НАСЛЕДЯВАНЕ

гл.ас., д-р. Нора Ангелова

НАСЛЕДЯВАНЕ

Наследяване

Наследяването е начин за създаване на нови класове чрез използване на компоненти и поведение на съществуващи класове.

НАСЛЕДЯВАНЕ

При създаване на нов клас, който има общи компоненти и поведение с вече дефиниран клас, вместо да дефинира повторно тези компоненти и поведение, програмистът може да определи новия клас като клас наследник на вече дефинирания.

- Базов и производен клас

Дефинираният клас се нарича базов или основен.

Новосъздаденият клас се нарича производен.

НАСЛЕДЯВАНЕ

- Единично наследавяне

НАСЛЕДЯВАНЕ

- Множествено наследяване

НАСЛЕДЯВАНЕ

- Множеството от компонентите на производен клас се състои от компонентите на неговите базови класове и компонентите, декларирани в самия производен клас.

- Наследяване - механизъм, чрез който производният клас получава компонентите на базовия.

НАСЛЕДЯВАНЕ

Процесът на наследяване се изразява в следното:

- Наследяват се член-данните и методите на основните класове.
- Получава се достъп до някои от наследените компоненти на основните класове.
- Производният клас „познава“ реализациите само на основните класове, от които произлиза.
- Производният клас може да е основен за други класове.

НАСЛЕДЯВАНЕ

Производният клас може да дефинира допълнително:

- свои член-данни;
- свои член-функции (методи), аналогични на тези на основните класове, а също и нови.

Дефинираните в производния клас член-данни и член-функции се наричат **собствени**.

НАСЛЕДЯВАНЕ

- Дефиниция на производен клас

<заглавие> ::=
class <име_на_производен_клас> :
[<атрибут_за_област>]$_{опц}$ <име_на_базов_клас>
{ , [<атрибут_за_област>] <име_на_базов_клас> }$_{опц}$

<име_на_производен_клас> ::= <идентификатор>
<атрибут_за_област> ::= public | private | protected
<име_на_базов_клас> ::= <идентификатор>

Наследяване

- Дефиниция на производен клас

Пример:

- Единично наследяване

```
class Point3D: public Point2 {
  // …
};
```

- Множествено наследяване

```
class CarPassport: private Car, private Passport {
  // …
};
```

Наследяване

- Дефиниция на производен клас

Атрибутът за достъп по подразбиране е private.

Пример:

```
class CarPassport: Car, Passport {
  // …
};
```

```
class CarPassport: private Car, private Passport {
  // …
};
```

НАСЛЕДЯВАНЕ

- Дефиниция на производен клас

Пример:

```
class CarPassport: public Car, Passport {
  // …
};
```

```
class CarPassport: public Car, private Passport {
  // …
};
```

НАСЛЕДЯВАНЕ

- Дефиниция на производен клас

Пример:

```
class CarPassport: Car, public Passport {
  // …
};
```

```
class CarPassport: private Car, public Passport {
  // …
};
```

НАСЛЕДЯВАНЕ

- **Директни основни класове**

Директните основни класове се изброяват в заглавието на производния клас, предшествани от двоеточие (:).

Пример:

Класът `Car` е директен основен клас на класа `CarPassport`.

- **Индиректни основни класове**

Не се изброяват в заглавието на производните класове, но се наследяват от две или повече по-високи нива.

НАСЛЕДЯВАНЕ

- Достъп до наследените компоненти

Атрибут public

Базов		Производен
public		public
private		private
protected		protected

Атрибут private

Базов		Производен
public		private
private		private
protected		private

НАСЛЕДЯВАНЕ

- Достъп до наследените компоненти

Атрибут protected

Базов		Производен
public		protected
private		private
protected		protected

НАСЛЕДЯВАНЕ

- Достъп до наследените компоненти

Атрибут public

public
private
protected

public
private
protected

Пример:

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

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

Резултат

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

НАСЛЕДЯВАНЕ

- Достъп до наследените компоненти

Атрибут private

public		private
private		private
protected		private

Пример:

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

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

Резултат

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

НАСЛЕДЯВАНЕ

- Достъп до наследените компоненти

Атрибут protected

public		protected
private		private
protected		protected

Пример:

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

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

Резултат

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

НАСЛЕДЯВАНЕ

- Достъп до наследените компоненти

Наследените компоненти се различават от собствените компоненти на производния клас по правата за достъп.

