

6. Moștenirea claselor (II)

Obiective

- Înțelegerea modului în care se utilizează constructorii și destructorii în clasele derivate
- Studiarea modului în care se folosește moștenirea multiplă pentru a deriva o clasă din mai multe clase de bază

- **Constructorii și destructorii în clasele derivate**

Deoarece o clasă derivată moștenește membrii clasei de bază, atunci când este instanțiat un obiect din clasa derivată trebuie apelat constructorul clasei de bază pentru a inițializa membrii care provin din clasa de bază. Inițializarea membrilor din clasa de bază trebuie realizată explicit în apelul constructorului clasei derivate. În caz contrar, pentru membrii care provin din clasa de bază se apelează automat constructorul implicit al clasei de bază.

Constructorii claselor de bază și operatorii de asignare din clasele de bază nu se moștenesc în clasele derivate. Constructorii și operatorii de asignare din clasele derivate pot, însă, să apeleze constructorii și operatorii de asignare din clasele de bază.

Constructorul unei clase derivate apelează întotdeauna mai întâi constructorii claselor de bază sau, în lipsa acestor apeluri, constructorii implicați pentru clasele de bază. Destructorii se apelează în ordine inversă apelurilor constructorilor, astfel încât constructorii claselor derivate sunt apeleți înaintea constructorilor claselor de bază.

Să presupunem că o clasă derivată câștigă și o clasă de bază, ambele conținând obiecte ale altor clase. Atunci când se creează un obiect dintr-o astfel de clasă derivată, automat se apelează prima dată constructorii obiectelor din clasa de bază, apoi constructorul clasei de bază, după care se apelează constructorii obiectelor din clasa derivată și, în final, constructorul clasei derivate. Destructorii se apelează, conform regulii enunțate mai sus, în ordinea inversă constructorilor.

Pentru clasele care conțin obiecte ale altor clase, ordinea de inițializare a obiectelor membre este ordinea în care acestea sunt declarate în clase. Ordinea în care sunt listate inițializările în lista de inițializare a constructorilor nu afectează această ordine.

Programul de mai jos demonstrează ordinea de apel al constructorilor și al destructorilor pentru clasele de bază și cele derivate.

Exemplu

point2.h

```
#ifndef POINT2_H
#define POINT2_H
class Point
{
public:
    Point(int = 0, int = 0); //constructor implicit
    ~Point(); //destructor
protected: //accesibil din clasele derivate
    int x, y; //x si y coordonatele unui punct
};
#endif
```

point2.cpp

```

#include <iostream>
using std::cout;
using std::endl;

#include "point2.h"
//Constructor pentru clasa Point
Point::Point(int a, int b)
{
    x = a;
    y = b;
    cout << "Constructorul obiectului Point: "
         << '[' << x << ", " << y << ']' << endl;
}
//Destructorul clasei Point
Point::~~Point()
{
    cout << "Destructorul obiectului Point: "
         << '[' << x << ", " << y << ']' << endl;
}
circle2.h
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"

class Circle : public Point
{
public:
    //constructor implicit
    Circle(double r = 0.0, int x = 0, int y = 0);
    ~Circle();
protected:
    double radius;
};
#endif
circle2.cpp
#include <iostream>
using std::cout;
using std::endl;

#include "circle2.h"
//Constructorul clasei Circle apeleaza
//constructorul pentru Point si apoi
//initializeaza raza
Circle::Circle(double r, int a, int b)
    : Point(a, b)
{
    radius = r; //ar trebui validata
    cout << "Constructorul obiectului Circle: raza este "
         << radius << " [" << x << ", " << y << ']' << endl;
}

```

```
}
Circle::~~Circle()
{
    cout << "Destructorul obiectului Circle: raza este "
          << radius << " [" << x << ", " << y << ']' << endl;
}
test_point2_circle2.cpp
#include <iostream>

using std::cout;
using std::endl;

#include "point2.h"
#include "circle2.h"

int main()
{
    //Demonstreaza apelul constructorului si al destructorului
    {
        Point p(11, 22);
    }

    cout << endl;
    Circle circle1(4.5, 72, 29);
    cout << endl;
    Circle circle2(10, 5, 5);
    cout << endl;

    return 0;
}
```

Rulând acest program obținem următorul rezultat:

```
Constructorul obiectului Point: [11, 22]
Destructorul obiectului Point: [11, 22]

Constructorul obiectului Point: [72, 29]
Constructorul obiectului Circle: raza este 4.5 [72, 29]

Constructorul obiectului Point: [5, 5]
Constructorul obiectului Circle: raza este 10 [5, 5]

Destructorul obiectului Circle: raza este 10 [5, 5]
Destructorul obiectului Point: [5, 5]
Destructorul obiectului Circle: raza este 4.5 [72, 29]
Destructorul obiectului Point: [72, 29]
```

Clasa Circle este derivată din Point prin moștenire public. Clasa Circle are un constructor, un destructor și o dată membră numită radius, pe lângă datele membre moștenite din clasa Point. Constructorul clasei Circle apelează constructorul clasei Point pentru inițializarea datelor membre x și y.

