { strcpy_s(Name, 100, name);
     }
     Man(const Man& m)     // конструктор копирования
     {  strcpy_s(Name, 100, m.Name);
     }
     virtual ~Man()                // деструктор
     { cout << " destructor man" << endl;
     }
```

```
protected:
     // функция fillдоступна только наследникам (protected)
     // вызов должен осуществляться через cin>>
     virtual void fill(istream& in)    // ввод Name
     {    cout << "> Enter Name: ";
          in >> Name;
     }
     // функция print доступна только наследникам (protected)
     // вызов должен осуществляться через cout<<
     virtual void print(ostream& out) const  // вывод
     {    out << "Type: " << get_type() << endl;
          out << "FIO: " << Name << endl;
     }
     // получение имени типа объекта
     virtual const char* get_type() const
     {    return "Man";  }
     // реализация ввода  через оператор >>
     // на friend функции не действуют модификаторы доступа:
     // они все считаются public
     friend istream& operator>>(istream& in, Man& me)
     {    // просто вызываем виртуальную функцию вывода
          me.fill(in);
          return in;
     }
    // реализация вывода через оператор <<
     // на friend функции не действуют модификаторы доступа:
    // они все считаются public
    friend ostream& operator<<(ostream& out, const Man& me)
     {
          me.print(out);
          return out;
     }
};

class Student : public Man   //объявление класса наследника 1
{ protected:
     double SrBall;          // средний балл
public:
     Student() : Man()      // конструктор без параметров
     {    SrBall = 0;
     }
     Student(char* name, double srBall) : Man(name)
     { // конструктор с параметрами
          SrBall = srBall;
     }
     Student(const Student& s) : Man(s)
     { // конструктор копирования
          SrBall = s.SrBall;
     }
     virtual ~Student() override      // деструктор
     {  cout << " destructor student" << endl;
     }
protected:
```

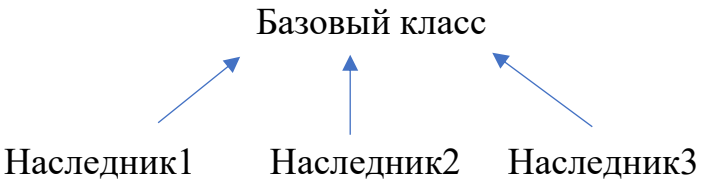
```
    virtual void fill(istream& in) override     // ввод
    {   Man::fill(in);
        cout << "> Enter grade point average: ";
        in >> SrBall;
    }
    virtual void print(ostream& out) const override// вывод
    {   Man::print(out);
        cout << "Grade point average: " << SrBall << endl;
    }
    virtual const char* get_type() const override
    {   return "Student";
    }
};

class Professor : public Man  //объявление класса наследника 2
{
protected:
    int Publications;    // число публикаций
public:
    Professor() : Man()   // конструктор без параметров
    {   Publications = 0;
    }
                        // конструктор с параметрами
    Professor(const char* name, int pblCnt) : Man(name)
    {   Publications = pblCnt;
    }
                          // конструктор копирования
    Professor(const Professor& p) : Man(p)
    {  Publications = p.Publications;
    }
    virtual ~Professor() override     // деструктор
    {   cout << " destructor professor" << endl;
    }
protected:
    virtual void fill(istream& in) override     // ввод
    {   Man::fill(in);
        cout << "> Enter Number Publications: ";
        in >> Publications;
    }
    virtual void print(ostream& out) const override// вывод
    {  Man::print(out);
      out << "Number of Publications: " << Publications << endl;
    }
    virtual const char* get_type() const override
    {   return "Professor";
    }
};



            // создание массива сотрудников
```

```
void Vvod(vector<Man*>& arr, unsigned int n)
{    ClearArr(arr);//если передан не пустой массив - очистить
     while (arr.size() < n)
     {    int type = 0;
          cout << "Input type of object:\n";
          cout << "Student - 1; Professor - 2 \n";
          cin >> type;
          switch (type)
          {
          case 1: // добавить студента
          {    Student* st = new Student();
               cin >> *st;
               arr.push_back(st);
               break;
          }
          case 2: // добавить профессора
          {    Professor* p = new Professor();
               cin >> *p;
               arr.push_back(p);
               break;
          }
          default:
               cout << "Error";
          }
     }
}
                    // вывод массива сотрудников
void Vyvod(const vector<Man*>& arr)
{ for (unsigned int i = 0; i < arr.size(); i++)
          cout << *arr[i];
}
// освободить память
void ClearArr(vector<Man*>& arr)
{
     for (unsigned int i = 0; i < arr.size(); i++)
          delete arr[i];
     arr.clear();
}

int main()
{    const char* FIO = "Ivanov Ivan Ivanovich";
     {
// ограничиваем область существования man1, man2, man3, stud1, prof1
          cout << "---------- man1 ----------" << endl;
          Man man1(FIO); // конструктор с параметрами
          cout << man1;
          cout << endl;
          cout << "---------- man2 ----------" << endl;
          Man man2;       // конструктор без параметров
          cin >> man2;
          cout << man2;
          cout << endl;
          cout << "---------- man3 ----------" << endl;
```

