Mostenirea multipla în C++

Ce este mostenirea multipla, ce probleme ridica si cum pot fi ele rezolvate sau evitate.

Mostenirea multipla reprezinta abilitatea de a defini o clasa derivata simultan din mai multe clase (numite clase de baza). A fost implementata pentru prima oara în 1989 în compilatorul *Cfront Release 2.0* si a fost prezentata de Bjarne Stroustrup în editia a II-a a cartii sale *"The C++ Programming Language"*, alaturi de alte caracteristici avansate ale limbajului C++ precum exceptiile si template-urile.

Înca de la început, mostenirea multipla a stârnit controverse. Oponentii sai au sustinut ca:

- orice model de proiectare care utilizeaza mostenirea multipla poate fi remodelat cu mostenire simpla;
- adauga limbajului un nivel de complexitate ne-necesar;
- complica procesul de scriere a compilatoarelor.

Cu siguranta, ultimul argument este adevarat. Sustinatorii mostenirii multiple afirma insa ca:

- mostenirea multipla apare natural în proiectarea aproape a oricarui sistem foarte mare si permite modelarea obiectelor cât mai fidel lumii reale;
- poate fi implementata eficient;

În privinta cresterii complexitatii limbajului, trebuie spus ca mostenirea multipla este optionala; cei care nu se simt confortabil cu ea, sunt liberi sa nu o utilizeze. Reamintim însa o metafora care a circulat în 1991 si care spune ca mostenirea multipla este ca o parasuta: nu ai nevoie de ea prea des, dar atunci când ai, este esentiala.

Din punct de vedere sintactic, mostenirea multipla se specifica astfel:

```
//exemplul 1
class base{};
class base1: public base{};
class base2: public base{};
class derived: public base1, public base2{};
```

Pentru derived, base1 si base2 sunt baze directe, iar base este baza indirecta. O clasa nu poate fi specificata ca baza directa de mai multe ori, dar poate fi baza indirecta de oricâte ori; de asemenea, o clasa poate fi, simultan, si baza directa, si baza indirecta.

Lista specificatorilor de baze indica tipurile sub-obiectelor continute de un obiect al clasei derivate; pentru fiecare aparitie a unei baze directe, obiectul derivat contine un sub-obiect de tipul corespunzator. Ordinea bazelor în lista este importanta doar din punct de vedere al ordinii de initializare a sub-obiectelor continute de obiectul clasei derivate.

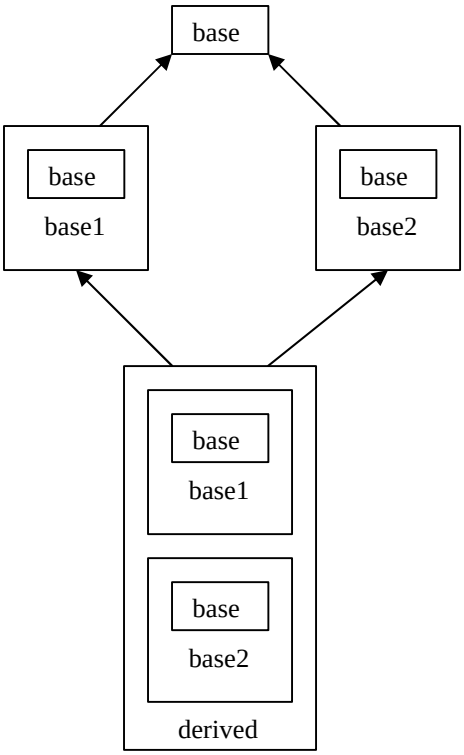


figura 1

Dupa cum se observa din figura 1, un obiect derived contine câte un sub-obiect base1 si base2; fiecare dintre acestea contine câte un subobiect base. Este deja vizibila una dintre problemele care apar: *mostenirea repetata* a clasei base. În functie de problema modelata, aceasta situatie poate sa aiba, sau nu, sens; vom discuta mai târziu despre cazul al doilea.

