Tutoriat 5

Virtual

Virtual este un cuvânt cheie care a apărut pentru a rezolva multe din problemele din C++ legate de *moștenire*.

Acest cuvânt cheie este folosit în fața unei metode și înseamnă că dacă acea metodă va fi rescrisă într-o clasă derivată, în momentul realizării upcasting-ului, metoda apelată va fi cea din clasa derivată.

Exemplu

```
1 class A {
2   public:
3     void f1() { cout << "A::f1()"; }
4     virtual void f2() { cout << "A::f2()"; }
5 };
6
7 class B : public A {
8   public:
9      void f1() { cout << "B::f1()"; }
10     void f2() { cout << "B::f2()"; }
11 };
12
13 int main() {
14   A* a = new B;
15   a -> f1(); // A::f1()
16   a -> f2(); // B::f2()
17 }
```

În situația asta, *virtual* ajută la apelarea metodei din clasa derivată după upcasting, nu invers cum s-ar fi întâmplat fără.

Obs. *Virtual* este folosit în clasele de bază, pentru că el ajută la moştenire. Nu este recomandată folosirea *virtual* fără moştenire, fiindcă îngreunează citirea codului. Totuşi, nu este interzis acest lucru.

Destructor virtual

O altă problemă rezolvată de *virtual* se referă tot la *upcasting*. Să urmărim exemplul:

```
1 class A {
2   public:
3     ~A() { cout << "~A()"; }
4 };
5
6 class B : public A {
7   public:
8      ~B() { cout << "~B()"; }
9 };
10
11 int main() {
12   A* a = new B;
13   delete a; // ~A()
14 }
```

Fără virtual (wrong way)

```
1 class A {
2   public:
3     virtual ~A() { cout << "~A() "; }
4 };
5
6 class B : public A {
7   public:
8      ~B() { cout << "~B() "; }
9 };
10
11 int main() {
12   A* a = new B;
13   delete a; // ~B() ~A()
14 }
```

Cu virtual (right way)

Putem observa în exemplul din stânga, că la distrugerea pointerului se apelează doar *destructorul* clasei de bază. Asta înseamnă că nu se distruge decât memoria ocupată de clasa A. Astfel, încă va rămâne în memorie, o porţiune alocată pentru clasa B.

Această problemă e rezolvată uşor cu un *destructor virtual* declarat în *clasa de bază*. Acum, se va apela întâi destructorul clasei derivate, asigurându-se distrugerea obiectului de tip clasă derivată, iar abia apoi se apelează destructorul clasei de bază.

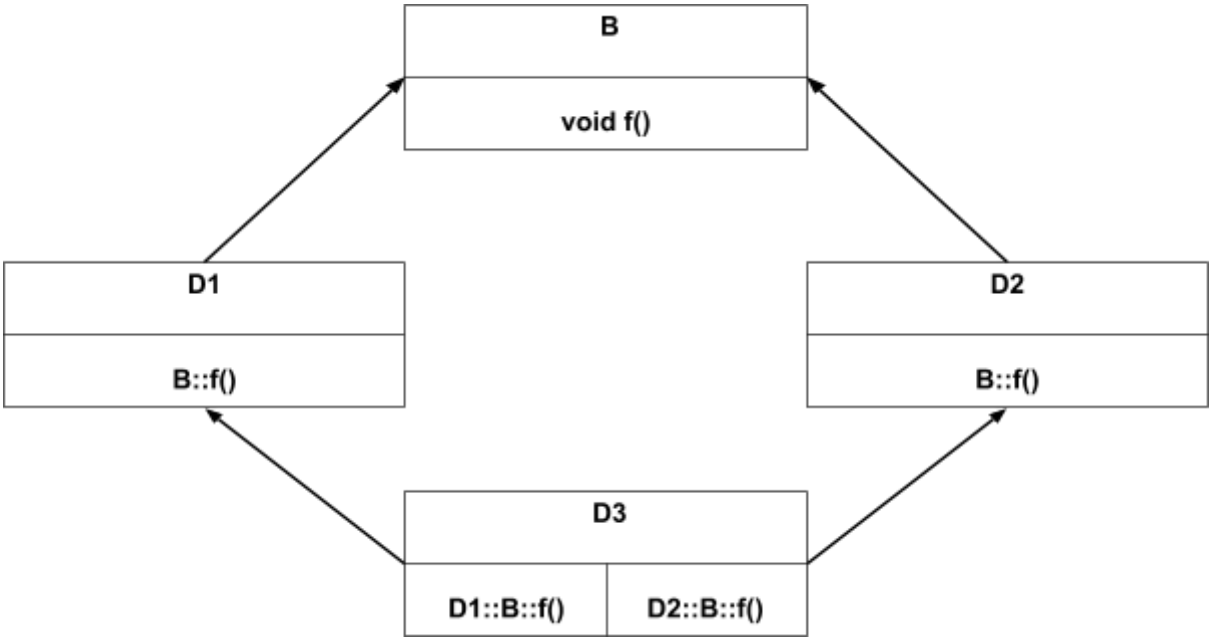
Moştenire virtuală

Urmează să vorbim despre una dintre cele mai clasice probleme cauzate de moştenire. Aceasta se numeşte ***Problema Diamantului.***

```
1 class B {
2   public:
3     void f() { cout << "f()"; }
4 };
5
6 class D1 : public B { };
7 class D2 : public B { };
8
9 class D3 : public D1, public D2 { };
10
11 int main() {
12   D3 d3;
13   d3.f(); // eroare
14 }
```

De ce apare această problemă?