```
            Man man3(man1);  // конструктор копирования
            cout << man3;
            cout << endl;
            cout << "----------- stud1 ----------" << endl;
            Student stud1;     // конструктор без параметров
            cin >> stud1;
            cout << stud1;
            cout << endl;
            cout << "----------- prof1 ----------" << endl;
            Professor prof1(FIO, 5); // конструктор с
            cout << prof1;
            cout << endl;
        }
        cout << "----------- array ----------" << endl;
        vector<Man*> members; // создание массива сотрудников
        int n;
        cout << "vvod number man: ";
        cin >> n;
        Vvod(members, n);
        Vyvod(members);
        ClearArr(members);     // освободить память
        return 0;
}
```

ПРИМЕР 31. Полиморфизм. Фабрика классов

Разработать иерархию наследования классов по схеме.



В каждом классе определяем: метод заполнения полей (*Input*), метод вывода значения полей и имени типа объекта на экран (*Print*), метод определения типа класса как строку, метод создания своей копии объекта.

Задача: написать функцию, которая принимает название класса в виде строки, а возвращает экземпляр этого класса.

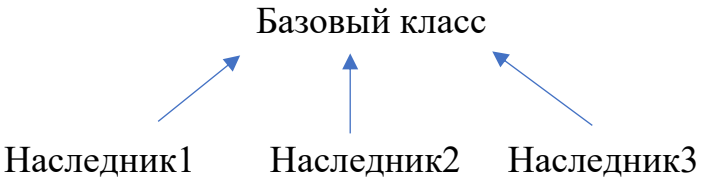
```
#include <iostream>
#include <map>
#include <string.h>
#include <string>
using namespace std;

class CEmployee     // базовый абстрактный класс
{    string m_Name;
     unsigned int m_ID;
public:
  void Input();                  // ввод
  void Print() const;            // вывод
               // абстрактный метод. Тип класса - строка
```

```
  virtual const char* GetType() const = 0;
                                  // абстрактный метод
  virtual CEmployee* Clone() const = 0; // создание своей копии
};

class CLaborer : public CEmployee  //наследник 1
{ public:
  virtual const char* GetType() const override
  { return "laborer"; } // тип класса - строка
  virtual CLaborer* Clone() const override // создание своей копии
  { return new CLaborer(*this); }
};

class CScientist : public CEmployee   //наследник 2
{   int PubsCnt;
public:
  void Input();         // ввод
  void Print() const;   // вывод
  virtual const char* GetType() const override
  { return "scientist"; } // тип класса - строка
  virtual CScientist* Clone() const override  // создание своей копии
  { return new CScientist(*this); }
};

class CManager : public CEmployee   //наследник 3
{    string m_Title;
public:
   void Input();          // ввод
   void Print() const;    // вывод
   virtual const char* GetType() const override
   { return "manage"; }  // тип класса - строка
   virtual CManager* Clone() const override  // создание своей копии
   { return new CManager(*this); }
};

// определение указателя на объект по имени типа класса
CEmployee* CreateEmployee1(const char* typeName)
{   if (strcmp(typeName, "laborer") == 0)
        return new CLaborer();
    if (strcmp(typeName, "manager") == 0)
        return new CManager();
    if (strcmp(typeName, "scientist") == 0)
        return new CScientist();
    return nullptr;
}
// определение указателя на объект по имени типа класса
// более интересный вариант для расширения вариантов кллассов
CEmployee* CreateEmployee2(const char* typeName)
{   static CEmployee* g_vEmpTmpls[3] =
   { new CLaborer(), new CScientist(), new CManager() };
    for (int i = 0; i < 3; i++)
    {   CEmployee* p = g_vEmpTmpls[i];
        if (strcmp(p->GetType(), typeName) == 0)
```

