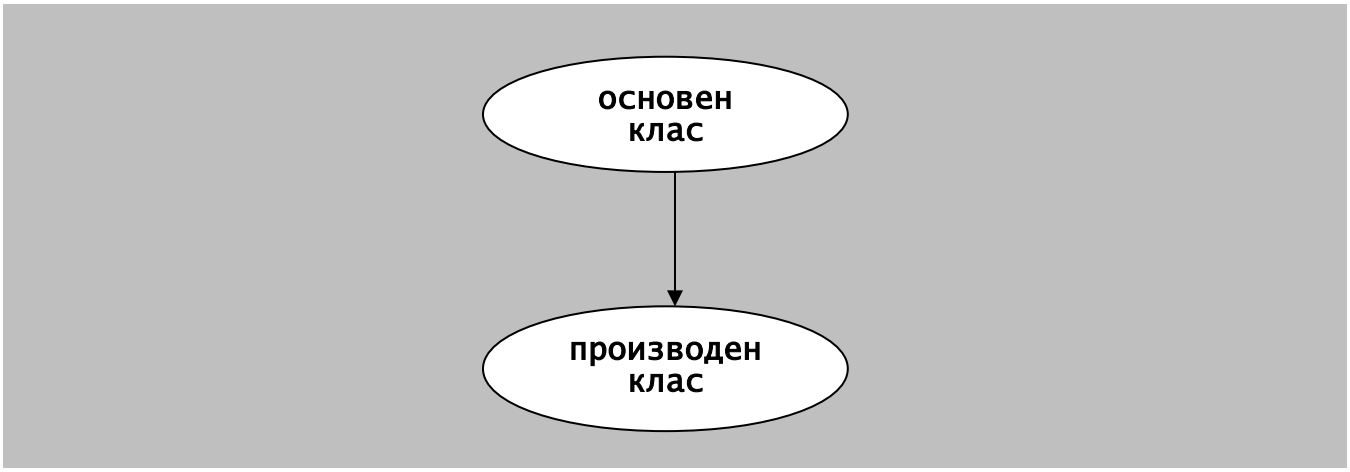
# 17

# Наследяване. Производни класове

Производните класове и наследяването са една от най-важните характеристики на обектно-ориентираното програмиране (ООП). Чрез механизма на наследяване от съществуващ клас се създава нов клас. Класът от който се създава се нарича **базов (основен) клас**, а този, който е създаден - **производен**. На фиг. 17.1 е показана най-проста връзка между основен и производен клас.




Фиг. 17.1 Най-проста връзка между основен и производен клас

Понятията основен и производен клас са относителни, тъй като производен клас може да е основен за други класове, а основен – да е производен от други основни класове. Производният клас може да наследи компонентите на един или няколко базови класа. В първия

случай наследяването се нарича **единично (просто)**, а във втория – **множествено**.

Дефинирането на производни класове е еквивалентно на конструирането на йерархии от класове. *Защо се налага дефинирането на производни класове? В кои случаи и как се прави това? Какви са предимствата от дефинирането на производни класове?* На тези въпроси ще дадем отговор в следващите разглеждания.

Ако множество от класове имат общи данни и методи, тези общи части могат да се обособят като основни класове, а всяка от останалите части да се дефинира като производен клас на съответния основен клас. Така се прави икономия на памет, тъй като се избягва многократното описание на едни и същи програмни фрагменти.

При конструирането на производни класове е достатъчно да се разполага само с обектните модули на основните класове, а не с техния програмен код. Това позволява да бъдат създавани библиотеки от класове, които да бъдат използвани при създаването на производни класове.

Тези предимства, а също възможността за реализиране на полиморфизъм, мотивират въвеждането на производни класове.

## 17.1 Дефиниране на производни класове

Подобно на обикновените, производните класове се дефинират като се *декларира класът* и се *дефинират неговите методи*. Синтаксисът на декларацията е даден на Фиг. 17.2.

```
Деклариране на производен клас
<декларация_на_производен_клас> ::=
class <име_на-производен_клас> :
      [<атрибут_за_област>]опц <име_на_базов_клас>
      {, [<атрибут_за_област>]опц <име_на_базов_клас>}опц
{<декларация_на_компоненти>
};
<име_на-производен_клас> ::= <идентификатор>
<атрибут_за_област> ::= public | private | protected
<име_на_базов_клас> ::= <идентификатор>
```

Фиг. 17.2 Деклариране на производен клас

Пред всяко име на базов клас *може* да се постави запазената дума public, private или protected. Нарича се **атрибут за област**, тъй като определя областта на наследените членове. Употребата на атрибутите за област е различна от тази за обявяване на секции в тялото на класа. Ако атрибут за област е пропуснат, подразбира се private. Атрибутът protected е включен в новите версии на езика и не се използва много често.

**Примери:**

1. Декларацията:

```
class der : base1, base2, base3
{...
};
```

определя производен клас der с три основни класа base1, base2 и base3. Тъй като атрибутът за област е пропуснат и за трите базови класа, подразбира се private, т.е. декларацията е еквивалентна на

```
class der : private base1, private base2, private base3
{...
};
```

2. Декларацията:

```
class der : public base1, base2, base3
{...
};
```

е еквивалентна на

```
class der : public base1, private base2, private base3
{...
};
```

3. Декларацията:

```
class der : protected base1, base2, public base3
{...
};
```

е еквивалентна на

```
class der : protected base1, private base2, public base3
{...
```

};

*Декларациите на компонентите на производен клас, а също дефинициите на неговите методи не се различават от съответните при обикновените класове.*

Множеството от компонентите на един производен клас се състои от компонентите на неговите базови класове и компонентите, декларирани в самия производен клас. Оттук произлиза и терминът наследяване. Механизмът, чрез който производният клас получава компонентите на базовия, се нарича наследяване. Когато производният клас има няколко базови класа, той наследява компонентите на всеки от тях. Наследяването в този случай е множествено.

*Процесът на наследяване се изразява в следното:*

- наследяват се данните и методите на основния клас;
- получава се достъп до някои от наследените членове на основния клас;
- производният клас "познава" реализацията само на основния клас, от който произлиза;
- производният клас може да е основен за други класове.

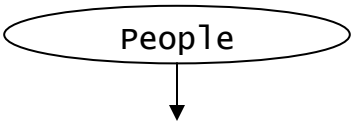
*Производният клас може да дефинира допълнително:*

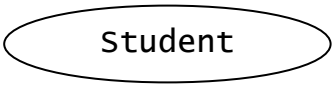
- свои член-данни;
- методи, аналогични на тези на основния клас, а също и нови.

Дефинираните в производния клас данни и методи се наричат **собствени**. Чрез примери ще илюстрираме дефинирането на производни класове с единично наследяване.

**Задача 157.** Да се напише програма, която дефинира клас People, определящ човек по име и единен граждански номер (ЕГН), а също производен клас Student на класа People, който определя понятието студент като човек, който има факултетен номер и среден успех. Да се дефинира обект от клас Student и се изведе дефинираният обект.

Програма Zad157.cpp решава задачата. Тя реализира йерархията:






```
// Program Zad157.cpp
#include <iostream.h>
#include <string.h>
// дефиниция на базовия клас People
class People
{public:
  void ReadPeople(char *, char *);
  void PrintPeople() const;
 private:
  char * name;
  char * egn;
};
void People::ReadPeople(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
void People::PrintPeople() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
}
// дефиниция на производния клас Student
class Student : People
{public:
  void ReadStudent(char *, char *, long, double);
  void PrintStudent() const;
 private:
  long facnom;
  double usp;
};
void Student::ReadStudent(char *str, char * num,
                          long facn, double u)
{ReadPeople(str, num);
```

```
 facnom = facn;
 usp = u;
}
void Student::PrintStudent() const
{PrintPeople();
 cout << "fac. nomer: " << facnom << endl;
 cout << "uspeh: " << usp << endl;
}
int main()
{Student stud;
 stud.ReadStudent("Ivan Ivanov", "8206123422", 42444, 6.0);
 stud.PrintStudent();
 return 0;
}
```

Чрез класа People е представено понятието човек, характеризиращо човек с име и ЕГН, реализирани чрез член-данните name и egn от тип char*. Капсулирани са чрез декларирането им като private. Освен член-данни класът съдържа и методите ReadPeople и PrintPeople, образуващи интерфейса на класа (обявени са като public). Чрез ReadPeople се инициализират обектите на класа People, а чрез PrintPeople се извеждат върху екрана стойностите на член-данните name и egn. Ще напомним, че заделената от методите динамична памет не се освобождава автоматично при унищожаване на обектите. Освобождаването на тази памет трябва да стане явно, чрез оператора delete. В тази част умишлено методът ReadPeople не е реализиран като конструктор, не е дефиниран също и деструктор. Това е заради някои особености при дефинирането на тези методи, които особености ще разгледаме по-късно в тази глава.

Класът Student, дефиниран в програмата, представя понятието студент, като реализира следното определение: Студент, това е човек, който има факултетен номер и се характеризира със среден успех. Това дава основание Student да бъде определен като производен клас на класа People. В резултат, класът Student има осем компоненти. Четири от тях (name, egn, ReadPeople и PrintPeople) са наследени от базовия клас People и четири (facnom, usp, ReadStudent и PrintStudent) са

декларирани в него. Тъй като не е указан атрибут за област на базовия клас People, подразбира се private. В този случай производният клас Student наследява всички член-данни и член-функции на основния клас като private. Освен това той получава възможността да използва всички компоненти на основния клас, които не са private (в случая ReadPeople и PrintPeople). Така член-функциите ReadStudent и PrintStudent нямат пряк достъп до наследените член-данни name и egn. Затова достъпът е реализиран чрез методите ReadPeople и PrintPeople.

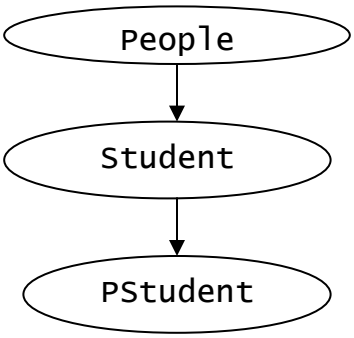
Производният клас е дефиниран след като вече е дефиниран базовият клас, от който той произлиза. Чрез него се разширява декларацията на съществуващ клас. Разширяемостта на класовете е една от важните характеристики на ООП.

Чрез следващата задача ще покажем възможността производен клас да е основен за друг клас, т.е. да бъде създадена верига от наследени класове.

**Задача 158.** Да се дефинира клас PStudent, производен на класа Student, реализиращ понятието студент от платена форма на обучение.

```
class PStudent : public Student
{public:
  void ReadPStudent(char *, char *, long, double, double);
  void PrintPStudent() const;
 private:
  double tax; // такса за обучението на студента
};
void PStudent::ReadPStudent(char *str, char *num, long facn,
                            double u, double t)
   {ReadStudent(str, num, facn, u);
    tax = t;
   }
   void PStudent::PrintPStudent() const
   {PrintStudent();
    cout << "Tax: " << tax << endl;
   }
```

Връзката между класовете People, Student и PStudent е следната:




Класът PStudent е производен на класа Student с атрибут за област public. В този случай PStudent наследява всички компоненти на класа Student (собствени и наследени от People) като запазва вида им, т.е. собствените методи ReadStudent и PrintStudent продължават да са public, а собствените член-данни facnom и usp и всички наследени от People продължават да са private в класа PStudent. Това е така, тъй като атрибутът за област на класа Student е private, заради което всички компоненти на People са наследени от Student като private и отново като private се наследяват и от класа PStudent.

