ОБЕКТНО-ОРИЕНТИРАНО ПРОГРАМИРАНЕ

Магдалина Тодорова

спец. Компютърни науки, I курс, I поток

ФМИ, СУ „Св. Климент Охридски“
2017/2018

Тема № 15

ВИРТУАЛНИ ФУНКЦИИ. ПОЛИМОРФИЗЪМ. АБСТРАКТНИ КЛАСОВЕ

СТАТИЧНО И ДИНАМИЧНО СВЪРЗВАНЕ.

ВИРТУАЛНИ ФУНКЦИИ. ПОЛИМОРФИЗЪМ

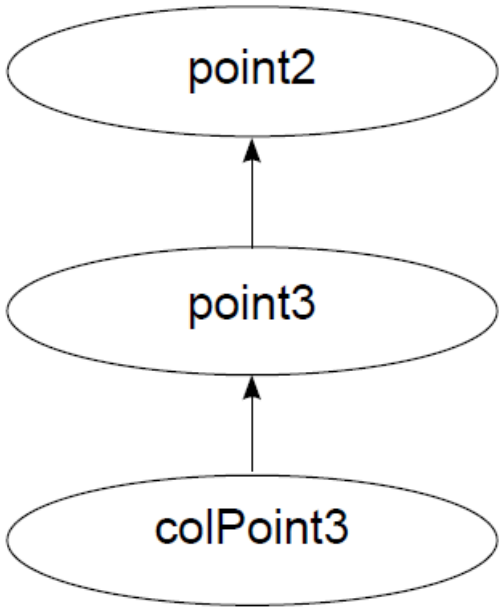
СТАТИЧНО И ДИНАМИЧНО СВЪРЗВАНЕ.

Статично разрешаване на връзката, статично или ранно свързване

При него разрешаването коя функция (член-функция) ще се изпълни се осъществява по време на компилация и не може да бъде променяно по време на изпълнение на програмата.

Пример. Следващата програма дефинира йерархия, определяща точка в равнината, точка в тримерното пространство и точка с цвят в тримерното пространство.

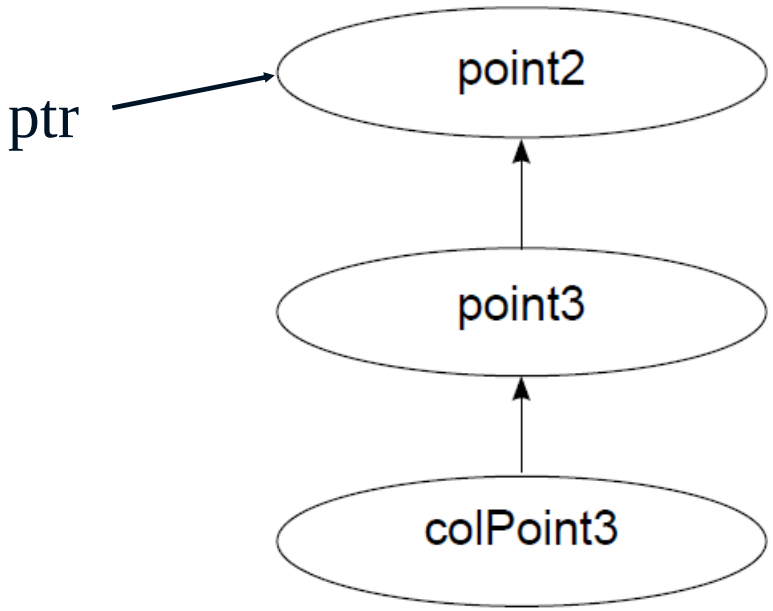
```
#include <iostream>
using namespace std;
class point2
{ public:
      point2(int a = 0, int o = 0)
      { x = a; y = o;
      }
      void print() const
      { cout << x << ", " << y;
      }
   private:
      int x, y;
};
```



```
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;
};
```