```
            return p->Clone();
    }
    return nullptr;
 }

// определение указателя на объект по имени типа класса
// через создание класса "фабрика классов"
class CEmplFactory
{  //элемент ассоциативного списка <имя-строка, указатель на объект>
    map<string, CEmployee*> m_EmpTmpls;
                              // вставка элемента в список
    void AddEmployeeType(CEmployee* pEmpl)
    {
 m_EmpTmpls.inser(pair<string, CEmployee*>(pEmpl->GetType(), pEmpl));
    }
public:
    CEmplFactory()// конструктор создания списка
    {   AddEmployeeType(new CLaborer());
        AddEmployeeType(new CManager());
        AddEmployeeType(new CScientist());
    }
// определение указателя на объект по имени типа класса
    CEmployee* Create(const char* typeName)
    {
        map<string, CEmployee*>::iterator it;
        it = m_EmpTmpls.find(typeName);
        if (it == m_EmpTmpls.end())
            return nullptr;
        return it->second->Clone();
    }
    ~CEmplFactory()  // деструктор освобождает память
    {
      for (map<string, CEmployee*>::iterator
    it = m_EmpTmpls.begin(); it != m_EmpTmpls.end(); ++it)
            delete it->second;
        m_EmpTmpls.clear();
    }
};
int main()
{ // определение указателя на объект по имени типа класса вариант 1
     CEmployee* pEmp2 = CreateEmployee1("manager");
    if (pEmp2 != NULL)
        cout << pEmp2->GetType() << endl;
// определение указателя на объект по имени типа класса вариант 2
    CEmployee* pEmp1 = CreateEmployee2("laborer");
    if (pEmp1 != NULL)
         cout << pEmp1->GetType() << endl;
// определение указателя на объект по имени типа класса вариант 3
    CEmplFactory factory;
    CEmployee* pEmp3 = factory.Create("scientist");
    if (pEmp3 != NULL)
        cout << pEmp3->GetType() << endl;
}
```

ПРИМЕР 32. Запись массива объектов в файл и чтение из файла

Иерархия наследования классов в примере 31.



В каждом классе определяем: метод заполнения полей (*Input*), метод вывода значения полей и имени типа объекта на экран (*Print*), метод определения типа класса как строку, метод создания своей копии объекта.

Добавляем метод заполнения полей из файла (*Input_file*), метод вывода значения полей и имени типа объекта в файл (*Print_file*),

Задача: используя фабрику классов, реализовать запись массива объектов в файл и заполнение массива объектов из файла.

Файл «Header.h» - содержит описание иерархии классов.

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

class CEmployee    // базовый абстрактный класс
{
     string m_Name;
     unsigned int m_ID;
public:
     virtual bool Input(istream&);                // ввод
     virtual bool Print(ostream&) const;          // вывод
     virtual bool Input_file(istream&);           // ввод из файла
     virtual bool Print_file(ostream&) const;     // вывод в файл
                    // абстрактный метод. Тип класса - строка
     virtual const char* GetType() const = 0;
                                          // абстрактный метод
     virtual CEmployee* Clone() const = 0; // создание своей копии
     // реализация ввода через оператор >>
     friend istream& operator>>(istream& in, CEmployee& s)
     {    // просто вызываем виртуальную функцию вывода
          s.Input(in);
          return in;
     }
     // реализация вывода через оператор <<
     friend ostream& operator<<(ostream& out, const CEmployee& s)
     {    s.Print(out);
          return out;
     }
};
```

```
class CLaborer : public CEmployee  //наследник 1
{ public:
     virtual const char* GetType() const override
     {    return "laborer";
     } // тип класса – строка
     // создание своей копии
     virtual CLaborer* Clone() const  override
     { return new CLaborer(*this); }
};

class CScientist : public CEmployee   //наследник 2
{    int PubsCnt;
public:
     virtual bool Input(istream&) override;         // ввод
     virtual bool Print(ostream&) const override;   // вывод
     virtual bool Input_file(istream&) override;    // ввод из файла
     virtual bool Print_file(ostream&) const override;// вывод в файл
     virtual const char* GetType() const override
     {    return "scientist";
     } // тип класса – строка
     // создание своей копии
     virtual CScientist* Clone() const override
     { return new CScientist(*this); }
};

class CManager : public CEmployee   //наследник 3
{    string m_Title;
public:
     virtual bool Input(istream&) override;          // ввод
     virtual bool Print(ostream&) const override;    // вывод
     virtual bool Input_file(istream&) override; // ввод  из файла
     virtual bool Print_file(ostream&) const override;// вывод в файл
     virtual const char* GetType() const override
     {  return "manager";
     }  // тип класса – строка
     // создание своей копии
     virtual CManager* Clone() const override
     { return new CManager(*this); }
};

// определение указателя на объект по имени типа класса
// через создание класса "фабрика классов"
class CEmplFactory
{  //элемент ассоциативного списка <имя-строка, указатель на объект>
     map<string, CEmployee*> m_EmpTmpls;
     // вставка элемента в список
     void AddEmployeeType(CEmployee* pEmpl)
     {
m_EmpTmpls.insert(pair<string, CEmployee*>(pEmpl->GetType(), pEmpl));
     }
public:
     CEmplFactory()// конструктор создания списка
     {    AddEmployeeType(new CLaborer());
```

```
            AddEmployeeType(new CManager());
            AddEmployeeType(new CScientist());
      }
      // определение указателя на объект по имени типа класса
      CEmployee* Create(const char* typeName)
      {     map<string, CEmployee*>::iterator it;
            it = m_EmpTmpls.find(typeName);
            if (it == m_EmpTmpls.end())   return nullptr;
            return it->second->Clone();
      }
      ~CEmplFactory()  // деструктор освобождает память
      {     for (map<string, CEmployee*>::iterator
                  it = m_EmpTmpls.begin(); it != m_EmpTmpls.end(); ++it)
                  delete it->second;
            m_EmpTmpls.clear();
      }
};
```

