7. Funcții virtuale şi polimorfism

Obiective

- Înțelegerea noțiunii de polimorfism
- Înțelegerea modului în care se declară şi se folosesc funcțiile virtuale
- Înțelegerea distincției dintre clasele abstracte şi clasele concrete
- Studierea modului în care se declară funcțiile virtuale pure pentru crearea claselor abstracte
- Înțelegerea modului în care polimorfismul face sistemele software extensibile şi mai uşor de întreținut

- *Introducere*

Cu ajutorul *funcțiilor virtuale* şi al *polimorfismului* este posibilă proiectarea şi implementarea sistemelor software care sunt mult mai uşor *extensibile*. Programele pot fi concepute să proceseze în mod generic, sub forma obiectelor din clasele de bază, a obiectelor tuturor claselor dintr-o ierarhie. Clasele care nu există în timpul dezvoltării inițiale a programului pot fi adăugate ulterior cu modificări minore sau chiar fără a face modificări părții generice a programului, atâta timp cât clasele sunt părți ale ierarhiei procesate generic. Singurele părți din program care trebuie modificate sunt cele care folosesc informații specifice despre o clasă adăugată în ierarhie.

- *Instrucțiunile `switch` pentru lucrul cu diverse tipuri de date*

O variantă de tratare a obiectelor care au diverse tipuri de date este folosirea instrucțiunii `switch` prin care se pot apela acțiuni diferite pentru fiecare tip de obiect. De exemplu, într-o ierarhie de forme geometrice in care fiecare clasă are o dată membră care conține numele formei, o instrucțiune `switch` ar putea determina care funcție `print` să fie apelată pentru un obiect particular.

Există, însă, probleme legate folosirea instrucțiunii `switch`. Programatorul ar putea uita, de exemplu, să testeze toate posibilele tipuri de dată care există în ierarhia de clase. După adăugarea unei noi clase în ierarhie, programatorul ar putea uita să adauge şi această clasă în lista de cazuri `switch`. Fiecare adăugare sau ştergere a unei clase înseamnă o modificare a instrucțiunii `switch` care este o potențială sursă de erori.

Funcțiile virtuale şi programarea polimorfică pot elimina necesitatea logicii switch. Programatorul poate folosi mecanismul funcțiilor virtuale pentru a automatiza implementarea acestei logici.

- *Funcțiile virtuale*

Să presupunem că avem mai multe clase care reprezintă forme geometrice: `Point`, `Circle`, `Triangle`, `Rectangle`, `Square` derivate din clasa de bază `Shape`. Fiecare dintre aceste clase trebuie să permită afişarea numelui formei pe care o reprezintă. Deşi fiecare clasă are o funcție `printShapeName`, implementarea funcției `printShapeName` pentru fiecare formă este diferită. Ar fi de dorit ca toate formele să poată fi tratate generic, fiind derivate din clasa de bază `Shape`. Prin acest mecanism, pentru oricare dintre forme s-ar apela funcția `printShapeName` din clasa

de bază `Shape`, urmând ca, după tipul de dată al obiectului, programul să determine dinamic (în timpul execuției) care dintre funcțiile `printShapeName` din clasele derivate se foloseşte.

Pentru a permite acest lucru, funcția `printShapeName` trebuie declarată virtual în clasa de bază, iar fiecare clasă din ierarhia de derivare trebuie să o suprascrie pentru ca să implementeze un comportament particular.

Declararea unei funcții virtuale se face prin adăugarea cuvântului cheie `virtual` înaintea prototipului în clasa de bază:

```
virtual void printShapeName() const;
```

Acest prototip de funcție este cel care trebuie declarat in clasa de bază `Shape`.

Exemplu

```
#include <iostream>
using namespace std;
class Shape{
  public:
    virtual void printShapeName() const
    {
      cout << "Shape::printShapeName()" << endl;
    }
    void init() const
    {
      cout << "Shape::init()" << endl;
    }
};

class Point : public Shape{
  public:
    void printShapeName() const
    {
      cout << "Point::printShapeName()" << endl;
    }
    void init() const
    {
      cout << "Point::init()" << endl;
    }
};
int main()
{
  cout << "Functii apelate prin pointer la Shape:" << endl;
  Shape* shapePtr = new Shape();
  shapePtr->printShapeName();
  shapePtr->init();

  cout << "\nFunctii apelate prin pointer la Shape "
       << "initializat prin pointer la Point:" << endl;
  Point* pointPtr = new Point();
  shapePtr = pointPtr;
  cout << "Comportament polimorfic: ";
     shapePtr->printShapeName();
  shapePtr->init();
```

```
    cout << "\nFunctii apelate prin obiect de tip Shape:"
         << endl;
    Shape shape;
    shape.printShapeName();
    shape.init();

    cout << "\nFunctii apelate prin obiect de tip Point:"
         << endl;
    Point point;
    point.printShapeName();
    point.init();

    cout << "\nFunctie non-virtuala apelata prin pointer la "
         << "Shape:" << endl;
    shapePtr = &point;
    cout << "Comportament non-polimorfic: ";
    shapePtr->init();

    return 0;
}
```

