Facultatea de Matematică și Informatică,
Universitatea din București

Tutoriat 5

Programare Orientată pe Obiecte - Informatică, Anul I

Tudor-Gabriel Miu
Radu Tudor Vrînceanu
04-12-2024

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Cuprins

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Despre mosteniri am discutat data trecută (informații și în Seminarul IV). E un concept esențial pentru a înțelege acest tutoriat.

Funcții virtuale

Introducere

Funcțiile virtuale sunt o caracteristică fundamentală a programării orientate pe obiecte în limbajul C++. Ele permit implementarea polimorfismului în cadrul ierarhiilor de clase. O funcție virtuală este o funcție declarată într-o clasă de bază și suprascrisă (override) în clasele derivate.

Principalele caracteristici ale funcțiilor virtuale includ:

- Declararea: Funcțiile virtuale sunt declarate în clasa de bază folosind cuvântul cheie `virtual`. O clasă poate avea mai multe funcții virtuale, care pot fi suprascrise de clasele derivate.
- Suprascrierea: Clasele derivate pot suprascrie funcțiile virtuale declarate în clasa de bază. Suprascrierea se face prin definirea unei funcții cu același nume și semnătură în clasa derivată.
- Polimorfism: Funcțiile virtuale permit polimorfismul în C++. Acest lucru înseamnă că, atunci când apelăm o funcție virtuală printr-un pointer sau o referință la o clasă de bază, se apelează versiunea corespunzătoare a funcției în cadrul clasei derivate. Practic se apelează funcția din clasa „cea mai derivată”.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Iată un exemplu simplu pentru a ilustra utilizarea funcțiilor virtuale în C++:

```cpp
#include <iostream>

class ClasaDeBaza {
public:
    virtual void afisare() {
        std::cout << "Aceasta este o metoda din clasa de baza.\n";
    }
};

class ClasaDerivata : public ClasaDeBaza {
public:
    void afisare() override {
        std::cout << "Aceasta este o metoda din clasa derivata.\n";
    }
};

int main() {
    ClasaDeBaza* pointerBaza = new ClasaDerivata();
    pointerBaza->afisare(); // Apelează funcția afisare() din clasa derivată

    delete pointerBaza;
    return 0;
}
```

În acest exemplu, `ClasaDeBaza` are o funcție virtuală `afisare()`, iar `ClasaDerivata` o suprascrie pentru a afișa un mesaj diferit. Atunci când apelăm `afisare()` prin intermediul unui pointer la clasa de bază care referă un obiect al clasei derivate, se apelează funcția suprascrisă din clasa derivată. Acesta este un exemplu simplu de polimorfism realizat prin funcții virtuale în C++.

Constrângeri

În această secțiune vorbim doar despre funcții virtuale. Funcțiile virtuale trebuie să aibă același antet și în bază, și în derivate. Există o singură excepție de la regulă pe care o discutăm mai târziu.

Există câteva funcții într-o clasă care nu pot fi funcții virtuale:

- constructorii
- funcțiile statice: doar funcțiile membre nestatice pot fi virtuale
- funcțiile friend: același motiv ca mai sus

Pot fi virtuali și operatorii binari, dar în practică nu ne ajută să îi facem virtuali din cauză că trebuie să păstrăm același antet:

- nu am putea primi ca argument un obiect de tip derivat
- nu s-ar păstra simetria între operanzi

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

- nu este nevoie să facem operatorii virtuali ca să apelăm în interiorul lor funcții virtuale

Exemple de operatori pe care nu are rost să îi supraîncărcăm:

- `operator=` și alți operatori de atribuire
- operatori de comparație și de egalitate
- operatori aritmetici și logici

De ce nu doar funcții virtuale?

Declarația tuturor funcțiilor ca fiind virtuale într-o ierarhie de clase ar putea fi o opțiune tentantă la prima vedere, dar ar avea unele consecințe nedorite și ar putea duce la o proiectare și implementare ineficientă a codului.

În Java, toate metodele sunt implicit virtuale, ceea ce înseamnă că comportamentul de rezoluție dinamică a legăturii este implicit activat pentru toate apelurile de metode. Aceasta înseamnă că, în mod implicit, atunci când o clasă derivată suprascrie o metodă definită într-o clasă de bază, metoda suprascrisă este apelată la rulare.