Файл «Metody.cpp» - содержит реализацию методов классов.

```
#include "Header.h"
// метод для ввода значений полей класса CEmployee с клавиатуры
bool CEmployee::Input(istream& in)
{     cout << "> Enter Name, ID ";
      in>> m_Name;   in>> m_ID;
      return in.good();
}
// метод для ввода значений полей класса CEmployee из файла
bool CEmployee::Input_file(istream& in)
{     in >> m_Name;  in >> m_ID;
      string s; getline(in, s);
      return in.good();
}
// метод для вывода значений полей класса CEmployee в файл
bool CEmployee::Print_file(ostream& out) const
 {     out << GetType() << endl;
       out << m_Name << endl;
       out << m_ID << endl;
       return out.good();
 }
// метод для вывода значений полей класса CEmployee на экран
bool CEmployee::Print(ostream& out) const
{     out << "Type: " << GetType() << endl;
      out << "Name: " << m_Name << endl;
      out << "ID: " << m_ID << endl;
      return out.good();
}
// метод для ввода значений полей класса  CScientis с клавиатуры
bool CScientist::Input(istream& in)
{     CEmployee::Input(in);
      cout << "> Enter PubsCnt ";
      in >> PubsCnt;
      return in.good();
}
```

```
// метод для ввода значений полей класса  CScientis из файла
bool CScientist::Input_file(istream& in)
{    CEmployee::Input_file(in);
     in >> PubsCnt;
     string s; getline(in, s);
     return in.good();
}
// метод для вывода значений полей класса  CScientisв файл
bool CScientist::Print_file(ostream& out) const
{    CEmployee::Print_file(out);
     out << PubsCnt << endl;
     return out.good();
}
// метод для вывода значений полей класса  CScientis на экран
bool CScientist::Print(ostream& out) const  // вывод
{    CEmployee::Print(out);
     out << "PC: " << PubsCnt << endl;
     return out.good();
}
// метод для ввода значений полей класса  CManager с клавиатуры
bool CManager::Input(istream& in)
{    CEmployee::Input(in);
     cout << "> Enter Title ";
     in >> m_Title;
     return in.good();
}
// метод для ввода значений полей класса  CManager из файла
bool CManager::Input_file(istream& in)
{    CEmployee::Input_file(in);
     in >> m_Title;
     string s; getline(in, s);
     return in.good();
}
// метод для вывода значений полей класса  CManager в файл
bool CManager::Print_file(ostream& out) const
{    CEmployee::Print_file(out);
     out << m_Title << endl;
     return out.good();
}
// метод для вывода значений полей класса  CManager на экран
bool CManager::Print(ostream& out) const  // вывод
{    CEmployee::Print(out);
     out << "Title: " << m_Title << endl;
     return out.good();
}
```

