PROGRAMARE ORIENTATĂ PE OBIECTE

LABORATOR 9

Anul universitar 2023 – 2024

Semestrul II

1 Noțiuni teoretice

1.1 Mecanisme intermediare în ierarhiile de clase

1.1.1 Comportamentul în ierarhiile de clase

Pe măsură ce lucrăm cu ierarhii de clase, observăm că simpla moștenire nu este mereu suficientă. Uneori avem nevoie să tratăm obiecte diferite într-un mod comun, alteori vrem să revenim la funcționalități specifice unei clase, sau ne confruntăm cu metode care par la fel, dar se comportă diferit. Toate aceste situații apar natural când lucrăm cu clase înrudite și ne pregătesc pentru un mod mai flexibil de a gestiona comportamentul obiectelor în funcție de cum sunt construite, nu doar de cum sunt declarate.

În sistemele bazate pe moștenire, putem grupa mai multe tipuri de obiecte sub o clasă comună, tratându-le unitar. Acest lucru este util pentru a organiza codul, a evita redundanța și a lucra cu funcții care operează generic pe obiecte similare. Totuși, pe măsură ce ierarhiile devin mai complexe, apare o nevoie esențială: deși obiectele împărtășesc o structură comună, ele trebuie să se comporte diferit în anumite situații. De exemplu, vrem ca același apel de funcție să ducă la reacții specifice pentru fiecare tip de obiect real, chiar dacă îl apelăm printr-un tip general. Pentru a răspunde acestei nevoi de flexibilitate și adaptare la tipul concret al obiectului, apare conceptul de *polimorfism*, despre care vom vorbi în laboratoarele următoare.

1.1.2 Tipuri de mecanisme

Într-un sistem bazat pe moștenire, upcasting-ul și downcasting-ul sunt două mecanisme importante care permit lucrul flexibil cu obiecte dintr-o ierarhie de clase, facilitând astfel dezvoltarea de cod generalizat, reutilizabil și ușor extensibil. Aceste conversii sunt frecvent utilizate în situații în care obiectele derivate trebuie manipulate printr-o interfață comună — de exemplu, într-o colecție de obiecte eterogene, într-o funcție care primește parametri de tip bază, sau în implementarea unor modele de design precum Factory, Strategy sau Visitor. Totodată, ele permit trecerea controlată de la abstractizare (prin upcasting) la comportamente specifice (prin downcasting), ceea ce este esențial în aplicații care necesită atât generalitate cât și acces la funcționalități concrete, cum ar fi sistemele de gestiune, framework-urile grafice sau motoarele de jocuri.

În practica dezvoltării software, upcasting-ul și downcasting-ul sunt mecanisme fundamentale care permit scrierea de cod flexibil, extensibil și reutilizabil în contexte unde este necesară manipularea

obiectelor prin intermediul unei ierarhii de clase. Aceste conversii sunt întâlnite frecvent în aplicații care folosesc principii de proiectare precum polimorfismul, injecția de dependență, sau colecții de obiecte eterogene (ex. containere STL în C++). Prin upcasting, programatorii pot trata un grup de obiecte derivate printr-un pointer sau o referință la clasa de bază comună, facilitând astfel dezvoltarea modulară și orientată pe interfețe. Downcasting-ul, deși mai rar și mai riscant, este util în situații în care se impune accesul la funcționalități specifice ale unei clase derivate, după ce obiectul a fost manipulat generic. Aceste mecanisme contribuie direct la implementarea unor modele de design consacrate, precum Factory, Visitor, sau Composite, în care relațiile ierarhice și comportamentele specializate joacă un rol central.

În programarea orientată pe obiecte, conceptele de upcasting și downcasting sunt esențiale pentru gestionarea relațiilor dintre clasele aflate într-o ierarhie de moștenire. Upcasting-ul este utilizat pentru a trata obiecte de clase derivate printr-o interfață comună definită în clasa de bază. Acest mecanism permite scrierea de cod generalizat, care funcționează cu mai multe tipuri de obiecte, fără a fi nevoie să cunoască detaliile fiecărei clase derivate. Este frecvent folosit în contexte precum colecții de obiecte eterogene, apeluri către funcții care operează pe tipuri generice, sau pentru a respecta principiul „programării împotriva interfeței, nu implementării”.