Rulând acest program obținem următorul rezultat:

```
Functii apelate prin pointer la Shape:
Shape::printShapeName()
Shape::init()

Functii apelate prin pointer la Shape initializat prin
pointer la Point:
Comportament polimorfic: Point::printShapeName()
Shape::init()

Functii apelate prin obiect de tip Shape:
Shape::printShapeName()
Shape::init()

Functii apelate prin obiect de tip Point:
Point::printShapeName()
Point::init()

Functie non-virtuala apelata prin pointer la Shape:
Comportament non-polimorfic: Shape::init()
```

Funcția `printShapeName` din clasa de bază poate fi invocată printr-un pointer sau o referință la clasa de bază:

```
Shape* shapePtr = new Shape();
shapePtr->printShapeName();
```

Inițializând pointerul la clasa de bază cu un pointer la o clasa derivată, programul va alege dinamic (în timpul rulării) implementarea funcției `printShapeName` din clasa derivată bazându-se pe tipul obiectului apelant:

```
Point* pointPtr = new Point();
shapePtr = pointPtr;
```

```
shapePtr->printShapeName();
```

Acest proces se numeşte *legare dinamică* (*dynamic binding*).

Când o funcție virtuală este apelată referind un obiect prin nume şi selecția funcției membre prin operatorul punct `.`, referința este rezolvată la compilare (*static binding*), iar funcția virtuală apelată este cea definită pentru clasa căreia îi aparține obiectul:

```
Point point;
point.printShapeName();
```

- *Clase de bază abstracte şi clase de bază concrete*

Când ne gândim la o clasă ca la un tip de dată, ne aşteptăm ca în aplicațiile care folosesc acea clasă să fie instanțiate obiecte ale sale. Există, totuşi, cazuri în care este util să se declare clase pentru care programatorul nu are intenția să instanțieze obiecte. Acestea sunt *clase abstracte*. Deoarece acestea sunt folosite drept clase de bază în ierarhii de moştenire, obişnuim să le numim *clase de bază abstracte*. Aşadar, nu pot fi instanțiate obiecte din clasele de bază abstracte.

Rolul unei clase abstracte este acela de a crea o clasă de bază din care alte clase pot moşteni interfața sau implementarea. Clasele din care pot fi instanțiate obiecte sunt *clase concrete*.

Clasă abstractă de bază	**Clase derivate**

`FormaBidimensionala`	`Cerc`
	`Triunghi`
	`Dreptunghi`

`FormaTridimensionala`	`Cub`
	`Sfera`
	`Cilindru`

Clasele de bază abstracte sunt de regulă prea generice pentru a defini obiecte reale. Este nevoie de clase mai specifice pentru a putea justifica posiblitatea de a instanția obiecte.

O clasă devine abstractă dacă una sau mai multe funcții virtuale este declarată *pură*. O funcție virtuală devine pură dacă declarația sa este urmată de `=0`:

```
virtual void earnings() const = 0; //functie virtuala pura
```

Dacă o clasă este derivată dintr-o clasă de bază care conține o funcție virtuală pură şi nu defineşte acea funcție, atunci funcția este pură şi în clasa derivată, iar clasa derivată este şi ea abstractă.

O ierarhie de clase nu trebuie să conțină neapărat clase abstracte. Atunci când se întâmplă, însă, acest lucru, clasele abstracte se găsesc în vârful ierarhiei, iar la baza ierarhiei sunt clase concrete.

- *Polimorfismul*

Limbajul C++ oferă suport pentru *polimorfism* care înseamnă posibilitatea ca obiecte din diverse clase care sunt legate prin relații de moştenire să răspundă diferit la acelaşi mesaj, adică la acelaşi apel de funcție.

Polimorfismul este implementat prin funcțiile virtuale. Atunci când programul cere folosirea unei funcții printr-un pointer sau o referință la o clasa de bază, C++ alege suprascrierea corectă din clasa derivată corespunzătoare obiectului care o apelează.

Uneori, o funcție non-virtuală este definită în clasa de bază şi suprascrisă într-o clasă derivată. Dacă o astfel de funcție membră este apelată printr-un pointer la clasa de bază inițializat cu adresa unui obiect al unei clase derivate, este apelată

versiunea din clasa de bază. Dacă funcția membră este apelată printr-un pointer la clasa derivată, este folosită versiunea funcției din clasa derivată. Acesta este un comportament non-polimorfic.

