ОБЕКТНО-ОРИЕНТИРАНО ПРОГРАМИРАНЕ

Магдалина Тодорова

спец. Информационни системи,
I курс, I поток

ФМИ, СУ „Св. Климент Охридски“
2018/2019

Тема № 10

Множествено наследяване.
Преобразуване на типове.

I. Множествено наследяване.

1. *Дефиниране на производен клас*

<декларация_на_производен_клас> ::=

<заглавие>

<тяло>

<заглавие> ::=

class <име_на_производен_клас> **:**

[<атрибут_за_област> **]** <име_на_базов_клас>,

[<атрибут_за_област> **]** <име_на_базов_клас>

{ , [<атрибут_за_област> **]** <име_на_базов_клас> **}**

<тяло> ::= { <декларация_на_компонента>;

{ <декларация_на_компонента>; **}**

} **[** <списък_от_обекти> **]**;

1. *Дефиниране на производен клас*

<декларация_на_компонента> ::=

<декларация_на_конструктор> |

<декларация_на_мутатор> |

<декларация_на_функция_за_достъп> |

<декларация_на_член_данна>

<име_на_производен_клас> ::= <идентификатор>

<атрибут_за_област> ::= public | protected |

private

<име_на_базов_клас> ::= <идентификатор>

Дефиниране на производен клас

Примери.

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

Еквивалентна е на

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

Дефиниране на производен клас

Производният клас наследява компоненти на всички базови класове като видът на наследяване се определя от атрибута за област на базовия клас. Изключения има при каноничните представяния.

Правилата за наследяване, за пряк (локален, вътрешен) и външен достъп са същите като при единичното наследяване.

Дефиниране на производен клас

```
class base1
{ public: void b11();
  protected: void b12();
  private: int b13;
};
```

```
class base2
{ public: void b21();
  protected: void b22();
  private: int b23;
};
```

```
class base3
{ public: void b31();
  protected: void b32();
  private: int b33;
};
```

Дефиниране на производен клас

```
class der : public base2, base3, protected base1
{ public: void d1();
   protected: void d2();
   private: int d3;
} d;
```

```
d
b23   b33  b13  d3
 -      -    -    -
```

Дефиниране на производен клас

```
class der
{ public:
     void b21();
     void d1();
  protected:
      void b22();
      void b11();
      void b12();
      void d2();
  private:
      void b31(); void b32();
      int b23;
      int b33;
      int b13;
      int d3;
} d;
```

Дефиниране на производен клас

Канонично представяне

За член-функциите на голямата четворка на производен клас с множествено наследяване са в сила аналогични правила, като при производен клас с единично наследяване. ***В общия случай тези член-функции на основните класове не се наследяват от производния им клас.***

Конструктор на производен клас

<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$_i$>) }
{ , <член-данна>(<параметри$_i$>) }

Конструктор на производен клас

При обръщение към конструктор на производен клас последователно се изпълняват:

1) конструкторите на базовите му класове в *реда на тяхното задаване в декларацията на производния клас*, а не в инициализиращия списък на конструктора.

Ако за някой основен клас не е посочен конструктор в инициализиращия списък, изпълнява се конструкторът по подразбиране на класа, ако такъв е дефиниран, или се съобщава за грешка.

Конструктор на производен клас

2) конструкторите по подразбиране на класовете, чиито обекти са член-данни на производния клас, в случай, че в инициализиращият списък не е указано как да се инициализират.
Редът на извикване съответства на реда на деклариране на тези член-данни в тялото на производния клас;

3) тялото на конструктора на производния клас.

Конструктор на производен клас

***Възможни са*:**

а) ***В някой от основните класове не е дефиниран конструктор в т.ч. за присвояване***

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

Конструктор на производен клас

б) ***В някой от основните класове е дефиниран конструктор с параметри, от който не следва подразбиращият се конструктор***

Тогава:

✓ ако в производния клас е дефиниран конструктор, в инициализиращия му списък задължително трябва да има обръщение към конструктора с параметъри на този основен клас.

✓ ако в производния клас не е дефиниран конструктор, компилаторът ще съобщи за грешка.

Конструктор на производен клас

в) ***В някой от основните класове са дефинирани няколко конструктора в т. ч. подразбиращ се***

✓ ако в производния клас е дефиниран конструктор, в инициализиращия му списък може да не се посочи конструктор за този основен клас. Ще се използва подразбиращият се конструктор на основния клас.