Статично свързване



Статично свързване

```
int main()
{ point2 p2(15, 10);
  point3 p3(21, 41, 63);
  colPoint3 p4(12, 24, 36, 11);
  point2 *ptr = &p2;
  ptr->print(); cout << endl;
  ptr = &p3;    // атрибутът на point2 е public
  ptr->print(); cout << endl;
  ptr = &p4;    // атрибутът на point3 е public
  ptr->print(); cout << endl;
  return 0;
}
```

```
point2 *ptr = &p2;
ptr->print(); cout << endl;    // point2::print()

ptr = &p3;
ptr->print(); cout << endl;    // point2::print()

ptr = &p4;
ptr->print(); cout << endl;    // point2::print()
```

Резултат:

15, 10
21, 41
12, 24

Ако искаме след свързването на *ptr* с адреса на *p3* да се изпълни член-функцията *print* на *point3*, а също след свързването на *ptr* с адреса на *p4* да се изпълни член-функцията *print* на *colPoint3*, са необходими явни преобразувания от вида

```
ptr = &p3;
((point3*)ptr)->print();
cout << endl;

ptr = &p4;
((colPoint3*)ptr)->print();
```

Недостатък на статичното свързване:

По време на създаването на клас трябва да се предвидят възможните обекти, указатели и псевдоними на обекти, чрез които ще се извикват член-функциите му.

При сложни йерархии от класове това е не само трудно, но понякога и невъзможно.

Късно* или *динамично свързване

При него *избирането на функцията, която трябва да се изпълни, се осъществява по време на изпълнение на програмата*.

При динамичното свързване не се налага явно преобразуване на типове. 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.

Прилагането на механизма на късното свързване се осъществява над специални член-функции на класове, наречени

виртуални член-функции (***виртуални методи*** **или само** ***виртуални функции***).

Виртуалните член-функции се декларират чрез поставяне на ключовата дума ***virtual*** пред декларацията им, т.е.

virtual [<тип_на_резултата>**]** <име_на_метод>
(<параметри>) **[**const**]**;

Пример.

```
#include <iostream>
using namespace std;
class point2
{ public:
      point2(int a = 0, int o = 0)
      { x = a; y = o;
      }
      virtual void print() const
      { cout << x << ", " << y;
      }
   private:
      int x, y;
};
```

Динамично свързване

```
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;
};
```

Динамично свързване

```
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;
};
```

Динамично свързване

```
int main()
{ point2 p2(15, 10);
  point3 p3(21, 41, 63);
  colPoint3 p4(12, 24, 36, 11);
  point2 *ptr = &p2;
  ptr->print(); cout << endl;
  ptr = &p3;
  ptr->print();
  ptr = &p4;
  ptr->print(); cout << endl;
  return 0;
}
```

Резултат:
15, 10
21, 41, 63
12, 24, 36
colour: 11

Динамично свързване

Декларирането на член-функциите *print* като виртуални причинява трите обръщения към *print* чрез указателя *ptr* да определят функцията, която ще бъде извикана по време на изпълнението на програмата.

Определянето е в зависимост от типа на обекта, към който сочи указателят, а не от класа, от който е указателят.

Динамично свързване

В случая *ptr = &p3*, указателят *ptr* е от класа *point2*, но сочи обекта *p3*, който е от класа *point3*. Затова обръщението *ptr->print()* ще изпълни *point3::print()*, ако е възможен достъп.

В случая *ptr = &p4*, указателят *ptr* сочи обекта *p4*, който е от класа *colPoint3*. Затова обръщението *ptr->print()* ще изпълни *colPoint3::print()*, ако е възможен достъп.

И в двата случая достъпът е възможен, тъй като *ptr* е от тип *point2**, а в този клас член-функцията *print* е в *public* секцията му.

Динамично свързване

Свойства на виртуалните член-функции:

1. Само член-функции на класове могат да се декларират като виртуални. От технически съображения конструкторите не могат да се декларират като виртуални.

2. Ако в клас е декларирана виртуална функция, декларираните член-функции със същия прототип (име, типове на явно указаните параметри и тип на върнатата стойност) в производните на класа класове също са виртуални дори ако ключовата дума *virtual* бъде пропусната.

Динамично свързване

3. Ако в производен клас е дефинирана член-функция със същото име като на вече определена в основния му клас като виртуална член-функция, но с други типове на параметрите и/или тип на резултата, то това е друга член-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 член-функция на основен клас, то те се интерпретират като различни член-функции.

Динамично свързване

5. Възможно е виртуална член-функция да се дефинира извън клас. Тогава заглавието ѝ не започва с ключовата дума *virtual*.