Polimorfismul promovează extensibilitatea. Aplicațiile software scrie în manieră polimorfică sunt independente de tipurile obiectelor către care se trimit mesaje. Astfel, noile tipuri de obiecte care răspund la mesajele existente pot fi adăugate într-un astfel de sistem fără a modifica sistemul de bază. Atunci când codul client intanțiază, însă, obiecte noi, programul trebuie recompilat.

Cu toate că nu se pot instanția obiecte din clasele de bază abstracte, se pot declara pointeri sau referințe la aceste clase. Aceşti pointeri sau referințe pot fi utilizați pentru a permite manipulările polimorfice ale obiectelor din clasele derivate când se instanțiază astfel de obiecte din clase concrete.

Polimorfismul şi funcțiile virtuale sunt utile şi atunci când, într-o fază intermediară a proiectării şi implementării unei aplicații software, nu sunt cunoscute toate clasele care vor fi folosite în versiunea finală. Noile clase care sunt adăugate sistemului sunt integrate prin legarea dinamică (*dynamic binding*, numit uneori şi *late binding*). Tipul unui obiect care apelează o funcție virtuală nu este nevoie să fie cunoscut la compilare. La rulare, funcția apelată virtual este identificată cu funcția membră din clasa căreia îi aparține obiectul.

- *Destructorii virtuali*

Atunci când se foloseşte polimorfismul pentru procesarea obiectelor alocate dinamic dintr-o ierarhie de clase apar probleme legate de distrugerea acestora. Dacă un obiect cu un destructor non-`virtual` este distrus explicit prin aplicarea operatorului `delete` unui pointer la clasa de bază care pointează către obiect, se apelează destructorul clasei de bază, indiferent de tipul actual al obiectului. O soluție simplă la această problemă este declararea `virtual` a destructorului clasei de bază. Această declarare face ca toți destructorii claselor derivate să devină virtuali, chiar dacă, evident, au nume diferite de cel al destructorului clasei de bază. Dacă acum un obiect din ierarhie este distrus explicit prin aplicarea operatorului delete unui pointer către clasa de bază care a fost inițializat cu adresa unui obiect dintr-o clasă derivată, se apelează, în mod corect, destructorul clasei derivate corespunzătoare pointerului cu care a fost inițializat pointerul la clasa de bază.

Spre deosebire de destructori, constructorii nu pot fi declarați `virtual`.

- *Studiu de caz: Moştenirea interfeței şi a implementării*

Vom rescrie ierarhia de clase care cuprinde tipurile `Point`, `Circle` şi `Cylinder` prin adăugarea, în vârful ierarhiei, a clasei de bază `Shape` care are două funcții virtuale pure: `printShapeName` şi `print`, de aceea `Shape` este o clasă de bază abstractă. Clasa `Shape` conține încă două funcții virtuale, `area` şi `volume`, fiecare dintre ele cu câte o implementare implicită care returnează valoarea `0`. Clasa `Point` este derivată din clasa `Shape`. Interpretarea dată valorii funcțiilor `area` şi `volume` este, astfel, corectă. Clasa `Circle` moşteneşte funcția `volume` de la clasa `Point`, dar are propria implementare pentru funcția `area`. Clasa `Cylinder` are propriile implementări atât pentru funcția `area`, cât şi pentru funcția `volume`.

Deşi clasa `Shape` este o clasă abstractă, ea conține implementări ale unor funcții membre şi aceste implementări sunt moştenite de clasele derivate. Clasa `Shape` are o interfață sub forma a patru funcții virtuale care este moştenită de clasele derivate. Aceste funcții pot fi utilizate de toate clasele din ierarhia de moştenire.

Exemplu

shape.h

```
#ifndef SHAPE_H
#define SHAPE_H

class Shape
{
  public:
    virtual double area() const {return 0.0;}
    virtual double volume() const {return 0.0;}

    //functii virtuale pure suprascrise in clasele derivate
    virtual void printShapeName() const = 0;
    virtual void print() const = 0;
};

#endif
```

point1.h

```
#ifndef POINT1_H
#define POINT1_H

#include <iostream>
using std::cout;
#include "shape.h"

class Point : public Shape
{
  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
    virtual void printShapeName() const {cout << "Point: ";}
    virtual void print() const;
  private:
    int x, y; //x si y coordonatele unui punct
};
#endif
```

point1.cpp

```
#include <iostream>
#include "point1.h"

Point::Point(int a, int b)
  { setPoint(a, b); }

void Point::setPoint(int a, int b)
{
  x = a;
  y = b;
}
```

```
void Point::print() const
  { cout << '[' << x << ", " << y << ']'; }
```