Както при класовете без наследяване,
собствените компоненти на **производния** клас имат пряк достъп помежду си.

Собствените компоненти на производния клас имат пряк достъп до компонентите, декларирани като public и protected в основния му клас, но **нямат** пряк достъп до декларираните като private компоненти на основния клас.

Достъпът до **private** компонентите на базовия клас може да се извърши чрез неговия **интерфейс**.

НАСЛЕДЯВАНЕ

- Пряк достъп (ПД) на член-функции на производен клас до компонентите на базовия му клас.
- Външен достъп (ВД) на обект на производен клас до компонентите на базовия му клас.

```
class derivativeClassName : <атрибут_за_достъп> baseClass {
  … // ПД до BaseClass
};
derivativeClassName obj; // ВД до BaseClass obj.
```

компонента на базов клас	производен клас - атрибут **public** на базовия му клас		производен клас - атрибут **private** на базовия му клас		производен клас - атрибут **protected** на базовия му клас	
	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

Предефиниране на компоненти

- Проблем

Производният клас може да наследи член-функция, която не трябва да има.

- Решение

В производния клас се предефинира (дефинира се повторно) член-функцията с подходяща реализация.

Предефиниране на компоненти

- Проблем

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 собствената на производния клас компонента, т.е. името на собствената компонента е с по-висок приоритет от това на наследената.

- Решение

За да се изпълни наследена компонента се указва пълното й име, т.е.

<име_на_основен_клас>::<компонента>

ПРЕДЕФИНИРАНЕ НА КОМПОНЕНТИ

```
class base {
  public:
    void init (int x) { data = x; }
    void display() const {
       cout << " class base: data= " << data << endl;
    }
  protected:
    int data;
};
class der : public base {
  public:
    void init (int x) {
      data = x;
      base::data = x + 5;
    }
    void display() const {
      cout << " class der: data = " << data;
      cout << " base::data = " << base::data << endl;
    }
  protected:
    int data;
};
```

```
int main() {
  base b;
  der d;
  b.init(5);        // base::init
  d.init(10);       // der::init
  b.display();      // base::display
  d.display();      // der::display
  d.base::init(20); // base::init
  d.base::display();
                    // base::display

  return 0;
}
```

Единично наследяване. Конструктори, деструктор и операторна функция за присвояване на производен клас

ЗАБЕЛЕЖКА

- Обикновените конструктори, конструкторът за присвояване, операторната функция за присвояване и деструкторът са методи, за които не важат правилата за достъп при наследяване.
- Тези методи на основния клас (с някой изключения) НЕ се наследяват от производния клас.

КОНСТРУКТОР

- Конструкторите на производния клас инициализират само собствените член-данни на класа.
- Наследените член-данни на производния клас се инициализират от конструктор на основния му клас.

Реализация:
Това се осъществява като в дефиницията на конструктора на производния клас се укаже обръщение към съответен конструктор на основния клас.

КОНСТРУКТОР

```
<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>)
{ , <член-данна>(<параметри>) }
```

ЗАБЕЛЕЖКА

- При единичното наследяване инициализиращият списък на конструктора на производния клас може да съдържа не повече от едно обръщение към конструктор на основен клас.

- Ако инициализиращият списък не съдържа обръщение към конструктор на основния клас, чрез което да укаже как да се инициализира наследената част, в базовия клас трябва да е дефиниран конструктор по подразбиране.

- Освен обръщение към конструктор на базовия клас, инициализиращият списък на конструктора на производния клас може да съдържа инициализация на собствени за производния клас член-данни.

- Обръщението към конструктора на основния клас се записва в дефиницията на конструктора на производния клас, а не в неговата декларация в тялото на производния клас.

