Moșteniri. Cuvântul cheie virtual. Clase abstracte. Interfete

Anul Universitar 2023 – 2024, Semestrul II

Programare Orientată pe Obiecte

LABORATOR 6, GRUPA 133

Cuprins

Autor: Wagner Ștefan Daniel

1. Funcții virtuale

Funcțiile virtuale reprezintă unul dintre conceptele fundamentale ale programării orientate pe obiect (OOP) în C++, permițând polimorfismul la execuție (**runtime polymorphism**). Prin intermediul metodelor virtuale, o clasă derivată poate oferi o implementare proprie pentru o funcție definită într-o clasă de bază, și poate apela acea funcție prin intermediul unui pointer/referință către obiect bază, asigurând flexibilitate și extensibilitate în designul software.

Atunci când o funcție este declarată ca virtuală în clasa de bază, aceasta poate fi suprascrisă în orice clasă derivată, oferind o implementare specifică.

De reținut: Mecanismul de funcții virtuale este fundamentul polimorfismului la timpul execuției programului în C++.

1.1. Definirea funcțiilor virtuale

O funcție virtuală este declarată într-o clasă de bază folosind cuvântul cheie ***virtual*** și permite claselor derivate să o "suprascrie" (***override***), oferind o nouă implementare. Atunci când apelăm o funcție virtuală prin intermediul unui pointer sau referință la clasa de bază, mecanismul de runtime decide care versiune a funcției să execute, bazându-se pe tipul obiectului la care pointerul sau referința efectivă pointează, cu ajutorul V-Table/VPTR.

1.2. Exemplu de funcție virtuală

```cpp
#include <iostream>

class Forma {
public:
    virtual void Deseneaza() const
    {
        std::cout << "Desensnd o forma generica.\n";
    }
};


class Cerc : public Forma {
public:
    void Deseneaza() const override
    {
        std::cout << "Desenand un cerc.\n";
    }
};
```

```
int main()
{
    // obiecte derivate memorate drept pointeri catre obiecte tip clasa de baza
(upcasting)
    Forma *forma = new Forma();
    Forma *cerc  = new Cerc();
    // se va afisa: Desensnd o forma generica.
    forma->Deseneaza();
    // se va afisa: Desenand un cerc.
    cerc->Deseneaza();

    delete forma;
    delete cerc;

    // obiecte derivate memorate drept referinte catre obiecte tip clasa de
baza (upcasting)
    Forma& forma2 = Forma();
    Cerc cercTmp = Cerc();
    Forma& cerc2 = cercTmp;

    // se va afisa: Desensnd o forma generica.
    forma2.Deseneaza();
    // se va afisa: Desenand un cerc.
    cerc2.Deseneaza();
    return 0;
}
```

1.3. Funcții virtuale pure

Funcțiile virtuale pure în C++ sunt declarate atribuind **0** funcției virtuale în clasa de bază în antetul metodei. O clasă care conține **cel puțin o funcție virtuală pură** este considerată o **clasă abstractă** și **nu poate fi instanțiată** direct.

O clasă abstractă servește drept clasă de bază pentru alte clase care furnizează implementări concrete pentru funcțiile virtuale pure. Aceasta este o tehnică fundamentală pentru a defini interfețe în C++. Vom vedea ulterior asemănările precum și diferențele dintre o clasă abstractă și o interfață în capitolul 5 din laboratorul curent aici.

1.4. Exemplu de funcție virtuală pură

```
##include <iostream>

class IForma {
public:
    virtual void Deseneaza() const = 0;
};

class Cerc : public IForma {
public:
    void Deseneaza() const override {
        std::cout << "Desenand un cerc.\n";
    }
};

int main()
{
    // apel de functie din derivata prin pointer catre baza
    IForma *forma1 = new Cerc();
    // Afisare: Desenand un cerc.
    forma1->Deseneaza();

    // apel de functie din derivata prin referinta catre baza
    Cerc cerc = Cerc();
    IForma& forma2 = cerc;
    // Afisare: Desenand un cerc.
    forma2.Deseneaza();

    delete forma1;

    return 0;
}
```

2. Explicații VPTR (Virtual Pointer) și V-Table

2.1. Introducere în conceptul de VPTR

În contextul programării orientate pe obiecte în C++, gestionarea corectă a polimorfismului, în special a funcțiilor virtuale, este esențială. Mecanismul de bază care permite C++ să suporte **polimorfismul la runtime** este reprezentat de tabelele virtuale (V-Table), împreună cu pointerii la aceste tabele (VPTR) încorporați în fiecare obiect al claselor care au **cel puțin o funcție virtuală**.