Конструктор на производен клас

✓ ако в производния клас не е дефиниран конструктор, компилаторът автоматично създава за него подразбиращ се конструктор.

В този случай всички основни класове на производния клас трябва да имат конструктори по подразбиране.

Деструктор на производен клас

Всеки деструктор трябва да разруши само онези собствени компоненти, които са реализирани в динамичната памет.

Извикването на деструкторите на базовите класове и производния им клас се осъществява автоматично в следната последователност:

1) извиква се деструкторът на производния клас,

2) в обратен ред, се извикват деструкторите на класовете на обектите, които са член-данни на производния клас (ако има такива) и

3) изпълнявят се деструкторите на основните му класове, отново в обратен ред на реда на извикване на техните конструктори.

Деструктор на производен клас

Пример.

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

class base1
{ public:
      base1(int x = 0)
      { cout << "base1(" << x << ")\n";
        b1 = x;
      }
      ~base1()
      { cout << "~base1()\n";
      }
   private:
      int b1;
};
```

Деструктор на производен клас

```
class base2
{ public:
      base2(int x = 0)
      { cout << "base2(" << x << ")\n";
        b2 = x;
      }
      ~base2()
      { cout << "~base2()\n";
      }
   private:
      int b2;
};
```

Деструктор на производен клас

```
class base3
{ public:
      base3(int x = 0)
      { cout << "base3(" << x << ")\n";
        b3 = x;
      }
      ~base3()
      { cout << "~base3()\n";
      }
   private:
      int b3;
};
```

Деструктор на производен клас

```
class der : public base2, base1, protected base3
{ public:
      der(int x = 0) : base1(x), base2(x), base3(x)
      { cout << "der:\n";
        d1 = base1(1);
        d2 = base2(2);
        d3 = base3(3);
      }
      ~der()
      { cout << "~der()\n";
      }
   private:
      base1 d1;
      base2 d2;
      base3 d3;
};
```

Деструктор на производен клас

```
int main()
{ der d(5);
  return 0;
}
```

```
base2(5)
base1(5)
base3(5)
base1(0)
base2(0)
base3(0)
der:
base1(1)
~base1()
base2(2)
~base2()
base3(3)
~base3()
```

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

Деструктор на производен клас

По-добра реализация на конструктора на производния клас е следната:

```
der(int x = 0) : base1(x), base2(x), base3(x),
                 d1(1),  d2(2), d3(3)
{ cout << "der:\n";
}
```

```
base2(5)
base1(5)
base3(5)
base1(1)
base2(2)
base3(3)
```

```
der:
~der()
~base3()
~base2()
~base1()
~base3()
~base1()
~base2()
```

Конструктор за присвояване на производен клас

Дефиниране на конструктор за присвояване на производен клас

<име_на_произв._клас>::<име_на_произв._клас>
(const <име_на_производен_клас>& **p**)
<инициализиращ_списък>
{ <тяло>
}

<инициализиращ_списък> ::=
<празно> |
: <име_на_основен_клас>(**p**)
{ , <име_на_основен_клас>(**p**) **}**
{ , <член-данна>(<параметри>) **}**

p неявно се преобразува до типа на основния клас

Конструктор за присвояване на производен клас

Възможни са:

а) ***в производния клас не е дефиниран конструктор за присвояване***

Тогава компилаторът автоматично генерира за него 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.

В този случай конструкторите за присвояване (копиране) на основните класове се наследяват от производния клас.

Конструктор за присвояване на производен клас

б) ***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 член-данни на този клас става чрез указания конструктор.

Конструктор за присвояване на производен клас

Ако не е указано обръщение към конструктор за някой от основните класове, използва се ***конструкторът по подразбиране на основния клас***, ако такъв *съществува* или се съобщава за отсъствието на подходящ конструктор за този основен клас, ако в него не е дефиниран конструктор по подразбиране.

Операторна функция за присвояване на производен клас

```
<производен_клас>& <производен_клас>::operator=(const
                                        производен_клас>& p)
{ if (this != &p)
    { // дефиниране на присвояването
      // за наследените член-данни
      <основен_клас1>::operator=(p);
      <основен_клас2>::operator=(p);
      ...
      <основен_класN>::operator=(p);
```

Операторна функция за присвояване на производен клас