6. Виртуалните член-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 на него класове.

Динамично свързване

8. Виртуалните член-функции се извикват чрез указател към обект или чрез псевдоним на обект на клас.

9. Локалният и външният достъпът до виртуална член-функция имат някой особености, които ще разгледаме.

Достъп до виртуална член-функция

а) *локален (вътрешен) достъп*

На локално ниво (чрез методи на класове) достъпът до виртуални член-функции се определя по традиционните правила. Виртуална член-функция на производен клас има пряк достъп както до собствените на производния клас компоненти, така и до компонентите, декларирани в *public* и *protected* секциите на основните си класове.

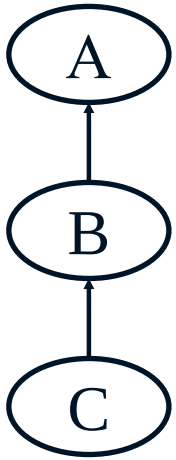
Достъп до виртуална член-функция

Допълнение.

Всяка член-функция на клас, в който е дефинирана виртуална член-функция, има пряк достъп както до виртуалната член-функция на самия клас, така и до виртуалните член-функции със същия прототип на производните му директни и индиректни класове без значение на вида на секциите, в които са дефинирани виртуалните член-функции.

Динамично свързване

Пример. Член-функцията *usual* на класа *A*, дефиниран по-долу, има пряк достъп както до виртуалната член-функция *f* на класа *A*, така и до виртуалните член-функции *f* на класовете *B* и *C*.



Динамично свързване

```
#include <iostream>
using namespace std;
class A
{ private:
      virtual void f() const
      { cout << "A\n";
      }
  public:
      void usual() const
      { f();        // разрешава се динамично
      }
};
```

Динамично свързване

```
class B : public A
{ protected:
      void f() const
      { cout << "B\n";
      }
};

class C : public B
{ private:
      virtual void f() const
      { cout << "C\n";
      }
};
```

Динамично свързване

```
int main()
{ A a; B b; C c;
   A *ptr = &a;
   ptr->usual();
   ptr = &b;
   ptr->usual();
   ptr = &c;
   ptr->usual();
   return 0;
}
```

Резултатът от изпълнението:

```
A
B
C
```

Динамично свързване

б) *външен достъп*

✓външният достъп до виртуална член-функция на клас чрез обект на класа се определя по традиционните правила. Връзката се разрешава статично.

✓ обръщение към виртуална функция чрез указател

клас1* ptr = & обект;

ptr -> виртуална_функция_на_клас1(...);

Пример.

```
#include <iostream>
using namespace std;
class base
{ public:
      virtual void pub()
      { cout << "pub()\n";
      }
      void usual()
      { cout << "usual()\n";
        pub();    // разрешава се диманично
        pri();    // разрешава се диманично
        pro();    // разрешава се диманично
      }
```

Динамично свързване

```
    protected:
        virtual void pro()
        { cout << "pro()\n";
        }
     private:
        virtual void pri()
        { cout << "pri()\n";
        }
};
```

```
class der : public base
{ public:
      virtual void pri()
      { cout << "Derived-pri()\n";
      }
      virtual void pro()
      { cout << "Derived-pro()\n";
      }
   protected:
      virtual void pub()
      { cout << "Derived class\n";
        base::pub();      // разрешава се статично
        base::pro();      // разрешава се статично
      }
};
```

```
int main()
{ base *p = new base;
   base *q = new der;
   p->pub();
   q->pub();
    // p->pri();
    // q->pri();
    // p->pro();
    // q ->pro();
    der *r = new der;
     r->pri();
    // r->pub();
    p->usual();
    delete p;
    delete q;
    delete r;
    return 0;
}
```

Динамично свързване

Резултат:

pub()

Derived class

pub()

pro()

Derived-pri()

usual()

pub()

pri()

pro()

Динамично свързване

- обръщение към виртуална функция чрез псевдоним на обект

клас1 &pps = инициализатор;
pps.виртуална_функция_на_клас1(...);

Динамично свързване

Пример.

```
int main()
{ base x; der y;
  base& z = y;
  z.pub();
  // z.pri();
  // z.pro();
  z.usual();
  return 0;
}
```

ПОЛИМОРФИЗЪМ

Полиморфизъм

Изразява се в това, че едни и същи действия (в общия смисъл) се реализират по различен начин в зависимост от обектите, над 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.

