18

Множествено наследяване. Виртуални класове и функции. Полиморфизъм

18.1 Множествено наследяване

В случаите, когато производният клас наследява няколко основни класа, се казва, че класът е с множествено наследяване. Този вид наследяване е мощен инструмент на ООП, тъй като чрез него се изграждат графовидни йерархични структури.

В параграф 17 дефинирахме производен клас. Сега ще конкретизираме дефиницията за случая на множественото наследяване.

Дефиницията на производен клас се състои от декларация на класа и дефиниции на методите му.

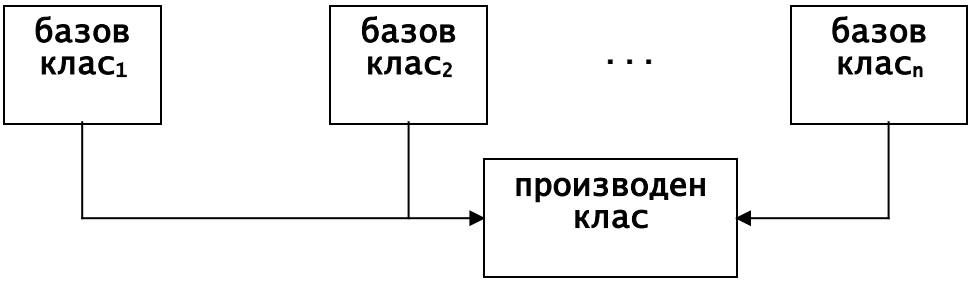
Деклариране на производен клас с множествено наследяване

<декларация_на_производен_клас> ::=
class <име_на-производен_клас> :
[<атрибут_за_област>]$_{опц}$ <име_на_базов_клас$_1$>,
[<атрибут_за_област>]$_{опц}$ <име_на_базов_клас$_2$>{,
[<атрибут_за_област>]$_{опц}$ <име_на_базов_клас$_n$>}$_{опц}$
{<декларация_на_компоненти>
};
<име_на-производен_клас> ::= <идентификатор>
<атрибут_за_област> ::= public | private | protected
<име_на_базов_клас$_i$> ::= <идентификатор>

18.1 Деклариране на производен клас с множествено наследяване

Атрибутите за област се задават за всеки базов клас. Семантиката им беше вече пояснена при разглеждане на производни класове с единично наследяване. Ако за някои клас атрибутът за област е пропуснат, подразбира се private.

Определената на фиг. 18.1 декларация задава следната йерархия:



Производният клас наследява компонентите на всички базови класове като видът на наследяване private, public или protected се определя от атрибута за област на базовия клас. Правилата са същите като при единичното наследяване.

За член-функциите на голямата четворка на производен клас с множествено наследяване са в сила същите правила, като при производен клас с единично наследяване. В общия случай тези член-функции за основните класове не се наследяват от производния им клас. Изключенията отново са при конструкторите за присвояване и операторните функции за присвояване.

Ще напомним дефиницията на конструктора на производен клас с множествено наследяване (Фиг. 18.2).

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$_i$>)}$_{опц}$
<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.
В <инициализиращ_списък> може да има повече от едно обръщение към конструктор на основен клас.

Фиг. 18.2 Конструктор на производен клас

При извикването на този конструктор последователно се извикват:

а) Конструкторите на базовите класове по реда на тяхното задаване не в инициализиращия списък на конструктора, а в декларацията на производния клас.

б) Конструкторите на класовете, чиито обекти са членове на производния клас. Редът на извикване съответства на реда на деклариране на тези членове в тялото на производния клас.

в) Конструкторът на производния клас.

Отново са възможни:

1) В основен клас не е дефиниран конструктор

В този случай в инициализиращия списък на конструктора на производния клас не се прави обръщение към конструктора на този клас и наследената му част остава неинициализирана.

2) В основен клас има един конструктор с параметър, от който не следва подразбиращия се

Тогава ако в производния клас е дефиниран конструктор, в инициализиращия му списък задължително трябва да има обръщение към конструктора с параметър на този основен клас. Ако в производния клас не е дефиниран конструктор за присвояване, компилаторът ще сигнализира грешка.

3) Основен клас има няколко конструктора в т. число подразбиращ се

Ако в производния клас е дефиниран конструктор, в инициализиращия му списък може да не се посочва конструктор за този основен клас. Ще се използва подразбиращия се. Ако в производния клас не е дефиниран

конструктор, компилаторът автоматично създава за него подразбиращ се конструктор. В този случай обаче всички основни класове на производния клас трябва да имат подразбиращ се конструктор.

Дефинирането на деструкторите става по описания вече начин. Всеки деструктор се грижи само за собствените си компоненти. Извикването им става в обратен ред – първо се извиква деструкторът на производния клас, след това, в обратен ред, се извикват деструкторите на класовете на обектите – членове на производния клас и най-накрая на основните му класове, отново в обратен ред на реда на извикване на техните конструктори.

Пример: Резултатът от изпълнението на програмата:

```
#include <iostream.h>
class base1
{public:
 base1(int x = 0)
 {cout << "base1:\n";
  b1 = x;
 }
 ~base1()
 {cout << "~base1()\n";
 }
 private:
  int b1;
 };
class base2
{public:
 base2(int x = 0)
 {cout << "base2:\n";
  b2 = x;
 }
 ~base2()
 {cout << "~base2()\n";
 }
 private:
  int b2;
```

```
 };
class base3
{public:
 base3(int x = 0)
 {cout << "base3:\n";
  b3 = x;
 }
 ~base3()
 {cout << "~base3()\n";
 }
 private:
  int b3;
 };
class der : public base2, base1, base3
{public:
 der(int x = 0) : base3(x), base1(x), base2(x)
 {d = 5;
 }
 private:
  int d;
 };
void main()
{der d1(5);
}
```

e

```
base2
base1
base3
~base3()
~base1()
~base2()
```

Същият е резултатът от изпълнението на програмата ако от инициализиращия списък на класа der пропуснем някое от обръщенията към основните класове base1, base2 или base3, или даже всичките. В тези случаи се използват конструкторите по подразбиране на основните класове. Резултатът от изпълнението не се променя ако в производния

клас der не е дефиниран конструктор. В този случай компилаторът създава за der конструктор по подразбиране, който се обръща към конструкторите по подразбиране на основните класове в реда, указан в декларацията на производния клас.

В тази програма класът der има две собствени и 3 наследени компоненти (конструкторите и деструкторите не се наследяват). Може да си мислим за него като клас от вида:

```
class der : public base2, base1, base3
{public:
 der(int x = 0) : base3(x), base1(x), base2(x)
 {d = 5;
 }
private:
  int d;
  int b1, b2, b3;
};
```

Дефиницията der d1(5); предизвиква създаване на обект d1 с компоненти от вида:

и извикване на конструктора на класа der с параметър 5. Преди да се изпълни неговото тяло се изпълняват конструкторите на базовете класове base2, base1 и base3 с параметър 5. Това инициализира отделената памет за член-данните на d1 с 5.

Дефинирането на конструктора за присвояване и операторната функция за присвояване на производен клас с множествено наследяване се извършва по същия начин както при единичното наследяване.

Ще напомним, че ако в производния клас не е дефиниран конструктор за присвояване, компилаторът генерира за него “служебен” конструктор за копиране, който преди да се изпълни активира и изпълнява

конструкторите за присвояване (копиране) на всички основни класове в реда, указан в декларацията на производния клас. Ако в производния клас е дефиниран конструктор за присвояване, препоръчва се в инициализиращия му списък да има обръщения към конструкторите за присвояване на основните класове (ако такива са дефинирани). Ако за някои основен клас не е указано такова обръщение, а е указан обикновен негов конструктор, инициализирането на наследените компоненти на този клас става чрез указания конструктор. Ако не е указано обръщение към конструктор, използва се конструкторът по подразбиране на основния клас, ако такъв съществува или се съобщава за отсъствието на подходящ конструктор за този основен клас, ако не съществува конструктор по подразбиране.