```
        // дефиниране на присвояването
        // за собствените член-данни
      Del();    // разрушаване на собствени член-
                // данни на подразбиращия се обект,
                // които са разположени в ДП
      Copy(p); // копиране на собствените член-
                 // данни на обекта p в съответните
                 // член-данни на подразбиращия
                 // се обект
   }
   return *this;
}
```

Операторна функция за присвояване на производен клас

Възможни са:

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

Тогава компилаторът създава такава. Тя изпълнява операторните функции за присвояване (дефинирани или генерирани от компилатора) на всички основни класове на производния клас.

Операторна функция за присвояване на производен клас

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

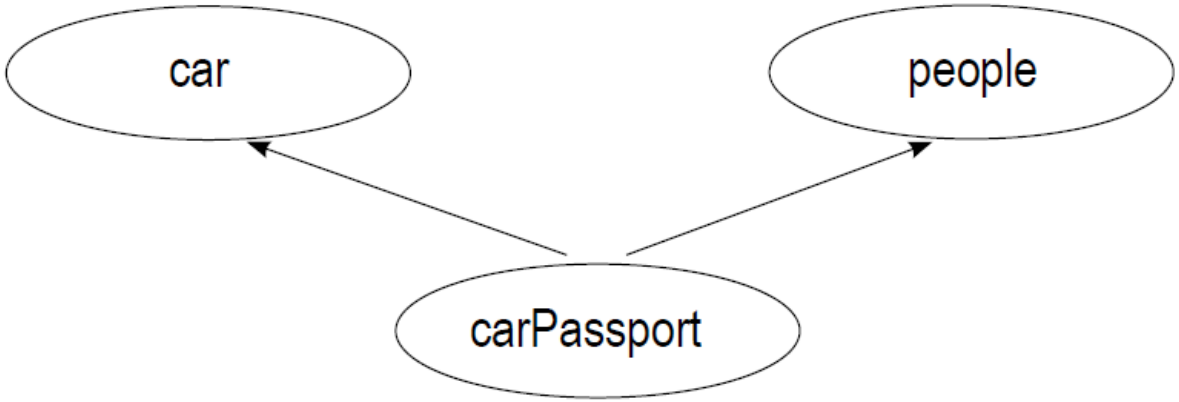
Тази функция трябва да се погрижи за присвояването на всички наследени член-данни.

Ако това не е направено явно за някой основен клас, ***стандартът на езика не уточнява как ще стане присвояването на наследените от този клас член-данни.***

Канонично представяне при множествено наследяване

Задача.

Да се реализира йерархията



Канонично представяне при множествено наследяване

в която класовете *car* и *people* определят съответно *автомобил* (по указани марка, година на производство и регистрационен номер) и *човек* (по указани име и единен граждански номер).
Класът *carPassport* (без собствени член-данни), производен на класовете *car* и *people*, определя *паспорт на автомобил*.
За всеки от класовете да се дефинира каноничното представяне.

Канонично представяне при множествено наследяване

```
#include <iostream>
#include <cstring>
#include <cassert>
using namespace std;
class car
{ public:
      car(const char* = "", unsigned int = 0, unsigned int = 0);
      ~car();
      car(const car&);
      car& operator=(const car&);
      void display() const;
   private:
      char* brand;              // марка
      unsigned year;            // година на производство
      unsigned reg_numb;        // регистрационен номер
};
```

Канонично представяне при множествено наследяване

```
car::car(const char* br, unsigned int y, unsigned int r_n)
{ brand = new char[strlen(br)+1];
   assert(brand != NULL);
   strcpy(brand, br);  // strcpy_s(brand, strlen(br) + 1, br);
   year = y;
   reg_numb = r_n;
}

car::~car()
{ cout << "~car()\n";
   delete [] brand;
}
```

Канонично представяне при множествено наследяване

```
car::car(const car& c)
{ brand = new char[strlen(c.brand)+1];
  assert(brand != NULL);
  strcpy(brand, c.brand);  // strcpy_s(brand, strlen(c.brand) + 1, c.brand);
  year = c.year;
  reg_numb = c.reg_numb;
}
```

```
car& car::operator=(const car& c)
{ if (this != &c)
   { delete [] brand;
     brand = new char[strlen(c.brand)+1];
     assert(brand != NULL);
     strcpy(brand, c.brand); // strcpy_s(brand, strlen(c.brand) + 1, c.brand);
     year = c.year;
     reg_numb = c.reg_numb;
   }
   return *this;
}
```