Полиморфизъм

За да се реализира полиморфно действие, класовете над които то ще се прилага, трябва да имат общ родител или прародител.
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 класа.

Полиморфизъм

Активирането на полиморфното действие се осъществява чрез указател към базовия клас или чрез псевдоним на обект на базовия клас.

В зависимост от типа на обекта, към който сочи указателят, ще бъде изпълняван един или друг виртуален метод.

Ако класовете, над които ще се реализира полиморфно действие, нямат общ родител, такъв може да бъде създаден изкуствено чрез дефиниране на т.н. *абстрактен клас*.

Полиморфизъм

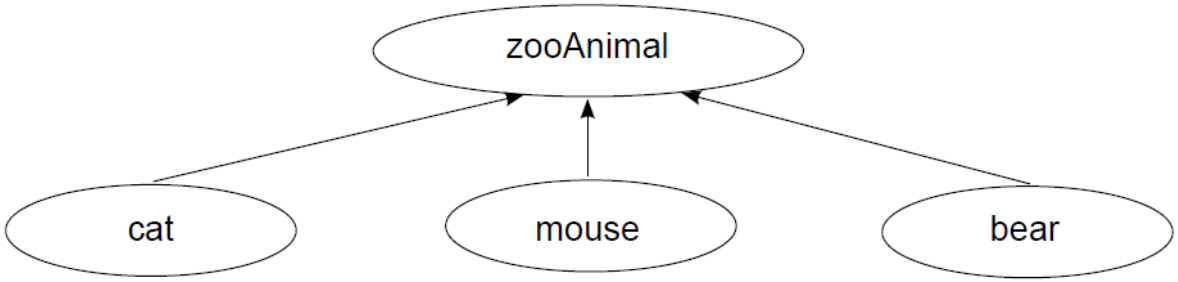
Още един пример за дефиниране на полиморфно действие

В йерархия от 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 за класовете.

Полиморфизъм

Още един пример за дефиниране на полиморфно действие



Полиморфизъм

```
#include <iostream>
using namespace std;
class zooAnimal
{ public:
      void print() const
      { cout << "ZooAnimal\n";
        cout << "Address:\n Sofia, Bulgaria\n";
      }
};
class cat : public zooAnimal
{ public:
      void print() const
      { cout << "ZooAnimal\n";
        cout << "Cat\n";
      }
};
```

Полиморфизъм

```
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";
      }
};
```

Полиморфизъм

```
int main()
{ zooAnimal zoo; zoo.print();
  cat c; c.print();
  mouse m; m.print();
  bear b; b.print();
  return 0;
}
```

Резултат:

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

Полиморфизъм

Модификация на програмата

Полиморфизъм

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

class zooAnimal
{ public:
      virtual void spec() const
      { cout << "Address:\nSofia, Bulgaria\n";
      }
      void print() const
      { cout << "ZooAnimal\n";
        spec();   // разрешава се динамично
      }
};
```

Полиморфизъм

```
class cat : public zooAnimal
{ public:
      virtual void spec() const
      { cout << "Cat\n";
      }
};

class mouse : public zooAnimal
{ public:
      virtual void spec() const
      { cout << "Mouse\n";
      }
};
```

Полиморфизъм

```
class bear : public zooAnimal
{ public:
      virtual void spec() const
      { cout << "Bear\n";
      }
};

int main()
{ zooAnimal zoo; zoo.print();
  cat c; c.print();
  mouse m; m.print();
  bear b; b.print();
  return 0;
}
```

Полиморфизъм

Същият резултат се получава и след изпълнение на програмата, като тялото на главната функция се замени с 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();
```

Полиморфизъм

Съществуват три случая, при които обръщение към виртуална член-функция се разрешава статично (по време на компилация):

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

Пример.

```
cat c; c.spec();
mouse m; m.spec();
bear b; b.spec();
```

Полиморфизъм

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

Извикването на виртуална функция чрез оператора :: **изключва** механизма на динамичното свързване.