Файл «Massiv.cpp» - содержит функции для работы с массивом объектов.

```
#include "Header.h"
// определение указателя на объект по имени типа класса
CEmployee* CreateEmployee1(const char* typeName)
{    if (strcmp(typeName, "laborer") == 0)
          return new CLaborer();
```

```
     if (strcmp(typeName, "manager") == 0)
          return new CManager();
     if (strcmp(typeName, "scientist") == 0)
          return new CScientist();
     return nullptr;
}

// определение указателя на объект по имени типа класса
// более интересный вариант для расширения вариантов классов
CEmployee* CreateEmployee2(const char* typeName)
{    static CEmployee* g_vEmpTmpls[3] =
     { new CLaborer(), new CScientist(), new CManager() };
     for (int i = 0; i < 3; i++)
     {
          CEmployee* p = g_vEmpTmpls[i];
          if (strcmp(p->GetType(), typeName) == 0)
               return p->Clone();
     }
     return nullptr;
}

void ClearArr(vector<CEmployee*>& arr)    // освобождение памяти
{
     for (unsigned int i = 0; i < arr.size(); i++)
          delete arr[i];
     arr.clear();
}
// занесение элементов в массив через выбор типа и ввода с клавиатуры
значений полей
bool Vvod(vector<CEmployee*>& arr, unsigned int n)
{
     ClearArr(arr);        //если передан не пустой массив - очистить
     while (arr.size() < n)
     {    int type = 0;
          cout << "Input type of object:\n";
          cout << "laborer - 1; manager- 2 ; scientist - 3\n";
          cin >> type;
          CEmployee* pEmpl = nullptr;
          if (type == 1) // добавить laborer
               pEmpl = new CLaborer();
          else if (type == 2) // добавить manager
               pEmpl = new CManager();
          else if (type == 3) // добавить scientist
               pEmpl = new CScientist();
          else
          {
               cout << "Error";
               return false;
          }
          cin >> *pEmpl;       // ввод полей
          arr.push_back(pEmpl);
     }
     return true;
```

```
}
// занесение элементов в массив из файла
bool Vvod_file(vector<CEmployee*>& arr, istream& f)
{    ClearArr(arr);//если передан не пустой массив - очистить
     // загрузить количество элементов  из файла
     int n;
     f >> n;  // чтение из файла количества элементов
     string t; getline(f, t);   // очистка буффера
     for (unsigned int i = 0; i < n; i++)
     {    string tip;
          getline(f, tip);   // чтение типа класса (строка)
          if (tip.length() == 0)
               return false;
// определение указателя на объект по имени типа класса
          CEmployee* p = CreateEmployee1(tip.c_str());
//CEmployee* p = CreateEmployee2(tip.c_str());  // или так
//CEmplFactory factory; CEmployee* p = factory.Create(tip.c_str());
// или так
          if (!p)
               return false;
          if (!p->Input_file(f))   // метод чтения объекта из файла
          {    delete p;
               return false;
          }
          arr.push_back(p);  // добавляем в массив
     }
     return true;
}

// вывод элементов массива на экран
void Vyvod(const vector<CEmployee*>& arr)
{
     for (unsigned int i = 0; i < arr.size(); i++)
          cout << *arr[i];
}
// вывод элементов массива в файл
void Vyvod_file(const vector<CEmployee*>& arr, ofstream& f)
{
 f << arr.size() << endl; // запись в файл количество объектов массива
 for (unsigned int i = 0; i < arr.size(); i++) {
     arr[i]->Print_file(f);  // метод записи объекта в файл
 }
}
```

Файл «Fabric_class.cpp» - содержит основную программу.

```
#include "Header.h"
// освобождение памяти
void ClearArr(vector<CEmployee*>&);
// занесение элементов в массив через выбор типа и ввода с клавиатуры
значениц полей
bool Vvod(vector<CEmployee*>& , unsigned int);
// занесение элементов в массив из файла
bool Vvod_file(vector<CEmployee*>& , istream&);
```

```
// вывод элементов массива на экран
void Vyvod(const vector<CEmployee*>& );
// вывод элементов массива в файл
void Vyvod_file(const vector<CEmployee*>& , ofstream&);
```

```
int main()
{     vector<CEmployee*> members1; // массив объектов
      int n;
      cout << "vvod n= ";
      cin >> n;

      Vvod(members1, n);  // ввод n объектов в массив
      Vyvod(members1);  // вывод на экран массива объектов
      cout << "---------------" << endl;

      ofstream file("1.txt");
      Vyvod_file(members1, file);   // запись массива в файл
      file.close();

      ClearArr(members1);     // освободить память

      vector<CEmployee*> members2; // массив объектов

      ifstream ifile("1.txt");
      bool p = Vvod_file(members2, ifile); // чтение объектов из файла
      ifile.close();

      if (!p)
            return 0;

      Vyvod(members2); // вывод на экран массива объектов

      ClearArr(members2); // освободить память

      return 1;
}
```