```
void car::display() const
{ cout << "Brand: " << brand << endl;
  cout << "Year: " << year << endl;
  cout << "Reg. Number: " << reg_numb
       << endl;
}
```

```
class people
{ public:
      people(const char * = "", const char * = "");
      ~people();
      people(const people&);
      people& operator=(const people& p);
      void display() const;
   private:
      char* name;          // име
      char* ucn;           // ЕГН
};
```

Канонично представяне при множествено наследяване

```
people::people(const char *na, const char *uc)
{ name = new char[strlen(na)+1];
   assert(name != NULL);
   strcpy(name, na);  // strcpy_s(name, strlen(na) + 1, na);
   ucn = new char[strlen(uc)+1];
   assert(ucn != NULL);
   strcpy(ucn, uc);    // strcpy_s(ucn, strlen(uc) + 1, uc);
}

people::~people()
{ cout << "~people()\n";
   delete [] name;
   delete [] ucn;
}
```

Канонично представяне при множествено наследяване

```
people::people(const people& p)
{ name = new char[strlen(p.name)+1];
  assert(name != NULL);
  strcpy(name, p.name);  // strcpy_s(name, strlen(p.name) + 1, p.name);
  ucn = new char[strlen(p.ucn)+1];
  assert(ucn != NULL);
  strcpy(ucn, p.ucn);      // strcpy_s(ucn, strlen(p.ucn) + 1, p.ucn);
}
```

Канонично представяне при множествено наследяване

```
people& people::operator=(const people& p)
{ if (this != &p)
   { delete[] name;
     delete [] ucn;
     name = new char[strlen(p.name)+1];
     assert(name != NULL);
     strcpy(name, p.name); // strcpy_s(name, strlen(p.name) + 1, p.name);
     ucn = new char[strlen(p.ucn)+1];
     assert(ucn != NULL);
     strcpy(ucn, p.ucn);    // strcpy_s(ucn, strlen(p.ucn) + 1, p.ucn);
   }
   return *this;
}
```

Канонично представяне при множествено наследяване

```
void people::display() const
{ cout << "Name: " << name << endl;
  cout << "UCN: " << ucn << endl;
}

class carPassport : public car, public people
{ public:
      carPassport(const char* = "", unsigned int = 0,
                  unsigned int = 0, const char* = "",
                  const char* = "");
      ~carPassport();
      carPassport(const carPassport&);
      carPassport& operator=(const carPassport&);
      void display() const;
};
```

Канонично представяне при множествено наследяване

```
carPassport::carPassport(const char* br, unsigned int y,
        unsigned int reg_nu, const char* na, const char *uc) :
        car(br, y, reg_nu), people(na, uc)
{}

carPassport::~carPassport()
{ cout << "~carPassport()\n";
}

carPassport::carPassport(const carPassport& cp) :
                                        car(cp), people(cp)
{}
```

Канонично представяне при множествено наследяване

```
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();
}
```

Канонично представяне при множествено наследяване

```
int main()
{ carPassport x("FORD FIESTA", 2006, 4444,
                    "Petar Popov", "8008080000");
   x.display();
   carPassport y("MERCEDES-BENZ", 2010, 8888,
                    "Inna Ivanova", "8807071111");
    y = x;
    y.display();
    return 0;
}
```

Канонично представяне при множествено наследяване

Резултат:

```
Brand: FORD FIESTA
Year: 2006
Reg. Number: 4444
Name: Petar Popov
UCN: 8008080000
Brand: FORD FIESTA
Year: 2006
Reg. Number: 4444
Name: Petar Popov
UCN: 8008080000
~carPassport()
~people()
~car()
~carPassport()
~people()
~car()
```

II. Преобразуване на типове

1. *Преобразуване – основни елементи*

Ако основният клас, който се наследява от производния клас е с атрибут за област ***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.

1. *Преобразуване – основни елементи*

Замяната „производен с основен“ е безопасна, докато замяната „основен с производен“ може да предизвика проблеми.

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

1. *Преобразуване – основни елементи*

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

1. Преобразуване – основни елементи

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

2. Преобразуване от производен в 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 при атрибут ***public*** производният клас наследява всички свойства на базовия клас и може да бъде използван вместо него.

2. Преобразуване от производен в основен

Пример.

der d;	d.f_der();	class der: d: 6 b: 0
base x = d;	x.f();	b: 1
der &d1 = d;	d1.f_der();	class der: d: 7 b: 0
base &y = d1; *	y.f();	b: 1
der *d2 = &d;	d2->f_der();	class der: d: 8 b: 1
base *z = d2;	z->f();	b: 2

*** Забележка:** Вместо d1 може да се използва d.

3. Преобразуване от основен в производен

Осъществява се (ако е възможно) чрез ***явно преобразуване.***

а) ***Инициализиране на обект, указател към обект и псевдоним на обект на производен клас със съответно обект, указател към обект и псевдоним на обект на основния му клас***