Полиморфизъм

Пример. Нека сме в означенията на йерархията с базов клас *zooAnimal*.

```
zooAnimal *pz;
cat c; bear b; mouse m;

pz = &c; pz -> spec(); // динамично свързване
pz = &b; pz -> spec(); // динамично свързване
pz = &m; pz -> spec(); // динамично свързване
pz->zooAnimal::spec(); // статично свързване
```

Полиморфизъм

3. ***Виртуалната член-функция се активира в тялото на конструктор или деструктор на основен клас***

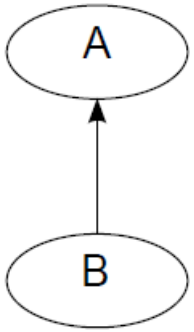
В този случай се изпълнява виртуалната член-функция на основния клас.

Това е така, защото виртуалната функция в конструктора или деструктора на основния клас се извиква когато обектът от производния клас още не е създаден или вече е разрушен.

ВИРТУАЛНИ ДЕСТРУКТОРИ

Пример.

```
#include <iostream>
#include <cstring>
#include <cassert>
using namespace std;
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];
  assert(x != NULL);
  strcpy(x, s);
}
A::~A()
{ cout << "~A()\n";
  delete [] x;
}
```

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

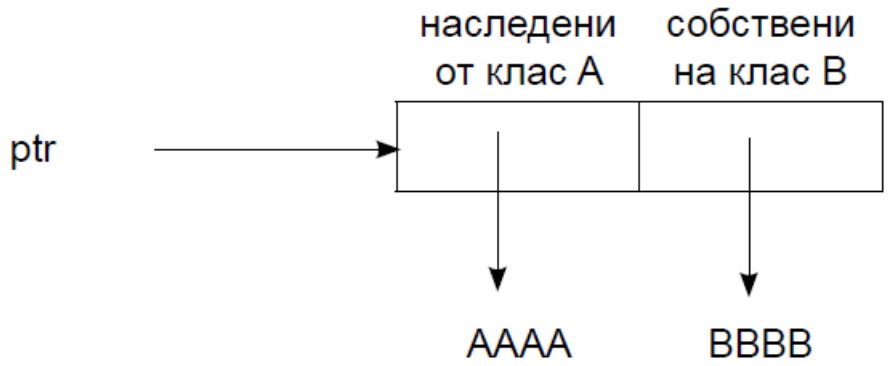
```
void A::print() const
{ cout << "A::x " << x << endl;
}

class B : 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];
  assert(x != NULL);
  strcpy(x, b);
}
B::~B()
{ cout << "~B()\n";
  delete [] x;
}
B::B(const B& p) : A(p)
{ x = new char[strlen(p.x)+1];
  assert(x != NULL);
  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];
      assert(x != NULL);
      strcpy(x, p.x);
    }
    return *this;
}
void B::print() const
{ A::print();
  cout << "B::x " << x << endl;
}
```

Виртуални деструктори

```
int main()
{ A *ptr = new B("AAAA", "BBBB");
  assert(ptr != NULL);
  ptr->print();
  delete ptr;
  return 0;
}
```



Виртуални деструктори

Последователно са се изпълнили:
– конструкторът на базовия клас *A*;
– конструкторът на производния клас *B*.

Тъй като *ptr* е от тип *A**, обръщението *delete ptr*; ще изпълни деструктора на *A*, след което ще разруши връзката на *ptr* с ДП. В резултат динамичната памет, заета от низа *AAAA*, се освобождава, а динамичната памет, заета от низа *BBBB* не се освобождава.

Реализирани са следните обръщения към:
– конструктора на класа *A*;
– конструктора на класа *B*;
– деструктора на класа *A*.

Виртуални деструктори

***Проблемът се решава като деструкторът на базовия клас A на йерархията се обяви за виртуален*.**

Особеност. Обявяването на деструктор на клас на йерархия за виртуален причинява деструкторите на всички класове в наследствената за този основен клас йерархия да са виртуални.

