Обектно ориентирано програмиране

ВИРТУАЛНИ ФУНКЦИИ.

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**.

Динамично свързване. Виртуални функции ...

Пример: В следващата програма е дефинирана йерархията: **Point2 -> Point3 -> ColPoint3** определяща точка в равнината, точка в тримерното пространство и точка в тримерното пространство с цвят.

```
#include <iostream>
using namespace std;
class Point2
{
public:
   Point2(int a = 0, int b = 0) : x(a), y(b) {}
   void print() const
   {
      cout << x << ", " << y;
   }
private:
   int x;
   int y;
};
```

Динамично свързване. Виртуални функции ...

```
class Point3 : public Point2
{
public:
   Point3(int a = 0, int b = 0, int c = 0) : Point2(a, b), z(c) {}
   void print() const
   {
      Point2::print();
      cout << ", " << z << endl;
   }
private:
   int z;
};
```

Динамично свързване. Виртуални функции ...

```
class ColPoint3 : public Point3
{
public:
   ColPoint3(int a = 0, int b = 0, int c = 0, int col = 0):
             Point3(a, b, c), color(col) {}
   void print() const
   {
      Point3::print();
      cout << "color: " << color << endl;
    }
private:
   int color;
};
```

Динамично свързване. Виртуални функции ...

```
void main()
{
   Point2 p2(5, 10);
   Point3 p3(2, 4, 6);
   ColPoint3 p4(12, 24, 36, 11);
   Point2 *ptr1 = &p3; // атрибутът на Point2 е public
   ptr1->print();
   cout << endl;
   Point2 *ptr2 = &p4; // атрибутът на Point2 е public
   ptr2->print();
   cout << endl;
}
```

Резултат:

2 4

12 14

Динамично свързване. Виртуални функции ...

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.

Динамично свързване. Виртуални функции ...

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.

Динамично свързване. Виртуални функции ...

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**.

Динамично свързване. 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.

Динамично свързване. Виртуални функции ...

```
#include <iostream>
using namespace std;
class Point2
{
public:
   Point2(int a = 0, int b = 0) : x(a), y(b) {}
   virtual void print() const
   {
      cout << x << ", " << y;
   }
private:
   int x;
   int y;
};
```

Динамично свързване. Виртуални функции ...

```
class Point3 : public Point2
{
public:
   Point3(int a = 0, int b = 0, int c = 0) : Point2(a, b), z(c) {}
   virtual void print() const
   {
      Point2::print();
      cout << ", " << z << endl;
   }
private:
   int z;
};
```

Динамично свързване. Виртуални функции ...

```
class ColPoint3 : public Point3
{
public:
   ColPoint3(int a = 0, int b = 0, int c = 0, int col = 0) :
             Point3(a, b, c), color(col) {}
   virtual void print() const
   {
      Point3::print();
      cout << "color: " << color << endl;
   }
private:
   int color;
};
```

Динамично свързване. Виртуални функции ...

```
void main()
{
   Point2 p2(5, 10);
   Point3 p3(2, 4, 6);
   ColPoint3 p4(12, 24, 36, 11);
   Point2 *ptr1 = &p3; // атрибутът на Point2 е public
   ptr1->print();
   cout << endl;
   Point2 *ptr2 = &p4; // атрибутът на Point2 е public
   ptr2->print();
   cout << endl;
}
```

Резултат:

```
2, 4, 6
12, 24, 36
color: 11
```

Динамично свързване. Виртуални функции ...

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 декларират като виртуални.

Динамично свързване. Виртуални функции ...

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.

Динамично свързване. Виртуални функции ...

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.

Динамично свързване. Виртуални функции ...

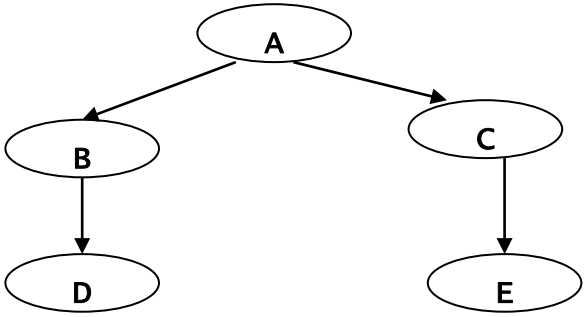
Някои предимства на виртуалните функции

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:

Динамично свързване. Виртуални функции ...



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.

Динамично свързване. Виртуални функции ...

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.

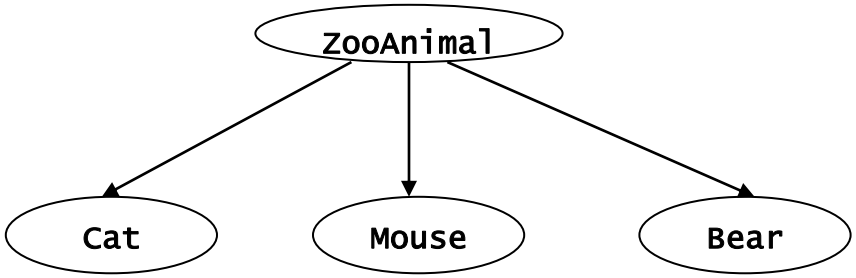
Във всеки от производните класове този метод може да бъде предефиниран съобразно особеностите на този клас.

Активирането на 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**.

Динамично свързване. Виртуални функции ...

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:



Динамично свързване. Виртуални функции ...

```
#include <iostream>

using namespace std;

class ZooAnimal

{

public:

   void print() const

   {

      cout << "ZooAnimal\n";

      cout << "Address:\n"

           << "Sofia, Bulgaria\n";

   }

private:

   //...

};
```

Динамично свързване. Виртуални функции ...

```
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";
   }
//...
};
```

Динамично свързване. Виртуални функции ...

```
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();
}
```

Динамично свързване. Виртуални функции ...

Резултат:

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

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 Bear в производните класове Cat, Mouse или Dog, съответно.

Динамично свързване. Виртуални функции ...

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().

Динамично свързване. Виртуални функции ...

```
using namespace std;
class ZooAnimal
{
public:
   void print() const
   {
      cout << "ZooAnimal\n";
      spec();
   }
   virtual void spec() const
   {
      cout << "Address:\n"
           << "Sofia, Bulgaria\n";
   }
private:
//...
};
```

Динамично свързване. Виртуални функции ...

```
class Cat : public ZooAnimal
{
public:
   virtual void spec() const
   {
      cout << "Cat\n";
   }
//...
};
class Mouse : public ZooAnimal
{
public:
   virtual void spec() const
   {
      cout << "Mouse\n";
   }
//...
};
```

Динамично свързване. Виртуални функции ...

```
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
Bear
```

Динамично свързване. Виртуални функции ...

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();
```

Динамично свързване. Виртуални функции ...

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 ѝ характер произлиза от съдържащата се в нея виртуална функция spec().