Операторната функция за присвояване на производен клас с множествено наследяване има вида:

```
<производен_клас>&
      <производен_клас>::operator=(const<производен_клас>& p)
{if (this != &p)
 {<основен_клас1>::operator=(p);
  <основен_клас2>::operator=(p);
    ...
  <основен_класN>::operator=(p);
    // дефиниране на присвояването
    // за собствените за класа компоненти
    Del();  // изтриване от ДП на собствените компоненти на
            // подразбиращия се обект
   Copy(p); // копиране на собствените компоненти на p в
            // подразбиращия се обект
 }
 return *this;
}
```

Ако в производния клас не е дефинирана операторна функция за присвояване, компилаторът създава такава. Тя изпълнява операторните функции за присвояване на всички основни класове на производния клас. Ако в производния клас е дефинирана операторна функция за присвояване, тя трябва да се погрижи за присвояването на наследените

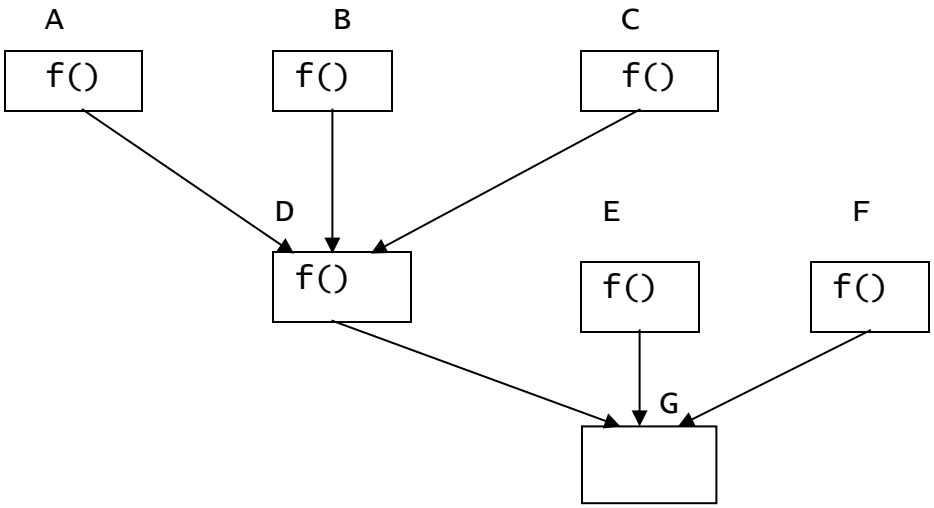
компоненти. Ако това не е направено явно, стандартът на езика не уточнява как ще стане присвояването на наследените компоненти.

Чрез обект на производния клас могат да се викат всички негови public компоненти – собствени и наследени. Ако в базовите класове са дефинирани компоненти с еднакви имена, необходимо е да се използва пълното име на компонентата, т.е. <име_на_клас>::<компонента>. Ако всички основни класове $base_i$ на производния клас der имат член-функция f() и член-данна x с еднакви имена, различаването им в производния клас става чрез пълните им имена:

base1::f(), base2::f(), ...

base1::x, base2::x, ...

Пример: Нека имаме йерархията:



Класът G наследява метода f() на класовете A, B, C, D, E и F. В резултат от изпълнението на програмата:

```
#include <iostream.h>
class A
{public:
 A(int x = 1)
 {a = x;
 }
 void f() const
 {cout << "A: " << a << endl;
 }
 private:
  int a;
```

```
};
class B
{public:
 B(int x = 2)
 {b = x;
 }
 void f() const
 {cout << "B: " << b << endl;
 }
 private:
  int b;
};
class C
{public:
 C(int x = 3)
 {c = x;
 }
 void f() const
 {cout << "C: " << c << endl;
 }
 private:
  int c;
};
class D : public A, public B, public C
{public:
 D(int x = 4)
 {d = x;
 }
 void f() const
 {cout << "class D: \n" ;
  A::f(); B::f(); C::f();
  cout << "D: " << d << endl;
 }
 private:
  int d;
};
```

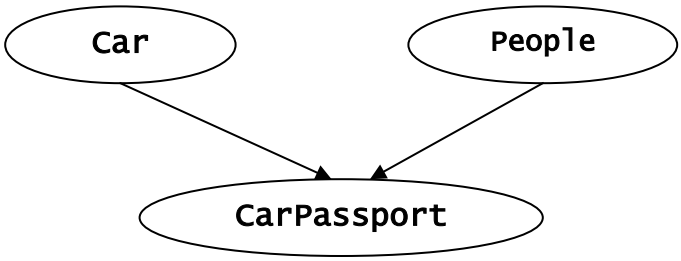
```
class E
{public:
 E(int x = 5)
 {e = x;
 }
 void f() const
 {cout << "E: " << e << endl;
 }
 private:
  int e;
};
class F
{public:
 F(int x = 6)
 {fi = x;
 }
 void f() const
 {cout << "F: " << fi << endl;
 }
 private:
  int fi;
};
class G : public D, public E, public F
{public:
 G(int x = 7)
 {g = x;
 }
 void f() const
 {cout << "G: " << endl;
  D::f(); E::f(); F::f();
  cout << "G: " << g << endl;
 }
 private:
  int g;
};
void main()
```

```
   {G ge;
    ge.A::f();
    ge.B::f();
    ge.C::f();
    ge.f();
   }
```

се получава:

```
   A: 1
   B: 2
   C: 3
   G:
   class D:
   A: 1
   B: 2
   C: 3
   D: 4
   E: 5
   F: 6
   G: 7
```

В следващата задача е реализирана следната йерархия:



Задача 166. Да се дефинират класове Car и People, определящи понятията “автомобил” и “човек”. Да се дефинира клас CarPassport, производен на класовете Car и People и определящ понятието “паспорт на автомобил”. За всеки от класовете да се определи голямата четворка.

Програма Zad166.cpp решава задачата.

```
// Program Zad166.cpp
```

```
#include <iostream.h>
#include <string.h>
class Car
{public:
  Car(char * = "", unsigned = 0, unsigned = 0);
  ~Car();
  Car(const Car&);
  Car& operator=(const Car&);
  void display() const;
 private:
  char *mark;
  unsigned year;
  unsigned reg_numb;
};
Car::Car(char *m, unsigned y, unsigned r_n)
{mark = new char[strlen(m)+1];
 strcpy(mark, m);
 year = y;
 reg_numb = r_n;
}
Car::~Car()
{cout << "~Car()\n";
 delete mark;
}
Car::Car(const Car& c)
{mark = new char[strlen(c.mark)+1];
 strcpy(mark, c.mark);
 year = c.year;
 reg_numb = c.reg_numb;
}
Car& Car::operator=(const Car& c)
{if (this != &c)
 {delete mark;
  mark = new char[strlen(c.mark)+1];
  strcpy(mark, c.mark);
  year = c.year;
```

```
  reg_numb = c.reg_numb;
 }
  return *this;
}
void Car::display() const
{cout << "Mark: " << mark << endl;
 cout << "Year: " << year << endl;
 cout << "Reg. Number: " << reg_numb << endl;
}
class People
{public:
  People(char * = "", char * = "");
  People(const People&);
  People& operator=(const People& p);
  void display() 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::display() const
{cout << "Ime: " << name << endl;
 cout << "EGN: " << egn << endl;
}
People::~People()
{cout << "~People()\n";
 delete name;
 delete egn;
}
class CarPassport: public Car, public People
{public:
  CarPassport(char* = "", unsigned = 0, unsigned = 0,
              char* = "", char* = "");
  ~CarPassport();
  CarPassport(const CarPassport&);
  CarPassport& operator=(const CarPassport&);
  void display() const;
};
CarPassport::CarPassport(char *mark, unsigned year,
           unsigned reg_numb, char* name, char *egn) :
           Car(mark, year, reg_numb), People(name, egn)
{}
CarPassport::~CarPassport()
{cout << "~CarPassport()\n";
}
CarPassport::CarPassport(const CarPassport& cp) : Car(cp),
                                                  People(cp)
{}
CarPassport& CarPassport::operator=(const CarPassport& cp)
```

```
{if (this != &cp)
 {Car::operator =(cp);
  People::operator =(cp);
 }
 return *this;
}
void CarPassport::display() const
{Car::display();
 People::display();
}
void main()
{CarPassport x("FORD FIESTA", 2000, 2295,
               "Vassil Todorov", "8012174586");
 x.display();
 CarPassport y("LADA", 1900, 8817,
               "Sonia Todorova", "8203314576");
 y = x;
 y.display();
}
```