Pe de altă parte, downcasting-ul este necesar atunci când, având un pointer sau o referință la o clasă de bază, programatorul dorește să acceseze comportamente sau membri specifici unei clase derivate. Deși este mai rar folosit și implică riscuri (în special fără mecanisme de verificare a tipului la runtime), downcasting-ul oferă flexibilitatea de a reveni la funcționalitatea specializată a clasei derivate, acolo unde este necesar. Prin urmare, cele două tipuri de conversii reflectă echilibrul dintre abstractizare și specificitate, contribuind la reutilizarea codului și la extinderea comportamentului aplicației într-un mod controlat.

Shadowing-ul este un fenomen care apare în ierarhiile de clase atunci când un membru (atribut sau metodă) definit într-o clasă derivată poartă același nume ca un membru din clasa de bază. În astfel de cazuri, membrul din clasa derivată ascunde (maschează) membrul moștenit, chiar dacă acesta continuă să existe în obiect. Accesul direct la membrul mascat se va referi întotdeauna la versiunea locală, iar pentru a utiliza explicit varianta din clasa de bază este necesar să folosim calificatori de tipul `A::nume`. Shadowing-ul evidențiază o limitare importantă a moștenirii: simpla definire a unui element cu același nume nu înlocuiește comportamentul original, ci doar îl ascunde, ceea ce poate duce la confuzii sau erori dacă nu este înțeles corect.

1.1.2.1 Upcasting

Upcasting apare atunci când un pointer sau o referință către o clasă derivată este convertit într-un pointer sau o referință către clasa de bază. Upcasting-ul este implicit și sigur în C++. Dacă o clasă este derivată

dintr-o altă clasă de bază (relație „is a") ⇒ obiectele de tipul clasei derivate sunt, în același timp, și obiecte de tipul clasei de bază. Acest fapt ne permite următoarea atribuire:

```
obiect_clasa_baza = obiect_clasa_derivata (upcasting, conversie la tipul obiectului de bază)
```

Dacă facem acest lucru (`obiect_clasa_baza = obiect_clasa_derivata`), vom pierde informațiile suplimentare stocate în obiect_clasa_derivata. Este permisă atribuirea în care se „pierd date". Nu sunt permise cele în care nu se știe cu ce să se completeze câmpurile suplimentare (`obiect_clasa_derivata = obiect_clasa_baza`).

Putem face acest lucru posibil (`obiect_clasa_derivata = obiect_clasa_baza`) doar dacă supradefinim operatorul de atribuire din clasa derivată.

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

class Animal {
public:
    void vorbeste() {
        cout << "Animalul face un sunet.\n";
    }
};

class Caine : public Animal {
public:
    void latra() {
        cout << "Cainele latra: Ham ham!\n";
    }
};

int main() {
    Caine c;

    // upcasting: tratam un Caine ca un Animal
    Animal* a = &c;
    a->vorbeste();

    // upcasting prin copiere cu slicing
    Caine c1;
    Animal a1 = c1;
    a1.vorbeste();

