10 Derivarea claselor. Supraîncărcarea funcțiilor cu argumente de tip clasă

10.1 Conversii standard

Între clasele derivate şi clasele publice de bază există 4 conversii implicite predefinite:

Clasa derivată public		Clasa publică de bază
obiect	->	obiect
referință	->	referință
pointer	->	pointer
pointer la membru	<-	pointer la membru

În plus, un pointer la orice obiect este convertit implicit la un pointer generic, `void *`. Pentru a asigna un pointer generic unui pointer de orice alt tip este necesar un cast explicit !

Conversiile din clasa derivată într-o clasă publică de bază sunt considerate sigure pentru că orice obiect de tipul clasei derivate conține un obiect de tipul clasei publice de bază (conține toți membrii claei publice de bază). Din acelaşi motiv, pointerii la membri se comportă exact invers.

10.2 Inițializări şi asignări

Am vazut până acum că inițializarea sau asignarea unui obiect oarecare cu un alt obiect de acelaşi tip se face membru cu membru. Regula se păstreaza şi pentru obiectele de tip clasă derivată: membrii se inițializează în ordinea declarării lor, de la clasele de bază spre cele derivate. Cum s-a mai vazut, inițializările membru cu membru pot genera probleme în manipularea obiectelor care conțin pointeri. Soluția constă în redefinirea constructorului de copiere `X( const X& )` şi a operatorului de atribuire `operator=(const X&)`.

În continuare ne vom ocupa de constructorul de copiere `X( const X& )` în contextul derivării claselor. Vom discuta următoarele 3 cazuri:

1) clasa de bază a definit un constructor de copiere, iar clasa derivată nu.

Astfel, membrii clasei derivate vor fi inițializati membru cu membru.

Exemplu:

```
class Carnivor
{
   public:
      Carnivor();
...
};

class AnimalZoo
{
   public:
      AnimalZoo();
      AnimalZoo( const AnimalZoo& );
...
};

class Felina : public AnimalZoo, public Carnivor
{
   public:
      Felina();
...
};

Felina Felix;
Felina Fritz = Felix; // - apel constructor clasa de baza
                      //   AnimalZoo( const AnimalZoo& )
                      // - initializare membru cu membru pentru
                      //   clasa de baza Carnivor
```

2) clasa derivată a definit constructor de copiere;

În acest caz, acesta trebuie să transmită explicit parametri pentru inițializarea tipului de bază.

Exemplu:

```
class Urs : public AnimalZoo
{
   public:
      Urs();
      Urs( const Urs& ) { ... };
};

Urs Yogi;
Urs Stanley = Yogi;
```

Inițializarea obiectului `Stanley` cu obiectul `Yogi` se face prin următoarele invocari de constructori: `AnimalZoo(); Urs( Urs& );`.

3) atât clasa de bază cât şi clasa derivată au definit câte un constructore de copiere

Constructorul clasei de bază este invocat întotdeauna înaintea execuției constructorului clasei derivate. La fel se întâmplă şi în cazul constructorului de copiere. În plus, dacă există constructor al clasei de bază cu un singur argument, el trebuie specificat în lista de inițializare a membrilor. Pentru că `AnimalZoo` defineşte propria instanță a constructorului de copiere, este preferabil ca el să fie invocat. Deci corect ar fi fost:

Exemplu:

```
Urs( const Urs& u) : AnimalZoo(u)
{
}
```

Dacă clasa de bază nu are definit un constructor `X( const X& )`, se aplică recursiv inițializare membru cu membru pentru clasa de bază.

Analog se tratează şi cazurile de asignare, deci se defineşte o instanță a operatorului `operator = (const X&)`.

- Dacă o clasă derivată nu defineşte `operator = (const X&)`, atunci pentru fiecare clasă de bază sau obiect membru care are definit un obiect propriu de asignare, acesta va fi apelat ori de câte ori un obiect de tipul clasei derivate este asignat altui obiect de tipul clasei derivate.

- Dacă o clasă derivata defineşte o instanță pentru `operator = (const X&)`, el va fi invocat să rezolve cazurile de asignare a unui obiect altui obiect de tipul clasei.

Observație:

Dacă şi clasa de bază şi clasa derivată definesc `operator = (const X&)`, atunci pentru

```
Urs u;
AnimalZoo A;
A = u;
```

tipul operandului stâng determină tipul asignării. Partea de AnimalZoo din obiectul `Urs u` va fi asignata obiectului `A`.

Am văzut în capitolele anterioare că programarea orientată pe obiecte este caracterizată prin *moştenire* şi *legături dinamice*. Moştenirea defineşte relații de forma *tip/subtip* între clase, iar funcțiile virtuale definesc operații dependente de tip într-o ierarhie.