В резултат се получава:

```
Mark: FORD FIESTA
Year: 2000
Reg. Number: 2295
Ime: Vassil Todorov
EGN: 8012174586
Mark: FORD FIESTA
Year: 2000
Reg. Number: 2295
Ime: Vassil Todorov
EGN: 8012174586
~CarPassport()
~People()
~Car()
~CarPassport()
~People()
~Car()
```

Ще отбележим, че в конструктора на производния клас CarPassport тялото е празно, тъй като класът е без собствена член-данна и единственото му предназначение е обръщение към конструкторите на базовите класове Car и People със съответните им параметри. Тялото на конструктора за присвояване на класа CarPassport също е празно. Присвояването на наследените компоненти се осъществява чрез конструкторите за присояване на основните класове. Това е указано в инициализиращия списък на конструктора за присвояване на CarPassport. Операторната функция за присвояване на CarPassport също се обръща само към операторните функции на основните си класове Car и People. Дефинирането на конструктора за присвояване и на операторната функция за присвояване на производния клас е излишно, тъй като ако бъдат пропуснати, компилаторът ще създаде "служебни", които ще се обърнат към съответните член-функции на основните класове. Същото ще се отнася за конструктора CarPassport(char*, unsigned, unsigned, char*, char*) ако в класа CarPassport не е дефиниран и конструктор за присвояване.

В тази програма е показано също как се осъществява достъпът до наследени компоненти с еднакви имена. И в базовите класове Car и People, и в производния клас CarPassport са дефинирани методи с едно и също име display(). По такъв начин класът CarPassport има три метода с името display() – два наследени от Car и People съответно и един собствен. Проблемът с идентифицирането на имената е разрешен чрез използване на пълните имена, както е показано в дефиницията на метода display() – собствен на класа CarPassport.

Забележка: Няма значение как се изброяват основните класове в списъка на производния клас с множествено наследяване. Но вече фиксирана, тази последователност се използва при:

- разполагане на наследените части на производния клас в паметта;
- неявното извикване на конструкторите (деструкторите) на основните класове.

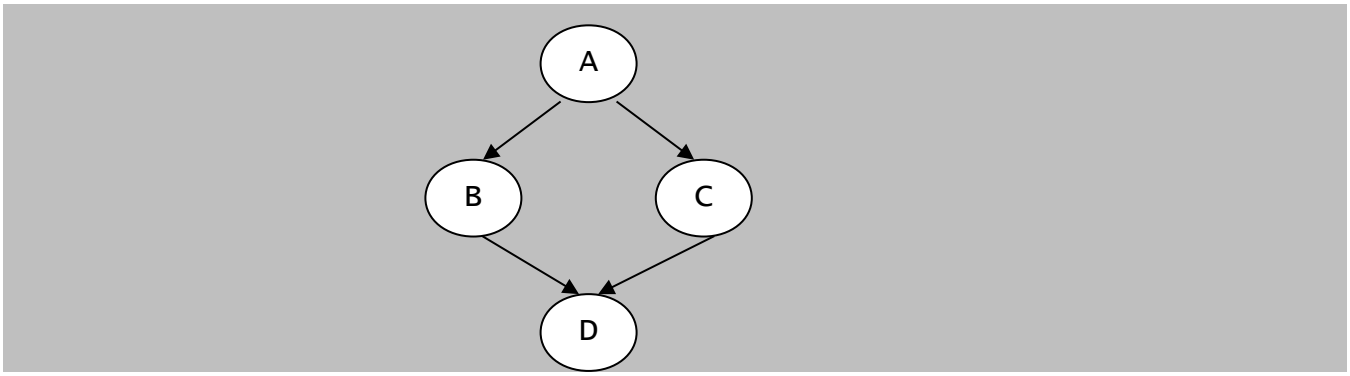
Освен това редът на записване на конструкторите на основния клас в инициализиращия списък на конструктора на производния клас няма значение. Конструкторите ще се активират в реда им в декларацията на производния клас. Ще отбележим още веднъж, че когато се използва множествено наследяване, изискванията към основните класове и

съгласуването им с производния клас са както в случая на единичното наследяване.

18.2 Виртуални класове

Стандартният механизъм за наследяване дефинира йерархия, която се представя чрез дърво. В този случай, при всяко срещане на основен клас се създава копие на неговите член-данни. Виртуалните основни класове са механизъм за отменяне на стандартния наследствен механизъм.

При реализиране на йерархии на класове с множествено наследяване е възможно един производен клас да наследи многократно даден базов клас. Например, ако класът A има два производни класа B и C, които са базови за класа D (Фиг. 18.3), т.е.



Фиг. 18.3 Една йерархия на класове

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

```
};
```

Като производни на класа A, класовете B и C наследяват неговите компоненти. От друга страна класовете B и C са базови за класа D. Следователно класът D ще наследи два пъти компонентите на класа A, веднъж чрез класа B и още веднъж – чрез класа на C. За член-функциите двойното наследяване не е от значение, тъй като за всяка член-функция се съхранява само едно копие. Член-променливите обаче се дублират и обект на класа D ще наследи двукратно всяка член-променлива, дефинирана в класа A. Класът D в паметта ще има вида:

Този пример илюстрира един от недостатъците на многократното наследяване на клас – *неефективността от поддържането на множество копия на наследени компоненти*. Ще покажем и други недостатъци на многократното наследяване на класове. Да разгледаме още веднъж горната йерархична схема като попълним декларацията на класовете A, B, C и D. Конструктор по подразбиране няма да дефинираме за нито един от тези класове.

```
class A
{public:
  A(int a)
  {x = a;
  }
  int f() const;
```

```
  void print() const;
  int x;
 };
int A::f() const
{return x;
}
void A::print() const
{cout << "A:: x " << x << endl;
}
class B: public A
{public:
  B(int a, int b): A(a)
  {x = b;
  }
  int f() const;
  void print() const;
  int x;
};
int B::f() const
{return x;
}
void B::print() const
{A::print();
 cout << "B:: x " << x << endl;
}
class C: public A
{public:
  C(int a, int c): A(a)
  {x = c;
  }
  int f() const;
  void print() const;
  int x;
};
int C::f() const
{return x;
```

```
}
void C::print() const
{A::print();
 cout << "C:: x " << x << endl;
}
class D: public B, public C
{public:
  D(int a, int b, int c, int d): B(a, b), C(c, d)
  {}
void D::func() const;
void print() const;
};
void D::print() const
{B::print();
 C::print();
}
```

След дефиницията:

```
D d(1, 2, 3, 4);
```

се получава неоднозначност: наследената двукратно член-данна x на класа A има две стойности: 1 и 3. Този проблем можем да разрешим като дефинираме конструктора на класа D по следния начин:

```
D(int a, int b, int c): B(a, b), C(a, c)
  {}
```

но двукратно заделената памет си остава.

Друг проблем възниква при опит в класа D да се използват наследените компоненти A::x, A::f() или A::print(). В случая A::x причината е ясна – има две копия на A::x, но копията на A::f() и A::print() са единствени за класа D. Въпреки това съществува противоречие.

Пример: В резултат от компилирането на функцията:

```
void D::func() const
{A::print();
}
```

се получава грешката: *error C2385: 'D::A' is ambiguous*. Причината е, че единственият аргумент на func е this, който е и аргумент на print() в A::print(). this е указател към обект на класа D, който

обект съдържа два обекта от основния клас A. Кой от тях да се свърже с извиканата функция print()? Грижата да се посочи един от двата обекта си остава на програмиста. Ако е необходим само достъп до “конфликтните” наследени членове на клас A (без да се променят стойностите), осъществяването му става чрез последователно прилагане на операцията за явно преобразуване на типове. При атрибут за област public, обект на производен клас може да се преобразува в обект на основен клас с неявни преобразувания, но поради двата клона в йерархичната схема, обект от клас D не може да се преобразува директно в обект от клас A. Възможни са следните последователни преобразувания:

```
D d(1, 2, 3, 4);
(A)(B) d;
(A)(C) d;
```