От тези примери се вижда, че на атрибута за област е отредена важна роля. Семантиката му ще разгледаме в следващата част.

## 17.2 Наследяване и достъп до наследените компоненти

Ще напомним, че в рамките на един клас (без наследяване), protected частта има аналогична роля като тази на private частта. До компоненти от тип protected имат пряк достъп само член-функции и приятелски функции на класа.

Атрибутът за област на базовия клас в декларацията на производния клас (public, private или protected) управлява механизма на наследяване и определя какъв да бъде режимът на достъп до наследените членове. Таблицата от Фиг. 17.3 показва наследяванията на компоненти на основен клас в зависимост от атрибута за област.

| Атрибут за област | Компонента на основен клас, определена като | Наследява се като |
|---|---|---|
| public | private<br>public<br>protected | private<br>public<br>protected |
| private | private<br>public<br>protected | private<br>private<br>private |
| protected | private<br>public<br>protected | private<br>protected<br>protected |

Фиг. 17.3 Наследявания на компоненти на основен клас в производен

От таблицата се вижда, че:

· Ако базовият клас е деклариран като public в производния клас, всички private, public и protected компоненти на базовия клас се наследяват съответно като private, public и protected компоненти на производния клас.

**Пример:** Ако

```
class base                  class der1 : public base
{private: int b1;           {private: int d1;
 protected: int b2;          protected: int d2;
 public: int b3();           public: int d3();
};                          };
```

можем да си мислим, че der1 е клас от вида:

```
class der1
{private:
   int b1;
   int d1;
 protected:
   int b2;
   int d2;
 public:
   int b3();
   int d3();
```

```
};
```

· Ако базовият клас е деклариран като private в производния клас, всички негови компоненти се наследяват като private.

**Пример:** Ако

```
class base                  class der2 : private base
{private: int b1;           {private: int d1;
 protected: int b2;          protected: int d2;
 public: int b3();           public: int d3();
};                          };
```

можем да си мислим, че der2 е клас от вида:

```
class der2
{private:
  int b1;
  int b2;
  int b3();
  int d1;
 protected:
  int d2;
 public:
  int d3();
};
```

· Ако базовият клас е деклариран като protected в производния клас, private компонентите му се наследяват като private, а public и protected – като protected.

**Пример:** Ако

```
class base                  class der3 : protected base
{private: int b1;           {private: int d1;
 protected: int b2;          protected: int d2;
 public: int b3();           public: int d3();
};                          };
```

можем да си мислим, че der3 е клас от вида:

```
class der3
{private:
  int b1;
```

```
  int d1;
 protected:
  int b2;
  int d2;
  int b3();
 public:
  int d3();
};
```

*Наследените компоненти обаче се различават от декларираните в производния клас по правата за достъп. Производният клас има пряк достъп до компонентите, декларирани като public и protected, но няма пряк достъп до декларираните като private в базовия клас.* Достъпът до private компонентите на базовия клас се извършва чрез неговия интерфейс.

Таблицата от Фиг. 17.4 показва прекия достъп на член-функции на производния клас (ПД) и външния достъп на производния клас (ВД) до компонентите на базовия клас.

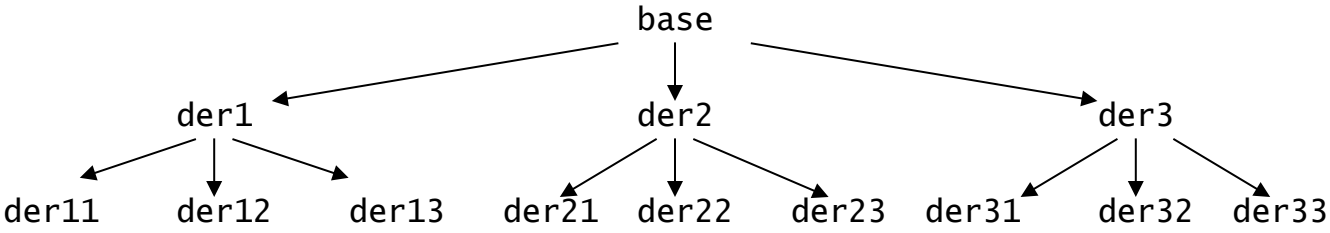
| компонента на базов клас | производен клас с атрибут public | | производен клас с атрибут private | | производен клас с атрибут protected | |
|---|---|---|---|---|---|---|
| | ПД | ВД | ПД | ВД | ПД | ВД |
| public | да | да | да | не | да | не |
| protected | да | не | да | не | да | не |
| private | не | не | не | не | не | не |

Фиг. 17.4 Достъп на член-функции и на обекти на производния клас до компонентите на базовия клас

Да се върнем към означенията от последните три примера. Собствените компоненти на класа der1 са видими навсякъде в класа. Те имат пряк достъп до компонентите b2 и b3() на base, но нямат пряк достъп до private-компонентата b1 на base. Същото се отнася и за класовете der2 и der3. Освен това, обект от клас der1 има пряк достъп public-компонентите b3() – наследена и d3() – собствена за der1; обект от клас der2 има пряк достъп единствено до собствената public-компонента d3(), тъй като всички наследени от base компоненти се наследяват като private и обект от клас der3 има също пряк достъп

единствено до собствената public-компонента d3(), тъй като public и protected компонентите на base се наследяват като protected в der3.

С цел разясняване на процеса на наследяване, ще разгледаме и следната йерархия на класове:




с декларации:

```
class base
{private: int a1;
 protected: int a2;
 public: int a3();
} b;
class der1: public base  class der2: private base  class der3:protected base
{private: int a4;         {private: int a7;        {private: int a10;
 protected: int a5;        protected: int a8;       protected: int a11;
 public: int a6();         public: int a9();        public: int a12();
} d1;                     } d2;                    } d3;
class der11:public der1  class der12:private der1 class der13:protected der1
{private: int a13;        {private: int a16;       {private: int a19;
 protected: int a14;       protected: int a17;      protected: int a20;
 public: int a15();        public: int a18();       public: int a21();
} d4;                     } d5;                    } d6;

class der21:public der2  class der22:private der2 class der23:protected der2
{private: int a22;        {private: int a25;       {private: int a28;
 protected: int a23;       protected: int a26;      protected: int a29;
 public: int a24();        public: int a27();       public: int a30();
} d7;                     } d8;                    } d9;

class der31:public der3  class der32:private der3 class der33:protected der3
{private: int a31;        {private: int a34;       {private: int a37;
 protected: int a32;       protected: int a35;      protected: int a38;
```

```
 public: int a33();        public: int a36();        public: int a39();
} d10;                    } d11                    } d12;
```

Всеки обект на класовете der1, der2 и der3 притежава 6 компоненти – 3 собствени и 3 наследени от базовия клас base, а всеки от класовете deri1, deri2 и deri3 притежава 9 компоненти: 3 собствени, 6 наследени от клас deri (i = 1, 2, 3).

Ще разгледаме достъпа на компонентите, дефинирани в класовете derij (i, j = 1, 2 3) до компонентите на класовете, предшестващи ги в йерархията.

Член-функциите на класа der1i (i = 1, 2, 3) имат пряк достъп до (могат да използват) всички собствени компоненти, до наследените като protected и public a5 и a6() на класа der1 и до наследените като protected и public a2 и a3() на класа base. Достъпът им до a1 и a4 се осъществява чрез интерфейса на съответните базови класове.

Член-функциите на класа der2i (i = 1, 2, 3) имат пряк достъп до (могат да използват) всички собствени компоненти и до наследените като protected и public a8 и a9() на класа der2. Тъй като базовият на класа der2 клас base е с атрибут за област private, компонентите на base се наследяват от der2 като private. Това е причината те да не са достъпни за член-функциите на класовете der2i (i = 1, 2, 3).

Член-функциите на класа der3i (i = 1, 2, 3) имат пряк достъп до (могат да използват) всички собствени компоненти, до наследените като protected и public собствени компоненти a11 и a12() на класа der3 и до наследените като protected a2 и a3() на класа base. Достъпът им до останалите компоненти се осъществява чрез интерфейса на съответните базови класове.

Външният достъп до компонентите на класовете ще определим чрез достъпа на дефинираните обекти на класовете от горната йерархия.

| **обект** | **възможност за достъп до компонента:** |
|---|---|
| b | a3() |
| d1 | a6(), a3() |
| d2 | a9() |
| d3 | a12() |
| d4 | a15(), a6(), a3() |
| d5 | a18() |

| | |
|---|---|
| d6 | a21() |
| d7 | a9(), a24() |
| d8 | a27() |
| d9 | a30() |
| d10 | a12(), a33() |
| d11 | a36() |
| d12 | a39() |

Ще се спрем на някои от често срещаните случаи за достъп до членове на производен и основен клас, а също на достъпа на външни функции до наследен компонент. Ще изкажем и някои правила за достъп до компоненти на базови и производни класове, които ще подкрепим с още примери.

**· Достъп до членове на основен клас чрез дефиниции на методи на производен клас**

**В сила са следните правила за достъп:**

*- Методите на производен клас (без значение на атрибута за област) нямат директен достъп до членовете от private-секцията на основния му клас*

**Примери:**

а) Класът Student е производен на класа People. Атрибутът за област не е указан явно, заради което се подразбира private. Методите на Student нямат пряк достъп до private членовете name и egn на People.

б) Класът PStudent е производен на Student. Атрибутът за област е public. От Фиг. 17.3 следва, че видът на наследените секции на Student се запазва. Методите на PStudent нямат пряк достъп както до собствените private компоненти facnom и usp на Student, така и до наследените от People private членовете name и egn.

Ще отбележим също,че тъй като атрибутът за област на класа People в Student е private, всички компоненти на People са private в Student и са недостъпни пряко в PStudent.

*- В дефинициите на собствени методи на производния клас могат да се използват методите от секциите public и protected на основния му клас*

**Примери:**

а) Тъй като методите на Student нямат пряк достъп до name и egn, инициализацията на тези компоненти в ReadStudent става чрез метода ReadPeople на класа People, който е обявен в public секцията на People.

б) Тъй като методите на PStudent нямат пряк достъп до facnom, usp, name и egn, инициализацията им в ReadPStudent става чрез метода ReadStudent на класа Student, който е обявен в public секцията на Student.

*- В дефинициите на собствени методи на производния клас може директно да се използват член-данните на секцията protected на основния му клас*

Ще илюстрираме това правило като извършим промени в програма Zad157.cpp.