Fiecare clasă care conține cel puțin o funcție virtuală primește un VPTR care indică către V-Table-ul specific clasei. Acest tabel conține adresele funcțiilor virtuale ale clasei, permițând apelul dinamic (la runtime) al metodei corespunzătoare.

2.2. Rolul VPTR și V-table în polimorfism

Când derivăm o clasă dintr-o clasă de bază care conține funcții virtuale, clasa derivată moștenește VPTR-ul clasei de bază, care este ajustat pentru a indica către propriul său V-Table. Aceasta permite ca, chiar dacă avem un pointer de tipul clasei de bază care indică către un obiect al clasei derivate, apelul unei funcții virtuale să fie direcționat către implementarea corectă a clasei derivate la **runtime**, prin intermediul mecanismului de **late-binding**.

2.3. Funcționarea VPTR și V-Table

- La compilare, pentru fiecare clasă care declară sau moștenește funcții virtuale, compilatorul C++ construiește o tabelă virtuală (V-Table). Această tabelă conține adresele funcțiilor virtuale asociate clasei.
- Fiecare instanță a unei astfel de clase conține un pointer ascuns (VPTR) care indică spre V-Table corespunzătoare clasei sale. Astfel, când este invocată o funcție virtuală prin intermediul unui pointer sau al unei referințe la clasă de bază, mecanismul de runtime folosește VPTR pentru a accesa V-Table și pentru a determina adresa corectă a funcției virtuale care trebuie apelată.

Acest mecanism asigură că apelurile funcțiilor virtuale sunt rezolvate corect la runtime prin late-binding, permițând obiectelor să utilizeze implementări specifice clasei derivate, chiar și atunci când sunt accesate prin pointeri sau referințe la o clasă de bază.

2.4. Diagramă ilustrativă VPTR și V-Table

Important: exemplul este preluat de pe https://www.learncpp.com/cpp-tutorial/the-virtual-table/, și vă recomand să îl citiți în intregime.

În diagrama de mai jos, este ilustrat cum fiecare clasă cu cel puțin o funcție virtuală, conține la începutul memoriei obiectului un tablou de pointeri la funcții (***__vptr***), generat de compilator, ascuns by default. Pentru cei curioși, vă recomand un video "C++ Under the Hood" – link aici (Creel pe YouTube).

Clasa ***Base*** conține doua funcții virtuale care returnează implicit ***int*** dacă nu avem tip de date de return menționat în antet (nu folosiți!), ***function1()*** respectiv ***function2()***. Ambele metode au un V-Table asociat în clasa ***Base***, iar clasele ***D1*** și ***D2*** care moștenesc din ***Base*** copiază implcit V-Table-ul părintelui.

Dacă clasa ***D1*** spre exemplu suprascrie (***override***) doar metoda ***function1()***, atunci metoda ***function2()*** va pointa în continuare către metoda din ***Base*** V-Table. Similar, dacă clasa ***D2*** suprascrie doar metoda ***function2()***, atunci metoda ***function1()*** va pointa în continuare catre metoda din ***Base***.