Practic, toate metodele din Java sunt analogul funcțiilor virtuale din C++. Acest lucru este consistent cu principiul de polimorfism în Java și face parte din natura sa de a fi orientat pe obiecte.

Și totuși, de ce nu și în C++? Pentru eficiență (Java e mai lent).

Iată câteva motive pentru care nu este recomandat să declarăm toate funcțiile ca fiind virtuale:

1. **Overheadul de memorie și performanță:** Declarând toate funcțiile ca fiind virtuale, fiecare obiect ar avea nevoie de un pointer suplimentar la o tabelă de funcții virtuale (VTable) pentru a stoca adresele funcțiilor virtuale. Acest lucru ar duce la un consum suplimentar de memorie și la un impact negativ asupra performanței, deoarece ar crește dimensiunea obiectelor și ar putea crește complexitatea execuției codului.
2. **Complexitatea și întreținerea codului:** O ierarhie de clase în care toate funcțiile sunt virtuale ar putea deveni rapid greu de gestionat și de înțeles. Mai mult, modificările în funcționalitatea unei funcții virtuale ar putea avea un impact asupra întregii ierarhii de clase, necesitând modificări extinse și testare suplimentară.

3. **Designul și claritatea codului:** Folosirea funcțiilor virtuale ar trebui să fie justificată de necesitățile specifice ale designului și comportamentului polimorfic. În multe cazuri, funcțiile pot fi definite ca non-virtuale și pot fi suprascrise doar atunci când este necesar polimorfismul.

4. **Semnalează intenția și contractul:** Declarația unei funcții ca fiind virtuală semnalează o intenție clară că acea funcție este supusă polimorfismului și că este destinată să fie suprascrisă în clasele derivate. Declarația tuturor funcțiilor ca fiind virtuale ar putea diminua semnificația acestui contract.

În concluzie, declarația tuturor funcțiilor ca fiind virtuale ar trebui făcută cu grijă și numai atunci când este necesar pentru a atinge obiectivele de proiectare și comportamentul dorit al aplicației. Este important să avem în vedere impactul asupra performanței, complexitatea și claritatea codului atunci când decidem să folosim funcții virtuale într-o ierarhie de clase.

Constructorii și virtualizarea

Constructorii **nu** pot fi declarați ca fiind virtuali în C++. Există mai multe motive pentru aceasta:

1. **Constructorii nu pot fi suprascrisi:** Constructorii sunt speciali în C++. Ei sunt responsabili pentru inițializarea obiectelor și nu pot fi suprascrisi sau mosteniti în aceeasi maniera ca metodele obisnuite.

2. **Ordinea apelării constructorilor:** Atunci când creați un obiect derivat, constructorii clasei de bază sunt apelați automat înainte de constructorul clasei derivate. Acest lucru se întâmplă indiferent dacă constructorul clasei de bază este virtual sau nu. Astfel, chiar dacă constructorul ar fi virtual, comportamentul nu ar fi cel așteptat, deoarece constructorul clasei de bază ar fi apelat oricum înainte de cel al clasei derivate, nici decum în locul lui.

3. **Semnificația constructorilor:** Constructorii au rolul lor specific de a inițializa obiectele și nu sunt concepuți pentru a fi polimorfici. Ideea constructorilor este să ofere un mod eficient și clar de a inițializa obiectele în cadrul ierarhiilor de clase.

4. **Consistența și predictibilitatea:** Permiteți constructorilor să fie virtuali ar duce la o lipsă de coerență și ar putea duce la situații dificile de gestionat și de înțeles în cod. Constructorii sunt folosiți pentru a crea și inițializa obiectele

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

într-un mod predictibil și eficient, iar virtualizarea lor ar putea perturba acest proces.

Destructorii și virtualizarea

În C++, destructorii pot fi declarați ca fiind virtuali. De fapt, în multe situații, este recomandat să declarați destructorul clasei de bază ca fiind virtual atunci când aveți o ierarhie de clase și intenționați să folosiți polimorfismul.

Iată câteva motive pentru a declara destructorii ca fiind virtuali:

1. **Eliberarea resurselor derivate:** Atunci când aveți o ierarhie de clase și folosiți pointeri la obiectele claselor derivate, este important ca destructorul corespunzător să fie apelat în mod corespunzător atunci când obiectele sunt eliminate. Declarația destructorului ca fiind virtual asigură că destructorul corespunzător clasei derivate este apelat atunci când un obiect este distrus prin intermediul unui pointer la clasa de bază.