**Задача 159.** Да се промени програма Zad157.cpp така, че освен класовете People и Student да включва и наследения от Student клас PStudent. Освен това, методите на производните класове да могат пряко да използват наследените член-данни на основните им класове.

```
// Program Zad159.cpp
#include <iostream.h>
#include <string.h>
class People
{public:
  void ReadPeople(char *, char *);
  void PrintPeople() const;
 protected: // вместо private:
  char * name;
  char * egn;
};
void People::ReadPeople(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
```

```
 strcpy(egn, num);
}
void People::PrintPeople() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
}
class Student : public People // вместо private:
{public:
  void ReadStudent(char *, char *, long, double);
  void PrintStudent() const;
 protected: // вместо private
  long facnom;
  double usp;
};
void Student::ReadStudent(char *str, char * num,
                          long facn, double u)
{name = new char[strlen(str)+1]; // използваме член-данните
 strcpy(name, str);              // name и egn
 egn = new char[11];
 strcpy(egn, num);
 facnom = facn;
 usp = u;
}
void Student::PrintStudent() const
{cout << "Ime: " << name << endl;   // използваме член-данните
 cout << "EGN: " << egn << endl;    // name и egn
 cout << "fac. nomer: " << facnom << endl;
 cout << "uspeh: " << usp << endl;
}
class PStudent : public Student
{public:
  void ReadPStudent(char *, char *, long, double, double);
  void PrintPStudent() const;
 protected:
  double tax;
};
```

```
void PStudent::ReadPStudent(char *str, char *num, long facn,
                            double u, double t)
{name = new char[strlen(str)+1]; // пряк достъп до
 strcpy(name, str);              // name, egn, facnom и usp
 egn = new char[11];
 strcpy(egn, num);
 facnom = facn;
 usp = u;
 tax = t;
}
void PStudent::PrintPStudent() const
{cout << "Ime: " << name << endl;      // пряк достъп до
 cout << "EGN: " << egn << endl;       // name, egn, facnom и usp
 cout << "fac. nomer: " << facnom << endl;
 cout << "uspeh: " << usp << endl;
 cout << "Tax: " << tax << endl;
}
int main()
{PStudent stud;
 stud.ReadPStudent("Ivan Ivanov", "8206123422", 42444, 6.0, 1000);
 stud.PrintPStudent();
 return 0;
}
```

Методите ReadStudent и PrintStudent на дефинирания в тази програма производен клас Student имат пряк достъп до член-данните name и egn на своя базов клас People, тъй като последните са декларирани като protected. Аналогично, методите ReadPStudent и PrintPStudent на дефинирания също в тази програма производен клас PStudent имат пряк достъп до собствените член-данни facnom и usp на базовия си клас Student, тъй като последните са декларирани като protected и до name и egn на базовия клас People за класа Student, тъй като те са декларирани в People също като protected, но атрибутът за област на People в Student е public, заради което се наследяват от Student като protected. Функцията main е външна както за класа Student, така и за PStudent и няма пряк достъп до техните protected компоненти. Затова

компилаторът ще сигнализира грешка при опит за пряк достъп до компонентите name, egn, facnom и usp в main.

· **Достъп до методи чрез обекти на основния и производния клас**

Обект на основен клас има пряк достъп до всички свои компоненти, обявени като public и няма пряк достъп до компонентите, обявени като private и protected. Обект на производен клас има пряк достъп до public компонентите на собствения си и компонентите на основния клас, наследени в производния клас като public. От фиг. 17.3 се вижда, че последното е възможно, ако атрибутът за област на производния клас е public и компонентата е в public секция на основния клас.

Нека сме в дефинициите на Zad159.cpp и дефинираме следните обекти:

```
People pe;
Student stud;
PStudent pstud;
```

Ще напомним, че Student е производен клас на основен клас People с атрибут за област public, а PStudent е производен клас на основния клас Student също с атрибут за област public. Обектът pe има пряк достъп до public методите на People, stud има пряк достъп до public методите на People и Student, а pstud има пряк достъп до public методите на People, Student и PStudent, т.е. допустими са обръщенията:

```
pe.ReadPeople("Ivan Ivanov", "5804134986");
pe.PrintPeople();
stud.ReadStudent("Pavel Dimov", "4806193046", 30100, 4.50);
stud.ReadPeople("Pavel Dimov", "4806193046");
stud.PrintPeople();
stud.PrintStudent();
pstud.ReadPStudent("Pavel Dimov","4806193046", 30100, 4.50, 500);
pstud.ReadStudent("Pavel Dimov", "4806193046", 30100, 4.50);
pstud.ReadPeople("Pavel Dimov", "4806193046");
pstud.PrintPeople();
pstud.PrintStudent();
pstud.PrintPStudent();
```

Да се върнем към програма Zad157.cpp. В нея класът People е основен на класа Student с атрибут за област private. Student наследява всички секции на People като private. Следователно обектите на Student нямат пряк достъп до методите на People. Ако имаме дефинициите:

```
People pe;
Student stud;
```

допустими са обръщенията

```
pe.ReadPeople("Ivan Ivanov", "5804134986");
pe.PrintPeople();
stud.ReadStudent("Pavel Dimov", "4806193046", 30100, 4.50);
stud.PrintStudent();
```

а обръщенията:

```
stud.ReadPeople("Ivan Ivanov", "5804134986");
stud.PrintPeople();
```

са недопустими.

**· Достъп на основен клас до членове на производен клас и обратно**

Методите на основния клас нямат достъп до членове на производен клас. Причината е, че когато основният клас се дефинира, не е ясно какви производни класове ще произхождат от него.

Производният клас също няма привилигирован достъп до членове на основния клас. От фиг. 17.3 се вижда, че производните класове нямат достъп до методите, обявени като private в основния клас.

Допустими са редица присвоявания между обекти на основния и производния клас. Ще ги разгледаме подробно в следващи части на главата. Засега ще отбележим само, че за реализирането им се извършват редица преобразувания.

**· Достъп на функции приятели на производен клас до компоненти на основния му клас**

Ще напомним, че функциите-приятели на клас не са елементи на класа, на който са приятели. Те са външни функции, получили привилигирован достъп до компонентите на класа.

Функциите приятели на производен клас имат същите права на достъп като член-функциите на производния клас. *Имат пряк достъп до всички компоненти, декларирани в класа и до public и protected компонентите на основния клас. Декларацията за приятелство не се наследява. функция приятел на базовия клас не е приятел (освен ако не е декларирана като такава) на производния клас.*

**Пример:** В резултат от изпълнението на програмата:

```
#include <iostream.h>
class base
{private: int a1;
 protected: int a2;
 public:
 void readbase(int x, int y)
 {a1 = x;
  a2 = y;
 }
 void a3() const
 {cout << "a1: " << a1 << endl
       << "a2: " << a2 << endl;
 }
};
class der : public base
{private: int d1;
 protected: int d2;
 public:
 friend void f(der &d, int x, int y, int z)
 {cout << "friend function f(): " << endl;
  d.d1 = x;
  d.d2 = y;
  d.a2 = z;
  cout << "d.a3(): " << endl;
  d.a3();
  cout << "d.d3(): " << endl;
  d.d3();
 }
 void readder(int x, int y, int z, int t)
```

```
     {readbase(x, y);
      d1 = z;
      d2 = t;
     }
    void d3() const
    {cout << "d1: " << d1 << endl
          << "d2: " << d2 << endl
          << "a2: " << a2 << endl;
     cout << "a3():" << endl;
     a3();
     }
    };
    void main()
    {der x;
     x.readder(10, 20, 30, 40);
     x.d3();
     f(x, 100, 200, 300);
    }
се получава:
    d1: 30
    d2: 40
    a2: 20
    a3():
    a1: 10
    a2: 20
    friend function f()
    d.a3()
    a1: 10
    a2: 300
    d.d3():
    d1: 100
    d2: 200
    a2: 300
    a3():
    a1: 10
    a2: 300
```

Ще отбележим, че ако der е основен клас за класа der1, функцията приятел f, дефинирана в примера по-горе, е видима в класа der1, но f не е функция приятел за der1.

**Забележки:**

1. Използването на секция protected позволява пряк достъп на производния клас до нейните компоненти. По такъв начин се нарушава принципът на капсулиране на данните, но пък дадената "привилегия" повишава ефективността на генерирания код.

2. Дефинирането на производни класове с атрибут за област private предизвиква забрана на достъпа на обект на класа до интерфейса на базовия му клас. Това се прави когато не трябва да се използва интерфейса на базовия клас, а се налага да се преработи и всички негови полезни функции да бъдат предефинирани.

## 17.3 Предефиниране на компоненти

Базовият и производният клас могат да притежават собствени компоненти с еднакви имена. В този случай производният клас ще притежава компоненти с еднакви имена. Обръщението към такава компонента чрез обект от производния клас извиква декларираната в класа компонента, т.е. *името на собствената компонента е с по-висок приоритет от това на наследената*. За да се изпълни "покритата" наследена компонента се указва пълното й име, т.е.

<име_на_клас>::<компонента>

където <име_на_клас> е името на основния клас.

**Пример:** В резултат от изпълнението на програмата

```
#include <iostream.h>
class base
{public:
 void init(int x)
 {bx = x;
 }
 void display() const
 {cout << " class base: bx= " << bx << endl;
 }
```

```
 protected:
  int bx;
 private:
// …
};
class der: public base
{public:
 void init(int x)
 {bx = x;
  base::bx = x + 5;
 }
 void display() const
 {cout << " class der: bx = " << bx;
  cout << " base::bx = " << base::bx << endl;
 }
 protected:
  int bx;
 private:
 //...
};
void main()
{base b;
 der d;
 b.init(5); d.init(10);
 b.display(); d.display();
 d.base::init(20);
 d.base::display();
 d.display();
 b.display();
}
```

се получава:

```
class base: bx = 5
class der: bx = 10 base::bx = 15
class base: bx = 20
class der: bx = 10 base::bx = 20
class base: bx = 5
```

## 17.4 Конструктори, операторни функции за присвояване и деструктори на производни класове

Обикновените конструктори, конструкторът за присвояване, операторната функция за присвояване и деструкторът са методи, за които не важат правилата за достъп при наследяване. ***Тези методи (с някои изключения) не се наследяват от производния клас.*** Ако например конструктор можеше да бъде наследен, той щеше да инициализира само наследената част. Нормално е конструкторът на производен клас да инициализира както наследената, така и собствената част на класа. Същото се отнася и за деструкторът. Това е причината, заради която конструкторите и деструкторът на основния клас не се наследяват от производния клас. Възможно е обаче конструкторът на производния клас да активира конструктор на основния клас, който пък да инициализира наследената част. Производният клас не наследява и създадените от програмиста конструктор за присвояване и предефинирания оператор за присвояване на обекти =.

Следователно, голямата четворка на основния клас не се наследява от производния клас. Това е следствие на особената роля на четворката.

Дефинирането и използването на голямата четворка за производния клас ще разгледаме на няколко стъпки. За да разграничим конструктора за присвояване от останалите конструктори, последните ще наричаме обикновени или само конструктори.