Като се използват подобни преобразувания, може да се осигури достъп до наследените от класа A членове на класа D. Ще ги илюстрираме с дефиниция на член-функцията func() на D.

```
void D::func() const
{cout << "Derived member x in a part A-B-D "
      << ((A)(B)*this).x << endl
      << "Derived member x in a part A-C-D "
      << ((A)(C)*this).x << endl
      << “Derived member-function f() in a part A-B-D ”
      << ((A)(B)*this).f() << endl
      << “Derived member-function f() in a part A-C-D ”
      << ((A)(C)*this).f() << endl
      << “Derived member-function f() in part C-D”
      << ((C)*this).f() << endl
      << “Derived member-function f() in part B-D”
      << ((B)*this).f() << endl;
}
```

Резултатът от изпълнението на фрагмента:

```
D d(1, 2, 3, 4);
d.func();
```

e:

```
Derived member x in a part A-B-D 1
```

Derived member x in a part A-C-D 3
Derived member-function f() in a part A-B-D 1
Derived member-function f() in a part A-C-D 3
Derived member-function f() in part C-D 4
Derived member-function f() in part B-D 2

Ще напомним, че указателят this сочи обект от клас D, *this е неговото съдържание, т.е. *this е обект от тип D. Чрез явните преобразувания (A)(B)*this този обект се превръща в обект отначало от клас B, а след това в обект от клас A. Чрез оператора . се прави достъп до член на съответния клас. *При тези преобразувания обектите от класове B и A са **временни**. Това е причината, заради която е възможен само достъп, а не промяна на стойности, т.е. при опит за промяна, тя ще е във временна, а не в постоянната памет*.

Ще отбележим също, че обръщението

```
d.print();
```

извиква два пъти метода A::print(), веднъж от обръщението B::print() и друг път от C::print().

Пример: Резултатът от изпълнението на фрагмента:

```
D d(1, 2, 3, 4);
d.print();
```

е:

```
A:: x 1
B:: x 2
A:: x 3
C:: x 4
```

Многократното наследяване на клас води от една страна до затруднен достъп до многократно наследените членове, а от друга до поддържане на множество копия на член-данните на многократно наследения клас, което не е ефективно.

Преодоляването на недостатъците на многократното наследяване на клас се осъществява чрез използването на т.н. ***виртуални основни класове***. Чрез тях се дава възможност да се “поделят” основни класове. Когато един клас е виртуален, независимо от участието му в няколко списъка на основни класове, се създава само едно негово копие. В

нашия случай, ако класът A се определи като виртуален за класовете B и C, класът D ще съдържа само един “поделен” основен клас A.

Декларация

Декларацията на основен клас като виртуален се осъществява като в декларацията на производния клас заедно с името и атрибута за област на основния клас се укаже и запазената дума virtual.

Пример: Ще променим декларацията на йерархичната схема от Фиг. 18.3 като определим класа A като виртуален на класовете B и C:

```
class A
{public:
  A(int a)
  {x = a;
  }
  int f() const;
  void print() const;
  int x;
 };
int A::f() const
{return x;
}
void A::print() const
{cout << "A:: x " << x << endl;
}
class B: virtual public A
{public:
  B(int a, int b): A(a)
  {x = b;
  }
  int f() const;
  void print() const;
  int x;
};
int B::f() const
{return x;
}
```

```
void B::print() const
{A::print();
 cout << "B:: x " << x << endl;
}
class C: virtual public A
{public:
  C(int a, int c): A(a)
  {x = c;
  }
  int f() const;
  void print() const;
  int x;
};
int C::f() const
{return x;
}
void C::print() const
{A::print();
 cout << "C:: x " << x << endl;
}
class D: public B, public C
{public:
  D(int a, int b, int c, int e): A(a), B(a, b), C(c, e)
  {}
void D::func() const;
void print() const;
};
void D::print() const
{B::print();
 C::print();
}
```

Така класът A е обявен за виртуален. Казва се, че **B и C наследяват класа A виртуално**. Виртуалното наследяване на класа A от класовете B и C оказва влияние само на производните на B и C класове. То не променя поведението на самите класове B и C. Забелязваме, че запазената дума virtual е поставена пред атрибута за област на

виртуалния клас A. Всъщност, няма значение редът на virtual и атрибута за област.

Дефинирането и използването на виртуални класове има редица особености. Една от тях касае дефиницията и използването на конструкторите на наследените класове. Нека A е виртуален основен клас за класа B, а класът B е основен за класа D, който пък е основен за класа E. Ако класът A има конструктор с параметри и няма подразбиращ се конструктор, то този конструктор трябва да бъде извикан не само от конструктора на класа B, но и от конструкторите на класовете D и E. Правилото за извикване на конструктори с параметри на виртуални класове може да се изкаже така: конструкторите с параметри на виртуални класове трябва да се извикват от конструкторите на всички класове, които са техни наследници, а не само от конструкторите на преките им наследници. С други думи, производният клас е отговорен за инициализирането на класовете, от които произлиза, както и на всички виртуални основни класове. Ако в инициализиращия списък на конструктора на производен клас няма обръщение към конструктор с параметър на виртуалния клас, използва се неговия подразбиращ се конструктор, ако такъв съществува или се съобщава за отсъстието на подходящ конструктор.

В примера по-горе са дефинирани конструктори с параметри за класовете A, B, C и D. Ще отбележим, че конструкторът с параметри на класа D:

```
D(int a, int b, int c, int d): A(a), B(a, b), C(c, d)
{}
```

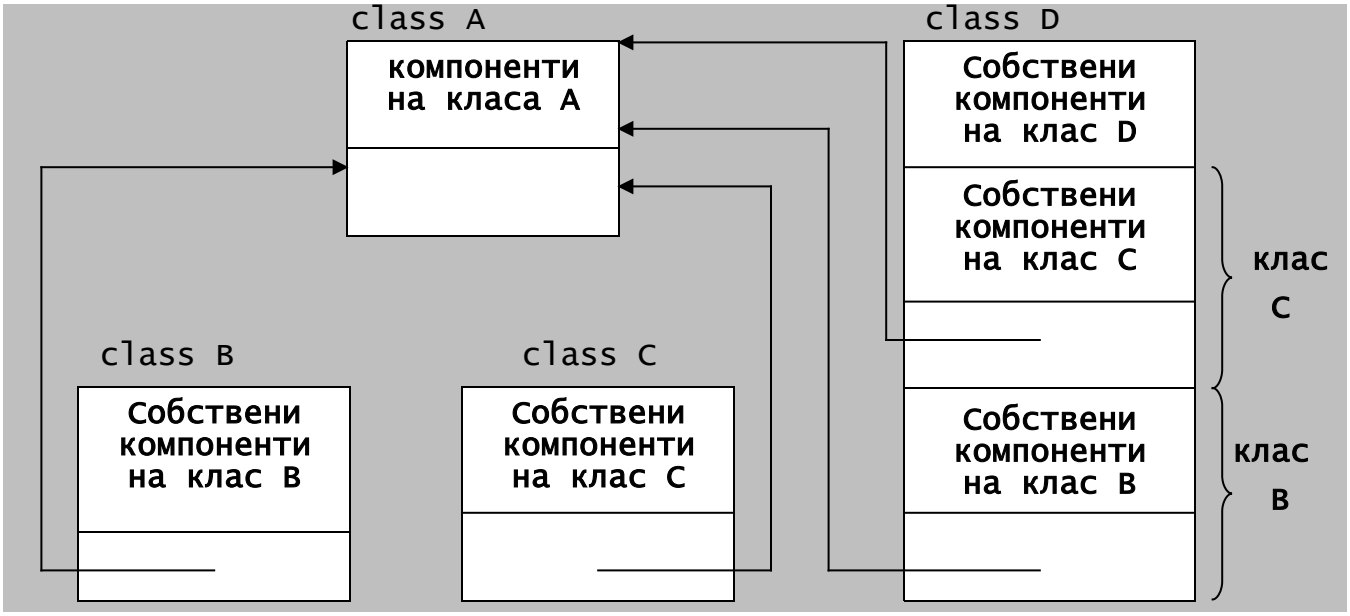
явно в инициализиращия си списък се обръща към конструктора на виртуалния основен клас A. Ако това не е направено, компилаторът ще съобщи за грешка, тъй като във виртуалния основен клас A не е определен конструктор по подразбиране. Ако класът A не беше виртуален неговият конструктор нямаше да се изисква в дефиницията на конструктора на класа D и употребата му щеше да доведе до грешка.