```
base x;
der y = x; // Допустимо ли е?
```

3. Преобразуване от основен в производен

Операцията е опасна, тъй като собствените компоненти на обекта *y* ще останат неинициализирани и *опитът за използването (промяната) им може да доведе до сериозни последици*. Затова **някои** реализации на езика, не реализират това преобразуване.

Други реализации го допускат, но чрез явно преобразуване на *x* в обект на клас *der*, т.е.

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

3. Преобразуване от основен в производен

Подобна е ситуацията при използване на указатели към обекти и псевдоними на обекти.

Пример.

```
base x;
base *pb = &x;
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*.

3. Преобразуване от основен в производен

```
base x;
base *pb = &x;
der* pd = (der*) pb;
```

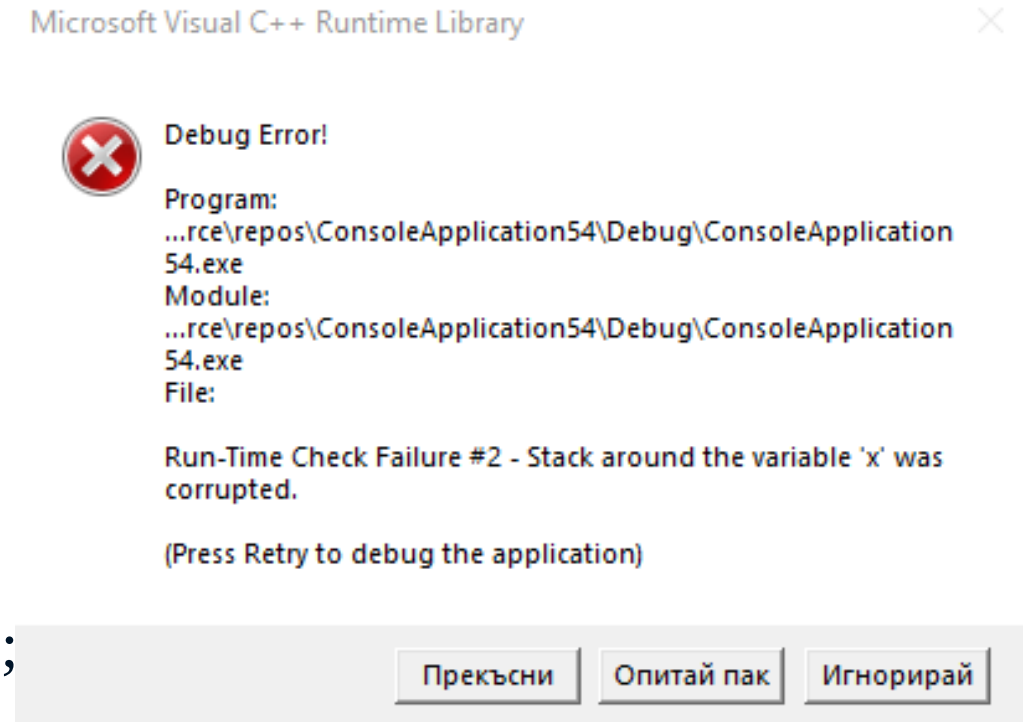
Областта в паметта, свързана с указателя *pd*, няма собствени компоненти на класа *der*. Опитът за използването им може да предизвика сериозни проблеми, тъй като ще се използва памет, която е определена за други цели.

Някои реализации на езика не извършват това преобразуване. Други го извършват.

