Tutoriat 5

Vîlculescu Mihai-Bogdan

5 aprilie 2020

1 Virtual

1.1 Introducere

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

Acest cuvânt cheie poate fi 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:

```
class A
{
public:
    void f1() { cout << "A::f1()"; }
    virtual void f2() { cout << "A::f2()"; }
};

class B : public A
{
```

```
public:
    void f1() { cout << "B::f1()"; }
    void f2() { cout << "B::f2()"; }
};

int main()
{
    A *a = new B; // upcasting
    a->f1();      // A::f1()
    a->f2();      // B::f2()

    return 0;
}
```

Exemplu de virtual

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

Observație:

Virtual este utilizat, în general, în **clasele de bază**, pentru că el ajută la **moştenire**.

NU este recomandată folosirea *virtual* într-o clasă care nu urmează să fie *moştenită*, fiindcă este îngreunată citirea codului.

Totuşi, **NU este interzis** acest lucru.

1.2 Destructor virtual

O altă problemă rezolvată de virtual se referă tot la **up-casting**.

Exemplu:

```cpp
class A {
    public:
        ~A() {
            cout <<
             ↪ "~A()";
        }
};

class B : public A {
    public:
        ~B() {
            cout <<
             ↪ "~B()";
        }
};

int main() {
    A *a = new B;
    delete a; // ~A()
    return 0;
}
```

Destructor nevirtual

```cpp
class A {
    public:
        virtual ~A() {
            cout <<
             ↪ "~A()";
        }
};
class B : public A {
    public:
        ~B() {
            cout <<
             ↪ "~B()";
        }
};

int main() {
    A *a = new B;
    delete a;
     ↪ // ~B() ~A()
    return 0;
}
```

Destructor virtual

STÂNGA (destructor nevirtual)

- La distrugerea pointerului se apelează doar **destructorul clasei de bază**.
- Se distruge doar memoria ocupată de **clasa A**.
- Rămân goluri în memorie (**memory leaks**).

DREAPTA (destructor virtual)

- Se apelează întâi **destructorul clasei derivate**.
- Se distruge obiectul de tipul **clasei derivate (B)**.
- Apoi, se apelează **destructorul clasei de bază**.
- Astfel, este eliberată **întreaga memorie alocată** fără memory leaks.

1.3 Moştenire virtuală

Este folosită pentru un caz particular de moştenire: **moştenire multiplă**. Să urmărim exemplul:

```
#include <iostream>
using namespace std;

class B {
    public:
        void f() { cout << "f()"; }
```

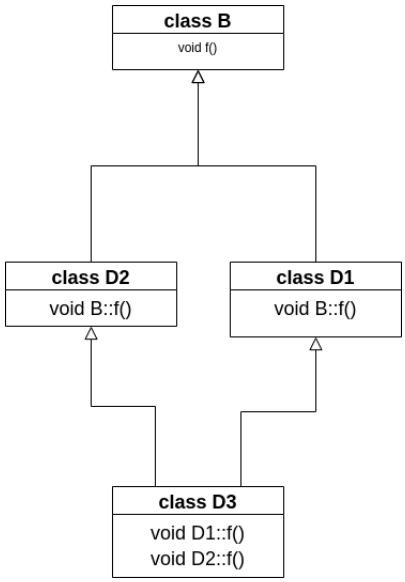
```
};

class D1 : public B {};
class D2 : public B {};
class D3 : public D1, public D2 {};

int main() {
    D3 d3;
    d3.f(); // eroare
    return 0;
}
```

Problema diamantului

Lanţul de moşteniri:



În momentul moștenirii multiple, clasa D3 va avea acces la metoda f, atât de pe **ramura moștenirii clasei D1**, cât și a **clasei D2**.

Din acest motiv, compilatorul nu va ști să distingă pe care metodă f dorim să o apelăm.

Există **2 soluţii posibile**:

```cpp
#include <iostream>
using namespace std;

class B {
    public:
        void f() { cout
        ↪  << "f()"; }
};

class D1 : public B {};
class D2 : public B {};

class D3 : public D1,
 ↪  public D2 {};

int main() {
    D3 d3;
    d3.D1::f(); // f()
    d3.D2::f(); // f()
    return 0;
}
```

Operator de rezoluţie

```cpp
#include <iostream>
using namespace std;

class B {
public:
    void f() { cout <<
     ↪  "f()"; }
};

class D1 : virtual
 ↪  public B {};
class D2 : virtual
 ↪  public B {};
class D3 : public D1,
 ↪  public D2 {};

int main() {
    D3 d3;
    d3.f(); // f()
    return 0;
}
```

Moştenire virtuală

Ambele metode rezolvă **ambiguitatea** cauzată de moştenirea multiplă:

- **Operatorul de rezoluție** spune explicit pe care ramură vrem să mergem când apelăm metoda f();
- **Virtual** asigură faptul că la moștenire, nu se va copia decât o singură dată metoda f() din clasa de bază.

Problema diamantului

- O clasica problema de *OOP* cauzată de moştenirea multiplă.
- Se poate rezolva prin operatorul de rezoluţie. **ATENTIE:** Rezolvarea aceasta **NU** este corectă din punct de vedere OOP.
- Se poate rezolva prin moştenirea virtuală.

Virtual (recap)

- Un **cuvânt cheie** folosit în cadrul **moştenirii**.
- O **metodă virtuală** asigură că este apelată metoda corectă pentru un obiect, indiferent de tipul pointer-ului sau referinţei (**upcasting**).
- **Moştenirea virtuală** este utilă pentru **moştenirea multiplă**, evitând crearea mai multor copii ale aceleiaşi metode sau variabile.

2 Downcasting

2.1 Introducere

Reprezintă trecerea de la un **pointer(sau referinţă)** de tipul *clasei de bază* la unul de tipul *clasei derivate*.

Sunt 2 situaţii de discutat la acest procedeu.

2.2 Clasa 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() {
    vector<Animal*> animals(10);
    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
        }
    }
    return 0;
}
```

Downcasting (fără metode virtuale)

Aceasta este situația mai nefavorabilă, pentru că nu putem realiza conversia decât într-un mod, **mai nesigur**.

Astfel, trebuie să ne asigurăm ce se află pe *fiecare poziție*, ca să știm că acea conversie se realizează cu succes ca să nu întâmpinăm erori (de aici acel **i % 2 == 0**)

În practică, nu prea este folosită această formă de *downcasting*, pentru că, de obicei, există **cel puţin o metodă virtuală în clasa de bază.**

2.3 Clasa de bază cu metode virtuale

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

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

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

int main() {
    vector<Animal*> animals(10);
    for(int i = 0; i < 10; ++i) {
        cout << "\n1. Cat\n 2. Dog\n";
        cout << "Choose your option: ";

        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();
        }
    }
    return 0;
}
```

Downcasting (cu metode virtuale)

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

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

```
dynamic_cast<catre_ce_convertim*>(ce_convertim)
```

Contents