### 17.4.1 Обикновени конструктори и деструктор

**Конструктори**

Обикновените конструктори на базовия и на производния клас изпълняват инициализиращи функции. Принципен е въпросът *как и от кого да се реализира инициализирането на наследената част на производния клас.* Най-естествено е това да се направи от конструкторите на производния клас. Но ако това е така, конструкторите на производния

клас трябва да имат достъп до наследените, при това най-често, private компоненти на основния клас. Това е в противоречие на принципа за капсулиране на информацията. Затова инициализирането на собствените и наследените членове е разделено между конструкторите на производния и основния клас.

Конструкторите на производния клас инициализират само собствените член-данни на класа. Наследените член-данни на производния клас се инициализират от конструктор на основния клас. Това се осъществява като в дефиницията на конструктора на производния клас се укаже обръщение към съответен конструктор на основния клас (Фиг. 17.4).

**Дефиниция на конструктор на производен клас**

**Синтаксис**

<дефиниция_на_конструктор_на_производен_клас> ::=
<име_на_производен_клас>::<име_на_производен_клас>(<параметри>)
<инициализиращ_списък>
{<тяло>
}
<инициализиращ_списък> ::= <празно>|
: <име_на_основен_клас>(<параметри$_i$>)
{,<име_на_основен_клас>(<параметри$_i$>)}$_{0}^{пц}$
<параметри> ::= <празно>|
<параметър>|
<параметри>, <параметър>
<параметър> ::= <тип> <име_на_параметър>
<име_на_параметър> ::= <идентификатор>
<параметри$_i$> е конструкция, която има синтаксиса на <фактически_ параметри> от дефиницията на обръщение към функция (глава 8), а <тяло> се определя като тялото на който и да е конструктор.

Фиг. 17.4 Дефиниция на конструктор на производен клас

При единично наследяване в инициализиращия списък на производния клас е указано не повече от едно обръщение към конструктор на основен клас. При множествено наследяване в инициализиращия списък може да са

указани няколко обръщения към конструктори на основни класове. За разделител се използва символът запетая.

**Забележка.** Обръщенията към конструкторите на основни класове се обявяват в дефиницията на конструктора на производния клас, а не в неговата декларация в тялото на производния клас.

В тази глава ще останем в означенията на единичното наследяване.

Ще отбележим също, че <параметри$_i$> са изрази или идентификатори, съответстващи по брой, тип и смисъл на формалните параметри на съответния конструктор на базовия клас, т.е. обръщението

<име_на_основен_клас>(<параметри$_i$>)

трябва да се оформи според дефиницията на конструктора на основния клас. Имената на параметри от конструктора на производния клас могат да се използват за фактически параметри в обръщението към конструктора на основния клас.

**Пример:** Да разгледаме следните изкуствени класове:

```
// дефиниция на базовия клас base
class base
{private: int a1;
 protected: int a2;
 public:
 base()  // конструктор по подразбиране
 {a1 = 0;
  a2 = 0;
 }
 base(int x)   // конструктор с един параметър
 {a1 = x;
 }
 base(int x, int y)  // конструктор с два параметъра
 {a1 = x;
  a2 = y;
 }
 void a3()
 {cout << "a1: " << a1 << endl
       << "a2: " << a2 << endl;
 }
};
```

```
// дефиниция на производния клас der
class der : public base
{private: int d1;
 protected: int d2;
 public:
 der(int x, int y, int z, int t) : base(x, y) // конструктор
 {d1 = z;
  d2 = t;
 }
 void d3()
 {cout << "d1: " << d1 << endl
       << "d2: " << d2 << endl
       << "a2: " << a2 << endl;
  cout << "a3():" << endl;
  a3();
  }
 };
```

Тъй като в инициализиращия списък на конструктора на класа der участва двуаргументният конструктор на base, наследените компоненти се инициализират от него. В резултат от изпълнението на фрагмента:

```
der x(1, 2, 3, 4);
x.d3();
```

се получава:

```
d1: 3
d2: 4
a2: 2
a3():
a1: 1
a2: 2
```

Ако вместо двуаргументния конструктор на основния клас дефиницията на конструктора на производния клас der използва подразбиращият се конструктор на base, т.е.

```
der(int x, int y, int z, int t) : base()
 {d1 = z;
  d2 = t;
 }
```

наследените компоненти от базовия клас base ще се инициализират от подразбиращия се за base конструктор. В резултат от изпълнението на фрагмента:

```
der x(1, 2, 3, 4);
x.d3();
```

се получава:

```
d1: 3
d2: 4
a2: 0
a3():
a1: 0
a2: 0
```

Тази дефиниция на конструктора на der е еквивалентна на дефиницията:

```
der(int z, int t)
 {d1 = z;
  d2 = t;
 }
```

т.е. подразбиращият се конструктор на основния клас може да бъде пропуснат в инициализиращия списък на конструктора на производния клас.

Ако вместо двуаргументния конструктор на основния клас дефиницията на конструктора на производния клас der използва едноаргументния конструктор на base, т.е.

```
der(int x, int y, int z, int t) : base(x)
 {d1 = z;
  d2 = t;
 }
```

наследената компонента a1 от базовия клас base ще се инициализира с x, а компонентата a2 ще остане неинициализирана и в резултат от изпълнението на фрагмента:

```
der x(1, 2, 3, 4);
x.d3();
```

ще се получи:

```
d1: 3
d2: 4
a2: -858993460
```

```
a3():
a1: 1
a2: -858993460
```

Към примера ще отбележим изрично, че в инициализиращия списък може да участва не повече от едно обръщение към конструктор на един и същ базов клас, т.е. дефиниция от вида:

```
der(int x, int y, int z, int t) : base(), base(x, y)
 {d1 = z;
  d2 = t;
 }
```

е недопустима.

***Семантика***

Дефинирането на обект от производен клас предизвиква създаване на "неявен" обект от базовия му клас и добавяне на декларираните в производния клас компоненти. Това означава, че ако и базовият, и основният клас имат конструктори, то първо се извиква конструкторът на базовия, а след това – конструкторът на производния клас.

В случая на множествено наследяване, извикването на конструктор на производен клас води до извикването на указаните в дефиницията му конструктори на неговите основни класове и след завършване на тяхното изпълнение се изпълнява <тяло> на конструктора на производния клас. Процедурата е следната:

- заменят се формалните с фактическите параметри във всяко обръщение към конструктор на основен клас, след което се изпълнява обръщението;
- изпълняват се операторите в тялото на конструктора на производния клас.

Ако производният клас има член-данни, които са обекти, техните конструктори се извикват след изпълнението на обръщенията към конструкторите на основните класове от инициализиращия списък и преди изпълнението на операторите в тялото на конструктора на производния клас. Конструкторите на обектите се извикват по реда на тяхното деклариране в тялото на производния клас.

**Пример:** Резултатът от изпълнението на програмата

```
#include <iostream.h>
class base
```

```
{private: int a1;
 protected: int a2;
 public:
 base()
 {cout << "constructor base() \n";
  a1 = 0;
  a2 = 0;
 }
 base(int x, int y)
 {cout << "constructor base(" << x << "," << y << ")\n";
  a1 = x;
  a2 = y;
 }
 void a3()
 {cout << "a1: " << a1 << endl
       << "a2: " << a2 << endl;
 }
};
class der : public base
{private: base d1; // не може base d1(1, 5). Защо?;
 protected: base d2;
 public:
 der(int x, int y) : base(x, y)
 {cout << "constructor der\n";
 }
 void d3()
 {d1.a3();
  d2.a3();
  cout << "a2: " << a2 << endl;
  cout << "a3():" << endl;
  a3();
 }
};
void main()
{der x(1, 2);
 x.d3();
```

```
   }
```
е следният:
```
   constrictor base(1, 2)
   constrictor base()
   constrictor base()
   constrictor der
   a1: 0
   a2: 0
   a1: 0
   a2: 0
   a2: 2
   a3():
   a1: 1
   a2: 2
```

Ще се отбележим специално на някои случаи:

· **В основния клас не е дефиниран конструктор**

В този случай в инициализиращият списък не се отбелязва нищо. Наследената част на производния клас остава неинициализирана.

**Пример:** Резултатът от изпълнението на програмата:

```
#include <iostream.h>
class base
{private: int a1;
 protected: int a2;
 public:
 void readbase(int x = 0, int y = 0)
 {a1 = x;
  a2 = y;
 }
 void a3()
 {cout << "a1: " << a1 << endl
       << "a2: " << a2 << endl;
 }
};
class der : public base
{private: int d1;
```

```
    protected: int d2;
    public:
    der(int x, int y)
    {cout << "constructor der\n";
     d1 = x;
     d2 = y;
    }
    void d3()
    {cout << "d1: " << d1 << endl;
     cout << "d2: " << d2 << endl;
     cout << "a2: " << a2 << endl;
     cout << "a3():" << endl;
     a3();
    }
   };
   void main()
   {der x(1, 2);
    x.d3();
   }
```

e:

```
   constructor der
   d1: 1
   d2: 2
   a2: -858993460
   a3():
   a1: -858993460
   a2: -858993460
```

отрицателните стойности на a1 и a2 показват, че тези данни са неинициализирани.

**· Основният клас има само един конструктор с параметри, който не е подразбиращия се**

Възможни са:

*а) в производния клас е дефиниран конструктор*

Тогава трябва да има задължително обръщение към него в инициализиращия списък. Изпълнява се по начина, описан по-горе.

**Пример:** Програмата

```
#include <iostream.h>
class base
{private: int a1;
 protected: int a2;
 public:
 base(int x, int y)
 {a1 = x;
  a2 = y;
 }
 void a3()
 {cout << "a1: " << a1 << endl
       << "a2: " << a2 << endl;
 }
};
class der : public base
{private: int d1;
 protected: int d2;
 public:
 der(int x, int y)      // няма обръщение към конструктор на base
 {cout << "constructor der\n";
  d1 = x;
  d2 = y;
 }
 void d3()
 {cout << "d1: " << d1 << endl;
  cout << "d2: " << d2 << endl;
  cout << "a2: " << a2 << endl;
  cout << "a3():" << endl;
  a3();
 }
};
void main()
{der x(1,2);
 x.d3();
}
```

съобщава грешката:

```
primer.cpp(19):error C2512: 'base' : no appropriate default
                            constructor available
```

*б) в производния клас не е дефиниран конструктор*

В този случай компилаторът ще сигнализира за грешка. Необходимо е да се създаде конструктор за производния клас, който да активира конструктора на основния клас.

**Пример:** Програмата

```
#include <iostream.h>
class base
{private: int a1;
 protected: int a2;
 public:
 base(int x, int y)
 {a1 = x;
  a2 = y;
 }
 void a3()
 {cout << "a1: " << a1 << endl
       << "a2: " << a2 << endl;
 }
};
class der : public base
{private: int d1;
 protected: int d2;
 public:
 void d3()
 {cout << "d1: " << d1 << endl;
  cout << "d2: " << d2 << endl;
  cout << "a2: " << a2 << endl;
  cout << "a3():" << endl;
  a3();
 }
};
void main()
```

```
{der x;
 x.d3();
}
```
издава следното съобщение за синтактична грешка:
```
primer.cpp(28):error C2512: 'der': no appropriate default
              constructor available
```

**· Основният клас има няколко конструктора в т. число подразбиращ се конструктор**

Възможни са:

*а) в производния клас е дефиниран конструктор*

Тогава може да не се посочва конструктор за основния клас в инициализиращия списък. Ако не е посочен, компилаторът се обръща към подразбиращия се конструктор на основния клас.