ПРИМЕР

```
class base {
 protected:
   int a2;
 private:
   int a1;
 public:
   // конструктор по подразбиране
   base() {
     a1 = 0;
     a2 = 0;
   }
   // конструктор с един параметър
   base(int x) {
     a1 = x;
   }
   // конструктор с 2 параметъра
   base(int x, int y) {
     a1 = x;
     a2 = y;
   }
   void a3() const {
     cout << "a1: " << a1 << endl
          << "a2: " << a2 << endl;
   }
};
```

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

   void d3() const {
     cout << "d1: " << d1 << endl
          << "d2: " << d2 << endl
          << "a2: " << a2 << endl;
     cout << "a3():" << endl;
     a3();
   }
};
```

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

Резултат:

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

ПРИМЕР

```
class base {
protected:
  int a2;
private:
  int a1;
public:
  // конструктор по подразбиране
  base() {
    a1 = 0;
    a2 = 0;
  }
  // конструктор с един параметър
  base(int x) {
    a1 = x;
  }
  // конструктор с 2 параметъра
  base(int x, int y) {
    a1 = x;
    a2 = y;
  }
  void a3() const {
    cout << "a1: " << a1 << endl
         << "a2: " << a2 << endl;
  }
};
```

```
class der : public base {
 protected:
   int d2;
 private:
   int d1;
 public:
   der(int x, int y, int z, int t) : base() {
     d1 = z;
     d2 = t;
   }
  // Наследените компоненти се
  // инициализират от подразбиращия се
  // конструктор

   void d3() const {
     cout << "d1: " << d1 << endl
          << "d2: " << d2 << endl
          << "a2: " << a2 << endl;
     cout << "a3():" << endl;
     a3();
   }
};
```

ПРИМЕР

```
class base {
protected:
  int a2;
private:
  int a1;
public:
  // конструктор по подразбиране
  base() {
    a1 = 0;
    a2 = 0;
  }
  // конструктор с един параметър
  base(int x) {
    a1 = x;
  }
  // конструктор с 2 параметъра
  base(int x, int y) {
    a1 = x;
    a2 = y;
  }
  void a3() const {
    cout << "a1: " << a1 << endl
         << "a2: " << a2 << endl;
  }
};
```

```
class der : public base {
 protected:
   int d2;
 private:
   int d1;
 public:
   der(int x, int y, int z, int t) {
     d1 = z;
     d2 = t;
   }
  // Наследените компоненти се
  // инициализират от подразбиращия се
  // конструктор

   void d3() const {
     cout << "d1: " << d1 << endl
          << "d2: " << d2 << endl
          << "a2: " << a2 << endl;
     cout << "a3():" << endl;
     a3();
   }
};
```

ПРИМЕР

```
class base {
protected:
  int a2;
private:
  int a1;
public:
  // конструктор по подразбиране
  base() {
    a1 = 0;
    a2 = 0;
  }
  // конструктор с един параметър
  base(int x) {
    a1 = x;
  }
  // конструктор с 2 параметъра
  base(int x, int y) {
    a1 = x;
    a2 = y;
  }
  void a3() const {
    cout << "a1: " << a1 << endl
         << "a2: " << a2 << endl;
  }
};
```

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

ГРЕШКА - не повече от 1 извикване към конструктор на базов клас

```
  void d3() const {
    cout << "d1: " << d1 << endl
         << "d2: " << d2 << endl
         << "a2: " << a2 << endl;
    cout << "a3():" << endl;
    a3();
  }
};
```

ЗАБЕЛЕЖКА

- Ако производният клас има собствени член-данни, които са обекти на класове и в инициализиращия списък на конструктора не е указано как те да се инициализират, техните конструктори по подразбиране се извикват след изпълнението на обръщението към конструктора на основния клас от инициализиращия списък и преди изпълнението на операторите в тялото на конструктора на производния клас.
- Редът на изпълнението им съвпада с реда на член-данните обекти в производния клас.

ПРИМЕР

```
class base {
 protected: int a2;
 private: int a1;
 public:
  base() {
    cout << "constructor base() \n";
    a1 = a2 = 0;
  }

  base(int x, int y) {
    cout << "constructor base("
         << x << ", " << y << ")\n";
    a1 = x;
    a2 = y;
  }

  void a3() const {
    cout << "a1: " << a1 << endl
         << "a2: " << a2 << endl;
  }
};
```

```
class der : public base {
 protected: base d2;
 private: base d1;
 public:
  der(int x, int y) : base(x, y) {
    cout << "constructor der\n";
  }

  void d3() const {
    d1.a3();
    d2.a3();
    cout << "a2: " << a2 << endl;
    cout << "a3():" << endl;
    a3();
  }
};
int main() {
  der x(1, 2);
  x.d3();
  return 0;
}
```