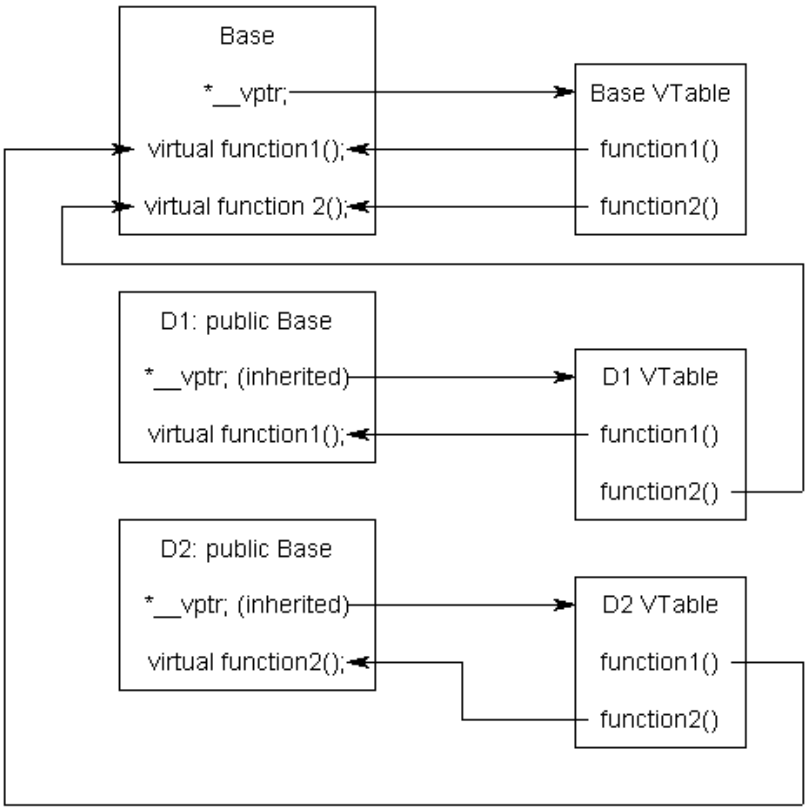

Diagrama preluată de pe https://www.learncpp.com/cpp-tutorial/the-virtual-table/

2.5. Exemplu ilustrativ VPTR și V-Table

```cpp
#include <iostream>

class Baza {
public:
    virtual void Afiseaza() { std::cout << "Afisare din Baza\n"; }
};

class Derivata : public Baza {
public:
    void Afiseaza() override { std::cout << "Afisare din Derivata\n"; }
};

int main()
{
    Baza *baza1 = new Derivata();
    // Utilizeaza VPTR pentru a apela Afiseaza() din Derivata prin pointer Baza
    baza1->Afiseaza();

    Derivata deriv = Derivata();
    Baza& baza2 = deriv;
    // Utilizeaza VPTR pentru a apela Afiseaza() din Derivata prin referinta
Baza
    baza2.Afiseaza();

    delete baza1;

    return 0;
}
```

În exemplul de mai sus, obiectul de tip ***Derivata*** este referențiat prin intermediul unui pointer de tipul clasei de bază ***Baza***. Datorită mecanismului VPTR și V-Table, funcția ***Afiseaza()*** din ***Derivata*** este cea care este apelată.

Afisaeaza() este o funcție virtuală în clasa ***Baza***, care este suprascrisă cu ajutorul cuvântului cheie ***override*** (care, deși poate fi omis, pentru claritate să îl puneți!) în clasa ***Derivata***. Acest lucru permite apelul metodei ***Afisaeaza()*** pe un pointer sau referință de tip ***Baza*** să execute codul din ***Derivata***, dacă obiectul referit este de fapt o instanță a ***Derivata***.

3. Clase abstracte în C++

3.1. Ce reprezintă clasele abstracte în C++?

În Programarea Orientată pe Obiecte în C++, clasele abstracte sunt clase care conțin cel puțin o metodă **virtuală pură (= 0)**, motiv pentru care devin **neinstanțiabile**, dar conțin și implementări de metode, definiții de variabile, sau alte metode non-virtuale.

Clasele abstracte sunt utilizate pentru a defini interfețe în C++, stabilind un "contract" pe care clasele derivate trebuie să-l respecte. Aceasta înseamnă că orice clasă derivată trebuie să furnizeze implementări concrete pentru toate metodele virtuale pure definite în clasa abstractă de bază, pentru a putea deveni clasă instanțiabilă.

3.2. Exemplu implementare clasă abstractă

```cpp
#include <iostream>

// Definirea clasei abstrate – este neinstantiabila (are metoda virtuala pura)
class FormaAbstracta {
private:
    // pot exista date membre in cadrul unei clase abstracte
public:
    // Metoda oarecare - nu mai avem o interfata ci o clasa abstracta
    // Metoda poate fi de asemenea virtuala dar cu corpul implementat
    void AfiseazaInformatii() { std::cout << "Forma\n"; }
    // Functie virtuala pura
    virtual void Deseneaza() const = 0;
};

// Implementarea interfetei
class Cerc : public FormaAbstracta {
public:
    void Deseneaza() const override
    {
        std::cout << "Desenand un cerc.\n";
    }
};

class Patrat : public FormaAbstracta {
public:
    void Deseneaza() const override
    {
        std::cout << "Desenand un patrat.\n";
    }
};
```

```
// Utilizarea clasei abstracte
void DeseneazaForma(const FormaAbstracta& forma) {
    // Polimorfism la executie (runtime)
    forma.Deseneaza();
}

int main() {
    Cerc cerc;
    Patrat patrat;

    cerc.AfiseazaInformatii();
    patrat.AfiseazaInformatii();

    DeseneazaForma(cerc);
    DeseneazaForma(patrat);

    return 0;
}
```

4. Interfețe în C++

4.1. Ce reprezintă o interfață

În C++, conceptul de **interfață** nu este definit explicit în limbaj precum în Java sau C#, dar poate fi realizat prin utilizarea **claselor abstracte complete**.