Пример вече беше даден.

*б) в производния клас не е дефиниран конструктор*

В този случай компилаторът автоматично създава подразбиращ се конструктор за производния клас. Последният активира и изпълнява подразбиращият се конструктор на основния клас. Собствените членове на подразбиращия се клас са инициализирани неопределено.

**Пример:** В резултат от изпълнението на програмата

```
#include <iostream.h>
class base
{private: int a1;
 protected: int a2;
 public:
 base()
 {a1 = 0;
  a2 = 0;
 }
 void a3()
 {cout << "a1: " << a1 << endl
       << "a2: " << a2 << endl;
 }
```

```
};
class der : public base
{private: int d1;
 protected: int d2;
 public:
 void d3()
 {cout << "d1: " << d1 << endl;
  cout << "d2: " << d2 << endl;
  cout << "a2: " << a2 << endl;
  cout << "a3():" << endl;
  a3();
 }
};
void main()
{der x;
 x.d3();
}
```

се получава:

```
d1: -858993460
d2: -858993460
a2: 0
a3():
a1: 0
a2: 0
```

**Деструктори**

Деструкторите на един производен клас и на неговите основни класове се изпълняват в ред, обратен на реда на изпълнение на техните конструктори. Най-напред се изпълнява деструкторът на производния клас, след това се изпълняват деструкторите на неговите основни класове.

**Пример:** Резултатът от изпълнението на програмата:

```
#include <iostream.h>
class A
{public:
 A(){cout << “Конструктор на клас A\n”;}
```

```
    ~A(){cout << “Деструктор на клас A\n”;}
   };
   class B : public A
   {public:
    B(){cout << “Конструктор на клас B\n”;}
    ~B(){cout << “Деструктор на клас B\n”;}
   };
   class C : public B
   {public:
    C(){cout << “Конструктор на клас C\n”;}
    ~C(){cout << “Деструктор на клас C\n”;}
   };
   void main()
   {C x;
   }
```

е:

```
   Конструктор на класа A
   Конструктор на класа B
   Конструктор на класа C
   Деструктор на клас C
   Деструктор на клас B
   Деструктор на клас A
```

При създаването на обекта x се извиква конструкторът на класа C. Тъй като C е производен на класа B и инициализиращият му списък е празен, се извиква конструкторът на класа B, който започва да създава “неявен” обект от клас B. Но класът B е производен на класа A и инициализиращият му списък е празен. Това предизвиква обръщение към подразбиращия се конструктор на класа A. Заради това отначало се изпълнява конструкторът на класа A, след това се довършва създаването на обекта от клас B като се извиква конструкторът му. Най-накрая се извиква конструкторът на класа C за да завърши създаването на обекта x. При завършване изпълнението на тялото на функцията main започва процес на разрушаване на обекта x. Това предизвиква обръщение към деструктора на класа C, след това – към деструктора на класа B, след него – към деструктора на класа A и най-накрая обектът x се разрушава.

Тъй като член-данните на класа People от задача 157 са реализирани в областта за динамично разпределение на паметта, добре е за този клас да се реализира голямата четворка. Това ще направим стъпка по стъпка в следващите разглеждания. Засега ще променим класовете People, Student и PStudent като включим в тях конструктори и деструктори.

**Задача 160.** Да се дефинират повторно класовете People, Student и PStudent от задача 159, така че инициализиращите действия да се изпълняват от подходящи конструктори. Разрушителните действия да се извършват от деструктори.

Програма Zad160.cpp решава задачата.

```
// Program Zad160.cpp
#include <iostream.h>
#include <string.h>
// декларация на класа People
class People
{public:
  People(char * = "", char * = "");
  void PrintPeople() const;
  ~People();
 private:
  char * name;
  char * egn;
};
// дефиниция на конструктора на People
People::People(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
// дефиниция на метода PrintPeople
void People::PrintPeople() const
{cout << "Ime: " << name << endl;
```

```
 cout << "EGN: " << egn << endl;
}
// дефиниция на деструктора на People
People::~People()
{cout << "~People(): " << endl;
 delete name;
 delete egn;
}
// декларация на класа Student
class Student : People
{public:
  Student(char * = “”, char * = “”, long = 0, double = 0);
  void PrintStudent() const;
  ~Student()
  {cout << "~Student(): " << endl;
  }
 private:
  long facnom;
  double usp;
};
//дефиниция на конструктора на класа Student
Student::Student(char *str, char * num, long facn,
  double u) : People(str, num)
{facnom = facn;
 usp = u;
}
// дефиниция на метода PrintStudent
void Student::PrintStudent() const
{PrintPeople();
 cout << "Fac. nomer: " << facnom << endl;
 cout << "Uspeh: " << usp << endl;
}
// декларация на класа PStudent
class PStudent : public Student
{public:
  PStudent(char * = “”, char * = “”, long = 0,
```

```
            double = 0, double = 0);
  ~PStudent()
  {cout << "~PStudent() \n";
  }
  void PrintPStudent() const;
 protected:
  double tax;
};
// дефиниция на конструктора на класа PStudent
PStudent::PStudent(char *str, char *num, long facn,
          double u, double t) : Student(str, num, facn, u)
{tax = t;
}
// дефиниция на метода PrintPStudent
void PStudent::PrintPStudent() const
{PrintStudent();
 cout << "Tax: " << tax << endl;
}
void main()
{People pe;
 pe.PrintPeople();
 PStudent PStud("Ivan Ivanov", "8206123422", 42444, 6.0, 4567);
 PStud.PrintPStudent();
}
```

**Резултат:**

```
Ime:
EGN:
Ime: Ivan Ivanov
EGN: 8206123422
Fac.nomer: 42444
Uspeh: 6
Tax: 4567
~PStudent()
~Student()
~People()
~People()
```

Тази програма илюстрира използването на конструктори и деструктори на производни класове. Тя се различава от програма Zad159.cpp по това, че методите ReadPeople, ReadStudent и ReadPStudent на класовете People, Student и PStudent са заменени с конструктори. Освен това е добавен деструктор на класа People, който освобождава паметта на динамичните член-данни name и egn на People. Деструкторите на класовете Student и PStudent са напълно излишни, тъй като собствените им член-данни не са динамични. Дефинирани са с цел илюстрация на реда на изпълнението на деструкторите на класовете. При създаването на обекта PStud се извиква конструкторът на класа PStudent. Преди изпълнението на тялото му се прави обръщение към конструктора Student("Ivan Ivanov", "8206123422", 42444, 6.0), т.е. започва създаване на обект от класа Student. Преди изпълнението на тялото на този конструктор се изпълнява обръщението People("Ivan Ivanov", "8206123422"), което инициализира компонентите name и egn с "Ivan Ivanov" и "8206123422" съответно. Процесът на оценяване продължава с изпълнение на тялото на конструктора на класа Student, в резултат на което компонентите facnom и usp се инициализират с 42444 и 6 съответно. Най-накрая се изпълнява тялото на конструктора на класа PStudent, при което компонентата tax на обекта PStud се инициализира с 4567. Обръщението Stud.PrintPStudent() извежда отначало наследените член-данни, а след това и собствените член-данни на класа PStudent. Накрая се разрушава обектът PStud като преди това се извикват деструкторите на PStudent, Student и People.

Ако заменим дефинирания в класа People конструктор с два подразбиращи се параметъра с конструкторите

```
People::People(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
```

и

```
People::People()
{name = “”;
```

```
    egn = “”;
   }
```
дефиницията
```
   People pe;
```
ще предизвика обръщение към конструктора People() и член-данните на pe ще се инициализират с празния низ. При завършване на блока, изпълнението на деструктора ~People() ще направи опит да разруши несъществуващи връзки на name и egn с динамичната памет. Това ще предизвика грешка по време на изпълнение.

Ще отбележим също, че в инициализиращия списък на производния клас участват обръщения само към неговите базови класове. Дефиниция от вида:
```
   PStudent::PStudent(char *str, char *num, long facn,
              double u, double t) : People(str, num)
   {tax = t;
   }
```
предизвиква синтактична грешка – People не е основен клас на класа PStudent.

Ще отбележим също още една **често допускана грешка.**

В основния клас динамична променлива е дефинирана като protected. По такъв начин тя е видима и може пряко да се използва от всички член-функции на производния клас. Тъй като тази динамична променлива е наследена член-данна на производния клас, често се прави опит заетата от тази променлива памет да се освободи два пъти – веднъж от деструктора на базовия и веднъж от деструктора на производния клас. Това предизвиква грешка по време на изпълнение.

**Пример:** Дефиницията на деструктора на класа Student:
```
   class People
   {public:
     People(char * = “”, char * = “”);
     void PrintPeople() const;
     ~People();
    protected:
     char * name;
     char * egn;
   };
```

```
People::People(char *str, char *num)
...
void People::PrintPeople() const
...
People::~People()
{cout << "~People(): " << endl;
 delete name;
 delete egn;
}
class Student : People
{public:
  Student(char * = “”, char * = “”, long = 0, double = 0);
  void PrintStudent() const;
  ~Student();
 private:
  long facnom;
  double usp;
};
Student::Student(char *str, char * num, long facn,
  double u) : People(str, num)
...
Student::~Student()
{cout << "~Student(): " << endl;
 delete name;
 delete egn;
}
void Student::PrintStudent() const
...
```

е неправилна заради двойното освобождаване на динамична памет. Някои дефиниции са пропуснати, тъй като са същите като в задача 160.

### 17.4.2 Конструктор за присвояване и операторна функция за присвояване

Член-данните на обект на производен клас могат да получат стойности и чрез инициализиране чрез присвояване на друг обект или направо чрез присвояване. Това се осъществява чрез конструктора за

присвояване и операторната функция за присвояване на производния клас.

В общия случай, производният клас **не** наследява от основния клас конструктора за присвояване и оператора за присвояване. Има някои изключения, на които ще се спрем по-долу.

**Конструктор за присвояване**

При конструкторите за присвояване се спазва същият принцип като при обикновените конструктори на производния и основния клас. Конструкторът за присвояване на производния клас инициализира чрез присвояване собствените член-данни на класа, а конструкторът за присвояване на основния клас инициализира наследените член-данни. Конструкторите за присвояване на производни класове се дефинират по същия начин като обикновените конструктори на производни класове. Ще напомним, че ако в клас не е дефиниран конструктор за присвояване, ролята на такъв се поема от генерирания системен *конструктор за копиране* <име_на_клас>(const <име_на_клас>&).

Ще отбележим някои случаи на използване на конструкторите за присвояване на производния и основния клас.