```
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
```

ПРИМЕР

```
class base {
 protected: int a2;
 private: int a1;
 public:
  base() {
    cout << "constructor base() \n";
    a1 = a2 = 0;
  }

  base(int x, int y) {
    cout << "constructor base("
         << x << ", " << y << ")\n";
    a1 = x;
    a2 = y;
  }

  void a3() const {
    cout << "a1: " << a1 << endl
         << "a2: " << a2 << endl;
  }
};
```

```
class der : public base {
 protected: base d2;
 private: base d1;
 public:
  der(int x, int y) : base(x, y) {
    cout << "constructor der\n";
    d1 = base(15, 25);
    d2 = base(35, 45);
  }

  void d3() const {
    d1.a3();
    d2.a3();
    cout << "a2: " << a2 << endl;
    cout << "a3():" << endl;
    a3();
  }
};

int main() {
  der x(1, 2);
  x.d3();
  return 0;
}
```

```
constrictor base(1, 2)
constrictor base()
constrictor base()
constrictor der
constructor base(15, 25)
constructor base(35, 45)
a1: 15
a2: 25
a1: 35
a2: 45
a2: 2
a3():
a1: 1
a2: 2
```

ПРИМЕР

```
class base {
 protected: int a2;
 private: int a1;
 public:
  base() {
    cout << "constructor base() \n";
    a1 = a2 = 0;
  }

  base(int x, int y) {
    cout << "constructor base("
         << x << ", " << y << ")\n";
    a1 = x;
    a2 = y;
  }

  void a3() const {
    cout << "a1: " << a1 << endl
         << "a2: " << a2 << endl;
  }
};
```

```
class der : public base {
 protected: base d2;
 private: base d1;
 public:
  // Избягва се двукратното инициализиране
  der(int x, int y) : base(x, y),
  d1(15, 25), d2(35, 45) {
    cout << "constructor der\n";
  }

  void d3() const {
    d1.a3();
    d2.a3();
    cout << "a2: " << a2 << endl;
    cout << "a3():" << endl;
    a3();
  }
};
int main() {
  der x(1, 2);
  x.d3();
  return 0;
}
```

```
constrictor base(1, 2)
constructor base(35, 45)
constructor base(15, 25)
constrictor der
a1: 15
a2: 25
a1: 35
a2: 45
a2: 2
a3():
a1: 1
a2: 2
```

Случаи

- В основния клас не е дефиниран конструктор в т.ч. конструктор за копиране

В този случай в инициализиращия списък на конструктор(ите) на производния клас не трябва да се задава инициализация за наследените от основния клас член-данни. Наследената част на производния клас остава неинициализирана.

Случаи

- В основния клас е дефиниран само един конструктор с параметри, който не е подразбиращият се

Възможни са:

а) в производния клас е дефиниран конструктор

В този случай в инициализиращия списък на конструктора на производния клас задължително трябва да има обръщение към конструктора с параметри на основния клас. Изпълнява се по начина, описан по-горе.

б) в производния клас не е дефиниран конструктор

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

СЛУЧАИ

- В основния клас са дефинирани няколко конструктора в т.ч. подразбиращ се конструктор

Възможни са:

а) в производния клас е дефиниран конструктор

Тогава в инициализиращия списък на конструктора на производния клас може да се посочи, но може и да не се посочи конструктор на основния клас. Ако не е посочен, компилаторът се обръща към конструктора по подразбиране на основния клас.

б) в производния клас не е дефиниран конструктор

В този случай компилаторът автоматично създава конструктор по подразбиране за производния клас. Последният активира и изпълнява конструктора по подразбиране на основния клас.

Собствените член-данни на производния клас остават неопределени.

ДЕСТРУКТОР

- Деструкторът на производен клас трябва да разруши само онези собствени на производния клас член-данни, които са разположени в динамичната памет.

  Деструкторите на производен клас и на неговия основен клас се изпълняват автоматично в ред, обратен на реда на изпълнението на техните конструктори.

  Най-напред се изпълнява деструкторът на производния клас, след това се изпълнява деструкторът на основния му клас.

Единично наследяване. Конструктори, деструктор, конструктор за копиране и операторна функция за присвояване на производен клас

Конструктор за копиране и операторна функция за присвояване

- В общия случай, производният клас **НЕ** наследява от основния си клас конструктора за копиране и оператора за присвояване.

Конструктор за копиране

- Конструкторът за копиране на производния клас инициализира собствените член-данни на класа.
- Конструкторът за копиране (или друг конструктор) на основния клас инициализира наследените член-данни.

- Конструкторът за присвояване на производен клас се дефинира по аналогичен начин като обикновените конструктори на производни класове.