3. Преобразуване от основен в производен

Пример.

```
base x;
base *pb = &x;
pb->f();
der *pd = (der*) pb;
pd->f_der();
cout << pd->get_b() << endl;
```



неопределено

***Резултат*:**

```
b: 1
class der: d: 7011897 b: 1
1
```

3. Преобразуване от основен в производен

Пример.

```
base x;
base & psx = x;
psx.f();
der & psd = (der&) psx;  // psd не е псевдоним на обект на der
psd.f_der();
cout << psd.get_b() << endl;
```

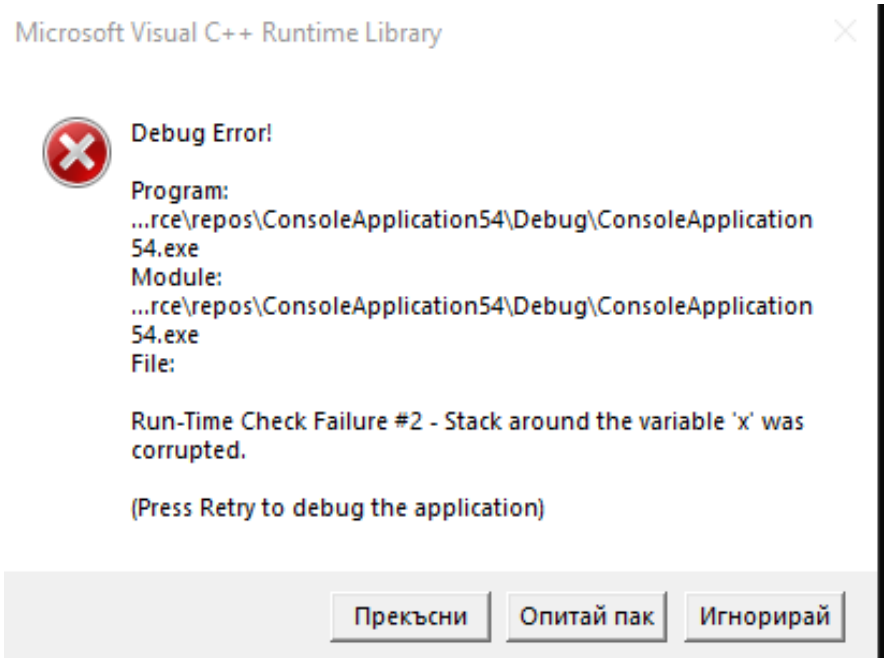
неопределено

***Резултат*:**

b: 1

class der: d: **7098997** b: 1

1



3. Преобразуване от основен в производен

б) ***Достъп до собствени компоненти на производен клас (намиращи се в public секция) чрез обект, указател към обект или псевдоним на обект на основния му клас***

Такъв достъп чрез обект не е възможен.

Непряк достъп е допустим и се осъществява чрез *указател* към обект или чрез *псевдоним* на обект и *преобразувания*.

3. Преобразуване от основен в производен

Пример. Във фрагмента

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

указателите *pb* и *pd* сочат обекта *y* на класа *der*.

```
pd->f_der();        // е допустимо
pb->f_der();        // е недопустимо
```

Трябва да се приложи **явно** преобразуване от вида:

((der*) pb) -> f_der();

Забележка: -> е с по-висок приоритет от С-преобразуването (<type>).

3. Преобразуване от основен в производен

Пример. Във фрагмента

```
der y;
der & psd = y;
base & psb = psd;      // неявно преобразуване

psd.f_der();           // е допустимо
psb.f_der();           // не е допустимо
```

Трябва да се приложи **явно** преобразуване от вида:

((der&) psb).f_der();

Забележка: **.** е с по-висок приоритет от С-преобразуването (<type>).

4. Присвояване на указатели към методи на класове

а) ***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 с тип на резултата *void*. За да се използва този указател е необходимо да се свърже с конкретен обект.

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

В резултат се осъществява обръщение към метода *f* на класа *base* чрез указателя *pb* към него и се получава:

b: 1

*Приоритет на оператора .**

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a--	Suffix/postfix increment and decrement	
	type() *type*{}	Functional cast	
	a()	Function call	
	a[]	Subscript	
	. ->	Member access	
3	++a --a	Prefix increment and decrement	Right-to-left
	+a -a	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	*a	Indirection (dereference)	
	&a	Address-of	
	sizeof	Size-of[note 1]	
	new new[]	Dynamic memory allocation	
	delete delete[]	Dynamic memory deallocation	
4	.* ->*	Pointer-to-member	Left-to-right

4. Присвояване на указатели към методи на класове

Чрез дефиницията

void (der::*pd)();

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

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

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

Отговорът е положителен.

4. Присвояване на указатели към методи на класове

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

Това присвояване 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.

4. Присвояване на указатели към методи на класове

Допустимо е присвояването

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

или

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

но използването му може да доведе до грешка или до нееднозначност.