    return 0;
}
```

Chiar dacă `a` pointează către un obiect de tip `Caine`, tipul său este `Animal*`. Prin urmare, se poate apela doar metoda definită în clasa `Animal`, iar metoda `latra()` nu este accesibilă. Acest comportament reflectă faptul că, prin upcasting, pierdem accesul la comportamentele specifice clasei derivate.

Upcasting prin pointer:

- Se creează un obiect `Caine c`.
- Se face upcasting implicit: `Animal* a = &c;`
- Pointerul a este de tip `Animal*`, dar pointează către un `Caine`.
- Se apelează metoda `vorbeste()` → OK, este disponibilă în `Animal`.

Upcasting prin copiere (cu slicing):

- Se creează un nou obiect `Caine c1`.
- Se face copiere într-un obiect `Animal a1 = c1;`
- Aici apare fenomenul de object slicing:
  - Obiectul `a1` este doar partea de `Animal` din `Caine`.
  - Informațiile specifice clasei `Caine` (ex: `latra()`) sunt pierdute.

Se apelează `a1.vorbeste();` – funcționează, dar:

- Nu mai este acces la nimic din `Caine`.
- Nu există legătură cu obiectul original după slicing.

Object slicing apare atunci când un obiect al unei clase derivate este copiat într-un obiect al clasei de bază. În timpul copierii, se păstrează doar partea corespunzătoare clasei de bază, iar restul – specific clasei derivate – se pierde.

Upcasting prin pointer este sigur și util, mai ales când se introduc funcții virtuale. Upcasting prin copiere duce la slicing, ceea ce înseamnă se pierd funcționalitățile derivatei.

1.1.2.2 Downcasting

Downcasting-ul este procesul prin care un pointer sau o referință la o clasă de bază este convertit într-un pointer sau referință către o clasă derivată. Acest tip de conversie nu este implicit sigur, deoarece nu orice obiect de tipul clasei de bază este și un obiect valid al clasei derivate. Astfel, downcasting-ul implică o asumare din partea programatorului că tipul real al obiectului este cel așteptat. În lipsa unei verificări explicite, această operație poate conduce la comportamente nedefinite sau erori de execuție. Downcasting-ul este folosit în mod obișnuit în situații în care un obiect a fost anterior convertit în tipul clasei de bază (prin upcasting) și se dorește revenirea la comportamentul specializat al clasei derivate.

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

class Animal {
public:
    void vorbeste() {
        cout << "Animalul face un sunet.\n";
```

```
    }
};

class Caine : public Animal {
public:
    void latra() {
        cout << "Cainele latra: Ham ham!\n";
    }
};

int main() {
    Animal* a = new Caine;

    // downcasting: incercam sa revenim la tipul derivat
    Caine* c = (Caine*)a;

    c->latra();

    delete a;
    return 0;
}
```

Pointerul `a` este de tip `Animal*`, dar pointează de fapt către un obiect de tip `Caine`. Prin downcasting (explicit), convertim `a` înapoi în `Caine*`, ceea ce ne permite să apelăm metoda specifică `latra()`. Conversia funcționează corect în acest caz, deoarece obiectul real este într-adevăr un `Caine`.

Dacă `a` pointează către un obiect care **nu este** de tip `Caine`, conversia este incorectă, dar compilatorul nu va da eroare. Apelul va duce la comportament nedefinit (poate chiar o eroare de execuție). De aceea, acest tip de conversie se folosește doar dacă suntem siguri de tipul real al obiectului.

1.1.2.3 Shadowing

În cadrul moștenirii, pot apărea situații în care o clasă derivată definește o metodă sau un atribut cu același nume ca în clasa de bază. Această redefinire nu înlocuiește comportamentul original, ci îl ascunde, iar accesul la versiunea din baza trebuie realizat explicit. Acest fenomen se numește shadowing și evidențiază o limitare importantă: atunci când un obiect este tratat printr-un pointer sau o referință la clasa de bază, metoda apelată va fi întotdeauna cea definită în acea clasă, chiar dacă obiectul este, în realitate, de tip derivat. Astfel, shadowing-ul scoate la iveală faptul că moștenirea, de una singură, nu permite adaptarea automată a comportamentului în funcție de tipul concret al obiectului, ci doar oferă o structură comună. Pentru a permite un control mai fin asupra comportamentului la nivelul ierarhiei, este necesar un mecanism care să rezolve această rigiditate.