Друга особеност е промяната на реда на инициализиране. Инициализирането на виртуални основни класове предхожда инициализирането на другите основни класове в декларацията на производния клас. Ако производен клас наследява основен и виртуален клас, конструкторът на виртуалния клас се извиква първи. При няколко

виртуални класа извикването на конструкторите става по реда им в декларацията на производния клас.

Как използването на виртуални основни класове преодолява недостатъците на многократното наследяване?

Виртуалният основен клас е общ за всички производни от него класове. Ще се върнем към разглежданата по-горе йерархия на класове (Фиг. 18.3) и схематично ще опишем как тя се реализира след обявяването на клас A за виртуален (Фиг. 18.4).



Фиг. 18.4 Реализация на виртуално наследяване

Фиг. 18.4 показва как е преодолян проблемът с многократното наследяване на член-данните на клас. Обръщенията A::x, A::f() или A::print() в класа D вече не създават проблем, тъй като класът A се наследява вече само веднъж. Резултатът от изпълнението на фрагмента:

```
   D d(1, 2, 3, 4);
   d.func();
e:
   Derived member x in a part A-B-D 1
   Derived member x in a part A-C-D 1
   Derived member-function f() in a part A-B-D 1
   Derived member-function f() in a part A-C-D 1
```

```
   Derived member-function f() in part C-D 4
   Derived member-function f() in part B-D 2
```
Използването на виртуални основни класове не премахва нееднозначността при достъп до някои член-функции. Например, обръщението
```
   d.print();
```
ще извика метода A::print() отново два пъти, т.е. резултатът от изпълнението на обръщението:
```
   d.print();
```
е:
```
   A:: x 1
   B:: x 2
   A:: x 1
   C:: x 4
```
За да се избегне двукратното изпълнение на A::print() може да се постъпи по следния начин. Дефинициите на методите print() се променят като всеки метод print() се състои от две части: print_own() и print(), използваща наследените print_own() от предшестващите класове.

Пример: Примерната програма ще променим до:
```
   #include <iostream.h>
   class A
   {public:
     A(int a)
     {x = a;
     }
    int f() const;
    void print() const
    {print_own();
    }
    private:
     int x;
    protected:
     void print_own() const;
   };
   int A::f() const
```

```
{return x;
}
void A::print_own() const
{cout << "A:: x " << x << endl;
}
class B: virtual public A
{public:
  B(int a, int b): A(a)
  {x = b;
  }
  int f() const;
  void print() const
  {A::print();
   print_own();
  }
 private:
  int x;
protected:
  void print_own() const;
};
int B::f() const
{return x;
}
void B::print_own() const
{cout << "B:: x " << x << endl;
}
class C: virtual public A
{public:
  C(int a, int c): A(a)
  {x = c;
  }
  int f() const;
  void print() const
  {A::print();
   print_own();
  }
```

```
 private:
  int x;
 protected:
  void print_own() const;
};
int C::f() const
{return x;
}
void C::print_own() const
{cout << "C:: x " << x << endl;
}
class D: public B, public C
{public:
  D(int a, int b, int c, int e): A(a), B(a, b), C(c, e)
  {
  }
void print() const
{print_own();
 A::print_own();
 B::print_own();
 C::print_own();
}
protected:
void print_own() const;
};
void D::print_own() const
{
}
void main()
{D d(1,2,3,4);
 d.print();
}
```

Резултат:

```
A:: x 1
B:: x 2
C:: x 4
```

Сега вече е решен и проблемът с двойното изпълнение на член-функцията print() на класа A.

Задача 166. Да се дефинира йерархията от фиг. 18.3 като член-данните x на класовете A, B, C и D са от тип низ, реализират се в ДП и са капсолирани. За всеки от класовете да се определи “голямата четворка”.

Програма Zad166.cpp решава задачата.

```
// Program Zad166.cpp
#include <iostream.h>
#include <string.h>
class A
{public:
  A(char* = "");
  ~A();
  A(const A&);
  A& operator=(const A &);
  void print() const;
 private:
  char* x;
 };
A::A(char* s)
{x = new char[strlen(s)+1];
 strcpy(x, s);
}
A::~A()
{delete x;
}
A::A(const A& p)
{x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
A& A::operator=(const A& p)
{if (this != &p)
```

```
{delete x;
 x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
 return *this;
}
void A::print() const
{cout << "A:: x " << x << endl;
}
class B: virtual public A
{public:
  B(char* = "", char* = "");
  ~B();
  B(const B&);
  B& operator=(const B&);
  void print() const;
 private:
  char* x;
};
B::B(char* a, char* b): A(a)
{x = new char[strlen(b)+1];
 strcpy(x, b);
}
B::~B()
{delete x;
}
B::B(const B& p) : A(p)
{x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
B& B::operator=(const B& p)
{if (this != &p)
{A::operator=(p);
 delete x;
 x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
```

```
}
 return *this;
}
void B::print() const
{A::print();
 cout << "B:: x " << x << endl;
}
class C: virtual public A
{public:
  C(char* = "", char* = "");
  ~C();
  C(const C&);
  C& operator=(const C&);
  void print() const;
 private:
  char* x;
};
C::C(char* a, char* b): A(a)
{x = new char[strlen(b)+1];
 strcpy(x, b);
}
C::~C()
{delete x;
}
C::C(const C& p) : A(p)
{x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
C& C::operator=(const C& p)
{if (this != &p)
{A::operator=(p);
 delete x;
 x = new char[strlen(p.x)+1];
 strcpy(x, p.x);
}
 return *this;
```

```
}
void C::print() const
{A::print();
 cout << "C:: x " << x << endl;
}
class D: public B, public C
{public:
  D(char* a = "", char* b = "", char* c = "", char* d = "") :
                                    A(a), B(a, b), C(a, c)
  {x = new char[strlen(d)+1];
   strcpy(x, d);
  }
  ~D()
  {cout << "~D(): \n";
   delete x;
  }
  D(const D& p): A(p), B(p), C(p)
  {x = new char[strlen(p.x)+1];
   strcpy(x, p.x);
  }
  D& operator=(const D&p)
  {if(this!=&p)
  {B::operator =(p);
   C::operator =(p);
   delete x;
   x = new char[strlen(p.x)+1];
   strcpy(x, p.x);
  }
  return *this;
  }
 void print() const;
 private:
  char* x;
};
void D::print() const
{B::print();
```

```
 C::print();
 cout << "D::x: " << x << endl;
}
void main()
{D d("Mimi", "Toni", "Liza", "Lora");
 d.print();
 D d1, d2;
 d2 = d1 = d;
 d1.print();
 d2.print();
}
```

Резултат:

Фрагментът

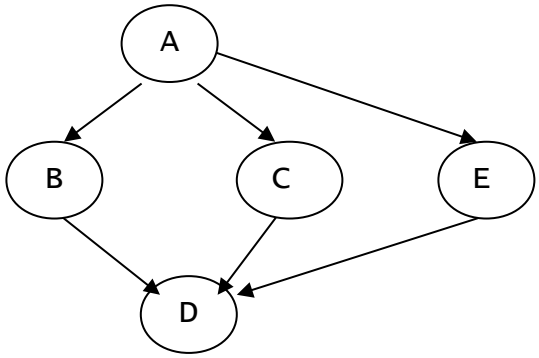
```
A::x Mimi
B::x Toni
A::x Mimi
C::x Liza
D::x Lora
```

се повтаря три пъти, след което също три пъти се извежда:

```
~D():
```

В тази реализация не се грижим за избягване на двойното извеждане на компонентите на класа A.

Характеристиките на виртуалните класове могат да се комбинират с тези на обикновените. Например, може да се разглежда йерархия от вида:



така, че A така, че A е виртуален за класовете B и C и не е виртуален за класа E. Класът D обединява три разклонения: два с виртуален основен клас A и един с наследени членове от обикновения клас A. Механизмът на виртуалните основни класове ще обедини двата виртуални

клона в едно виртуално копие на основния клас, но третият клон ще породи още едно наследено копие на основния клас A, използван като обикновен клас. В този случай се получават неоднозначности при достъп до членовете на двете копия. Преодоляването им става чрез преобразувания.

Нека накрая разгледаме отново познатата йерархична схема от Фиг. 18.3, в която класът A е виртуален за класовете B и C, но атрибутът му за област е public за класа B и private за класа C. Очевидно ситуацията е неоднозначна – обект на D няма достъп до компонентите на класа A по пътя A-C-D, но има такъв по пътя A-B-D. Тази неоднозначност е преодоляна с избора, че ако в някоя декларация виртуалният клас е обявен като public, счита се, че той е с атрибут public навсякъде.