```
<име_на_клас>::<име_на_клас>(const <име_на_клас>&)
<инициализиращ_списък> {
  <тяло>
}
```

КОНСТРУКТОР ЗА КОПИРАНЕ

- В производния клас НЕ е дефиниран конструктор за копиране

Възможни са:

а) в основния клас е дефиниран конструктор за копиране

В този случай компилаторът генерира конструктор за копиране на производния клас, който преди да се изпълни, активира и изпълнява конструктора за присвояване на основния клас.

В случая се казва, че конструкторът за присвояване на основния клас се наследява от производния клас.

б) в основния клас не е дефиниран конструктор за копиране

В този случай в основния и в производния му клас се генерират конструктори за копиране.

Конструкторът за копиране на производния клас активира конструктора за копиране на основния клас.

КОНСТРУКТОР ЗА КОПИРАНЕ

- В производния клас E дефиниран конструктор за копиране

Дефиницията на конструктора за копиране на производния клас определя как точно ще се инициализира наследената част.

В неговия инициализиращ списък може да има или да няма обръщение към конструктор (за копиране или обикновен) на основния му клас.

Препоръчва се в инициализиращия списък на производния клас да има обръщение към конструктора за копиране на основния клас, ако такъв е дефиниран.

Забележка:

Ако не е указано обръщение към конструктор на основния клас, инициализирането на наследените членове се осъществява от подразбиращия се конструктор на основния клас. Ако основният клас няма подразбиращ се конструктор, се съобщава за отсъствието на подходящ конструктор.

Операторна функция за присвояване

- Операторната функция за присвояване на производен клас трябва да указва как да се осъществи присвояването както на собствените, така и на наследените си член-данни.

- За разлика от конструкторите на производния клас тя прави това в тялото си, т.е. не поддържа инициализиращ списък.

ОПЕРАТОРНА ФУНКЦИЯ ЗА ПРИСВОЯВАНЕ

```
<производен_клас>& <производен_клас>::operator=
(<производен_клас> const & p) {
 if (this != &p) {
  // Дефиниране на присвояването за наследените член-данни
  <основен_клас>::operator=(p);

  // Дефиниране на присвояването за собствените член-данни
  del(); // разрушаване на онези собствени
  // член-данни на подразбиращия
  // се обект, които са разположени в ДП

  copy(p); // копиране на собствените член-данни
  // на p в съответните член-данни на
  // подразбиращия се обект
 }
 return *this;
}
```

Операторна функция за присвояване

- 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 (дефинираната или подразбиращата се), чрез която инициализира наследената част, след това инициализира чрез присвояване и собствените член-данни на производния клас.

Затова в този случай се казва, че операторът за присвояване на основния клас се наследява.

Операторна функция за присвояване

- В производния клас E е дефинирана операторна функция за присвояване

Тази член-функция трябва да се погрижи за присвояването на наследените компоненти. В тялото на нейната дефиниция трябва 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 операторът за присвояване на основния клас не се наследява.

ЗАДАЧА

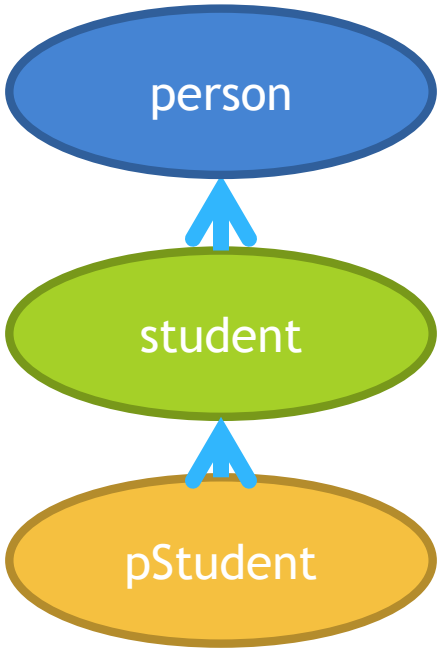
Да се дефинират класовете person, student и pStudent

person има информация за ЕГН и име.

student е човек, който има член-данна facNumb от тип char* и addr от тип char*, определяща адреса на студент.

В класа pStudent е клас за работещ студент, който има член-данна workplace от тип char*, определяща месторабота на студент от платена форма на обучение и такса.

Да се напишат функции за извеждане.



ЗАДАЧА