2. **Prevenirea scurgerilor de memorie:** Dacă nu declarați destructorul clasei de bază ca fiind virtual, este posibil să întâlniți probleme de scurgeri de memorie atunci când eliminați obiecte prin intermediul pointerilor la clase de bază care referă obiecte ale claselor derivate. Astfel, declarația destructorului ca fiind virtual ajută la prevenirea acestor scurgeri de memorie prin asigurarea apelului corect al destructorului clasei derivate.

3. **Polimorfismul și comportamentul dinamic:** Declarația destructorului ca fiind virtual este esențială pentru funcționarea corectă a polimorfismului în C++. Atunci când aveți o ierarhie de clase și utilizați pointeri sau referințe la clasele de bază pentru a accesa obiecte ale claselor derivate, asigurarea apelului corect al destructorului este crucială pentru comportamentul corect al programului și eliberarea corectă a resurselor.

Funcții virtuale pure

Funcțiile virtuale pure sunt funcții virtuale declarate într-o clasă de bază, dar nu au o implementare definită în acea clasă de bază. Ele sunt denumite "pure" deoarece nu conțin o implementare și trebuie să fie implementate în clasele derivate.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Pentru a declara o funcție virtuală pură într-o clasă de bază, folosim sintaxa următoare:

```
class ClasaDeBaza {
public:
   virtual void functieVirtualaPura() = 0;
};
```

Caracteristica cheie a unei funcții virtuale pure este adăugarea " = 0;" la sfârșitul declarației funcției virtuale în clasa de bază.

Clasele care conțin cel puțin o funcție virtuală pură sunt considerate clase abstracte. O clasă abstractă nu poate fi instantiată, adică nu se poate crea un obiect al acelei clase, dar poate fi folosită ca și tip de date pentru pointeri și referințe.

Pentru a deriva o clasă și pentru a implementa o funcție virtuală pură, trebuie să oferim o implementare pentru acea funcție în clasa derivată. Dacă nu o facem, clasa derivată va deveni și ea o clasă abstractă.

Iată un exemplu simplu care ilustrează utilizarea funcțiilor virtuale pure:

```
#include <iostream>


// Clasa abstracta cu functie virtuala pura
class Forma {
public:
   virtual void desenare() const = 0; // Functie virtuala pura
};


// Clasa derivata care implementeaza functia virtuala pura
class Cerc : public Forma {
public:
   void desenare() const override {
       std::cout << "Desenare cerc\n";
   }
};


int main() {
   // Nu se poate instantia un obiect de tipul clasei abstracte Forma
   // Forma forma; // EROARE


   // Putem folosi pointeri sau referinte la clasa abstracta pentru a
accesa obiecte ale claselor derivate
   const Forma* ptrForma = new Cerc();
   ptrForma->desenare(); // Apeleaza metoda desenare() a clasei Cerc


   delete ptrForma;
   return 0;
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
}
```

În acest exemplu, clasa Forma conține o funcție virtuală pură desenare(), iar clasa Cerc o suprascrie pentru a furniza o implementare. Putem folosi pointeri sau referințe la clasa Forma pentru a accesa obiecte ale claselor derivate, cum ar fi Cerc, permițând astfel utilizarea polimorfismului în cadrul codului nostru.

Clase abstracte

Clasele abstracte sunt clase care nu pot fi instanțiate direct, ci sunt concepute pentru a fi folosite ca șabloane pentru alte clase. Ele pot conține unele metode definite (metode implementate) și/sau metode virtuale pure (metode definite, dar fără implementare).

Iată câteva caracteristici cheie ale claselor abstracte:

- **Nu pot fi instanțiate:** Nu puteți crea un obiect al unei clase abstracte. Aceasta înseamnă că nu puteți folosi constructorul său pentru a crea o instanță a clasei.
- **(pot) conține(e) metode virtuale pure:** Clasele abstracte conțin metode virtuale pure, care sunt metode definite, dar fără implementare. Aceste metode trebuie să fie implementate în clasele derivate. Putem să nu avem metode virtuale pure în clasă și ea să fie abstractă: marcăm constructorul sau destructorul ca `protected`. Nu vom mai putea declara obiecte de acel tip, doar de tipuri derivate.
- **Pot conține metode definite:** Clasele abstracte pot conține metode definite, adică metode care au implementare. Acestea sunt metode obișnuite care sunt implementate în clasă.
- **Pot servi ca șabloane:** Clasele abstracte sunt adesea folosite pentru a defini un tip de bază comun pentru o serie de clase derivate. Ele furnizează un șablon comun pentru implementarea anumitor funcționalități, în timp ce permit flexibilitatea prin suprascrierea metodelor virtuale pure în clasele derivate.
- **Pot conține variabile de date:** O clasă abstractă poate conține și variabile de date, în plus față de metode. Acestea pot fi folosite pentru a stoca informații specifice clasei abstracte sau pentru a fi folosite în implementările metodelor.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Clasa din exemplul de mai sus e un exemplu de clasă abstractă pentru că are o funcție virtuală pură.

Destructori virtuali și destructori virtuali puri

În C++, destructorii pot fi declarați ca fiind virtuali puri la fel ca și alte funcții. Un destructor virtual pur este un destructor declarat ca virtual și care este declarat pur virtual prin adăugarea " = 0" la sfârșitul declarației sale.

Declarația unui destructor virtual pur într-o clasă abstractă este utilă atunci când clasa respectivă are resurse pe care trebuie să le elibereze, iar fiecare clasă derivată poate avea implementări specifice pentru eliberarea acestor resurse.

Iată un exemplu de utilizare a unui destructor virtual pur:

```
#include <iostream>

// Clasa abstracta cu destructor virtual pur
class Forma {
public:
    virtual ~Forma() = 0;
    virtual void desenare() const = 0;
};

// Implementarea destructorului virtual pur (o "implementare" goală)
Forma::~Forma() {}

// Clasa derivata
class Cerc : public Forma {
public:
    // Implementarea functiei virtuale pure
    void desenare() const override {
        std::cout << "Desenare cerc\n";
    }

    ~Cerc() {
        std::cout << "Destructor Cerc\n";
    }
};

int main() {
    // Nu se poate instantia o clasa abstracta
    // Forma* forma = new Forma(); // EROARE

    // Putem folosi pointeri la clasa abstracta pentru a accesa obiecte ale
claselor derivate
    const Forma* ptrForma = new Cerc();
    ptrForma->desenare(); // Apeleaza metoda desenare() a clasei Cerc

    delete ptrForma;
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
    return 0;
}
```

Folosim destructor virtual doar dacă avem nevoie și de alte funcții virtuale. Nu este obligatoriu să facem destructorii virtuali, chiar dacă facem moșteniri!

Aceste remarci au condus la următoarea convenție: destructorul ar trebui să fie public și virtual sau protected și non-virtual (in cazurile in care clasa e abstracta).

Să luăm pe rând cele două cazuri:

Destructor public și virtual

Să ne amintim ce fac operatorii `new` și `delete`:

- `new` apelează `malloc` pentru a aloca dinamic o zonă de memorie, apoi apelează constructorul
- `delete` apelează destructorul, apoi apelează `free` pentru a elibera zona de memorie

Avem nevoie să facem destructorul virtual dacă avem nevoie să alocăm dinamic obiecte din clase derivate la care să ne referim prin pointeri de bază:

Nevirtual:

```
#include <iostream>

class BazaNV {
public:
    ~BazaNV() { std::cout << "Destructor BazaNV\n"; }
};

class DerivataNV : public BazaNV {
public:
    ~DerivataNV() { std::cout << "Destructor DerivataNV\n"; }
};

int main() {
    BazaNV* ptrBazaNV = new BazaNV();
    delete ptrBazaNV;

    std::cout << "-----\n";

    BazaNV* ptrDerivataNV = new DerivataNV();
    delete ptrDerivataNV;

    return 0;
}
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Virtual:

```cpp
#include <iostream>

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

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

int main() {
    Baza* ptrBaza = new Baza();
    delete ptrBaza;

    std::cout << "-----\n";