Пример: Ако

```
D d(1, 2, 3, 4);
```

обръщенията: d.print(); d.A::print(); d.B::print(); d.C::print(); са коректни. А ако

```
C c(5, 7);
```

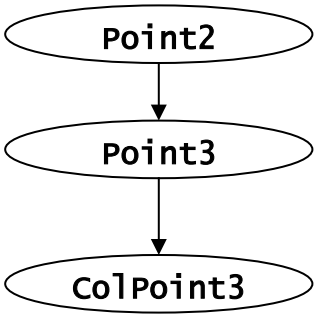
обръщението c.A::print(); не е допустимо заради това, че A е с атрибут за област private в класа C.

18.3. Динамично свързване. Виртуални функции

Вече използвахме функции с еднакви имена в т.ч. и методи на класове. В случая на обикновени функции за разпознаването на функцията се използва механизъм, който се изразява в следното: по време на компилация се сравняват формалните с фактическите параметри в обръщението и по правилото за най-доброто съвпадане се избира необходимата функция. След заместване на формалните с фактическите параметри се изпълнява тялото на функцията. При член-функциите на йерархията от класове, конфликтът между имената на наследените и собствените методи от един и същ тип и с едни и същи параметри се разрешава също по време на компилация чрез правилото на локалния приоритет и чрез явно посочване на класа, към който принадлежи методът. В тези два случая тъй като процесът на реализиране на обръщението към функцията приключва по време на компилация и не може

да бъде променян по време на изпълнение на програмата се казва, че има **статично разрешаване на връзката** или **статично свързване**.

Пример: В следващата програма е дефинирана йерархията:



определяща точка в равнината, точка в тримерното пространство и точка в тримерното пространство с цвят.

```
#include <iostream.h>
class Point2
{public:
 Point2(int a = 0, int o = 0)
 {x = a;
  y = o;
 }
 void Print() const
 {cout << x << ", " << y ;
 }
 private:
  int x, y;
 };
class Point3 : public Point2
{public:
 Point3(int a = 0, int o = 0, int b = 0) : Point2(a, o)
 {z = b;
 }
 void Print() const
 {Point2::Print();
  cout << ", " << z << endl;
 }
 private:
```

```
  int z;
};
class ColPoint3 : public Point3
{public:
 ColPoint3(int a = 0, int o = 0, int b = 0, int c = 0) :
                                          Point3(a, o, b)
 {col = c;
 }
 void Print() const
 {Point3::Print();
  cout << "colour: " << col << endl;
 }
 private:
  int col;
};
void main()
{Point2 p2(5, 10);
 Point3 p3(2, 4, 6);
 ColPoint3 p4(12, 24, 36, 11);
 Point2 *ptr1 = &p3; // атрубутът на Point2 е public
 ptr1->Print();
 cout << endl;
 Point2 *ptr2 = &p4; // атрубутът на Point2 е public
 ptr2->Print();
}
Резултат:
2 4
12 14
```

И в трите класа е дефинирана функция Print() без параметри и от тип void. В главната функция са дефинирани три обекта: p2, p3 и p4 от класове Point2, Point3 и ColPoint3 съответно. Освен това са дефинирани указатели и ptr1 и ptr2 към класа Point2. Указателят ptr1 е инициализиран е с адреса на обекта p3 от класа Point3, а ptr2 – с адреса на обекта p4 от класа ColPoint3. Тъй като атрибутът за област на Point2 в Point3 е public и атрибутът за област на Point3 в ColPoint3 също е public, преобразуванията са допустими. Обръщението:

ptr1->Print(); извежда първите две координати на точката p3, а ptr2->Print(); - първите две координати на точката с цвят p4, т.е. изпълнява се Print() на класа Point2 и в двата случая. Още по време на компилация член-функцията Print() на Point2 е определена като функция на обръщенията ptr1->Print() и ptr2->Print(). Определянето става от типа Point2 на указателите ptr1 и ptr2. Връзката е определена статично и не може да се промени по време на изпълнение на програмата.

Ако искаме след свързването на ptr1 с адреса на p3 да се изпълни член-функцията Print() на Point3, а също след свързването на ptr2 с адреса на p4 да се изпълни член-функцията Print() на ColPoint3 са необходими явни преобразувания от вида:

```
Point2 *ptr1 = &p3;
 ((Point3*)ptr1)->Print();
 cout << endl;
 Point2 *ptr2 = &p4;
 ((ColPoint3*)ptr2)->Print();
```

Отново връзките са разрешени статично.

При статичното свързване по време на създаването на класа трябва да се предвидят възможните обекти, чрез които ще се викат член-функциите му. При сложни йерархии от класове това е не само трудно, но и понякога невъзможно. Езикът C++ поддържа още един механизъм, прилаган върху специален вид член-функции, наречен **късно** или **динамично свързване**. При него изборът на функцията, която трябва да се изпълни, става по време на изпълнение на програмата.

Динамичното свързване капсулира детайлите в реализацията на йерархията. При него не се налага проверка на типа. Текстовете на програмите се опростяват, а промени се налагат много по-рядко. Разширяването на йерархията не създава проблеми. Това обаче е с цената на усложняване на кода и забавяне на процеса на изпълнение на програмата.

Наличието на двата механизма на свързване – статично и динамично, дава възможност на програмиста да се възползва от положителните им страни.

Прилагането на механизма на късното свързване се осъществява върху специални член-функции на класове, наречени **виртуални член-функции** или само **виртуални функции**.

Виртуалните методи се декларират чрез поставяне на запазената дума virtual пред декларацията им, т.е.

```
virtual <тип_на_резултата> <име_на_метод>(<параметри>);
```

Пример: В класовете Point2, Point3 и ColPoint3, на програмата от примера по-горе, член-функциите void Print()const са обявени за виртуални.

```
#include <iostream.h>
class Point2
{public:
 Point2(int absc = 0, int ord = 0)
 {x = absc;
  y = ord;
 }
 virtual void Print() const
 {cout << x << ", " << y ;
 }
 private:
  int x, y;
 };
class Point3 : public Point2
{public:
Point3(int a = 0, int o = 0, int b = 0) : Point2(a, o)
{z = b;
}
virtual void Print() const
{Point2::Print();
 cout << ", " << z << endl;
}
private:
  int z;
};
```

```
class ColPoint3 : public Point3
{public:
 ColPoint3(int a=0, int o=0, int b=0, int c=0) : Point3(a, o, b)
 {col = c;
 }
 virtual void Print() const
 {Point3::Print();
  cout << "colour: " << col << endl;
 }
 private:
  int col;
};
void main()
{Point2 p2(5, 10);
 Point3 p3(2, 4, 6);
 ColPoint3 p4(12, 24, 36, 11);
 Point2 *ptr1 = &p3;
 ptr1->Print();
 cout << endl;
 Point2 *ptr2 = &p4;
 ptr2->Print();
}
Резултат:
2, 4, 6
12, 24, 36
colour: 11
```

Декларирането на член-функцията Print() като виртуална причинява обръщенията ptr1->Print(); и ptr2->Print(); да определят функцията, която ще бъде извикана едва при изпълнението на програмата. ***Определянето е в зависимост от типа на обекта, към който сочи указателят, а не от класа към който е указателят.*** В случая, указателят ptr1 е към класа Point2, но сочи обекта p3, който е от класа Point3. Затова обръщението ptr1->Print(); изпълнява Point3::Print(). Указателят ptr2 е към класа Point2, но сочи обекта p4, който е от класа ColPoint3. Затова обръщението ptr2->Print(); изпълнява ColPoint3::Print().