Shadowing apare atunci când o variabilă locală sau un membru al unei clase are același nume cu o altă variabilă sau membru într-un context mai larg. În general, shadowing-ul este o practică nedorită,

deoarece poate crea confuzie și poate face codul mai dificil de înțeles. Când o variabilă sau membru este umbrit, accesul la variabila sau membrul mai larg devine nedisponibil în cadrul contextului de umbrire.

```
class A {
    protected:
        int atr;
};
class B : public A {
    private:
        //am mostenit si atributul ,,atr" din clasa A
        int atr; // are priotate fata de atributul cu acelasi nume din A
    public:
        void set_atr (int i, int j) {
            A::atr = i; //pentru a avea acces la atributul ,,atr" din A
                        //il apelam folosind A::
            atr = j;
        }
};
```

În exemplul prezentat, clasa B moștenește de la A atributul `atr`, dar definește și un atribut propriu cu același nume (`int atr;`). Astfel, atributul din clasa derivată maschează atributul moștenit din clasa de bază. Aceasta înseamnă că, în interiorul clasei B, accesul direct la atr se va referi întotdeauna la cel local, nu la cel moștenit. Pentru a accesa atributul original din A, este necesar să folosim calificatorul `A::atr`. Acest tip de shadowing este o consecință directă a regulii că numele declarate local au prioritate, și servește drept exemplu de cum moștenirea nu înseamnă întotdeauna substituție completă între clasele implicate.

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

class A {
protected:
    int atr;

public:
    void afisare() {
        cout << "A::atr = " << atr << endl;
    }
};

class B : public A {
private:
    // am mostenit și atributul "atr" din clasa A
    // acest atr mascheaza (shadow) pe cel din A
    int atr;

public:
    void set_atr(int i, int j) {
        // accesam explicit atr din A
```

```
        A::atr = i;
        // acesta este atr local din B
        atr = j;
    }

    void afisare() {
        // mascheaza metoda afisare din A
        cout << "B::atr = " << atr << endl;
    }
};

int main() {
    B obj;
    obj.set_atr(10, 20);

    cout << "Apel direct B:" << endl;
    // apelează metoda din B
    obj.afisare();

    cout << "Apel prin pointer la A (shadowing): " << endl;
    A* p = &obj;
    // apeleaza metoda din A (shadowing daca nu e virtuala)
    p->afisare();

    return 0;
}
```

Clasa B definește o metodă `afisare()` cu același nume ca în clasa de bază A, dar fără să o înlocuiască formal (nu este declarată virtual). În acest caz, metoda din B maschează pur și simplu pe cea din A. Atunci când obiectul este accesat direct printr-o instanță de tip B, se va apela metoda `afisare()` din B, dar dacă același obiect este tratat printr-un pointer sau referință la A, se va apela metoda din A, deoarece alegerea metodei se face în funcție de tipul pointerului, nu de tipul real al obiectului. Acest comportament evidențiază limitarea legării statice în ierarhiile de clase și arată că redefinirea unei metode în derivată nu este suficientă pentru a o face substituibilă în toate cazurile.

1.2 Funcții virtuale

O funcție virtuală este o metodă declarată în clasa de bază folosind cuvântul cheie virtual, care permite ca obiectele claselor derivate să își definească propria versiune a acelei metode. Scopul funcției virtuale este ca, atunci când un obiect derivat este tratat printr-un pointer sau o referință la clasa de bază, apelul metodei să se refere totuși la comportamentul specific al obiectului real. Astfel, metodele pot fi apelate printr-un tip general (clasa de bază), dar să se comporte corect în funcție de tipul concret (clasa derivată) al obiectului.

Funcțiile virtuale se folosesc în special împreună cu moștenirea, când mai multe clase au un comportament comun, dar unele dintre ele trebuie să-l modifice sau să-l specializeze. Fără virtual,

metoda apelată va fi întotdeauna cea din clasa de bază, chiar dacă obiectul este de fapt unul derivat. Acest lucru poate duce la situații în care codul pare corect, dar comportamentul este incorect sau neașteptat. Funcțiile virtuale rezolvă această problemă, permițând obiectelor să răspundă corect la apeluri, chiar și atunci când sunt tratate într-un mod generalizat.

În C++, o funcție virtuală se declară în clasa de bază astfel:

```
class Baza {
public:
    virtual void nume_functie();
};
```

În clasa derivată, funcția se poate suprascrie astfel:

```
class Derivata : public Baza {
public:
    void nume_functie() override;
};
```

Este important de reținut că funcțiile virtuale nu influențează apelurile directe prin obiecte (nu prin pointeri/referințe). Dacă se apelează o metodă virtuală direct pe un obiect, alegerea metodei se face tot static, iar virtual nu are niciun efect în acel caz. De asemenea, în cazul copierii unui obiect derivat într-unul de bază (prin obiect, nu prin pointer), apare fenomenul de object slicing, iar partea derivată se pierde, ceea ce anulează orice posibilitate de comportament specializat.

```
class Animal {
public:
    virtual void vorbeste() {
        cout << "Animalul face un sunet.\n";
    }
};

class Caine : public Animal {
public:
    void vorbeste() override {
        cout << "Cainele latra!\n";
    }
};

int main() {
    Caine c;
    c.vorbeste();
    return 0;
}
```

Funcțiile virtuale își dovedesc utilitatea în momentul în care se face upcasting – adică un obiect derivat este tratat ca un obiect de bază, de obicei printr-un pointer sau o referință. În astfel de cazuri, dacă metoda apelată este virtuală și a fost suprascrisă în clasa derivată, atunci apelul va fi direcționat corect

către implementarea din clasa derivată. Fără virtual, metodele sunt legate static (la compilare), iar apelul se face în funcție de tipul declarat, nu de cel real.

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

class Animal {
public:
    virtual void vorbeste() {
        cout << "Animalul face un sunet generic.\n";
    }
};

class Caine : public Animal {
public:
    void vorbeste() override {
        cout << "Cainele latra: Ham ham!\n";
    }
};

int main() {
    Caine c;
    // upcasting
    Animal* a = &c;

    // apel dynamic: "Cainele latra: Ham ham!"
    a->vorbeste();
    return 0;
}
```

Astfel, funcțiile virtuale sunt fundamentale pentru a permite flexibilitate și extensibilitate în ierarhiile de clase, făcând posibil ca metodele să se comporte corect în funcție de tipul real al obiectului, chiar și atunci când acesta este tratat generic.

1.3 Moștenire și polimorfism

Polimorfismul apare ca soluție naturală la o limitare a moștenirii simple: chiar dacă mai multe clase împărtășesc o structură comună (prin moștenire), comportamentul lor poate fi diferit. Moștenirea permite tratarea obiectelor printr-un tip comun (clasa de bază), dar fără polimorfism, metodele apelate prin pointeri sau referințe la această bază sunt întotdeauna cele ale clasei de bază — chiar dacă obiectul real este dintr-o clasă derivată. Cu alte cuvinte, fără polimorfism, tratăm toate obiectele la fel, chiar dacă ele ar trebui să se comporte diferit.

Polimorfismul răspunde la întrebarea:

„Cum pot apela aceeași funcție pe obiecte diferite, dar să obțin comportamente corecte, specifice fiecărui tip real?”

El permite ca apelurile la metode prin pointeri sau referințe la clasa de bază să fie rezolvate în funcție de tipul real al obiectului, nu doar în funcție de tipul declarării. Acest lucru face ca programele să fie mai flexibile, extensibile și corect structurate, respectând principiile de proiectare orientate pe interfețe, nu pe implementări concrete.