În funcția `main`, obiectul `p` este declarat într-un bloc. Imediat, domeniul de existență al obiectului `p` se încheie și acesta este distrus. Ilustrăm, astfel, apelul constructorului și al destructorului clasei `Point`. Este instanțiat apoi obiectul `circle1` pentru care se invocă mai întâi constructorul clasei `Point` cu valorile transmise de constructorul clasei `Circle`, apoi se rulează instrucțiunea de afișare din corpul constructorului clasei `Circle`. Aceste operații se repetă asemănător și pentru obiectul `circle2`. La încheierea funcției `main`, se apelează destructorii în ordine inversă celei în care au fost apelați constructorii. Este apelat destructorul clasei `Circle` și destructorul clasei `Point` pentru obiectul `circle2` și apoi destructorul clasei `Circle` și destructorul clasei `Point` pentru obiectul `circle1`.

- **Observații legate de conversia implicită a obiectelor din clasele derivate la obiectele din clasa de bază**

În ciuda faptului că un obiect al unei clase derivate “este un” obiect al clasei de bază, tipul introdus de clasa derivată și cel introdus de clasa de bază sunt diferite.

În cadrul moștenirii publice, obiectele clasei derivate pot fi tratate ca obiecte ale clasei de bază. Acest lucru este corect deoarece clasa derivată are membri corespunzători tuturor membrilor clasei de bază. Poate avea, însă, și alți membri, deci putem spune că o clasă derivată are mai mulți membri decât clasa ei de bază. Asignarea în sens invers, a unui obiect din clasa de bază unui obiect din clasa de derivată nu este permisă deoarece ar lăsa nedefiniți membrii adiționali ai clasei derivate. Deși această asignare nu este permisă în mod „natural”, ea ar putea fi implementată prin supraîncărcarea operatorului de asignare sau a constructorului de conversie.

Moștenirea `public` permite ca un pointer la un obiect dintr-o clasă derivată să fie convertit implicit într-un pointer la un obiect al clasei sale de bază deoarece un obiect al clasei derivate este și obiect al clasei de bază.

Există patru modalități de combinare și potrivire a pointerilor la clasa de bază și a celor la clasa derivată cu obiecte ale clasei de bază și obiecte ale clasei derivate:

1.Referirea la un obiect al clasei de bază printr-un pointer la clasa de bază este simplă;

2.Referirea unui obiect al clasei derivate printr-un pointer la clasa derivată este, de asemenea, simplă;

3.Referirea unui obiect al clasei derivate printr-un pointer la clasa de bază este acceptată deoarece un obiect al clasei derivate este și obiect al clasei de bază. Printr-o astfel de secvență de cod se pot referi doar membrii clasei de bază. Dacă se referă membrii clasei derivate printr-un pointer la clasa de bază, compilatorul semnalează eroare;

4.Referirea unui obiect al clasei de bază printr-un pointer la clasa derivată este eroare de sintaxă. Pointerul la clasa derivată trebuie convertit mai întâi la un pointer la clasa de bază.

- **Relațiile „uses a” și „knows a”**

Moștenirea și compunerea claselor încurajează reutilizarea codului prin crearea unor noi clase care au elemente în comun cu clasele existente. Există și alte moduri în care se pot folosi serviciile claselor. Deși un obiect persoană nu este o mașină și un obiect persoană nu conține o mașină, un obiect persoană *folosește* o mașină. O funcție folosește un obiect prin simplul apel al unei funcții membre `non-private` a

obiectului prin intermediul unui pointer, al unei referințe sau al obiectului însuși. Aceasta este o relație de tip „uses a”.

Un obiect poate să știe de existența altui obiect. Un obiect poate conține un pointer sau o referință pentru accesul la celălalt obiect. De această dată avem de a face cu o relație de tip „knows a” care, de regulă, se numește *asociere*. Diferența dintre compunere și asociere este că, în primul caz, obiectele, adică cel care conține un alt obiect și obiectul din interior, au aceeași perioadă de existență, în timp ce în cazul asocierii obiectele pot exista în mod independent.