Haideţi să urmărim lanţul de moşteniri:



În momentul moştenirii multiple, clasa *D3* va avea acces la metoda f, atât de pe ramura moştenirii clasei *D1,* cât şi a clasei *D2.* Din acest motiv, compilatorul nu va şti să distingă pe care metodă f dorim să o apelăm.

O primă rezolvare intuitivă ar fi folosirea operatorului de rezoluţie. Totuşi, aceasta complică puţin citirea codului.

Altă rezolvare implică utilizarea *moştenirii virtuale*. Vom vorbi în detaliu despre ambele metode mai jos:

```
1 class B {
2   public:
3     void f() { cout << "f()"; }
4 };
5
6 class D1 : public B { };
7 class D2 : public B { };
8
9 class D3 : public D1, public D2 { };
10
11 int main() {
12   D3 d3;
13
14   d3.D1::f(); // f()
15   d3.D2::f(); // f()
16 }
```

Fig. 1 (rezoluţie)

```
1 class B {
2   public:
3     void f() { cout << "f()"; }
4 };
5
6 class D1 : virtual public B { };
7 class D2 : virtual public B { };
8
9 class D3 : public D1, public D2 { };
10
11 int main() {
12   D3 d3;
13   d3.f(); // f()
14 }
```

Fig. 2 (virtual)

Ambele modalităţi rezolvă problema de ambiguitate cauzată de moştenirea multiplă.

- ***Operatorul de rezoluţie*** spune explicit pe care ramură vrem să mergem când apelăm metoda f();

- ***Virtual*** asigură faptul că la moştenire, nu se va copia decât o singură dată metoda f*()* din clasa de bază.

Downcasting

Acum că am lămurit care e treaba cu acest *virtual,* trebuie să discutăm mai în detaliu despre procesul de *downcasting.*

El reprezintă opusul *upcasting-ului,* adică trecerea de la un pointer de tipul *clasei de bază* la *clasa derivată.*

Există 2 situaţii care apar la *downcasting:*

Clasă de bază fără metode virtuale

```
class Animal {
   public:
     void sleep() { cout << "Sleep"; }
};

class Dog : public Animal{
   public:
      void bark() { cout << "Bark"; }
};

class Cat : public Animal{
   public:
      void meow() { cout << "Meow"; }
};

int main() {
   Animal* animals[10]; // vector de pointeri
   for (int i = 0; i < 10; ++i) {
     if (i % 2 == 0) {
        animals[i] = new Dog; // upcast
        animals[i] -> sleep(); // correct
     } else {
        animals[i] = new Cat; // upcast
        animals[i] -> sleep(); // correct
     }
   }

   for (int i = 0; i < 10; ++i) {
      if (i % 2 == 0) {
         Dog* d = (Dog*) animals[i]; // downcast
         d -> bark(); // correct
      } else {
         Cat* c = (Cat*) animals[i]; // downcast
         c -> meow(); // correct
      }
   }
}
```

Aceasta este situaţia mai nefavorabilă, pentru că nu putem realiza conversia decât într-un mod, mai nesigur şi neeficient. Astfel, trebuie să ne asigurăm ce se află pe fiecare poziţie, ca să ştim că acea conversie se realizează cu succes ca să nu întâmpinăm erori (de aici acel *i % 2 == 0)*.

În practică, acest lucru nu prea se întâmplă, dar, tot în practică, aproape mereu avem cel puţin o funcţie virtuală în clasa de bază.

Clasa de bază cu metode virtuale

```
class Animal {
   public:
     void sleep() { cout << "Sleep\n"; }
     virtual ~Animal() { }
};

class Dog : public Animal{
   public:
      void bark() { cout << "Bark\n"; }
};

class Cat : public Animal{
   public:
      void meow() { cout << "Meow\n"; }
};

void menu() {
   cout << "\n 1. Cat \n 2. Dog \n";
   cout << "Choose your option: ";
}

int main() {
   Animal* animals[10]; // vector de pointeri
   for (int i = 0; i < 10; ++i) {
      menu(); // afisam meniul, utilizatorul alege ce introduce

      int option;
      cin >> option;

      if (option == 1) {
         animals[i] = new Cat;
      } else {
         animals[i] = new Dog;
      }
   }

   for (int i = 0; i < 10; ++i) {
      if (Dog* d = dynamic_cast<Dog*>(animals[i])) {
         d -> bark();
      } else if (Cat* c = dynamic_cast<Cat*>(animals[i])) {
         c -> meow();
      }
   }
}
```

În acest caz, am considerat destructorul ca fiind o metodă virtuală. După cum putem observa, nu avem cum să ştim pe fiecare poziţie a vectorului ce fel de animal se află. Din cauza asta, nu mai funcţionează cast-ul obişnuit.

Astfel, ne folosim de operatorul *dynamic_cast. Sintaxa:*

```
1 dynamic_cast<tip_catre_care_se_converteste*>(pointer_de_convertit)
```

Acest *dynamic_cast* realizează înainte o verificare dacă pointerul transmis se poate converti la tipul de date specificat. ***Return:***

- 0 , dacă nu se poate realiza conversia;
- *Un pointer de tipul specificat între <>,* dacă se poate realiza.

Datorită verificării, *dynamic_cast* este foarte folosit în practică. Singura lui parte negativă este că necesită ca în *clasa de bază* să se găsească cel puţin o *metodă virtuală.*

Această problemă poate fi însă ocolită uşor, fie prin adăugarea unui destructor virtual (care este o *practică recomandată)* sau prin adăugarea unei metode virtuale care nu ar trebui neapărat să fie virtuale.

Totuşi, vom observa că sunt rare cazurile în proiecte mai mari în care să nu folosim metode virtuale.