Виртуални деструктори

Нека променим горната програма като обявим деструктора на класа *A* за виртуален. Деструкторът на класа *B* автоматично става виртуален. Тъй като типът на обекта, към който сочи *ptr* е *B*, обръщението

```
delete ptr;
```

отначало ще изпълни деструктора на класа *B*, след него ще изпълни деструктора на класа *A* и най-накрая ще разруши връзката на *ptr* с ДП, т.е. ще се реализира редицата от действия:

Виртуални деструктори

– конструктор на класа *A*;
– конструктор на класа *B*;
– деструктор на класа *B*;
– деструктор на класа *A*

и ще се разруши връзката на *ptr* с ДП.

АБСТРАКТНИ КЛАСОВЕ

Чисто виртуални член-функции

Възможно е виртуална член-функция да не е дефинирана, а само декларирана в клас. Такава виртуална член-функция се нарича *чисто виртуална*.

Декларация:

virtual [<тип>] <име_на_функция>
(<параметри>) [const] = 0;

Клас, в който е декларирана поне една чисто виртуална функция, се нарича ***абстрактен***.

Чисто виртуални член-функции

Абстрактните класове се характеризират със следните свойства:

а) Обекти от тези класове не могат да се създават, но е възможно да се дефинират указатели и псевдоними от такива класове.

б) Чисто виртуалните член-функции задължително трябва да бъдат предефинирани в класовете, производни на абстрактния клас или да бъдат обявени за чисто виртуални в тях.

Чисто виртуални член-функции

Предназначение:

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

Чисто виртуални член-функции

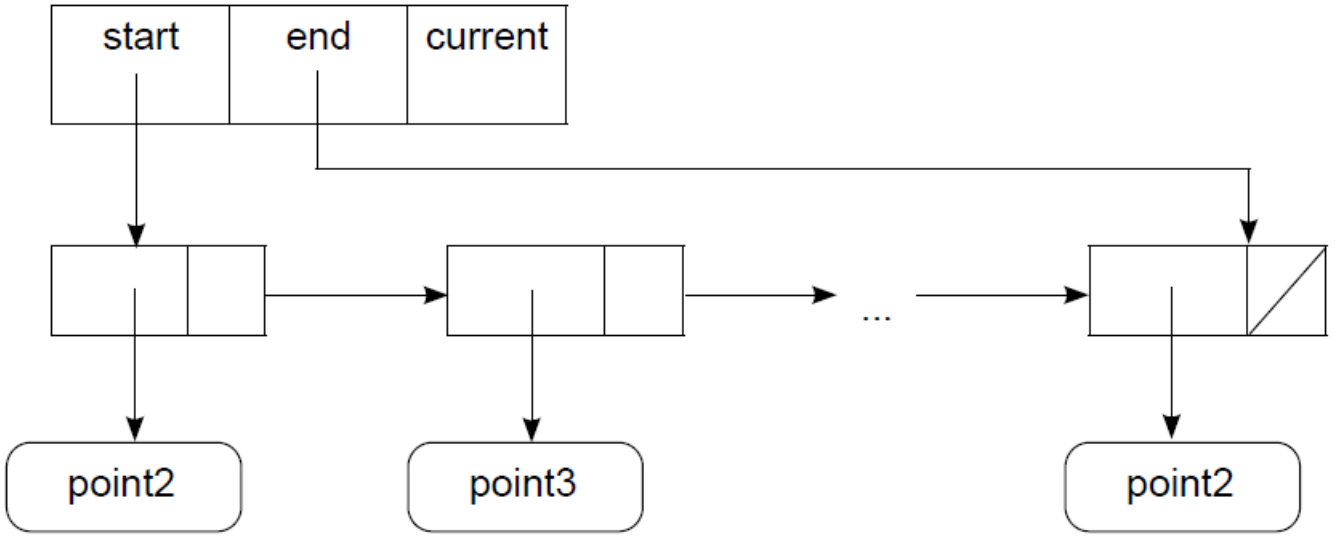
Полиморфизмът, с помощта на абстрактните класове, позволява създаването на *хетерогенни (полиморфни) структури от данни*.

Дефиниция. Съставна структура от данни, компонентите на която са от различни типове се нарича ***хетерогенна***.

Пример. Списък, елементите на който са от различен тип – точки в равнината и точки в пространството.

Чисто виртуални член-функции

Пример. Списък, елементите на който са от различен тип – точки в равнината и точки в пространството.