- **Studiu de caz: clasele Point, Circle și Cylinder**

În exemplele care urmează vom deriva clasa `Cylinder` din clasa `Circle` care, la rândul ei, a fost derivată din clasa `Point`. Vom relua clasele `Point` și `Circle` așa cum au fost ele definite în capitolul precedent.

Exemplu

```

point.h
#ifndef POINT_H
#define POINT_H

#include <iostream>
using std::ostream;
class Point
{
    friend ostream& operator<<(ostream&, const Point&);
public:
    Point(int = 0, int = 0); //constructor implicit
    void setPoint(int, int); //seteaza coordonatele
    int getX() const { return x; } //returneaza x
    int getY() const { return y; } //returneaza y
protected: //accesibil din clasele derivate
    int x, y; //x si y coordonatele unui punct
};
#endif

point.cpp
#include <iostream>
#include "point.h"
//Constructor pentru clasa Point
Point::Point(int a, int b)
    { setPoint(a, b); }
//Seteaza coordonatele x si y ale unui punct
void Point::setPoint(int a, int b)
{
    x = a;
    y = b;
}
//Afiseaza un punct
ostream& operator<<(ostream& output, const Point& p)
{
    output << '[' << p.x << ", " << p.y << '];'
    return output; //pentru cascadata apelurilor
}
    
```

circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include <iostream>
using std::ostream;
#include <iomanip>
using std::ios;
using std::setiosflags;
using std::setprecision;

#include "point.h"

class Circle : public Point //Circle derivata din Point
{
    friend ostream& operator<<(ostream&, const Circle&);
public:
    //constructor implicit
    Circle(double r = 0.0, int x = 0, int y = 0);
    void setRadius(double); //seteaza radius
    double getRadius() const; //intoarce radius
    double area() const; //calculeaza aria
protected:
    double radius;
};
#endif
```

circle.cpp

```
#include "circle.h"
//Constructorul clasei Circle apeleaza
//constructorul pentru Point si apoi
//initializeaza raza
Circle::Circle(double r, int a, int b)
    : Point(a, b)
    { setRadius(r); }
//Seteaza raza cercului
void Circle::setRadius(double r)
    { radius = (r > 0 ? r : 0); }
//Returneaza raza cercului
double Circle::getRadius() const
    { return radius; }
//Calculeaza aria cercului
double Circle::area() const
    { return 3.14159 * radius * radius; }
//Afiseaza datele despre cerc in forma
//Centrul = [x, y]; Raza = #.##
ostream& operator<<(ostream& output, const Circle& c)
{
    output << "Centrul = " << static_cast<Point>(c)
        << "; Raza = "
        << setiosflags(ios::fixed | ios::showpoint)
        << setprecision(2) << c.radius;
    return output;
}
```

```

}
cylinder.h
#ifndef CYLINDER_H
#define CYLINDER_H

#include <iostream>
using std::ostream;
#include "circle.h"

class Cylinder : public Circle
{
    friend ostream& operator<<(ostream&, const Cylinder&);
public:
    //constructor implicit
    Cylinder(double h = 0.0, double r = 0.0,
              int x = 0, int y = 0);
    void setHeight(double); //seteaza inaltimea
    double getHeight() const; //returneaza inaltimea
    double area() const; //calculeaza aria cilindrului
    double volume() const; //calculeaza volumul cilindrului
protected:
    double height; //inaltimea cilindrului
};
#endif
cylinder.cpp
#include "cylinder.h"
Cylinder::Cylinder(double h, double r, int x, int y)
    : Circle(r, x, y)
    { setHeight(h); }

void Cylinder::setHeight(double h)
    { height = (h >= 0 ? h : 0); }

double Cylinder::getHeight() const
    { return height; }

double Cylinder::area() const
{
    return 2 * Circle::area() +
           2 * 3.14159 * radius * height;
}

double Cylinder::volume() const
    { return Circle::area() * height; }

ostream& operator<<(ostream& output, const Cylinder& c)
{
    output << static_cast<Circle>(c)
           << "; Inaltimea = " << c.height;
    return output;
}

```

```
test_cylinder.cpp
#include<iostream>

using std::cout;
using std::endl;