**· В производния клас не е дефиниран конструктор за присвояване**

Възможни са:

***а) в основния клас е дефиниран конструктор за присвояване***

В този случай компилаторът генерира служебен конструктор за копиране на производния клас <име_на_производен_клас>(const <име_на_производен_клас>&), който преди да се изпълни, **активира и изпълнява** конструктора за присвояване на основния клас. Ще отбележим, че при обикновените конструктори този случай ще предизвика грешка, ако в основния клас няма подразбиращ се конструктор. Затова в случая се казва, че конструкторът за присвояване на основния клас се наследява от производния клас.

Ще илюстрираме казаното чрез следващата задача. Тя е обобщение на задача 160.

**Задача 161.** Да се допълни класът People от предната задача чрез конструктор за присвояване.

```
// Program Zad161.cpp
#include <iostream.h>
#include <string.h>
class People
{public:
  People(char *, char *);
  People(const People&);  // конструктор за присвояване
  void PrintPeople() const;
  ~People();   // деструктор
 private:
  char * name;
  char * egn;
};
// дефиниция на двуаргументния конструктор на класа People
People::People(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
// дефиниция на конструктора за присвояване на People
People::People(const People& p)
{name = new char[strlen(p.name)+1];
 strcpy(name, p.name);
 egn = new char[11];
 strcpy(egn, p.egn);
}
// дефиниция на метода PrintPeople
void People::PrintPeople() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
}
// дефиниция на деструктора
```

```
People::~People()
{cout << "~People()\n";
 delete name;
 delete egn;
}
// декларация на класа Student, в който не е дефиниран
// конструктор за присвояване
class Student : People
{public:
  Student(char *, char *, long, double); // обикновен конструктор
  void PrintStudent() const;
 private:
  long facnom;
  double usp;
};
// дефиниция на конструктора на класа Student
Student::Student(char *str, char * num,
                 long facn, double u) : People(str, num)
{facnom = facn;
 usp = u;
}
// дефиниция на метода PrintStudent
void Student::PrintStudent() const
{PrintPeople();
 cout << "fac. nomer: " << facnom << endl;
 cout << "uspeh: " << usp << endl;
}
// декларация на класа PStudent, в който също не е дефиниран
// конструктор за присвояване
class PStudent : public Student
{public:
  PStudent(char * = “”, char * = “”, long = 0,
           double = 0, double = 0);
  void PrintPStudent() const;
 protected:
  double tax;
```

```
};
// дефиниция на конструктора на класа PStudent
PStudent::PStudent(char *str, char *num, long facn,
          double u, double t) : Student(str, num, facn, u)
{tax = t;
}
// дефиниция на метода PrintPStudent
void PStudent::PrintPStudent() const
{PrintStudent();
 cout << "Tax: " << tax << endl;
}
void main()
{Student s1("Ivan Ivanov", "8206123422", 42444, 6.0);
 s1.PrintStudent();
 Student s2 = s1;
 s2.PrintStudent();
}
```

В резултат от изпълнението й се получава:

```
Ime: Ivan Ivanov
EGN: 8206123422
fac.nomer: 42444
uspeh: 6
Ime: Ivan Ivanov
EGN: 8206123422
fac.nomer: 42444
uspeh: 6
~People()
~People()
```

Инициализацията на обекта s1 се осъществява чрез конструктора Student(char*, char*, long, double), а инициализацията на обекта s2 – чрез генерирания от компилатора конструктор за *копиране* на класа Student. Конструкторът за копиране на класа Student се обръща към конструктора за присвояване на класа People (с аргумент s1). В резултат данните name и egn на s1 се копират в съответните полета на обекта s2. След това се изпълнява тялото на “служебния” конструктор за копиране на класа Student, при което числовите полета facnom и usp

на s2 се инициализират със стойностите 42444 и 6 на съответните полета на s1.

*б) в основния клас не е дефиниран конструктор за присвояване*

В този случай се генерират "служебни" конструктори за копиране за двата класа. Конструкторът за копиране на производния клас активира конструктора за копиране на основния клас.

Нека се върнем към решението на предходната задача. Класът PStudent е производен на класа Student и в двата класа не са дефинирани конструктори за присвояване. В резултат от изпълнението на функцията:

```
void main()
{PStudent s1("Ivan Ivanov", "8206123422", 42444, 6.0, 350);
 s1.PrintPStudent();
 PStudent s2 = s1;
 s2.PrintPStudent();
}
```

се получава:

```
Ime: Ivan Ivanov
EGN: 8206123422
fac.nomer: 42444
uspeh: 6
Tax: 350
Ime: Ivan Ivanov
EGN: 8206123422
fac.nomer: 42444
uspeh: 6
Tax: 350
~People()
~People()
```

Инициализацията на обекта s1 се осъществява чрез конструктора PStudent(char*, char*, long, double, double), а инициализацията на обекта s2 – чрез генерирания от компилатора конструктор за копиране на класа PStudent. Конструкторът за копиране на класа PStudent извиква конструктора за копиране на класа Student, който пък извиква конструктора за присвояване на класа People (с аргумент s1). В

резултат данните name и egn на s1 се присвояват на съответните полета на обекта s2, съгласно указания, в дефиницията на конструктора за присвояване на класа People, начин. След това се изпълнява тялото на конструктора за копиране на класа Student, при което полетата facnom и usp на s2 се инициализират с 42444 и 6 съответно и накрая се изпълнява тялото на конструктора за копиране на класа PStudent, при което полето tax се инициализира с 350.

· **В производния клас е дефиниран конструктор за присвояване**

В този случай отначало се активира конструкторът за присвояване на производния клас. В неговия инициализиращ списък може да има или да няма обръщение към конструктор (за присвояване или обикновен) на основния клас. Препоръчва се в инициализиращия списък на производния клас да има обръщение към конструктора за присвояване на основния клас, ако такъв е дефиниран. Ако не е указано обръщение към конструктор на основния клас, инициализирането на наследените членове става чрез подразбиращия се конструктор на основния клас. Ако основният клас няма такъв, ще се съобщи за отсъствието на подходящ конструктор.

Така конструкторът за присвояване на производния клас чрез дефиницията си определя как точно ще се инициализира наследената част.

**Задача 162.** Да се допълни и класът Student от предната задача с конструктор за присвояване.

В случая това не е необходимо, защото генерирания от компилатора служебен конструктор за копиране напълно ни устройва. Предложеното решение е заради технически съображения. Някои дефиниции на методи са пропуснати, тъй като са аналогични на тези от задача 161.

```
// Program Zad162.cpp
#include <iostream.h>
#include <string.h>
class People
{public:
  People(char *, char *);
```

```
  People(const People&);
  void PrintPeople() const;
  ~People()
 private:
  char * name;
  char * egn;
};
People::People(char *str, char *num)
{...
}
People::People(const People& p)
{...
}
void People::PrintPeople() const
{...
}
People::~People()
{...
}
class Student : People
{public:
  Student(char *, char *, long, double);
  Student(const Student& st);
  void PrintStudent() const;
 private:
  long facnom;
  double usp;
};
Student::Student(char *str, char * num,
                 long facn, double u) : People(str, num)
{...
}
// дефиниция на конструктора за присвояване на Student
Student::Student(const Student& st) : People(st)
{facnom = st.facnom;
 usp = st.usp;
```

```
}
void Student::PrintStudent() const
{...
}
void main()
{Student s1("Ivan Ivanov", "8206123422", 42444, 6.0);
 s1.PrintStudent();
 Student s2 = s1;
 s2.PrintStudent();
}
```

Забележете, аргументът на обръщението People(st), в инициализиращия списък на конструктора за присвояване на класа Student, е от тип const Student&, а не const People& (В т.17.5 ще разгледаме преобразуването на типовете).

Инициализацията на обекта s1 се осъществява чрез конструктора Student(char*, char*, long, double), а инициализацията на обекта s2 – чрез конструктора за присвояване Student(const Student&). При изпълнението на инициализацията на s2 отначало се изпълнява конструкторът за присвояване на класа People (с аргумент s1). В резултат данните name и egn на s1 се инициализират със съответните от обекта s2. След това се изпълнява самият конструктор за присвояване на класа Student, при което полетата facnom и usp на s2 се инициализират с 42444 и 6 съответно.

Друга реализация на конструктора за присвояване на класа Student е:

```
Student::Student(const Student& st) : People(st.name, st.egn)
{facnom = st.facnom;
 usp = st.usp;
}
```

***ако член-данните name и egn на класа People са обявени в секция protected.*** В този случай за инициализиране на наследените член-данни е използван двуаргументният конструктор на People.

Ако добавим подразбиращ се конструктор на класа People, т.е.

```
People::People()
 {name = new char[1];
  strcpy(name, "");
```

```
    egn = new char[1];
    strcpy(egn, "");
  }
```
и конструкторът за присвояване на класа Student има вида:
```
  Student::Student(const Student& st)
  {facnom = st.facnom;
   usp = st.usp;
  }
```
след изпълнението на фрагмента
```
  Student s1("Ivan Ivanov", "8206123422", 42444, 6.0);
  Student s2 = s1;
  s2.PrintStudent();
```
полетата name и egn на s2 се инициализират с празния низ, а facnom и успех с 42444 и 6 съответно.

Причината е, че в инициализиращия списък на конструктора за присвояване на класа Student не е указан начинът за инициализиране на член-данните на основния клас. В този случай инициализацията се осъществява чрез конструктора по подразбиране на основния клас People.

Ще дадем още един пример.
```
  #include <iostream.h>
  class base
  {public:
   base(int x = 99)
   {b = x;
   }
   void f() const
   {cout << "base: b: " << b << endl;
   }
   private:
    int b;
  };
  class der : public base
  {public:
  der(int x = 11) : base(x)
```

```
{d = x;
}
der(const der& p)
{d = p.d + 5;
}
void f() const
{base::f();
 cout << "der: d: " << d << endl;
}
private:
 int d;
};
void main()
{der d1;
 d1.f();
 der d2 = d1;
 d2.f();
}
```

В резултат от изпълнението му се получава:

```
base: b: 11
der: d: 11
base: b: 99
der: d: 16
```

В него основният клас не предефинира конструктора за копиране. В производния клас има конструктор за присвояване, в който явно не е казано как точно става инициализацията на наследената компонента. Тъй като в base има конструктор по подразбиране, инициализацията се осъществява чрез него. Ако променим конструктора на класа base в:

```
base(int x)
{b = x;
}
```

и конструкторът за присвояване на der стане:

```
der(const der& p)
{d = p.d + 5;
}
```

компилаторът ще съобщи за отсъствието на подходящ конструктор на класа base, тъй като конструктор по подразбиране за класа base не съществува.

**Операторна функция за присвояване**

Операторната функция за присвояване на производен клас трябва да указва как да стане присвояването както на собствените, така и на наследените си член-данни. За разлика от конструкторите на производни класове тя прави това в тялото си (не поддържа инициализиращ списък). Използването й зависи от това дали такава е дефинирана в производния клас. Ще напомним, че ако в клас не е дефинирана операторна функция за присвояване, компилаторът създава operator=(const <име_на_клас>&).

