Обектно ориентирано програмиране

ВИРТУАЛНИ ФУНКЦИИ. ПОЛИМОРФИЗЪМ. АБСТРАКТНИ КЛАСОВЕ

Динамично свързване. Виртуални функции ...

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

Преди да изкажем правилото, ще разгледаме следната примерна програма:

Динамично свързване. Виртуални функции ...

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

Динамично свързване. Виртуални функции ...

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

Динамично свързване. Виртуални функции ...

```
class Der : public Base
{
protected:
   virtual void pub()
   {
      cout << "Derived class\n";
      Base::pub();
      Base::pro();
   }
public:
   virtual void pri()
   {
      cout << "Derived-pri()\n";
   }
   virtual void pro()
   {
      cout << "Derived-pro()\n";
   }
};
```

Динамично свързване. Виртуални функции ...

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

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

и една обикновена член-функция:

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

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

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

Динамично свързване. Виртуални функции ...

```
void main()
{
   Base *p = new Base;
   Base *q = new Der;
   p->pub();
   q->pub();
   // p->pri();
   // q->pri();
   Der *r = new Der;
   r->pri();
   // q->pro();
   // r->pub();
   p->usual();
}
```

Динамично свързване. Виртуални функции ...

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

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

определят, че в обръщенията:

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

p ще активира Base::pub(), а q – Der::pub(), *ако е възможен достъп*.

Динамично свързване. Виртуални функции ...

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

Обръщенията:

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

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

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

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

Динамично свързване. Виртуални функции ...

Дефиницията

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

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

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

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

Обръщенията:

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

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

Динамично свързване. Виртуални функции ...

Обръщението

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

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

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

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

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

Динамично свързване. Виртуални функции ...

Пример: Фрагментът

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

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

Ще отбележим също, че ако

```
ZooAnimal *z;
```

обръщението:

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

е виртуално.

Динамично свързване. Виртуални функции ...

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

   Пример:

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

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

Динамично свързване. Виртуални функции ...

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

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

Динамично свързване. Виртуални функции ...

Още примери:

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.
class Instrument
{
public:
   void play(note) const
   {
      cout << "Instrument::play" << endl;
   }
};
```

Динамично свързване. Виртуални функции ...

```
// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument
{
public:
   // Redefine interface function:
   void play(note) const {
      cout << "Wind::play" << endl;
   }
};
void tune(Instrument& i)
{
   // ...
   i.play(middleC);
}
```

Динамично свързване. Виртуални функции ...

```
int main()
{
Wind flute;
tune(flute); // Upcasting
}
```

Резултат:

```
Instrument::play
```

Динамично свързване. Виртуални функции ...

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.
class Instrument
{
public:
   virtual void play(note) const
   {
      cout << "Instrument::play" << endl;
   }
};
```

Динамично свързване. Виртуални функции ...

```
// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument
{
public:
   // Redefine interface function:
   void play(note) const {
      cout << "Wind::play" << endl;
   }
};
```

Динамично свързване. Виртуални функции ...

```
void tune(Instrument& i) {
   // ...
   i.play(middleC);
}
int main()
{
   Wind flute;
   tune(flute); // Upcasting
}
```

Резултат:

```
Wind::play
```

Динамично свързване. Виртуални функции ...

Разширяемост

Тъй като play() е дефинирана като виртуална в базовия клас, ние може да добавяме нови класове без да променяме функцията tune. Такива програми се наричат *разширяеми* (*extensible*), защото можем да добавяме нова функционалност чрез дефиниране на нови класове от общ базов клас. Функциите, които използват интерфейса на базовия клас няма нужда да се променят.

Динамично свързване. Виртуални функции ...

```
#pragma once
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument
{
public:
   virtual void play(note) const {
      cout << "Instrument::play" << endl;
   }
   virtual char* what() const {
      return "Instrument";
   }
};
```

Динамично свързване. Виртуални функции ...

```
class Wind : public Instrument // духов инструмент
{
public:
   void play(note) const {
      cout << "Wind::play" << endl;
   }
   char* what() const { return "Wind"; }
};
class Percussion : public Instrument // ударен инструмент
{
public:
   void play(note) const {
      cout << "Percussion::play" << endl;
   }
   char* what() const { return "Percussion"; }
};
```

Динамично свързване. Виртуални функции ...

```
class Stringed : public Instrument // струнен инструмент
{
public:
   void play(note) const {
      cout << "Stringed::play" << endl;
   }
   char* what() const { return "Stringed"; }
};
class Brass : public Wind // меден инструмент
{
public:
   void play(note) const {
      cout << "Brass::play" << endl;
   }
   char* what() const { return "Brass"; }
};
```

Динамично свързване. Виртуални функции ...

```
class Woodwind : public Wind // дървен духов инсрумент
{
public:
   void play(note) const {
      cout << "Woodwind::play" << endl;
   }
   char* what() const { return "Woodwind"; }
};
void tune(Instrument& i)
{
   // ...
   i.play(middleC);
}
```