#include"point.h"
#include "circle.h"
#include "cylinder.h"

int main()
{
    Cylinder cyl(5.7, 2.5, 12, 23);

    cout << "Coordonata X este " << cyl.getX()
         << "\nCoordonata Y este " << cyl.getY()
         << "\nRaza este " << cyl.getRadius()
         << "\nInaltimea este " << cyl.getHeight() << "\n\n";

    cyl.setHeight(10);
    cyl.setRadius(4.25);
    cyl.setPoint(2, 2);
    cout << "Noua pozitie, raza si inaltimea lui cyl sunt:\n"
         << cyl << '\n';

    cout << "Aria lui cyl este: " << cyl.area() << '\n';

    Point& pRef = cyl;
    cout << "\nCilindrul tiparit ca un Point este: "
         << pRef << "\n\n";

    Circle& circleRef = cyl;
    cout << "Cilindrul tiparit ca un Circle este:\n"
         << circleRef
         << "\nAria: " << circleRef.area() << endl;

    return 0;
}
```

Rulând acest program obținem următorul rezultat:

```
Coordonata X este 12
Coordonata Y este 23
Raza este 2.5
Inaltimea este 5.7
```

```
Noua pozitie, raza si inaltimea lui cyl sunt:
Centrul = [2, 2]; Raza = 4.25; Inaltimea = 10.00
Aria lui cyl este: 380.53
```

```
Cilindrul tiparit ca un Point este: [2, 2]
```

```
Cilindrul tiparit ca un Circle este:
```



```
Centrul = [2, 2]; Raza = 4.25
Aria: 56.74
```

Acest exemplu demonstrează moștenirea public și definirea și referirea datelor membre `protected`.

Clasa `Cylinder` este derivată public din clasa `Circle` care, la rândul ei, este derivată public din clasa `Point`. Aceasta înseamnă că interfața publică a claselor `Circle` și `Point` se regăsește și în clasa `Cylinder`. La aceste funcții membre se adaugă funcțiile membre `setHeight`, `getHeight`, `area` (suprascrisere a funcției moștenite din clasa `Circle`) și `volume`. Constructorul clasei `Cylinder` trebuie să invoce constructorul clasei sale de bază directe care este `Circle`, dar nu și pe cel al clasei de bază indirecte `Point`. Constructorul unei clase derivate este responsabil doar de apelul constructorilor claselor de bază directe.

În funcția `main` se inițializează variabila `pRef` care este referință la `Point` cu obiectul `cyl` de tip `Cylinder`. Prin intermediul acestei referințe, obiectul `cyl` este afișat ca un `Point`. Asemănător, prin referința `circleRef` la `Circle`, obiectul `cyl` este afișat ca un `Circle`.

- **Moștenirea multiplă**

În acest capitol am discutat în detaliu despre moștenirea în care o clasă este derivată dintr-o singură clasă de bază. O clasă poate, însă, să fie derivată din mai multe clase de bază. În acest caz vorbim despre *moștenirea multiplă*, prin care clasa derivată moștenește membri de la mai multe clase de bază. Moștenirea multiplă este o facilitate puternică și se poate folosi atunci când există o relație de tip „*is a*” între un nou tip de dată și mai multe tipuri existente. Acest mecanism trebuie folosit, însă, cu atenție pentru că poate cauza o serie de ambiguități.

În exemplul de mai jos, clasa `Derived` este derivată public din clasele de bază `Base1` și `Base2`.

Exemplu

base1.h

```
#ifndef BASE1_H
#define BASE1_H
class Base1
{
public:
    Base1(int x){ value = x; }
    int getData() const { return value; }
protected: //accesibil claselor derivate
    int value; //mostenita de clasa derivata
};
```

base2.h

```
#ifndef BASE2_H
#define BASE2_H
class Base2
{
public:
    Base2(char c){ letter = c; }
    char getData() const { return letter; }
```

```

    protected: //accesibil claselor derivate
        char letter; //mostenita de clasa derivata
};
#endif
derived.h
#ifndef DERIVED_H
#define DERIVED_H
#include <iostream>
using std::ostream;

#include "base1.h"
#include "base2.h"

//Mostenire multipla
class Derived : public Base1, public Base2
{
    friend ostream& operator<<(ostream &, const Derived&);
public:
    Derived(int, char, double);
    double getReal() const;
private:
    double real;//data privata a clasei derivate
};
#endif
derived.cpp
#include "derived.h"
//Constructorul clasei Derived apeleaza
//constructorii claselor Base1 si Base2
Derived::Derived(int i, char c, double f)
    : Base1(i), Base2(c), real(f){}
//Intoarce valoarea lui real
double Derived::getReal() const { return real;}
//Afiseaza toate datele membre ale lui Derived
ostream& operator<<(ostream& output, const Derived& d)
{
    output << "    Intreg: " << d.value
        << "\n    Caracter: " << d.letter
        << "\n    Numar real: " << d.real;
    return output;
}
test_multiple_inheritance.cpp
#include <iostream>
using std::cout;
using std::endl;