Ще разгледаме следните два случая:

· **В производния клас не е дефинирана операторна функция за присвояване**

Компилаторът създава операторна функция за присвояване на производния клас. **Тя се обръща и изпълнява операторната функция за присвояване на основния клас**, чрез която инициализира наследената част, след това инициализира чрез присвояване и собствената част на производния клас. Затова в този случай се казва, че операторът за присвояване на основния клас се наследява, т.е. за наследените член-данни се използва подразбиращият се или предефинираният оператор за присвояване на основния клас.

· **В производния клас е дефинирана операторна функция за присвояване**

Дефинираният в производния клас оператор за присвояване **трябва да се погрижи за наследените компоненти. Налага се в тялото на неговата дефиниция да има обръщение към дефинирания оператор за присвояване на основния клас**, ако има такъв. Ако това не е направено **явно**, ***стандартът на езика не уточнява как ще стане присвояването на наследените компоненти***. В случая се казва, че операторът за присвояване на основния клас не се наследява.

**Пример:** Да разгледаме следната йерархична схема от 4 класа: базов клас base и три негови наследници: der1, der2 и der3, реализирана чрез програмата:

```
#include <iostream.h>
class base
{public:
 base(int x = 0)
 {b = x;
 }
 base& operator=(const base &x)
 {if(this!=&x) b = x.b + 1;
  return *this;
 }
 protected:
  int b;
};
class der1 : public base
{public:
 der1(int x = 1)
 {d = x;
 }
 der1& operator=(const der1& x)
 {if(this!=&x)
  {d = x.d + 2;
   b = x.b + 3;
  }
  return *this;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
  int d;
};
class der2 : public base
{public:
 der2(int x = 2)
 {d = x;
 }
```

```
 der2& operator=(const der2& x)
 {if(this !=&x)
  {d = x.d + 3;
  }
  return *this;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
  int d;
};
class der3 : public base
{public:
 der3(int x = 3)
 {d = x;
 }
 void Print()
 {cout << "der: " << d << "  base: " << b << endl;
 }
 private:
  int d;
};
void main()
{der1 d11(5), d12;
 der2 d21(5), d22;
 der3 d31(5), d32;
 d12 = d11;
 d22 = d21;
 d32 = d31;
 cout << "d11: "; d11.Print();
 cout << "d12: "; d12.Print();
 cout << "d21: "; d21.Print();
 cout << "d22: "; d22.Print();
 cout << "d31: "; d31.Print();
 cout << "d32: "; d32.Print();
```

```
   }
```
В резултат от изпълнението й се получава:
```
   d11: der: 5  base: 0
   d12: der: 7  base: 3
   d21: der: 5  base: 0
   d22: der: 8  base: 0
   d31: der: 5  base: 0
   d32: der: 5  base: 1
```
Класовете der1, der2 и der3 са дефинирани и използвани по идентичен начин с изключение на предефинирания оператор за присвояване. В класа der1 операторът за присвояване е предефиниран и се грижи за наследената част. В класа der2 операторът за присвояване е предефиниран, но не указва как става присвояването на наследената член-променлива. Тъй като операторът за присвояване на базовия клас в този случай не се наследява, стандартът на езика не уточнява стойността на наследената член-променлива на обекта d22. В този случай нейната стойност е тази от инициализацията der2 d22. В класа der3 операторът за присвояване не е предефиниран. Тогава за собствените на класа компонента се използва подразбиращият се, а за наследената – предефинираният оператор за присвояване на базовия клас се наследява и изпълнява.

**Задача 163.** Да се допълнят класовете People и Student от предишната задача с операторни функции за присвояване.

```
// Program Zad163.cpp
#include <iostream.h>
#include <string.h>
class People
{public:
  People(char * = "", char * = "");
  People(const People&);
  People& operator=(const People& p);
   void PrintPeople() const;
   ~People();
 private:
```

```
  char * name;
  char * egn;
};
People::People(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
People::People(const People& p)
{name = new char[strlen(p.name)+1];
 strcpy(name, p.name);
 egn = new char[11];
 strcpy(egn, p.egn);
}
People& People::operator=(const People& p)
{if(this!=&p)
 {delete name;
  delete egn;
  name = new char[strlen(p.name)+1];
  strcpy(name, p.name);
  egn = new char[11];
  strcpy(egn, p.egn);
}
 return *this;
}
void People::PrintPeople() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
}
People::~People()
{delete name;
 delete egn;
}
class Student : People
{public:
```

```
  Student(char * = "", char * = "", long = 0, double = 0);
  Student(const Student& st);
  Student& operator=(const Student& st);
  void PrintStudent() const;
 private:
  long facnom;
  double usp;
};
Student::Student(char *str, char * num,
                 long facn, double u) : People(str, num)
{facnom = facn;
 usp = u;
}
Student::Student(const Student& st) : People(st)
{facnom = st.facnom;
 usp = st.usp;
}
Student& Student::operator=(const Student& st)
{if(this!=&st)
{People::operator=(st);
 facnom = st.facnom;
 usp = st.usp;
 }
 return *this;
}
void Student::PrintStudent() const
{PrintPeople();
 cout << "fac. nomer: " << facnom << endl;
 cout << "uspeh: " << usp << endl;
}
// дефиниране на клас PStudent
class PStudent : public Student
{public:
  PStudent(char * = "", char * = "", long = 0,
  double = 0, double = 0);
  PStudent& operator=(const PStudent& st);
```

```
  void PrintPStudent() const;
 protected:
  double tax;
};
PStudent::PStudent(char *str, char *num, long facn,
          double u, double t) : Student(str, num, facn, u)
{tax = t;
}
PStudent& PStudent::operator=(const PStudent& st)
{if(this!=&st)
{Student::operator=(Student(st));
  tax = st.tax;
 }
 return *this;
}
void PStudent::PrintPStudent() const
{PrintStudent();
 cout << "Tax: " << tax << endl;
}
void main()
{PStudent s1("Ivan Ivanov", "8206123422", 42444, 6.0, 4444);
 s1.PrintPStudent();
 PStudent s2("Jonko Dimov", "9012074442", 43344, 5, 3434);
 s2.PrintPStudent();
 s2 = s1;
 s2.PrintPStudent();
}
```

Ще дадем още едно решение на задачата, което смятаме за полезно.

```
#include <iostream.h>
#include <string.h>
class People
{public:
  People(char * = "", char * = "");
  People(const People&);
  People& operator=(const People& p);
  ~People();
```

```
  void PrintPeople() const;
  void del();
  void CopyPeople(const People& p);
 private:
  char * name;
  char * egn;
};
People::People(char *str, char *num)
{name = new char[strlen(str)+1];
 strcpy(name, str);
 egn = new char[11];
 strcpy(egn, num);
}
People::People(const People& p)
{CopyPeople(p);
}
People& People::operator=(const People& p)
{if(this!=&p)
{del();
 CopyPeople(p);
}
 return *this;
}
void People::PrintPeople() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
}
People::~People()
{del();
}
void People::del()
{delete name;
 delete egn;
}
void People::CopyPeople(const People& p)
{name = new char[strlen(p.name)+1];
```

```
 strcpy(name, p.name);
 egn = new char[11];
 strcpy(egn, p.egn);
}
class Student : People
{public:
  Student(char * = "", char * = "", long = 0, double = 0);
  Student(const Student& st);
  Student& operator=(const Student& st);
  void PrintStudent() const;
 private:
  long facnom;
  double usp;
};
Student::Student(char *str, char * num,
             long facn, double u) : People(str, num)
{facnom = facn;
 usp = u;
}
Student::Student(const Student& st) : People(st)
{facnom = st.facnom;
 usp = st.usp;
}
Student& Student::operator=(const Student& st)
{if(this != &st)
{del();
 CopyPeople(st);
 facnom = st.facnom;
 usp = st.usp;
}
 return *this;
}
void Student::PrintStudent() const
{PrintPeople();
 cout << "Fac. nomer: " << facnom << endl;
 cout << "Uspeh: " << usp << endl;
```

```
}
class PStudent : public Student
{public:
  PStudent(char * = "", char * = "", long = 0,
           double = 0, double = 0);
  PStudent(const PStudent& st):Student(st),tax(st.tax)
  {}
  PStudent& operator=(const PStudent& st);
  void PrintPStudent() const;
 protected:
  double tax;
};
PStudent::PStudent(char *str, char *num, long facn,
               double u, double t) : Student(str, num, facn, u)
{tax = t;
}
PStudent& PStudent::operator=(const PStudent& st)
{if (this != &st)
 {Student p = st;
  Student::operator=(Student(st));
  tax = st.tax;
 }
 return *this;
}
void PStudent::PrintPStudent() const
{PrintStudent();
 cout << "Tax: " << tax << endl;
}
void main()
{PStudent s1("Ivan Ivanov", "8206123422", 42444, 6.0, 500);
 s1.PrintPStudent();
 PStudent s2("Jonko Dimov", "9012074442", 43333, 4.0, 700);
 s2.PrintPStudent();
 s2 = s1;
 s2.PrintPStudent();
}
```

## 17.5 Преобразуване на типовете

Ако основният клас, който се наследява от производния клас е с атрибут **public**, възможно е взаимно заменяне на обекти от двата класа. Заменянето може да се извършва при инициализиране, при присвояване и при предаване на параметри на функции. Могат да се заменят обекти, псевдоними на обекти, указатели към обекти и указатели към методи. Замяната в посока "производен с основен" се счита за безопасна, докато замяната в обратната посока "основен с производен" може да предизвика проблеми.

Процесът на замяна е свързан с преобразувания, които за различните случаи са явни или неявни.

За да покажем тези преобразувания ще използваме класовете base и der, дефинирани по следния начин:

```
class base
{public:
 base(int x = 0)
 {b = x;
 }
 int get_b() const
 {return b;
 }
 void f()
 {cout << "b: " << b << endl;
 }
 private:
  int b;
 };
class der : public base
{public:
 der(int x = 0) : base(x)
 {d = 5;
 }
 int get_d() const
 {return d;
```

```
    }
    void f_der()
    {cout << "class der: d: " << d
          << " b: " << get_b() << endl;
    }
    private:
     int d;
    };
```

### 17.5.1 Преобразуване в посока “производен с основен”

Обект, псевдоним на обект или указател към обект на производен клас се преобразуват съответно в обект, псевдоним на обект или указател към обект на основен клас чрез ***неявни стандартни преобразувания***. На практика тези преобразувания се свеждат до използване само на наследените компоненти на класа. Последното се основава на факта, че производният клас наследява всички свойства на базовия клас и може да бъде използван вместо него.

**Пример:** В резултат от изпълнението на фрагмента:

```
der d; d.f_der();
 base x = d; x.f();
 der &d1 = d; d1.f_der();
 base &y = d1; y.f();
 der *d2 = &d; (*d2).f_der();
 base *z = d2; (*z).f();
```

се получава

```
   class der: d: 5 b: 0
   b: 0
   class der: d: 5 b: 0
   b: 0
   class der: d: 5 b: 0
   b: 0
```

### 17.5.2 Преобразуване в посока “основен производен”

Тъй като основният клас не съдържа собствените компоненти на производния клас, това преобразуване се осъществява само чрез явно указване.