O **clasă abstractă completă** este o clasă care conține **doar funcții virtuale pure și nicio implementare**. Aceasta servește ca un ”contract” între clasa abstractă completă (interfața) și orice clasă derivată (implementarea), forțând clasele derivate să ofere implementări concrete pentru toate metodele virtuale pure specificate.

Prin definirea unei interfețe în acest mod, C++ permite dezvoltatorilor să specifice un set de metode care trebuiesc implementate explicit de către orice clasă care ”implementează” interfața, garantând astfel că obiectele de tipul claselor derivate pot fi utilizate acolo unde un pointer/referință către interfața este așteptat.

Dacă o clasă derivată implementează doar parțial metodele virtuale pure ale unei interfețe, clasa derivată va deveni clasă abstractă, în continuare neinstanțiabilă, pană când clasele derivate din clasa abstractă obținută implementează toate metodele virtuale pure rămase. Abia atunci, clasele devin instanțiabile.

4.2. Exemplu implementare interfață

```cpp
#include <iostream>

// Definirea interfetei
class IForma {
public:
    // Functie virtuala pura
    virtual void Deseneaza() const = 0;
};
// Implementarea interfetei
class Cerc : public IForma {
public:
    void Deseneaza() const override { std::cout << "Desenand un cerc.\n"; }
};


class Patrat : public IForma {
public:
    void Deseneaza() const override {  std::cout << "Desenand un patrat.\n"; }
};

// Utilizarea interfetei
void DeseneazaForma(const IForma& forma)
{
    // Polimorfism la executie (runtime)
    // Parametrul primit este de tip referinta catre clasa de baza
    // dar se apeleaza functia corecta din clasa derivata trimisa
    forma.Deseneaza();
}

int main() {
    Cerc cerc;
    Patrat patrat;

    // Afisare: Desenand un cerc.
    DeseneazaForma(cerc);
    // Afisare: Desenand un patrat.
    DeseneazaForma(patrat);

    return 0;
}
```

În exemplul de mai sus, ***IForma*** reprezintă interfața care definește contractul pentru orice formă geometrică prin metoda virtuală pură ***Deseneaza()***.

Clasele ***Cerc*** și ***Patrat*** "implementează" această interfață prin furnizarea de implementări concrete pentru metoda ***Deseneaza()***. Funcția ***DeseneazaForma()*** demonstrează utilizarea polimorfică a interfeței, permițând apelarea metodei ***Deseneaza()*** pe obiecte de tipuri diferite care implementează interfața ***IForma***.

5. Asemănări și diferențe între clase abstracte și interfețe

În timp ce ambele concepte sunt folosite pentru a defini interfețe în C++, există câteva diferențe:

- Clasele abstracte pot conține date membre și implementări parțiale (metode virtuale non-pure, sau chiar și metode non-virtuale), în timp ce "interfețele" (realizate ca clase abstracte complete) nu ar trebui să conțină stări/date membre/metode, și au doar funcții virtuale pure.
- O clasă poate moșteni (mai corect spus: implementa) mai multe "interfețe" , oferind flexibilitate în proiectarea software, dar moștenirea multiplă de la clase concrete sau abstracte poate duce la complicații, cum ar fi problema diamantului.

6. Destructori virtuali

6.1. Ce sunt destructorii virtuali și de ce sunt necesari?

Destructorii virtuali sunt necesari când o clasă de bază este moștenită și este posibil să se apeleze ***delete*** pe un pointer al clasei de bază care indică către un obiect al unei clase derivate. Acest lucru asigură că destructorul corect (al clasei derivate) este apelat, permițând o curățare adecvată a resurselor, prevenind memory leaks.

Fără un destructor virtual în clasa de bază, doar destructorul clasei de bază va fi apelat, ignorându-se destructorii claselor derivate. Acest lucru poate duce la memory leaks dacă clasa derivată alocă dinamic resurse.