#include "base1.h"
#include "base2.h"
#include "derived.h"

int main()
{

```

```

Base1 b1(10), *base1Ptr = 0;
Base2 b2('Z'), *base2Ptr = 0;
Derived d(7, 'A', 3.5);

//Afiseaza membrii obiectelor din clasele de baza
cout << "Obiectul b1 contine intregul " << b1.getData()
     << "\nObiectul b2 contine caracterul " << b2.getData()
     << "\nObiectul d contine:\n" << d << "\n\n";

//Afiseaza datele membre ale obiectului din clasa derivata
//Operatorul domeniu rezolva ambiguitatea
//in apelul functiilor getData
cout << "Datele membre ale lui Derived pot fi"
     << " accesate individual:"
     << "\n    Intreg: " << d.Base1::getData()
     << "\n    Caracter: " << d.Base2::getData()
     << "\n    Numar real: " << d.getReal() << "\n\n";

cout << "Derived poate fi tratat ca un obiect"
     << " al claselor de baza:\n";
//Derived tratat ca obiect al clasei Base1
base1Ptr = &d;
cout << "base1Ptr->getData() afiseaza "
     << base1Ptr->getData() << '\n';
//Derived tratat ca obiect al clasei Base2
base2Ptr = &d;
cout << "base2Ptr->getData() afiseaza "
     << base2Ptr->getData() << '\n';

return 0;
}

```

Rulând acest program obținem următorul rezultat:

```

Obiectul b1 contine intregul 10
Obiectul b2 contine caracterul Z
Obiectul d contine:
    Intreg: 7
    Caracter: A
    Numar real: 3.5

```

```

Datele membre ale lui Derived pot fi accesate individual:
    Intreg: 7
    Caracter: A
    Numar real: 3.5

```

```

Derived poate fi tratat ca un obiect al claselor de baza:
base1Ptr->getData() afiseaza 7
base2Ptr->getData() afiseaza A

```

Moștenirea multiplă se indică prin lista de clase de bază separate prin virgulă urmează semului : plasat după declarația `class Derived`. În acest program, constructorul clasei `Derived` apelează explicit constructorii claselor de bază, `Base1`

și `Base2`, pentru a inițializa membrii care provin din aceste clase. În lipsa acestor apeluri explicite, compilatorul ar fi apelat automat constructorii implicați ai celor două clase.

În funcția `main` se creează obiectul `b1` din clasa `Base1` care este inițializat cu valoarea întreagă `10`. Obiectul `b2` din clasa `Base2` este inițializat cu valoarea `'Z'` de tip `char`. Se creează în final obiectul `d` din clasa `Derived` care este inițializat cu valorile `7` de tip `int`, `'A'` de tip `char` și `3.5` de tip `double`.

Conținutul fiecărui obiect din clasa de bază este afișat prin apelul funcțiilor `getData` definite distinct în clasa `Base1` și în clasa `Base2`. Apelurile nu sunt ambigue pentru că se fac prin intermediul obiectelor `b1` și `b2`.

Problema ambiguității apare la tipărirea obiectului `d` pentru că funcția `getData` este moștenită atât din clasa `Base1` cât și din clasa `Base2`. Această problemă este rezolvată prin folosirea operatorului domeniu `::` ca în `d.Base1::getData()` pentru tipărirea datei membre `value` și `d.Base2::getData()` pentru tipărirea datei membre `letter`. Valoarea datei membre `real` este tipărită fără nicio ambiguitate prin apelul `d.getReal()`.

În final, programul demonstrează că relația „is a” din moștenirea simplă se aplică și în cazul moștenirii multiple. Adresa obiectului `d` din clasa derivată este asignat pointerului `base1Ptr` la clasa `Base1` și se tipărește valoarea datei membre `value` invocând funcția membră `getData` din clasa `Base1` prin pointerul `base1Ptr`. În mod asemănător este afișată valoarea datei membre `letter` prin apelul funcției membre `getData` din clasa `Base2` prin intermediul pointerului `base2Ptr`, după ce acestuia i-a fost asignată adresa obiectului `d`.