**Най-често се срещат следните случаи:**

*а) Присвояване и инициализиране на обект от производен клас с обект на основен клас*

Нека x е обект на класа base, а y е обект на производния му клас der. Искаме на y да присвоим x. Стандартно се реализира чрез явно преобразуване на x в обект на клас der, т.е.

```
base x;
der y = (der)x;
```

Операцията е опасна, тъй като собствените компоненти на обекта y ще останат неинициализирани и *опитът за промяната им може да доведе до сериозни последици*. Затова някои реализации на езика, в това число и Visual C++ 6.0, **не реализират това преобразуване**.

Подобна е ситуацията при използване на указатели към обекти:

```
base *pb = new base;
der* pd = (der*) pb;
```

Извършва се явно преобразуване на pb в указател към обект на клас der. Указателят pd към обект на der не сочи към *истински обект* от клас der. Областта в паметта, свързана с указателя pd, няма собствени компоненти на класа der. Опитът за използването им **може** да предизвика сериозни проблеми, тъй като ще се използва памет, която е определена за други цели. Някои реализации на езика не реализират това преобразуване. Visual C++ 6.0 го извършва. От примера по-долу се вижда, че се извежда случайна стойност, но опитът за промяна на тази памет, **може** да е с непредвидими за програмата последици. Дефиниция на указателя pd към der може да се използва за извикване на собствена член-функция на der.

**Пример:** Изпълнението на фрагмента:

```
base *pb = new base;
(*pb).f();     // или pb->f();
der *pd = (der*) pb;
(*pd).f_der();    // или pd->f_der();
cout << pd->get_b() << endl;
```

води до следния резултат:

```
b: 0
class der: d: -33686019 b: 0
0
```

*б) Присвояване на указател към метод на основен клас на указател към метод на производен клас*

Чрез дефиницията

```
void (base::*pb)() = base::f;
```

pb се обявява за указател към метода f на класа base, който няма параметри и е от тип void. За да се използва този указател е необходимо той да се свърже с конкретен обект, т.е.

```
base x;
(x.*pb)();
```

В резултат се осъществява обръщение към метода f чрез указателя pb към него и се получава:

```
b: 0
```

Чрез дефиницията

```
void (der::*pd)()
```

pd се определя като указател към метод на производния клас der като методът е без параметри и е от тип void.

*Възниква въпросът:* Може ли указателят pb към метод на основния клас да се присвои на указателя pd, т.е. може ли да запишем:

```
void (der::*pd)() = pb;
der y(20);
(y.*pd)();
```

Отговорът е положителен. При присвояването ще се извърши неявно присвояване на типовете. Обектът y на производния клас der съдържа наследената част от основния клас base и метода f() в частност. Затова указателят pd ще указва правилно. В резултат се получава:

```
b: 0
b: 20
```

Обратното присвояване – указател към член-функция на производния клас да се присвои на указател към член-функция на основен клас изисква явно преобразуване и дали ще се използва правилно зависи единствено от програмиста.

**Пример:** Фрагментът:

```
void (der::*pd)() = der::f_der;
void (base::*pb)() = pd;
```

е недопустим. Проблемът идва от това, че pb е указател към метод на base от тип void и без параметри. В същото време pd е указател към член-функцията f_der на der също от тип void и без параметри, но f_der има достъп до собствените членове на der, които не са членове на base.

Допустимо е присвояването:

```
void (base::*pb)();
pb = (void (base::*)()) der::f_der; //явно преобразуване
```

но използването му може да доведе до грешка или до неоднозначност, както е показано в примера по-долу.

**Пример:**

```
base x(5);
der y(10);
base *pf = &y; // pf сочи към обект на класа der
(pf->*pb)();   // извикване на метода der::f_der
pf = &x;       // pf сочи към обекта x на класа base
(pf->*pb)();   // грешка, зависи от реализацията
```

*в) Достъп до собствени членове на производния клас чрез обект на основния клас*

Такъв достъп директно не е възможен. Непряк достъп е възможен и се осъществява чрез указатели и преобразувания.

**Пример:**

```
der y;
der *pd = &y;  // инициализация на указателя pd към обект на der
base *pb = pd; // неявно преобразуване
```

Указателите pb и pd сочат към обекта y на класа der, но са различни. Компилаторът проверява допустимостта на използването им в зависимост **за кой клас се отнася**.

Обръщението

```
pd->f_der();
```

е допустимо и активира, чрез указателя pd към обекта y, метода f_der(), но обръщението

```
pb->f_der();
```
е недопустимо, тъй като pb е указател към base, който няма f_der() за свой метод. За да стане възможно последното трябва да се използва явно преобразуване
```
((der *) pb)->f_der();
```
Ще напомним, че казаното се отнася само за производни класове с атрибут за област public на основния клас. За производни класове с атрибути за област private и protected не са дефинирани подобни преобразувания. Това е естествено, тъй като в тези случаи производният клас не притежава всички свойства на базовия и не може да го замести.

## 17.6 Шаблони на класове и наследяване

Използването на шаблони на класове и наследяването може да се разгледа в следните случаи:

**· шаблон на клас – наследник на обикновен клас**

В този случай, ако base е обикновен клас, т.е.
```
class base
{<тяло>
};
```
декларацията на производния на base шаблон на клас der ще има вида:
```
template <class T>
class der:<атрибут_за_област> base
{<тяло>
};
```

**· обикновен клас – наследник на шаблон на клас**

Ако tempbase е шаблон на основен клас
```
template <class T>
class tempbase
{<тяло>
};
```

декларацията на обикновен производен клас на шаблона tempbase трябва да задава стойност на параметъра за тип на tempbase, например int в случая и има вида

```
class der:<атрибут_за_област> tempbase<int>
{<тяло>
};
```

· **шаблон на клас – наследник на шаблон на клас**

Нека е деклариран шаблон на клас tempbase с параметър T:

```
template <class T>
class tempbase
{<тяло>
};
```

Декларацията на производен шаблон на клас на шаблона tempbase може да стане по два начина. При първия, производният шаблон на клас запазва същия параметър за тип като базовия, т.е.

```
template <class T>
class tempder1: атрибут_за_област tempbase<T>
{<тяло>
};
```

В този случай на всеки възможен базов клас съответства точно един производен. При втория начин, производният шаблон на клас въвежда и други параметри за тип:

```
template <class T, class U, ...>
class tempder2 : атрибут_за_област tempbase<T>
{<тяло>
};
```

При тази декларация на всеки възможен базов клас съответства фамилия от производни класове.

**Задача 164.** Да се дефинира шаблон на клас “точка в равнината” с параметър T за тип (тип на координатите) и клас, определящ отсечка в равнината с краища – точки с цели координати. На тези класове да се дефинират шаблони на производни класове, определящи “точка в равнината с цвят” и “отсечка в равнината с цвят”.

Програма Zad164.cpp рашава задачата. Цветът на точка (отсечка) се задава с числов параметър.

```
// Program Zad164.cpp
#include <iostream.h>
// дефиниция на шаблон на клас Point, определящ точка
// в равнината с координати от тип T
template <class T>
class Point
{public:
 Point(T = 0, T = 0);
 void print() const;
 private:
  T x, y;
 };
template <class T>
Point<T>::Point<T>(T a, T o)
{x = a;
 y = o;
}
template <class T>
void Point<T>::print() const
{cout << "Point(" << x << ", " << y << ")\n";
}
// дефиниция на клас Segm, опреелящ отсечка в равнината
// с краища – точки с цели координати
class Segm
{public:
 Segm(int a1=0, int o1=0,int a2=0, int o2=0);
 void print() const;
 private:
  int x1, y1, x2, y2;
 };
Segm::Segm(int a1, int o1,int a2, int o2)
{x1 = a1;
 y1 = o1;
 x2 = a2;
```

```
 y2 = o2;
}
void Segm::print() const
{cout << "Segm(" << x1 << ", " << y1 << ")"
      <<"(" << x2 << ", " << y2 << ")\n";
}
// дефиниция на клас ColPoint, определящ точка
// с реални координати и цвят
class ColPoint : public Point<double>
{public:
 ColPoint(double = 0, double = 0, int = 0);
 void print() const;
 private:
  int col;
};
ColPoint::ColPoint(double a, double o, int c) :
                                              Point<double>(a, o)
{col = c;
}
void ColPoint::print() const
{Point<double>::print();
 cout << "Color: " << col << endl;
}
// дефиниция на шаблон на клас ColPoint1, наследник на шаблона
// на класа Point и запазващ параметъра на шаблона на Point
template <class T>
class ColPoint1 : public Point<T>
{public:
 ColPoint1(T = 0, T = 0, T = 0);
 void print() const;
 private:
  T col;
 };
template <class T>
ColPoint1<T>::ColPoint1<T>(T a, T o , T c) : Point<T>(a, o)
{col = c;
```

```
}
template <class T>
void ColPoint1<T>::print() const
{Point<T>::print();
 cout << "color(col1): " << col << endl;
}
// дефиниция на шаблон на клас ColPoint2 с два параметъра,
// наследник на шаблона на класа Point. Параметри:
// T – тип на координатите на точката
// U – тип на цвета
// на класа Point и запазващ параметъра на шаблона на Point
template <class T, class U>
class ColPoint2: public Point<T>
{public:
 ColPoint2(T a = 0, T o = 0, U c = 0);
 void print() const;
 private:
 U col;
};
template <class T, class U>
ColPoint2<T, U>::ColPoint2<T, U>(T a, T o, U c) : Point<T>(a, o)
{col = c;
}
template <class T, class U>
void ColPoint2<T, U>::print() const
{Point<T>::print();
 cout << "color(col2): " << col << endl;
}
template <class T>
class ColSegm : public Segm
{public:
 ColSegm(int a1=0, int o1=0, int a2=0, int o2=0,
   T c=0) : Segm(a1,o1,a2,o2)
 {col = c;
 }
 void print()
```

```
 {Segm::print();
  cout << "Color: " << col << endl;
 }
 private:
 T col;
};
```

В резултат от изпълнението й се получава:

```
Point(1.5, 3,5)
Segm(1,1) (5.5)
Point(3.8, -4.5)
Color: 8
Point(5, 10)
Color(col1): 15
Point(1.4, 3.5)
Color(col2): 7
Segm(0, 0) (1,1)
Color: 3.4
```

## 17.7 Компилация и свързване

Случаят, когато дефинициите на основния, на производния клас и програмата се намират в един файл, е тривиален. Файлът се компилира и свързва самостоятелно в готов за изпълнение програмен модул.

Когато декларациите на основния и производния клас, дефинициите на функциите им и текста на програмата, ползваща производния клас са разположени в различни файлове, компилацията и свързването се осъществяват на следните стъпки:

- Компилиране на декларацията на основния клас и дефинициите на функциите му;
- Компилиране на декларацията на производния клас и дефинициите на член-функциите му, заедно с декларацията на основния клас;
- Компилиране на програмата заедно с декларациите на основния и производния клас;
- Свързване на получените обектни модули.

## Допълнителна литература

1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.

2. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.