Daca nu sunt redefiniti în clasa derivata, membrii unei baze sunt membri si ai clasei derivate; membrii mosteniti se utilizeaza analog celor proprii, exceptând cazurile când ei sunt ascunsi sau când apare *ambiguitatea*.

Ambiguitatea se refera la receptionarea unui aceluiasi nume de membru de la mai multe clase de baza; ea poate fi accidentala sau mostenita. Sa consideram urmatorul exemplu:

```
//exemplul 1
class base{
	public:
	virtual void vf() const = 0;
	virtual ~base(){}
};

class base1: public base{
	public:
	void vf() const{
		cout << "base1::vf()\n";
	}
	virtual ~base1(){}
	private:
	void g() const{
		cout << "base1::g()\n";
	}
};
class base2: public base{
	public:
	void vf() const{
		cout << "base2::vf()\n";
	}
	virtual ~base2(){}
	int g() const{
		cout << "base2::g()\n";
	return 0;
	}
};
```

```
class derived: public base1, public base2{};

int main(){
	derived d;
	return 0;
}
```

În acest moment exista doua ambiguitati potentiale: vf() (mostenita) si g() (accidentala). Pentru un compilator de C++ ambiguitatea potentiala nu se constituie însa în eroare; din acest motiv, programul anterior se compileaza fara probleme. Efectuând însa modificarea:

```
int main(){
	derived d;
	d.vf();
	d.g()
	int i = d.g();
	return 0;
}
```

compilatorul C++ raporteaza

```
ex1.cpp: In function `int main()':
ex1.cpp:41: request for member `vf' is ambiguous
ex1.cpp:7: candidates are: virtual void base::vf() const
ex1.cpp:26:              virtual void base2::vf() const
ex1.cpp:13:              virtual void base1::vf() const
ex1.cpp:42: request for member `g' is ambiguous
ex1.cpp:30: candidates are: int base2::g() const
ex1.cpp:18:              void base1::g() const
ex1.cpp:43: request for member `g' is ambiguous
ex1.cpp:30: candidates are: int base2::g() const
ex1.cpp:18:              void base1::g() const
```

Apelul metodelor vf() si g() forteaza compilatorul sa faca o alegere, pe care acesta nu o poate însa efectua fara o rezolvare explicita a ambiguitatii. Foarte important, se observa în cazul metodei g() ca valoarea returnata nu se ia în considerare (analog mecanismului de supraîncarcare a functiilor). De asemenea, restrictiile de accesibilitate nu intra în calcul; avem ambiguitate desi base1::g() este private si deci inaccesibila în derived! Aceasta deoarece determinarea semnificatiei unui nume în contextul unei clase precede controlul accesului!

Ambiguitatile pot fi rezolvate explicit astfel:

```
int main(){
	derived d;
	d.base1::vf();
	d.base2::g()
	int i = d.base2::g();
	return 0;
}
```

Redefinirea unui nume dintr-o clasa de baza într-o clasa derivata ascunde versiunea din clasa de baza; din acest motiv, ambiguitatea poate fi rezolvata si astfel:

```
//...
class derived: public base1, public base2{
        public:
        void vf() const{
                cout << "derived::vf()\n";
        }
        void g() const{
                cout << "derived::g()\n";
        }
};
//...
int main(){
        derived d;
        d.vf();
        d.g()
        return 0;
}
```

O discutie speciala trebuie purtata în legatura cu vf(). Pe de o parte, utilizarea numelui clasei pentru explicitarea ambiguitatii (adica d.base1::vf()) anuleaza comportamentul virtual al lui vf()! Pe de alta parte, metodele virtuale reprezinta calea prin care C++ asigura polimorfismul dinamic; utilizatorul are libertatea de o oferi o definitie noua unei funtii virtuale daca nu este multumit de definitia mostenita de la clasa de baza. Se observa insa ca, desi derived mosteneste doua versiuni ale lui vf() (base1::vf() si base2::vf()), doar una singura poate fi supraîncarcata! Aceasta deoarece într-o clasa poate exista o singura metoda cu prototipul void vf() const.