```
class person {
 public:
   // канонично представяне
   person(char* = "", char* = "");
   ~person();
   person(person const &);
   person& operator=(person const & p);
   // член-функция за извеждане
   void printPerson() const;
 private:
   char* name; // име
   char* ucn; // ЕГН
   // помощни член-функции за копиране и изтриване
   void copyPerson(char*, char*);
   void delPerson();
};
```

ЗАДАЧА

```
void person::copyPerson(char* nameStr, char* ucnStr) {
  name = new char[strlen(nameStr)+1];
  assert(name != NULL);
  strcpy(name, nameStr);
  ucn = new char[strlen(ucnStr)+1];
  assert(ucn != NULL);
  strcpy(ucn, ucnStr);
}

void person::delPerson() {
  delete [] name;
  delete [] ucn;
}
```

ЗАДАЧА

```
void person::copyPerson(char* nameStr, char* ucnStr) {
  name = new char[strlen(nameStr)+1];
  assert(name != NULL);
  strcpy(name, nameStr);
  ucn = new char[strlen(ucnStr)+1];
  assert(ucn != NULL);
  strcpy(ucn, ucnStr);
}

void person::delPerson() {
  delete [] name;
  delete [] ucn;
}
```

Unlike strncpy and strcpy_s, strcpy doesn't do any length checking of the destination buffer which may cause stack overflow allowing an attacker to execute malicious code if exploited or just crach your application.

Задача

```
person::person(char* nameStr, char* ucnStr) {
  copyPerson(nameStr, ucnStr);
}

person::~person() {
  delPerson();
}

person::person(person const & p) {
  copyPerson(p.name, p.ucn);
}

person& person::operator=(person const & p) {
  if (this != &p) {
    delPerson();
    copyPerson(p.name, p.ucn);
  }
  return *this;
}
```

ЗАДАЧА

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

ЗАДАЧА

```
class student : public person {
 public:
  // канонично представяне
  student(char* = "", char* = "", char* = "", char* = "");
  ~student();
  student(student const &);
  student& operator=(student const &);
  // член-функция за извеждане
  void printStudent() const;
 private:
  char* facNumb; // факултетен номер
  char* addr;    // адрес
  // помощни член-функции
  // за копиране и изтриване
  void copyStudent(char*, char*);
  void delStudent();
};
```

ЗАДАЧА

```
void student::copyStudent(char* fNumb, char* addrStr) {
  facNumb = new char[strlen(fNumb)+1];
  assert(facNumb != NULL);
  strcpy(facNumb, fNumb);

  addr = new char[strlen(addrStr)+1];
  assert(addr != NULL);
  strcpy(addr, addrStr);
}

void student::delStudent() {
  delete [] facNumb;
  delete [] addr;
}
```

ЗАДАЧА

```
student::student(char* nameStr, char* ucnStr, char* fNumb, char*
addrStr) : person(nameStr, ucnStr) {
  copyStudent(fNumb, addrStr);
}

student::~student() {
  delStudent();
}

student::student(student const & st) : person(st) {
  copyStudent(st.facNumb, st.addr);
}



student& student::operator=(student const & st) {
  if (this != &st) {
    person::operator=(st);
    delStudent();
    copyStudent(st.facNumb, st.addr);
  }
  return *this;
}
```

ЗАДАЧА

```
void student::printStudent() const {
  printPerson();
  cout << "Fac. nomer: " << facNumb << endl;
  cout << "Address: " << addr << endl;
}
```

ЗАДАЧА

