Tutoriat 5
Virtual, moștenire diamant și downcasting

1. Virtual

a. Ce este?
Virtual este un **keyword** care a apărut pentru a rezolva multe din problemele din C++ legate de moștenire.

b. Când se folosește?
În fața unei metode din clasa de bază(dar și la mostenirea claselor – vezi moștenirea diamant) și înseamnă că dacă acea metoda va fi **suprascrisă(overriding)** într-o clasă derivată, în momentul realizării **upcasting-ului**, metoda apelată va fi **cea din clasa derivată**.

Obs:
- **Virtual** este utilizat, în general, în clasele de bază, pentru că el ajută la moștenire.
- Nu este recomandată folosirea lui într-o clasă care nu urmează să fie moștenită, dar totuși, nu este interzis acest lucru.

```
class A
{
public:
    void f1() { cout << "f1 normal din A"<<endl; }
    virtual void f2() { cout << "f2 virtual din A"<<endl; }
};
class B : public A
{
public:
    void f1() { cout << "f1 din B care suprascrie"<<endl; }
    void f2() { cout << "f2 din B care suprascrie virtual din A"<<endl; }
};
int main()
{
    A *a = new B; // upcasting
    a->f1(); // f1 normal din A
    a->f2(); // f2 din B care suprascrie virtual din A
    return 0;
}
```

c. **Destructor virtual** vs destructor obișnuit(doar la realizarea upcasting-ului)

În mod normal, la moștenire, destructorii sunt *apelați de la clasa derivată spre clasa de bază*. Însă, la **upcasting**, la distrugerea pointerului/referinței prin care se realizează upcating-ul, se va apela **doar** destructorul clasei de bază, cel din clasa derivată rămânând neapelat => *memory leaks*.

Dacă destructorul din clasa de bază este declarat ca fiind și virtual, la distrugerea pointerului/referinței prin care s-a reazlizat upcastingul se vor apela ambii destructori în oridinea corectă, evitând memory leaks.

```
// ------------------- Exemplu normal ------------------
class B {
public:
   ~B() {
      cout <<
          "~B()";
   }
};
class D : public B {
public:
   ~D() {
      cout <<
          "~D()";
   }
};
// ---------------------- Exemplu destructor virtual ------------
class BV {
public:
  virtual ~BV() {
      cout <<
          "~BV()";
   }
};
class DV : public BV {
public:
   ~DV() {
      cout <<
          "~DV()";
   }
};
int main() {
   B *p = new D(); // upcasting
   BV *pv = new DV();
   delete p; // ~B()
   cout<<endl;
   delete pv; // ~DV()~BV()
```

```
    return 0;
}
```

2. Moștenirea diamant

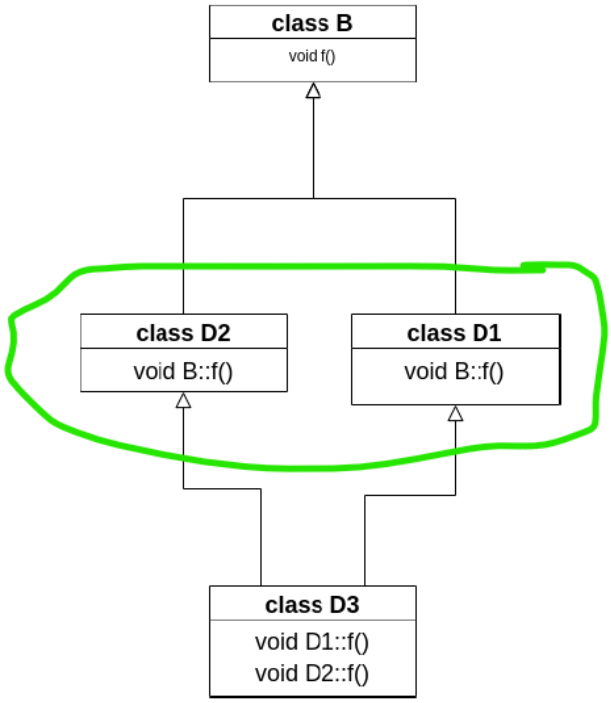
a. Moștenirea virtuală

Este folosită pentru un caz particular de moștenire și anume **moștenirea multiplă**.