Dar daca este necesar a fi supraîncarcate în derived atât base1::vf() cât si base2::vf()? Este o întrebare îndreptatita, care ar fi putut conduce la modificarea limbajului C++ daca nu ar fi fost gasita urmatoarea tehnica:

```
//exemplul 2
class base{
        public:
        virtual void vf() = 0;
};
class base1: public base{
        public:
        void vf(){
                cout << "base1::vf()\n";
        }
};
class base2: public base{
        public:
        void vf() {
                cout << "base2::vf()\n";
        }
};
```

```
class aux_base1: public base1{
	public:
	virtual void aux1_vf() =0;
	void vf(){
		aux1_vf();
	}
};

class aux_base2: public base2{
	public:
	virtual void aux2_vf() =0;
	void vf(){
		aux2_vf();
	}
};

class derived: public aux_base1, public aux_base2{
	public:
	void aux1_vf(){
		cout << "derived::aux1_vf()\n";
	};
	void aux2_vf(){
		cout << "derived::aux2_vf()\n";
	};
};

int main(){
	derived d;
	base1* p1 = &d;
	base2* p2 = &d;
	p1->vf();
	p2->vf();
	return 0;
}
```

si output-ul acestui program este:

```
derived::aux1_vf()
derived::aux2_vf()
```

Practic, au fost introduse doua noi clase; fiecare dintre acestea declara un nume nou pentru vf() sub forma unei functii virtuale pure, si supraîncarca vf() ca sa apeleze aceasta functie. În final, derived ofera definitii celor doua functii virtuale pure pe care le mosteneste. În felul acesta, un nume singular a fost separat în doua nume operational echivalente si ne-ambigue.

O alta ambiguitate se obtine daca încercam sa obtinem un pointer (sau referinta) la base dintr-un pointer (referinta) la derived; de exemplu, secventa de cod

```
derived d;
base* pb = &d;
```

va genera mesajul de eroare

```
ex1.cpp: In function `int main()':
ex1.cpp:41: `base' is an ambiguous base of `derived'
```

Ambiguitatea se datoreaza faptului ca un obiect derived contine doua sub-obiecte base; conversia lui &d se poate realiza fie catre sub-obiectul base mostenit prin intermediul base1, fie catre sub-obiectul base mostenit prin intermediul base2. O conversie intermediara explicita rezolva aceasta situatie:

```
derived d;
base * pb1 = static_cast<base*>(static_cast<base1*>(&d));
base * pb2 = static_cast<base*>(static_cast<base2*>(&d));
cout << pb1 << '\t' << pb2 << endl;
```

Output-ul acestei secvente de cod este

```
0xbfffdcc0      0xbfffdcc4
```

si cele doua adrese distincte afisate evidentiaza faptul ca d contine doua sub-obiecte base.

Dupa cum am mentionat anterior, în functie de problema modelata, are sens ca o clasa derivata sa contina câte un sub-obiect corespunzator fiecarei aparitii a unei clase de baza. De exemplu, are sens ca un obiect apartinând unei clase MultiScrollWindow, derivata din HorizontalScrollbar si VerticalScrollbar, ambele derivate din Scrollbar, sa contina doua sub-obiecte ale bazei indirecte Scrollbar. Exista însa si situatii când un astfel de fenomen nu este de dorit; de exemplu, o ierarhie RadioCasetofon, Radio, Casetofon, DispozitivElectric. Într-o astfel de situatie se poate utiliza *mostenirea virtuala*.