10.3 Supraîncărcarea funcțiilor cu argumente de tip clasă

În cele ce urmează vom detalia procesul de *potrivire a argumentelor* (*argument matching*). Aria posibilităților de potrivire a argumentelor se restrânge la *potrivire exactă* a

parametrilor rezultați 1) în urma aplicării unei conversii standard, sau 2) în urma invocării unui operator de conversie definit de utilizator.

10.3.1 Potrivire exactă de argumente (*exact match*)

Un obiect se potriveşte exact doar unui argument formal de acelaşi tip cu el.

Exemplu:

```
f(Urs&);
f(Panda&);
Panda Beep;
f(Beep); //potrivire exacta, f(Panda&).
```

Analog, un pointer la un obiect se potriveşte exact doar unui argument formal de acelaşi tip.

10.3.2 Conversii standard

Dacă argumentul clasei nu se potriveşte exact, se încearcă o potrivire aplicând conversii standard predefinite:

1) Un obiect, o referință sau un pointer al unei clase derivate este implicit convertit la tipul corespunzător clasei de bază.

Exemplu:

```
f( AnimalZoo& );
f( Screen& );
f( Beep ); // f( AnimalZoo& );
```

2) Un pointer la orice tip de clasă este convertit implicit într-un pointer de tip `void *`. În sens invers, este nevoie de *cast* explicit.

Exemplu:

```
f( Screen& );
f( void* );
f( &Beep ); // f( void* );
```

Observații:

- NU se aplică o conversie de la un tip de bază (obiect, referință sau pointer) într-un tip derivat.

Exemplu:

```
f( Urs& );
f( Panda& );
AnimalZoo az;
f( az ); // eroare, nepotrivire.
```

- Prezența mai multor tipuri de bază (publice) poate genera ambiguități.

Exemplu:

```
f( Urs& );
f( Ocrotit& );
f( Beep ); // eroare,  ambiguitate.
```

Soluție: Programatorul trebuie să specifice explicit:

```
f( Urs(Beep) );
```

- Conversiile la tipurile claselor de bază se fac începând cu cele mai apropiate în ierarhia de derivare.

Exemplu:

```
f( AnimalZoo& );
f( Urs& );
f( Beep ); // f( Urs& );
```

- Aceeaşi regulă este valabilă şi pentru pointeri. La pointeri generici se ajunge doar dacă nu mai este altă posibilitate.

Exemplu:

```
f( void* );
f( AnimalZoo* );
f( &Beep ); // f( AnimalZoo* );
```

10.3.3 Conversii utilizator

Acestea pot fi efectuate de către constructorii cu un argument sau de către operatori de conversie expliciți. Conversiile utilizator se aplică numai dacă prin conversii standard nu a fost posibilă o potrivire exactă a argumentelor. Pentru a exemplifica acest lucru, să adăugăm clasei `AnimalZoo` următoarele două conversii utilizator.

Exemplu:

```
class AnimalZoo
{
   public:
      AnimalZoo( long ); // pentru conversia long->AnimalZoo
      operator char*();  // pentru conversia AnimalZoo->char*
...
};
```

Date fiind funcțiile supraîncărcate:

```
f( AnimalZoo& );
f( Screen& );
long lval;
f(lval); // f( AnimalZoo& ); prin conversie utilizator
```

Observații:

- Algoritmul de *matching* aplică conversii utilizator standard dacă astfel se poate ajunge la conversii definite de utilizator:

Exemplu:

```
f( 1024 ); // f( AnimalZoo& ); ( 1024->long->AnimalZoo)
```

- O conversie utilizator se aplică doar când tipurile nu mai pot fi potrivite altfel:

Exemplu:

```
f( AnimalZoo& );
f( char );
long lval;
f( lval ); // f( char );
f( AnimalZoo(lval) ); // f( AnimalZoo& ); prin cast explicit
```

- Algoritmul de *matching* aplică conversii standard rezultatelor conversiilor utilizator dacă în acest mod se poate obține o potrivire de tip.

Exemplu:

```
f( Panda* );
f( void* );
AnimalZoo az;
f( az ); // f(void *); (az->char*->void*)
```

- Tipul de bază nu se convertește la tipuri derivate
- Operatorii de conversie (NU şi constructorii!) sunt moşteniți în aceeaşi manieră ca şi ceilalți membri.
- Constructorul de conversie şi operatorul de conversie au aceeaşi precedență.

Exemplu:

Să definim pentru clasa `Ocrotit` un operator de conversie la tipul `int`:

```
class Ocrotit
{
   public:
```

```
      operator int(); // Ocrotit->int
...
};
```

Presupunem că `Disparut` este o clasă care defineşte un constructor de conversie care acceptă o referință la un obiect de tip `Ocrotit`:

```
class Disparut
{
   public:
      Disparut( Ocrotit& );   // Ocrotit->Disparut
...
};
```

În următoarea situație apare o ambiguitate generată de faptul că operatorul de conversie şi constructorul de conversie au aceeaşi precedență:

```
f( Disparut& );
f( int );
Ocrotit oc;
f( oc ); // eroare, ambiguitate.
```