```
class pStudent : public student {
 public:
  // канонично представяне
  pStudent(char* = "", char* = "", char* = "",
char* = "", double = 0, char* = "");
  ~pStudent();
  pStudent(pStudent const &);
  pStudent& operator=(pStudent const &);
  // член-функция за извеждане
  void printPStudent() const;
 private:
  double fee;       // такса
  char* workplace; // месторабота
  // помощни член-функции за копиране и изтриване
  void copyPStudent(double, char*);
  void delPStudent();
};
```

ЗАДАЧА

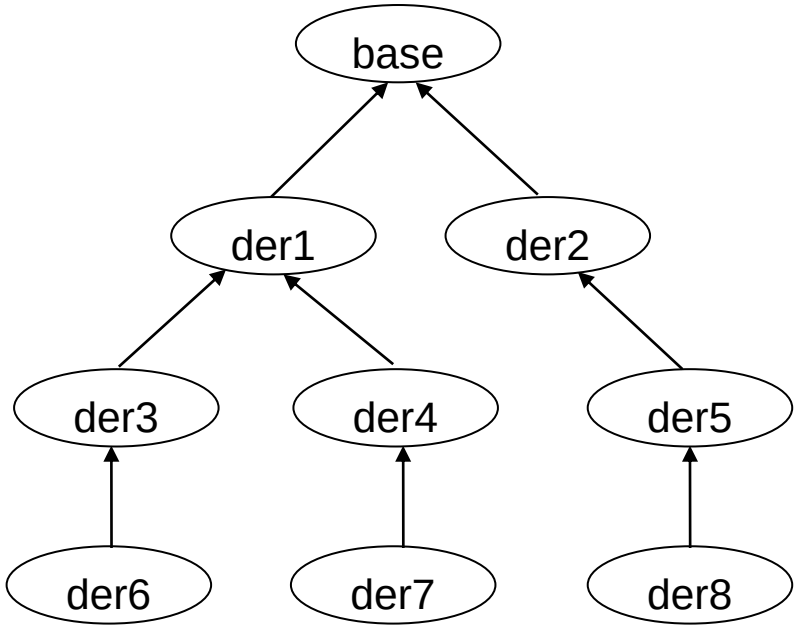
```
void pStudent::copyPStudent(double feeData, char* workplaceStr) {
  fee = feeData;
  workplace = new char[strlen(workplaceStr)+1];
  assert(workplace!= NULL);
  strcpy(workplace, workplaceStr);
}

void pStudent::delPStudent() {
  delete [] workplace;
}
```

ЗАДАЧА

```
pStudent::pStudent(char* nameStr, char* ucnStr, char* fNumb, char*
addrStr, double feeData, char* workplaceStr)
: student(nameStr, ucnStr, fNumb, addrStr) {
  copyPStudent(feeData, workplaceStr);
}

pStudent::~pStudent() {
  delPStudent();
}

pStudent::pStudent(pStudent const& ps) : student(ps) {
  copyPStudent(ps.fee, ps.workplace);
}

pStudent& pStudent::operator=(pStudent const & ps) {
  if (this != &ps) {
    student::operator=(ps);
    delPStudent();
    copyPStudent(ps.fee, ps.workplace);
  }
  return *this;
}
```

ЗАДАЧА

```
void pStudent::printPStudent() const {
  printStudent();
  cout << "Fee: " << fee << endl
       << " Workplace:" << workplace<< endl;
}
```




```
class base {
   public:    int b1;
   protected: int b2;
   private:   int b3;
} b;

class der1 : protected base {
   public:    int d11;
   protected: int d12;
   private:   int d13;
} d1;

class der2 : public base {
   public:    int d21;
   protected: int d22;
   private:   int d23;
} d2;
```

```
class der3 : public der1 {
   public:    int d31;
   protected: int d32;
   private:   int d33;
} d3;

class der4 : der1 {
   public:    int d41;
   protected: int d42;
   private:   int d43;
} d4;

class der6 : protected der3 {
   public:    int d61;
   protected: int d62;
   private:   int d63;
} d6;

class der7 : public der4 {
   public:    int d71;
   protected: int d72;
   private:   int d73;
} d7;

class der5 : protected der2 {
   public:    int d51;
   protected: int d52;
   private:   int d53;
} d5;

class der8 : public der5 {
   public:    int d81;
   protected: int d82;
   private:   int d83;
} d8;
```

ПД & ВД

Решение:

ВД

b -
d1 -
d2 -
d3 -
d4 -
d5 -
d6 -
d7 -
d8 -

ПД

base -
der1 -
der2 -
der3 -
der4 -
der5 -
der6 -
der7 -
der8 -

КРАЙ