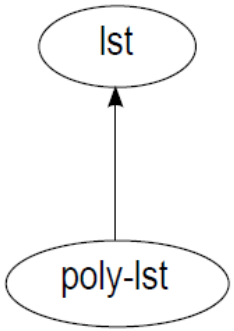


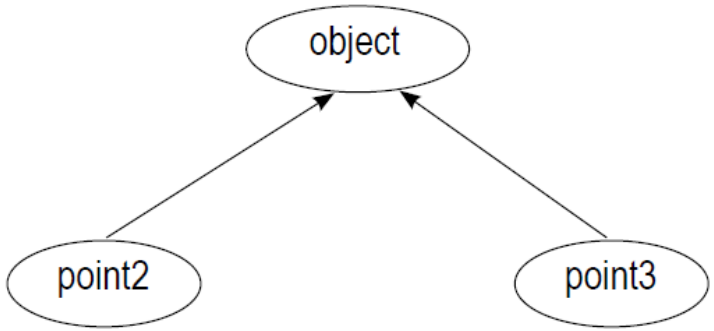
Чисто виртуални член-функции

Задача.

Да се напише програма, която създава хетерогенен едносвързан списък, елементите на който могат да съхраняват обекти от два класа: *point2*, представящ точка в равнината и *point3*, представящ точка в тримерното пространство. Да се изведат елементите на хетерогенния списък.

Чисто виртуални член-функции





Чисто виртуални член-функции - пример

```
#include <iostream>
#include <cassert>
#include "LList.cpp"
using namespace std;

class object                    // абстрактен клас
{ public:
      // полиморфно действие
      virtual void print() const = 0;
};
typedef LList<object*> lst;


class poly_lst : public lst
{ public:
      void print();
};
```

Пример

```
void poly_lst::print()
{ iterStart();
  elem_link<object*> *p = iter();
  object *ptr;
  while(p)
  { ptr = p->inf;
    ptr->print();   // разрешава се динамично
    p = p->link;
  }
  cout << endl;
}
```

Пример

```
class point2 : public object
{ public:
      point2(int a = 0, int o = 0)
      { x = a;
        y = o;
      }
      void print() const
      { cout << x << ", " << y << endl;
      }
   private:
      int x, y;
};
```

Пример

```
class point3 : public object
{ public:
      point3(int a = 0, int o=0, int b=0)
      { x = a;
        y = o;
        z = b;
      }
      void print() const
      { cout << x << ", " << y << ", " << z << endl;
      }
   private:
      int x, y, z;
};
```

Пример

```
int main()
{ poly_lst lh;
  point2 p21(1,5), p22(2,6);
  point3 p31(10, 20, 30), p32(11, 21, 31),
         p33(2, 4, 10);
  lh.toEnd(&p21); lh.toEnd(&p31);
  lh.toEnd(&p22); lh.toEnd(&p32);
  lh.toEnd(&p33);
  lh.print();
  return 0;
}
```

Пример

Резултатът от изпълнението на програмата е:

1, 5
10, 20, 30
2, 6
11, 21, 31
2, 4, 10

Чисто виртуални член-функции - пример

Връзката между двата клона на йерархията може да се осъществи и чрез *псевдоним*.

```
void poly_lst::print()
{ iterStart();
  elem_link<object*> *p = iter();
  while(p)
  { object& ps = *(p->inf);
    ps.print();      // разрешава се динамично
    p = p->link;
  }
  cout << endl;
}
```