Динамично свързване. Виртуални функции ...

```
int main() {
    Wind flute; // флейта
    Percussion drum;
    Stringed violin;
    Brass flugelhorn; // флигорна
    Woodwind recorder; // права флейта
    tune(flute);
    tune(drum);
    tune(violin);
    tune(flugelhorn);
    tune(recorder);
```

Динамично свързване. Виртуални функции ...

```
    Instrument* A[] = {
        new Wind,
        new Percussion,
        new Stringed,
        new Brass,
    };
    for (int i = 0; i < 4; i++)
    {
        tune(*(A[i]));
        cout << A[i]->what();
    }
    for (int i = 0; i < 4; i++)
        delete A[i];
}
```

Виртуални деструктори

Виртуални деструктори

Да разгледаме следния пример:

```
#include <iostream>
using namespace std;
class Base1
{
public:
   Base1() { cout << "Base1()\n"; }
   ~Base1() { cout << "~Base1()\n"; }
};
class Derived1 : public Base1
{
public:
   Derived1() { cout << "Derived1()\n"; }
   ~Derived1() { cout << "~Derived1()\n"; }
};
```

Виртуални деструктори ...

```
class Base2
{
public:
   Base2() { cout << "Base2()\n"; }
   virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2
{
public:
   Derived2() { cout << "Derived2()\n"; }
   ~Derived2() { cout << "~Derived2()\n"; }
};
```

Виртуални деструктори ...

```
int main()
{
Base1* bp = new Derived1; // Upcast
delete bp;
Base2* b2p = new Derived2; // Upcast
delete b2p;
}
```

Резултат:

```
Base1()
Derived1()
~Base1()
Base2()
Derived2()
~Derived2()
~Base2()
```

Виртуални деструктори ...

Особеност: Обявяването на деструктора на клас на йерархия за виртуален води до това, деструкторите на всички класове в наследствената за този основен клас йерархия да се разглеждат като виртуални.

Абстрактни класове

Възможно е виртуална член.функция да има само декларация, а не дефиниция. Такава виртуална член-функция се нарича чисто виртуална. За да се определи една виртуална функция като чисто виртуална, се използва следния синтаксис:

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

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

Абстрактни класове ...

Абстрактните класове се характеризират със следните свойства:

а) Обекти от тези класове не могат да се създават, но е възможно да се дефинират указатели към такива класове;

б) Чисто виртуалните функции задължително трябва да бъдат дефинирани в производните класове или да бъдат обявени за чисто виртуални. В последния случай класът наследник също е абстрактен.

Абстрактните класове са предназначени да служат за базови на други класове. Чрез тях могат да се обединят в обща структура различни йерархии.

Абстрактни класове ...

Абстрактен клас, който няма член-данни и всички член-функции са виртуални чисти се нарича **интерфейс**

Ако наследник на абстрактен клас не реализира някоя чиста виртуална функция, то той също остава абстрактен

Абстрактните класове могат да имат конструктори и деструктори – но те винаги се извикват косвено, от наследник

Можем да имаме указатели и псевдоними към абстрактни класове

Абстрактни класове не могат да са

- параметри на шаблон
- тип на връщан резултат

Що е интерфейс?

Множество от операции, които поддържа даден тип

- не включва физическото представяне на типа
- не включва реализацията на операциите
- включва имената на операциите
- включва брой и типове на параметрите

Ако няколко класа имат общ интерфейс, с тях може да се работи унифицирано

- но затова всички операции от интерфейса трябва да са виртуални, т.е. с динамично свързване

Абстрактни класове ...

Полиморфизмът, с помощта на абстрактните класове позволява създаването на класове с различна логическа структура. Пример за такава структура е списък, елементите на който са от различен тип – стек, опашка, дърво и др. контейнери. Такива структури се наричат **хетерогенни** или **полиморфни**. Логическата структура на класа, представляващ хетерогенна конструкция, се реализира отделно от обектите, които се включват в него. Връзката между тях се осъществява чрез указатели към контейнери, които съхраняват обекти от различни класове.

Хетерогенни контейнери

Контейнери, които съдържат обекти от различен тип

- Реализират се чрез използване на указател към полиморфен тип

– защо указател, а не директно обект?

– защо указател, а не псевдоним?

- Над хетерогенните контейнери могат да се изпълняват масови операции от общия интерфейс

Абстрактни класове ...

Пример:

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument
{
public:
   // Pure virtual functions:
   virtual void play(note) const = 0;
   virtual char* what() const = 0;
};
```

Абстрактни класове ...

```
class Wind : public Instrument // духов инструмент
{
public:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
    char* what() const { return "Wind"; }
};
...
```

Абстрактни класове ...

```
int main() {
   Instrument* A[] = { new Wind, new Percussion,
      new Stringed, new Brass,
   };
   for (int i = 0; i < 4; i++) {
      tune(*(A[i]));
      cout << A[i]->what();
   }
   for (int i = 0; i < 4; i++)
      delete A[i];
}
```