6.2. Exemplu utilizare destructori virtuali

```
#include <iostream>

class Baza {
public:
    Baza() { std::cout << "Constructor Baza\n"; }
    // Destructor virtual
    virtual ~Baza()
    {
        std::cout << "Destructor Baza\n";
    }
};
```

```
class Derivata : public Baza {
public:
    Derivata() { std::cout << "Constructor Derivata\n"; }
    // Destructor
    ~Derivata() { std::cout << "Destructor Derivata\n"; }
};

int main()
{
    Baza* ptr = new Derivata();

    // Fara destructor virtual in Baza, doar destructorul lui Baza ar fi
apelat. In acest caz, este apelat si destructorul derivatei.
    delete ptr;

    return 0;
}
```

7. Moștenire virtuală

7.1. Despre moștenire virtuală, rolul ei, ce probleme rezolvă

Moștenirea virtuală în C++ este un mecanism esențial utilizat pentru a rezolva problema diamantului, care apare în contextul moștenirii multiple. Problema diamantului se referă la o situație în care o clasă derivată moștenește de la două clase care au o clasă de bază comună. Fără moștenire virtuală, clasa de bază ar fi inclusă de două ori în clasa derivată, ceea ce ar duce la ambiguități și potențiale erori de compilare sau execuție.

Rolul moștenirii virtuale este de a asigura că, în ierarhia de clasă, clasa de bază este inclusă o singură dată, indiferent de câte ori este moștenită indirect prin clase intermediare. Aceasta se realizează prin crearea unei singure instanțe a clasei de bază, pe care toate clasele derivate o împărtășesc. Prin urmare, moștenirea virtuală permite structurarea eficientă a codului și evită duplicarea inutilă a membrilor clasei de bază.

7.2. Exemplu cod moștenire virtuală

```
#include <iostream>

// Clasa de baza comuna
class Baza {
public:
    Baza() { std::cout << "Constructor Baza\n"; }
    virtual void afiseaza() const { std::cout << "Afisare din Baza\n"; }
    virtual ~Baza() { std::cout << "Destructor Baza\n"; }
};

// Prima clasa derivata, mosteneste virtual Baza
```

```
class Derivata1 : virtual public Baza {
public:
    Derivata1() { std::cout << "Constructor Derivata1\n"; }
    void afiseaza() const override { std::cout << "Afisare din Derivata1\n"; }
    ~Derivata1() { std::cout << "Destructor Derivata1\n"; }
};

// A doua clasa derivata, mosteneste virtual Baza
class Derivata2 : virtual public Baza {
public:
    Derivata2() { std::cout << "Constructor Derivata2\n"; }
    void afiseaza() const override { std::cout << "Afisare din Derivata2\n"; }
    ~Derivata2() { std::cout << "Destructor Derivata2\n"; }
};

// Clasa care mosteneste de la ambele clase derivate
class MostenireDubla : public Derivata1, public Derivata2 {
public:
    MostenireDubla() { std::cout << "Constructor MostenireDubla\n"; }
    void afiseaza() const override { std::cout << "Afisare din
MostenireDubla\n"; }
    ~MostenireDubla() { std::cout << "Destructor MostenireDubla\n"; }
};

int main() {
    MostenireDubla obj;
    obj.afiseaza();  // Demonstreaza apelarea metodei corecte
    return 0;
}
```

În acest exemplu, ***MostenireDubla*** moștenește de la ***Derivata1*** și ***Derivata2***, ambele derivând virtual de la ***Baza***. Acest lucru asigură că există o singură instanță a clasei ***Baza*** în ***MostenireDubla***, evitând duplicarea și potențialele probleme asociate cu moștenirea multiplă fără utilizarea moștenirii virtuale.

8. Exercițiu lucrat împreună – clasa *Shape* și derivatele: *Circle, Rectangle, Triangle*

- **Definirea clasei de bază *Shape***

    - Atribute: culoare, poziție etc.
    - Metode (virtuale pure): ***GetArea()***, **GetPerimeter()** pentru afișarea detalilor, etc.
- **Derivarea claselor *Circle, Rectangle, Triangle* din *Shape***
    - Adăugarea de atribute specifice fiecărei clase derivată.
    - Suprascrierea metodelor din clasa de bază în fiecare din clasele derivate cu implementare specifică.