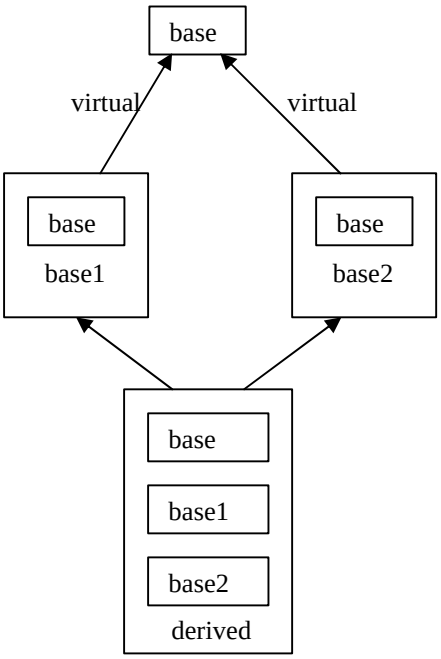


fig. 2

Dupa cum se observa în figura 2, instantele unei clase derivate direct sau indirect din baza virtuala base contin un singur sub-obiect base, indiferent de numarul cailor prin care clasa mosteneste base. Sintactic, mostenirea virtuala se specifica astfel:

```
//exemplul 3
class base{};
class base1: public virtual base{};
class base2: public virtual base{};
class derived: public base1, public base2{};
```

Cum asigura compilatorul C++ existenta unei singure instante a unei baze virtuale? Standardul nu precizeaza nimic despre layout-ul obiectelor în memorie si lasa la latitudinea dezvoltatorilor compilatorului rezolvarea acestei chestiuni; uzual, se utilizeaza un nivel suplimentar de indirectare, accesându-se baza virtuala prin intermediul unui pointer, ceea ce face ca localizarea sub-obiectelor bazei virtuale sa nu fie cunoscuta la momentul compilarii, rezultând costuri în spatiu si timp (în anumite circumstante, RTTI poate fi necesara pentru a accesa sub-obiectele bazei virtuale). Pentru exemplificare, consultati figura 3.

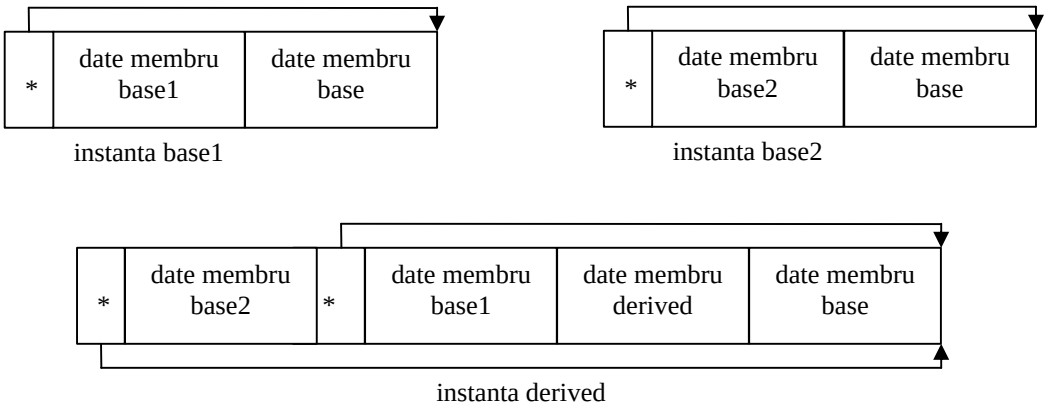


figura 3

Executând secventa de cod

```
//exemplul 4
derived d;
base * pb1 = static_cast<base*>(static_cast<base1*>(&d));
base * pb2 = static_cast<base*>(static_cast<base2*>(&d));
cout << pb1 << '\t' << pb2 << endl;
base* pb3 = &d;
cout << pb3 << endl;
```

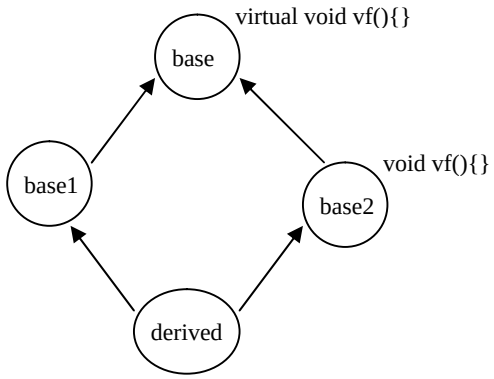
se obtine output-ul

```
0xbffff840      0xbffff840
0xbffff840
```

si se observa ca cele doua adrese coincid; mai mult decât atât, încercarea de a converti direct &d la base* nu mai este ambigua, din moment ce d contine un singur sub-obiect base.