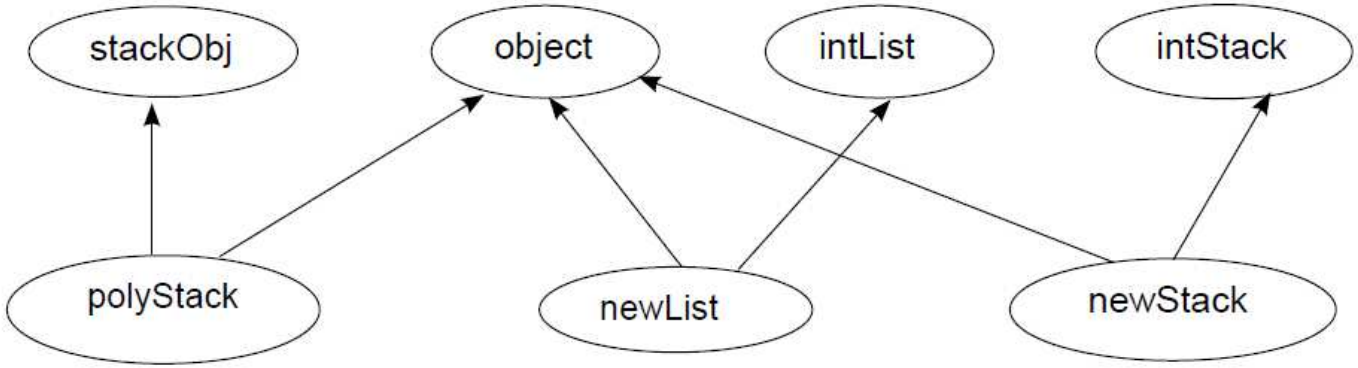
Чисто виртуални член-функции

Задача.

Да се напише програма, която създава хетерогенен стек *polyStack*, елементите на който са обекти от класовете *newList* (списък от цели числа) и *newStack* (стек от цели числа), както и *хетерогени стекове*, съдържащи списъци и стекове с цели числа.

Програмата да извежда контейнерите на хетерогенния стек, а също и броя на елементите, както на хетерогенния стек, така и на всеки негов контейнер.

Чисто виртуални член-функции



Чисто виртуални член-функции

```
#include <iostream>
#include <cassert>
#include "LList.cpp"
#include "stack.cpp"
using namespace std;

class object
{ public:
      // полиморфни действия
      virtual void print() = 0;
      virtual int length() = 0;
};
```

Чисто виртуални член-функции

```
typedef stack<object*> stackObj;
typedef LList<int> intList;
typedef stack<int> intStack;

class polyStack : public stackObj, public object
{ public:
      void print();
      int length();
};
```

Чисто виртуални член-функции

```
void polyStack::print()
{ polyStack ps(*this);      // копира хетерогенния
                            // стек в стека ps
  object *x;
  while(!ps.empty())
  { ps.pop(x);
    x->print();             // разрешава се динамично
  }
}
```

Чисто виртуални член-функции

```
int polyStack::length()
{ polyStack ps(*this);
  int num = 0;
  object *x;
  while(!ps.empty())
  { ps.pop(x);
    cout << x->length() << " ";
    num++;
   }
   cout << endl;
   return num;
}
```

Чисто виртуални член-функции

```
class newList : public intList, public object
{ public:
      void print();
      int length();
};

void newList::print()
{ intList::print();
}

int newList::length()
{ return intList::length();
}
```

Чисто виртуални член-функции

```
class newStack : public intStack, public object
{ public:
      void print();
      int length();
};
void newStack::print()
{ intStack temp(*this);
  temp.print();
}
int newStack::length()
{ intStack temp(*this);
   return temp.length();
}
```

Чисто виртуални член-функции

```
int main()
{  polyStack ps1;
   newList il1;
   il1.toEnd(1); il1.toEnd(2); il1.toEnd(3);
   il1.toEnd(4); il1.toEnd(5);
   ps1.push(&il1);
   newList il2;
   il2.toEnd(300); il2.toEnd(400);
   ps1.push(&il2);
   newStack is1;
   is1.push(10); is1.push(20); is1.push(30);
   is1.push(40);
   ps1.push(&is1);
```

Чисто виртуални член-функции

```
ps1.push(&il2);
newStack is1;
is1.push(10); is1.push(20);
is1.push(30); is1.push(40);
ps1.push(&is1);
cout << "Извеждане на елементите "
        "на хетерогенния стек ps1\n";
ps1.print();
```

Чисто виртуални член-функции

```
cout << "Извеждане на броя на елементите на всеки "
        "от контейнерите и на броя на контейнерите "
        "на хетерогенния стек ps1\n";
cout << ps1.length() << endl;
polyStack ps;
ps.push(&ps1); ps.push(&is1); ps.push(&il2);
cout << "Извеждане на елементите на "
        " хетерогенния стек ps\n";
ps.print();
```

```
    cout << "Извеждане на броя на елементите на всеки "
            "от контейнерите и на броя на контейнерите "
            "на хетерогенния стек ps\n";
    cout << ps.length() << endl;
    return 0;
}
```