Ще отбележим, че:

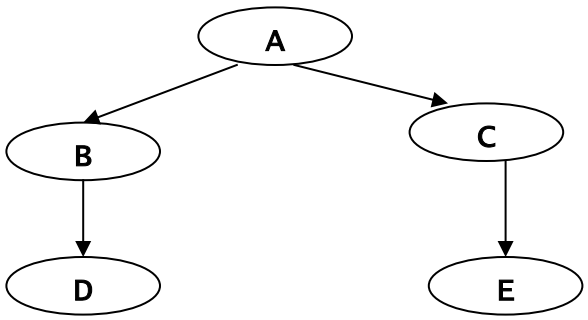
1. Само член-функции на класове могат да се декларират като виртуални. По технически съображения конструкторите не могат да се деларират като виртуални.
2. Ако в даден клас е декларирана виртуална функция, декларираните член-функции със същия прототип (име, параметри и тип на върнатата стойност) в производните на класа класове също са виртуални дори ако запазената дума virtual бъде пропусната.
3. Ако в производен клас е дефинирана функция със същото име като определена вече в основен клас като виртуална член-функция, но с други параметри и/или тип, то това ще е друга функция, която може да бъде или да не бъде декларирана като виртуална.
4. Ако в производен клас е дефинирана виртуална функция със същия прототип като на невиртуална функция на основен клас, то те се интерпретират като различни функции.
5. Възможно е виртуална функция да се дефинира извън клас. Тогава заглавието ѝ не започва със запазената дума virtual, т.е. запазената дума virtual може да се среща само в тялото на клас.
6. Виртуалните функции се наследяват като другите компоненти на класа.
7. Основният клас, в който член-функция е обявена за виртуална, трябва да е с атрибут public в производните от него класове.
8. Виртуалните функции се извикват чрез указател към или псевдоним на обект от някакъв клас.
9. Виртуалната функция, която в действителност се изпълнява, зависи от типа на аргумента.
10. Виртуалните функции не могат да бъдат декларирани като приятели на други класове.

Някои предимства на виртуалните функции

1. *Производният клас наследява всяка виртуална функция на базовия клас, за която няма собствена дефиниция*

От тук следва, че не е задължително виртуалните функции да се декларират във всеки клас от йерархията. Ако виртуална функция е дефинирана в базов клас и логиката на производния клас не изисква нейното предефиниране, декларацията й може да се пропусне. Когато бъде извикана виртуална функция за обект от даден клас, тя се търси в него. Ако не е дефинирана в класа, търсенето продължава в базовия клас и нагоре по йерархията.

Пример: Нека виртуалната функция void f() е дефинирана като виртуална само в класовете A и B на йерархията:



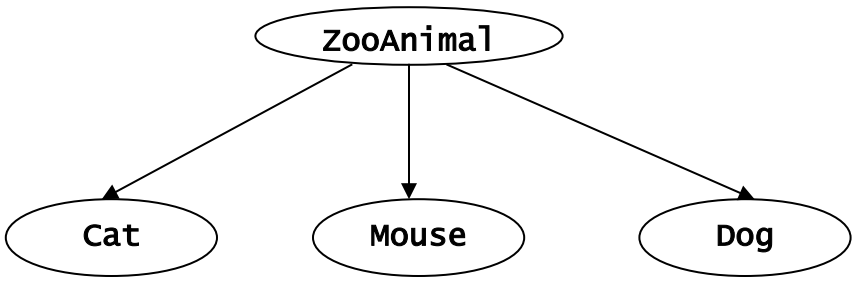
Извикването на функцията f() от обекти от класовете A, C и E ще доведе до изпълнението на функцията A::f(), а нейното извикване за обекти от класовете B и D – ще изпълни функцията B::f(). Ако в класа C бъде дефинирана функция от вида: void f(){}, то функцията A::f() ще се извика само за обект на класа A. За класовете C и E ще бъде извикана празната виртуална функция.

2. Реализират се полиморфни действия

Полиморфизмът е важна характеристика на ООП. Изразява се в това, че едни и същи действия (в общия смисъл) се реализират по различен начин в зависимост от обектите, върху които се прилагат, т.е. действията са полиморфни (с много форми). Полиморфизмът е свойство на член-функциите на обектите и в езика C++ се реализира чрез виртуални функции. За да се реализира полиморфно действие, класовете върху които то ще се прилага, трябва да имат общ родител или прародител, т.е. да бъдат производни на един и същ клас. В този клас трябва да

бъде дефиниран виртуален метод, съответстващ на полиморфното действие. Във всеки от производните класове този метод може да бъде предефиниран съобразно особеностите на този клас. Активирането полиморфното действие става чрез указател към базовия клас, на който могат да се присвоят адресите на обекти на който и да е от производните класове от йерархията. Ще бъде изпълнен методът на съответния обект, т.е. в зависимост от обекта към който сочи указателят ще бъде изпълняван един или друг метод. Ако класовете, в които трябва да се дефинират виртуални методи нямат общ родител, такъв може да бъде създаден изкуствено чрез дефиниране на т.н. **абстрактен клас**.

Пример: В йерархия на класове еднотипни действия са описани с член-функции с еднакви прототипи. Член-функциите на производните класове обикновено извършват редица общи действия. В този случай в основния клас може да се реализира една невиртуална функция, която извършва общите действия и след или преди това извиква виртуалната функция, извършваща специфичните действия на класовете. В следващата програма е дефинирана йерархията от класове:




```
#include <iostream.h>
class ZooAnimal
{public:
 void print() const
 {cout << "ZooAnimal\n";
  cout << "Address:\n"
      << "Sofia, Bulgaria\n";
 }
 private:
 //...
 };
```

```
class Cat : public ZooAnimal
{public:
 void print() const
 {cout << "ZooAnimal\n";
  cout << "Cat\n";
 }
 //...
};
class Mouse : public ZooAnimal
{public:
 void print() const
 {cout << "ZooAnimal\n";
  cout << "Mouse\n";
 }
 //...
};
class Bear : public ZooAnimal
{public:
 void print() const
 {cout << "ZooAnimal\n";
  cout << "Bear\n";
 }
 //...
};
void main()
{ZooAnimal zoo; zoo.print();
 Cat c; c.print();
 Mouse m; m.print();
 Bear b; b.print();
 }
```

Резултат:

```
ZooAnimal
Address:
Sofia, Bulgaria
ZooAnimal
Cat
```

```
ZooAnimal
Mouse
ZooAnimal
Dog
```

Член-функцията void print()const; на всеки един от класовете извежда общата за всички класове информация:

```
ZooAnimal
```

и специфична за всеки клас информация – определяща: адреса на зоологическата градина (в клас ZooAnimal) и вида на животното Cat, Mouse или Dog в производните класове Cat, Mouse или Dog, съответно. Следващата програма е модификация на горната. В класа ZooAnimal е дефинирана обикновена член-функция void print() const, която извежда повтарящия се текст, след което се обръща към виртуалната функция void spec() const;. Тази функция описва специфичните за класовете ZooAnimal, Cat, Mouse и Dog действия. Функцията spec() има един параметър – this. Когато this сочи обект от клас Cat, spec() е функцията Cat::spec(), когато this сочи обект от клас Mouse, spec() е функцията Mouse::spec(), а когато this сочи обект от клас Dog, spec() е функцията Dog::spec().

```
#include <iostream.h>
class ZooAnimal
{public:
 virtual void spec() const
 { cout << "Address:\n"
        << "Sofia, Bulgaria\n";
 }
 void print() const
 {cout << "ZooAnimal\n";
  spec();
 }
 private:
 //...
 };
class Cat : public ZooAnimal
{public:
 virtual void spec() const
```

```
 {cout << "Cat\n";
 }
 //...
};
class Mouse : public ZooAnimal
{public:
virtual void spec() const
 {cout << "Mouse\n";
 }
 //...
};
class Bear : public ZooAnimal
{public:
virtual void spec() const
 {cout << "Bear\n";
 }
 //...
};
void main()
{ZooAnimal zoo; zoo.print();
 Cat c; c.print();
 Mouse m; m.print();
 Bear b; b.print();
}
```

В случая, общият повтарящ се код е малък по обем, но има йерархии, където това не е така.

Същият резултат се получава след изпълнение на фрагмента:

```
ZooAnimal zoo, *pzoo;
Cat c; Mouse m; Bear b;
pzoo = &zoo; pzoo->print();
pzoo = &c; pzoo->print();
pzoo = &m; pzoo->print();
pzoo = &b; pzoo->print();
```

Забелязваме, че едно и също обръщение: pzoo->print(); е извикано четири пъти и всеки път изпълнява член-функцията print() с различни обръщения към виртуалната функция spec(). Обръщението pzoo->print()

се разрешава статично, тъй като print() не е виртуална. Полиморфният характер произлиза от съдържащата се в нея виртуална вункция spec().