    Baza* ptrDerivata = new Derivata();
    delete ptrDerivata;

    return 0;
}
```

Rulați codul!

Veți observa că în primul caz nu se apelează destructorul pentru derivată (adică aveți un memory leak). În al doilea se vor apela corect.

Pointerii și referințele către bază văd doar funcțiile din bază! Dacă aceste funcții sunt virtuale, se apelează la momentul execuției funcția cea mai derivată a tipului efectiv al obiectului.

Dacă avem o funcție virtuală, am plătit deja costul virtualizării, deci este gratuit să facem și destructorul virtual.

Dacă uităm să facem destructorul virtual, deși ar fi trebuit, nu se apelează toți destructorii! Acest aspect este deosebit de grav dacă în destructorii din derivate eliberăm resurse.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Are sens să facem destructorii virtuali doar dacă avem și alte funcții virtuale. Reciproca nu este adevărată!

„Limbajul ne permite să avem funcții virtuale fără să facem și destructorii virtuali. Totuși, nu văd utilitatea acestei abordări, deoarece nu pot fi reținute decât adresele unor variabile locale și apare foarte ușor riscul de referințe/pointeri agățate/agățați (dangling reference/pointer). Poate avea sens atunci când avem legături între clase în ambele direcții, dar tot mi se pare forțat. Dacă găsiți un exemplu cu sens, vă rog să îmi spuneți și mie.” (Marius Micluță - câteva paragrafe din tutoriatul acesta sunt luate din materialul lui, nu doar acesta)

Destructor protected și non-virtual

Pentru situațiile în care doar vrem să grupăm atribute și funcționalități comune, însă nu avem nevoie de funcții virtuale și am folosi doar clase derivate, avem posibilitatea să nu plătim prețul virtualizării.

Din moment ce nu avem funcții virtuale, nici destructorul din bază nu este nevoie să fie virtual.

Totuși, întrucât nu vrem să construim decât obiecte din clase derivate, destructorul bazei nu trebuie să fie public: dacă destructorul unei clase nu este public, nu avem voie să construim obiecte din acea clasă, deoarece resursele asociate unui astfel de obiect nu ar putea fi eliberate.

Destructorul clasei de bază nu poate fi privat, deoarece trebuie apelat de clasele derivate. Prin urmare, destructorul din bază trebuie să fie protected. Dacă suntem paranoici, putem face protected și constructorii din bază.

Downcasting

Downcasting-ul în C++ se referă la convertirea unui pointer sau referință la o clasă de bază către o clasă derivată. Este important să se utilizeze downcasting-ul cu precauție, deoarece poate duce la comportamente nedefinite sau la erori la momentul rulării dacă nu este gestionat corect.

Iată un exemplu simplu de downcasting:

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```cpp
#include <iostream>

// Clasa de bază
class Baza {
public:
    virtual ~Baza() {}
};

// Clasa derivată
class Derivata : public Baza {
public:
    void functieDerivata() {
        std::cout << "Functie din clasa Derivata\n";
    }
};

int main() {
    // Creăm un pointer la clasa de bază și îl asignăm unui obiect de tipul Derivata
    Baza* ptrBaza = new Derivata();

    // Efectuăm downcasting
    Derivata* ptrDerivata = dynamic_cast<Derivata*>(ptrBaza);

    // Verificăm dacă downcasting-ul a fost realizat cu succes
    if (ptrDerivata) {
        // Putem apela metodele specifice clasei Derivata
        ptrDerivata->functieDerivata();
    } else {
        std::cout << "Downcasting-ul a eșuat!\n";
    }

    // Eliberăm memoria
    delete ptrBaza;

    return 0;
}
```

În acest exemplu, creăm mai întâi un pointer la clasa de bază Baza și îl asignăm unui obiect de tipul Derivata. Apoi, folosim dynamic_cast pentru a efectua downcasting-ul către clasa Derivata. Dacă downcasting-ul este reușit, putem apela metodele specifice clasei Derivata folosind pointerul rezultat. Este important să folosim dynamic_cast pentru downcasting în locul altor tipuri de casting pentru că acesta oferă verificări suplimentare pentru a evita comportamentele nedefinite sau erorile la rulare.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Casting

dynamic_cast

dynamic_cast este un operator din C++ care este utilizat pentru convertirea pointerilor și referințelor între tipuri de date la rulare. Este utilizat în special pentru a efectua conversii între pointeri la clase în ierarhii de moștenire polimorfică (adică când avem funcții virtuale în clasele respective).

Principalele utilizări ale dynamic_cast includ:

Downcasting: Conversia unui pointer sau a unei referințe la o clasă de bază către o clasă derivată.

Verificarea tipului: Determinarea dacă un pointer sau o referință la o clasă de bază arată către un obiect de o anumită clasă derivată.