Programul de mai jos **nu compliează**, deoarece nu știe care metodă *f()* să apeleze pentru instanța clasei D3(cea venită din D1/ cea venită din D2).

```
class B {
public:
    void f() { cout << "f() din B"; }
};
class D1 : public B {};
class D2 : public B {};
class D3 : public D1, public D2 {};
int main() {
    D3 d3;
    d3.f(); // eroare
    return 0;
}
```

Lanțul de moșteniri pentru programul de mai sus ar arăta așa:



b. Probleme la moștenirea multiplă
   În exemplul de mai sus, în momentul moștenirii multiple, clasa D3 va avea acces la metoda *f()*, atât de pe ramura moștenirii clasei D1, cât și pe ramura clasei D2.

c. Soluții pentru rezolvarea problemei diamantului

   i. Folosirea operatorului de rezoluție(::) împreună cu numele clasei din care folosim metoda f() moștenită, adăugate la apelul metodei (nu respectă principiile POO, deși este posibilă, **nu** este recomandată).

   ii. Folosirea **moștenirii virtuale** (recomandată) : keyword-ul **virtual**(adăugat la moștenirea claselor încercuite cu verde în figura de mai sus) asigură faptul că nu se va copia decât o singură dată metoda *f()* din clasa de bază.

```
class B {
public:
    void f() { cout << "f() din B"; }
};
class D1 : virtual public B {}; //aici
class D2 : virtual public B {}; //aici
class D3 : public D1, public D2 {};
int main() {
    D3 d3;
    d3.f();  // f() din B
    return 0;
}
```

3. Downcasting

a. Ce este?
   Reprezintă trecerea de la un pointer/referință de tipul clasei de bază la unul de tipul clasei derivate.

   Adică?

Transform un obiect de tipul clasei de bază într-un obiect de tipul clasei derivate.

b. Cum se realizează?

- Prin intermediul operatorului **dynamic_cast**:

clasa_derivată * pointer = dynamic_cast<clasa_derivată *>(obiect_clasă_de_bază)

!Obs:

- *Instrucțiunea returnează NULL dacă nu se poate face conversia cu succes.*
- *Este necesar ca în clasa de bază sa existe **cel puțin o metodă virtuală**, altfel utilizarea operatorului va duce la o **eroare de compilare**.*

```
class Animal {
public:
   void sleep() { cout << "Sleep"<<endl; }
   virtual ~Animal() {}
};
class Dog : public Animal {
public:
   void bark() { cout << "Bark"<<endl; }
};
class Cat : public Animal {
public:
   void meow() { cout << "Meow"<<endl; }
};
int main() {
   Animal* animals[2];
   animals[0] = new Cat();
   animals[1] = new Dog();

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

- Același lucru se poate realiza și prin intermediul **operatorului static_cast<>()**, însă este folosit doar în anumite situații. (mai multe puteți citi aici).

- Downcasting fără metode virtuale și fără operatorul dynamic_cast

  Este posibil doar în cazurile în care știm sigur ce tip de obiect se află pe fiecare poziție a vectorului.

```
class Animal {
public:
    void sleep() { cout << "Sleep"<<endl; }
};
class Dog : public Animal {
public:
    void bark() { cout << "Bark"<<endl; }
};
class Cat : public Animal {
public:
    void meow() { cout << "Meow"<<endl; }
};
int main() {
    Animal* animals[2];
    animals[0] = new Dog();
    animals[1] = new Cat();
    for (int i = 0; i < 2; ++i) {
        if (i == 0) {
            Dog* d = (Dog*)animals[i]; // downcast
            d->bark(); // correct
        } else {
            Cat* c = (Cat*)animals[i]; // downcast
            c->meow(); // correct
        }
    }
    return 0;
}
```