Преди да разгледаме абстрактните класове, ще се спрем на още един важен въпрос – **достъпът до виртуална функция**. Всяка член-функция на клас, в който е дефинирана виртуална функция, има пряк достъп до виртуалната функция, т.е. на локално ниво достъпът се определя по традиционните правила. На глобално ниво достъпът е малко по-различен. Преди да изкажем правилото, ще разгледаме следната примерна програма:

```
#include <iostream.h>
class Base
{public:
 virtual void pub()
 {cout << "pub()\n";
  //....
 }
 void usual()
 {cout << "usual()\n";
  pub();
  pri();
  pro();
 }
private:
 virtual void pri()
 {cout << "pri()\n";
  //....
 }
 protected:
 virtual void pro()
 {cout << "pro()\n";
  //....
 }
};
class Der : public Base
{protected:
 virtual void pub()
```

```
    {cout << "Derived class\n";
     Base::pub();
     Base::pro();
    }
    public:
    virtual void pri()
    {cout << "Derived-pri()\n";
    }
    virtual void pro()
    {cout << "Derived-pro()\n";
    }
   };
   void main()
   {Base *p = new Base;
    Base *q = new Der;
    p->pub();
    q->pub();
   // p->pri();
   // q->pri();
    Der *r = new Der;
    r->pri();
   // q->pro();
   // r->pub();
    p->usual();
   }
```

В нея е реализирана йерархията Base -> Der, като в класа Base са дефинирани три виртуални функции:

```
   void pub(); - в секция public
   void pri(); - в секция private
   void pro(); - в секция protected
```

и една обикновена член-функция:

```
   void usual(); - в секция public, която ги използва.
```

В класа Der са предефинирани трите виртуални функции, но с променен достъп:

```
   void pub(); - в секция protected
   void pri(); и void pro(); - в секция public.
```

В главната функция са дефинирани два указателя p и q към класа Base и указател r към производния клас Der. Тъй като функцията pub() е виртуална, десните страни на дефинициите:

```
Base *p = new Base;
Base *q = new Der;
```

определят, че в обръщенията:

```
p->pub();
q->pub();
```

p ще активира Base::pub(), а q – Der::pub(), ако е възможен достъп. Достъпът се определя от вида на секцията на метода pub() в класовете към които сочат указателите. Тъй като и p, и q са от тип Base* (т.е. сочат към клас Base) и в Base pub() е в секция public, независимо, че Der::pub() е в секция protected, обръщенията се изпълняват.

Обръщенията:

```
// p->pri();
// q->pri();
```

са коментирани, тъй като не са успешни. Член-функцията pri() е виртуална и тъй като p и q не са променени, десните страни на дефинициите:

```
Base *p = new Base;
Base *q = new Der;
```

определят, че p ще активира Base::pri();, а q – Der::pri(), ако е възможен достъп. Достъпът се определя от вида на секцията, в която се намира pri() в класа Base (към него сочат и p, и q). Тъй като секцията е private, достъпът е невъзможен.

Дефиницията

```
Der *r = new Der;
```

определя r като указател към Der и го свързва с обект от него. Функцията pri() е виртуална. Според дясната страна на дефиницията на r, обръщението:

```
r->pri();
```

активира Der::pri(). Тъй като r е указател към Der, а в класа Der функцията pri() е дефинирана в секция public, достъпът е възможен.

Обръщенията:

```
// q->pro();
// r->pub();
```

отново са коментирани, тъй като не са успешни. В първия случай виртуалната функция pro() е дефинирана в секция protected в класа Base, към който сочи указателят q. Във втория случай виртуалната функция pub() е дефинирана в секция protected в класа Der, към който сочи указателят r.

Обръщението

```
p->usual();
```

е допустимо, тъй като обикновената член-функция на класа Base е дефинирана в секция public.

Ще заключим, че достъпът до виртуална функция на глобално ниво зависи от секцията, в която е дефинирана тя, на класа към който сочи указателят, чрез който се активира функцията.

Съществуват три случая, при които обръщението към виртуална функция се решава статично (по време на компилация):

1. *Виртуалната функция се извиква чрез обект на класа, в който е дефинирана*

Пример: Фрагментът

```
Cat c; c.spec();
Mouse m; m.spec();
Bear b; b.spec();
```

е допустим. Независимо че spec() е виртуална, предварително (по време на компилация) се определя, че в c.spec() се извиква spec() на класа Cat, че в m.spec() се извиква spec() на класа Mouse и че в b.spec() се извиква spec() на класа Bear. Ще отбележим изрично, че методите void spec() са обявени в секция public. В противен случай достъпът е невъзможен.

Ще отбележим също, че ако

```
ZooAnimal *z;
```

обръщението:

```
(*z).spec();
```

е виртуално.

2. *Виртуалната функция се активира чрез указател към или псевдоним на обект, но явно, чрез операцията ::, е посочена конкретната функция*

Пример:

```
ZooAnimal *pz;
```

```
Bear b; Cat c; Mouse m;
pz = &b; pz -> spec(); // динамично свързване
pz = &c; pz -> spec(); // динамично свързване
pz = &m; pz -> spec(); // динамично свързване
pz->ZooAnimal::spec();  // статично свързване
```

Отново ще отбележим, че методът void spec() е в секция public за всеки от класовете на йерархията с основен клас ZooAnimal.

3. Виртуалната функция се активира с тялото на конструктор или деструктор на основен клас.

Това е така, защото обектът от производния клас още не е създаден или вече е разрушен.

18. 4. Абстрактни класове. Контейнерни класове

Възможно е виртуалните функции да имат само декларация, а не дефиниция. Такива виртуални член-функции се наричат **чисти**. За да се определи една виртуална функция като чиста се използва следният синтаксис:

```
virtual <тип> <име_на_функция>(<параметри>) = 0;
```

Клас, в който е декларирана поне една чиста виртуална функция се нарича **абстрактен**. Абстрактните класове се характеризират със следните свойства:

а) Обекти от тези класове не могат да се създават;

б) Чистите виртуални функции, задължително трябва да бъдат предефинирани в производните класове със същите прототипи или да бъдат овявени за чисто виртуални в тях. В последния случай, класът наследник също е абстрактен.

Абстрактните класове са предназначени да служат като базови на други класове. Чрез тях се обединяват в обща структура различни йерархии.

Полиморфизмът позволява създаването на класове с различна логическа структура, които могат да включват обекти от други класове. Логическата структура на класа се реализира отделно от обектите, които се включват в него. Връзката между тях се реализира чрез указатели към контейнери, които съхраняват обекти от различни класове.

Логическата структура на контейнерните класове може да е най-различна – масив, списък, множество и т.н. Ще предложим реализацията на едносвързан списък, контейнерите на който могат да съхраняват обекти от два класа: Point2 и Point3.

Задача 167. Да се напише програма, която създава хетерогенен едносвързан списък, контейнерите на който могат да съхраняват обекти от два класа: Point2 и Point3.

```
// Program Zad167.cpp
#include <iostream.h>
#include "L-List.cpp"
typedef LList<void*> lst;
class Object
{public:
 virtual void Print()=0;
};
class lst_het: public lst, public Object
{public:
 lst_het(){}
 void Print();
};
void lst_het::Print()
{IterStart();
 elem<void*> *p = Iter();
 Object *ptr;
 while(p)
 {ptr = (Object*)(p->inf);
  ptr->Print();
  p = p->link;
 }
 cout << endl;
}
class Point2:Object
{public:
 Point2(int absc = 0, int ord = 0)
```

```
 {x = absc;
  y = ord;
 }
 void Print()
 {cout << x << ", " << y << endl;
 }
 private:
  int x, y;
 };
class Point3 : Object
{public:
 Point3(int a, int o, int b)
 {x = a;
  y = o;
  z = b;
 }
 void Print()
 {cout << x << ", " << y << ", " << z << endl;
 }
 private:
  int x, y, z;
};
void main()
{lst_het lh;
 Point2 p21(1,5), p22(2,6);
 Point3 p31(10, 20, 30), p32(11, 21, 31);
 lh.ToEnd(&p21); lh.ToEnd(&p31);
 lh.ToEnd(&p22); lh.ToEnd(&p32);
 lh.Print();
}
```

(Виртуални деструктори)

Задачи

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.