Conversii între tipuri de date de la o clasă de bază la o clasă derivată în timpul rulării.

Exemplul anterior folosește dynamic_cast

static_cast

static_cast este un operator din limbajul C++ folosit pentru efectuarea unor tipuri de conversii între tipuri de date. Este utilizat în principal pentru conversii care sunt considerate sigure la compilare și care nu necesită verificări suplimentare la rulare.

Principalele utilizări ale static_cast includ:

- Conversii între tipuri numerice, cum ar fi între tipuri întregi și tipuri în virgulă mobilă, precum și între pointeri și întregi de dimensiune diferită.
- Conversii între pointeri la clase în ierarhii de moștenire, cu anumite restricții.
- Conversii între tipuri de referințe, cum ar fi între referințe la clase de bază și clase derivat.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Iată un exemplu simplu de utilizare a static_cast pentru conversii între pointeri la clase:

```cpp
#include <iostream>

class Baza {
public:
    virtual ~Baza() {}
};

class Derivata : public Baza {
public:
    void functieDerivata() {
        std::cout << "Functie din clasa Derivata\n";
    }
};

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

    // Efectuăm conversia statică de la Baza* la Derivata*
    Derivata* ptrDerivata = static_cast<Derivata*>(ptrBaza);

    // Apelăm o metodă specifică clasei Derivata
    ptrDerivata->functieDerivata();

    delete ptrBaza;

    return 0;
}
```

reinterpret_cast

reinterpret_cast este un operator de tip din limbajul C++ care permite conversia între tipuri de date care nu sunt implicit convertibile. Este cel mai puternic și periculos tip de cast, deoarece nu face verificări de siguranță la compilare sau la rulare și poate duce la comportamente nedefinite sau la erori.

Principalele utilizări ale reinterpret_cast includ:

- Conversii între tipuri de pointeri sau referințe, care sunt complet diferite, cum ar fi între un pointer la un obiect și un pointer la un alt tip de obiect, între pointeri la obiecte și pointeri la funcții, între pointeri la obiecte și pointeri la întregi, etc.
- Conversii între tipuri întregi și pointeri, cum ar fi între adrese de memorie și valori întregi.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

- Conversii între tipuri de pointeri și pointeri la void.

Fiind atât de puternic, reinterpret_cast trebuie folosit cu precauție și doar atunci când este absolut necesar, deoarece poate face codul dificil de înțeles și poate duce la comportamente nedefinite dacă este utilizat incorect.

Iată un exemplu simplu de utilizare a reinterpret_cast pentru a converti un pointer la int într-un pointer la char:

```
int x = 65; // Codul ASCII pentru litera 'A'
char* ptrChar = reinterpret_cast<char*>(&x);

std::cout << *ptrChar; // Va afișa litera 'A'
```

const_cast

const_cast este un operator din limbajul C++ folosit pentru a elimina sau a adăuga calificatori const și volatile unui obiect. Acesta este utilizat pentru a schimba constanța unui obiect, permițând modificările asupra acelui obiect, sau pentru a adăuga constanță unui obiect, împiedicând modificările acestuia în anumite contexte.

Principalele utilizări ale const_cast includ:

Eliminarea constanței: Acest lucru este util atunci când avem un pointer sau o referință la un obiect constant și dorim să efectuăm modificări asupra acestui obiect.

Adăugarea constanței: Acest lucru este util atunci când dorim să prevenim modificările asupra unui obiect într-un context specific, cum ar fi în transmiterea unui obiect constant către o funcție care nu ar trebui să îl modifice.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Iată un exemplu simplu de utilizare a const_cast pentru a elimina constanța unui pointer la un obiect constant:

```cpp
#include <iostream>

int main() {
    const int x = 5;

    // Eliminăm constanța lui x pentru a permite modificările
    int* ptr = const_cast<int*>(&x);

    // Modificăm valoarea lui x
    *ptr = 10;

    // Afisăm noua valoare a lui x
    std::cout << "Noua valoare a lui x: " << x << std::endl;

    return 0;
}
```

O resursă bună ca să mai lucrăm la finalul tutoriatului (pe model de examen, s-ar putea ca nu toate erorile din aceste exemple sa se poata corecta intr-o singura linie): FMI/Year 1 Sem 2/pooExamQuestions/probleme at master · mihainsto/FMI (github.com)