Daca la mostenirea ne-virtuala vf() conducea la ambiguitate potentiala, în cazul mostenirii virtuale vf() cauzeaza ambiguitate efectiva (un singur sub-obiect base si doua versiuni, base1::vf(), respectiv base2::vf()); din acest motiv, în derived, vf() trebuie supraîncarcata explicit.

Ce se întâmpla însa în urmatoarea situatie?




```
derived* pd = new derived;
pd->vf();  //base::vf() sau base2::vf() ?
```

Metoda virtuala vf() din base este supraîncarcata doar în base2 (base1 pastreaza varianta din base); derived nu supraîncarca vf(). Care versiune se va apela?

Raspunsul depinde de tipul derivarii. Daca base este baza ne-virtuala, atunci avem ambiguitate. Dar daca base este baza virtuala, supraîncarcarea lui vf() în base2 *domina* definitia originala din base; din acest motiv, se va apela base2::vf()!

O ultima observatie: la mostenirea ne-virtuala, clasele de pe un nivel n paseaza argumente constructorilor claselor de pe nivelul n-1pentru initializarea sub-obiectelor membru. La mostenirea virtuala, derived (si toate clasele derivate din ea) contin o singura instanta base; cine paseaza argumente pentru initializarea acestei instante: un constructor al lui base1 sau un constructor al lui base2? Raspunul este acela ca ultima clasa din ierarhie (numita *most derived class*) este responsabila cu initializarea bazei virtuale. În cazul nostru, la crearea unei instante base1, baza virtuala din aceasta instanta va fi initializata de un constructor al clasei base1; la crearea unei instante base2, baza virtuala din aceasta instanta va fi initializata de un constructor al clasei base2; la crearea unei instante derived, baza virtuala din aceasta instanta va fi initializata de un constructor al clasei derived.

Aceasta însemna ca responsabila cu initializarea bazei virtuale poate fi o clasa arbitrar de îndepartata. Cu alte cuvinte, utilizatorul unei biblioteci de clase trebuie sa se familiarizeze cu bazele virtuale ale claselor din care doreste sa deriveze, pentru a le putea initializa. De aceea, se prefera ca bazele virtuale sa fie dotate cu constructori impliciti sau sa nu contina date membru (analog interfetelor din Java).

În final, urmatorul exemplu demonstreaza cum poate fi utilizata mostenirea virtuala pentru a scrie o clasa din care nu se mai poate deriva:

```
//exemplul 5
class aux{
        protected:
        aux(int i){};                   //constructorul accesibil în clasele derivate
};

class demo: virtual private aux{        //mostenirea private face constructorul clasei aux
        public:                         //inaccesibil în clasele derivate din demo!
        demo(): aux(0){}                //initializarea bazei virtuale
};

class test: public demo{                //încercare de a deriva din demo
   public:
   test(): aux(0){};                    //obligativitatea de a initializa baza virtuala aux
};

int main(){
test obj;                               //eroare la compilare
return 0;
}
```

Încercarea de a compila acest program conduce la afisarea urmatorului output:

```
ex5.cpp: In constructor `test::test()':
ex5.cpp:6: `aux::aux(int)' is protected
ex5.cpp:16: within this context
```

Norocel PETRACHE

Bibliografie

- ISO/IEC 14882 – Programming Languages – C++
- Bjarne Stroustrup - The C++ Programming Language, ed. III
- Scott Meyers – Effective C++
- Scott Meyers – More Effective C++