2 Aplicații

2.1 Cazare

Într-un proiect, implementați următoarele clase:

```
class Camera
{
private:
    int nr;
    int capacitate;
    double pret;

public:
    Camera();
    Camera(int, int, double);
    Camera(const Camera&);

    friend istream& operator>>(istream&, Camera&);
    friend ostream& operator<<(ostream&, const Camera&);
    Camera& operator=(const Camera&);
};

class UnitateCazare
{
protected:
    int nrCam;
    Camera* camere;
    char denumire[100];

public:
    UnitateCazare();
    UnitateCazare(int);
    UnitateCazare(int, Camera*, char*);
    UnitateCazare(const UnitateCazare&);

    ~UnitateCazare();

    friend istream& operator>>(istream&, UnitateCazare&);
    friend ostream& operator<<(ostream&, const UnitateCazare&);

    // va fi folosita inainte de upcasting
    /// void clasificare();

    // folosita dupa upcasting
    virtual void clasificare();
};

class Hotel : public UnitateCazare
{
private:
    int stele;
```

```
public:
    Hotel();
    void clasificare() override;
};

class Pensiune : public UnitateCazare
{
private:
    int margarete;

public:
    Pensiune();
    void clasificare() override;
};
```

3 Rezolvări

3.1 Cazare

```
#include <iostream>
#include <cstring>

using namespace std;

class Camera
{
private:
    int nr;
    int capacitate;
    double pret;

public:
    Camera();
    Camera(int, int, double);
    Camera(const Camera&);

    friend istream& operator>>(istream&, Camera&);
    friend ostream& operator<<(ostream&, const Camera&);
    Camera& operator=(const Camera&);
};

class UnitateCazare
{
protected:
    int nrCam;
    Camera* camere;
    char denumire[100];

public:
    UnitateCazare();
    UnitateCazare(int);
    UnitateCazare(int, Camera*, char*);
    UnitateCazare(const UnitateCazare&);

    ~UnitateCazare();

    friend istream& operator>>(istream&, UnitateCazare&);
    friend ostream& operator<<(ostream&, const UnitateCazare&);

    // va fi folosita inainte de upcasting
    /// void clasificare();

    // folosita dupa upcasting
    virtual void clasificare();
};

class Hotel : public UnitateCazare
{
private:
```

```
    int stele;

public:
    Hotel();
    void clasificare() override;
};

class Pensiune : public UnitateCazare
{
private:
    int margarete;

public:
    Pensiune();
    void clasificare() override;
};

Camera::Camera() : nr(0), capacitate(0), pret(0.0) {}
Camera::Camera(int a, int b, double c) : nr(a), capacitate(b), pret(c) {}
Camera::Camera(const Camera& c): nr(c.nr), capacitate(c.capacitate), pret(c.pret) {}

istream& operator>>(istream& in, Camera& c)
{
    cout << "Introduceti datele pentru camera: \n";
    cout << "Numarul: ";
    in >> c.nr;
    cout << "Capacitatea: ";
    in >> c.capacitate;
    cout << "Pretul: ";
    in >> c.pret;
    return in;
}

ostream& operator<<(ostream& out, const Camera& c)
{
    out << "Aceasta este camera cu urmatoarele caracteristici: \n";
    out << "Numar: " << c.nr << "; Capacitate: " << c.capacitate << "; Pret: " << c.pret <<
"\n";
    return out;
}

Camera& Camera::operator=(const Camera& c)
{
    if(this != &c)
    {
        nr = c.nr;
        capacitate = c.capacitate;
        pret = c.pret;
    }
    return *this;
}

UnitateCazare::UnitateCazare() : nrCam(0), camere(nullptr)
```

```
{
    strcpy(denumire, "");
}

UnitateCazare::UnitateCazare(int nr) : nrCam(nr)
{
    camere = new Camera[nr];
    strcpy(denumire, "");
}

UnitateCazare::UnitateCazare(int nr, Camera* c, char d[100]) : nrCam(nr)
{
    camere = new Camera[nr];
    for(int i = 0; i < nr; i++)
    {
        camere[i] = c[i];
    }
    strcpy(denumire, d);
}

UnitateCazare::UnitateCazare(const UnitateCazare& uc) : nrCam(uc.nrCam)
{
    camere = new Camera[uc.nrCam];
    for(int i = 0; i < uc.nrCam; i++)
    {
        camere[i] = uc.camere[i];
    }
    strcpy(denumire, uc.denumire);
}

UnitateCazare::~UnitateCazare()
{
    delete[] camere;
}

istream& operator>>(istream& in, UnitateCazare& uc)
{
    cout << "Introduceti datele pentru unitatea de cazare: \n";
    cout << "Numarul de camere: ";
    in >> uc.nrCam;

    uc.camere = new Camera[uc.nrCam];

    for(int i = 0; i < uc.nrCam; i++)
    {
        cout << "Camera " << i << ": ";
        in >> uc.camere[i];
    }
    cout << "Denumirea: ";
    in >> uc.denumire;
    return in;
}
```

```
ostream& operator<<(ostream& out, const UnitateCazare& uc)
{
    out << "Unitatea de cazare " << uc.denumire << " are urmatoarele caracteristici: \n";
    out << "- numar de camere: " << uc.nrCam << "\n";
    out << "- camerele: \n";
    for(int i = 0; i < uc.nrCam; i++)
    {
        out << " * camera " << i << ": " << uc.camere[i];
    }
    out << "\n";
    return out;
}

void UnitateCazare::clasificare()
{
    cout << "Unitate de cazare generica \n";
}

Hotel::Hotel()
{
    cout << "Numarul de stele este: ";
    cin >> stele;
}

void Hotel::clasificare()
{
    cout << "Numarul de stele al hotelului este: " << stele << "\n";
}

Pensiune::Pensiune()
{
    cout << "Numarul de margarete este: ";
    cin >> margarete;
}

void Pensiune::clasificare()
{
    cout << "Numarul de margarete ale pensiunii este: " << margarete << "\n";
}

int main()
{

    Hotel h;
    Pensiune p;

    cin >> h;
    cout << h;

    cin >> p;
    cout << p;

    // object slicing hotel
```

```
    UnitateCazare uh1 = h;
    uh1.clasificare();

    // shadowing inainte de virtual hotel
    UnitateCazare* uh2 = &h;
    uh2->clasificare();

    // upcasting dupa virtual hotel
    UnitateCazare* uh3 = &h;
    uh3->clasificare();

    // upcasting dupa virtual hotel
    UnitateCazare* uh4 = new Hotel;
    uh4->clasificare();

    delete uh4;

    // object slicing pensiune
    UnitateCazare up1 = p;
    up1.clasificare();

    // shadowing inainte de virtual pensiune
    UnitateCazare* up2 = &p;
    up2->clasificare();

    // upcasting dupa virtual pensiune
    UnitateCazare* up3 = &p;
    up3->clasificare();

    // upcasting dupa virtual hotel
    UnitateCazare* up4 = new Pensiune;
    up4->clasificare();

    delete up4;

    return 0;
}
```

Notă: În tabel sunt menționate conceptele folosite în rezolvare și modul în care sunt implementate.

Concept	Demonstrație
Moștenire	`Hotel` și `Pensiune` moștenesc `UnitateCazare`
Upcasting	`UnitateCazare* uh3 = &h;` și celelalte cu *
Object slicing	`UnitateCazare uh1 = h;` și `UnitateCazare up1 = p;`
Shadowing (fără virtual)	`uh2->clasificare();` ar fi apelat baza, dar acum este virtual
Funcții virtuale	`virtual void clasificare()` în `UnitateCazare`
Supradefinire (override)	`void clasificare() override` în `Hotel` și `Pensiune`

4 Teme

4.1 Laturi

Rescrieți rezolvarea problemei *Laturi*, ținând cont de următoarele cerințe:

1. Se cere definirea claselor corespunzătoare, astfel încât:
    a. Clasa Figura să conțină metoda virtual double arie();
    b. Clasa Triunghi să calculeze aria pe baza bazei și înălțimii: (b * h) / 2
    c. Clasa Dreptunghi să calculeze aria ca: l * L
2. Să se demonstreze comportamentul corect al funcției virtuale prin:
    a. Crearea unor obiecte Triunghi și Dreptunghi
    b. Apelarea metodei arie() prin pointeri la Figura (upcasting)
    c. Afișarea rezultatelor în funcție de tipul concret al obiectelor