circle1.h

```
#ifndef CIRCLE1_H
#define CIRCLE1_H
#include "point1.h"

class Circle : public Point
{
  public:
    //constructor implicit
    Circle(double r = 0.0, int x = 0, int y = 0);
    void setRadius(double); //seteaza radius
    double getRadius() const; //intoarce radius
    virtual double area() const; //calculeaza aria
    virtual void printShapeName() const {cout << "Circle: ";}
    virtual void print() const;
  private:
    double radius;
};
#endif
```

circle1.cpp

```
#include <iostream>
using std::cout;
#include "circle1.h"
Circle::Circle(double r, int a, int b)
  : Point(a, b)
  { setRadius(r); }

void Circle::setRadius(double r)
  { radius = (r > 0 ? r : 0); }

double Circle::getRadius() const
  { return radius; }

double Circle::area() const
  { return 3.14159 * radius * radius; }

void Circle::print() const
{
  Point::print();
  cout << "; Raza = " << radius;
}
```

cylinder1.h

```
#ifndef CYLINDER1_H
#define CYLINDER1_H
#include "circle1.h"

class Cylinder : public Circle
{
  public:
```

```
    //constructor implicit
    Cylinder(double h = 0.0, double r = 0.0,
                  int x = 0, int y = 0);
    void setHeight(double);
    double getHeight() const;
    virtual double area() const;
    virtual double volume() const;
    virtual void printShapeName() const {cout << "Cylinder: ";}
    virtual void print() const;
  private:
    double height;
};
#endif
```

cylinder1.cpp

```
#include <iostream>
using std::cout;
#include "cylinder1.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; }

void Cylinder::print() const
{
  Circle::print();
  cout << "; Inaltimea = " << height;
}
```

test_shape.cpp

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

#include <iomanip>
using std::ios;
using std::setiosflags;
using std::setprecision;
```

```
#include "shape.h"
#include"point1.h"
#include "circle1.h"
#include "cylinder1.h"

void virtualViaPointer(const Shape*);
void virtualViaReference(const Shape&);

int main()
{
  cout << setiosflags(ios::fixed | ios::showpoint)
       << setprecision(2);

  Point point(7, 11);
  Circle circle(3.5, 22, 8);
  Cylinder cylinder(10, 3.3, 10, 10);

  point.printShapeName();    //legare statica (static binding)
  point.print();             //legare statica
  cout << '\n';

  circle.printShapeName();   //legare statica
  circle.print();            //legare statica
  cout << '\n';

  cylinder.printShapeName(); //legare statica
  cylinder.print();          //legare statica
  cout << '\n';

  //tablou de pointeri la clasa de baza
  Shape *arrayOfShapes[3];

  //arrayOfShapes[0] pointeaza la obiectul point
  //din clasa derivata Point
  arrayOfShapes[0] = &point;

  //arrayOfShapes[1] pointeaza la obiectul circle
  //din clasa derivata Circle
  arrayOfShapes[1] = &circle;

  //arrayOfShapes[2] pointeaza la obiectul cylinder
  //din clasa derivata Cylinder
  arrayOfShapes[2] = &cylinder;

  //Se parcurge tabloul arrayOfShapes
  //si se apeleaza virtualViaPointer
  //pentru tiparirea numelui formei, atributelor,
  //ariei si volumului fiecarui obiect prin legare dinamica
  cout << "\nApeluri de functii virtuale prin "
       << "pointeri la clasa de baza\n";
  for(int i = 0; i < 3; i++)
```

```
    virtualViaPointer(arrayOfShapes[i]);

  //Se parcurge tabloul arrayOfShapes
  //si se apeleaza virtualViaReference
  //pentru tiparirea numelui formei, atributelor,
  //ariei si volumului fiecarui obiect prin legare dinamica
  cout << "Apeluri de functii virtuale prin "
       << "referinte la clasa de baza\n";
  for(int i = 0; i < 3; i++)
    virtualViaReference(*arrayOfShapes[i]);

  return 0;
}

//Permite apelul functiilor virtuale printr-un pointer
//la clasa de baza folosind legarea dinamica
void virtualViaPointer(const Shape* baseClassPtr)
{
  baseClassPtr->printShapeName();
  baseClassPtr->print();
  cout << "\nAria = " << baseClassPtr->area()
       << "\nVolumul = " << baseClassPtr->volume() << "\n\n";
}

//Permite apelul functiilor virtuale printr-o referinta
//la clasa de baza folosind legarea dinamica
void virtualViaReference(const Shape& baseClassRef)
{
  baseClassRef.printShapeName();
  baseClassRef.print();
  cout << "\nAria = " << baseClassRef.area()
       << "\nVolumul = " << baseClassRef.volume() << "\n\n";
}
```

Rulând acest program obținem următorul rezultat:

```
Point: [7, 11]
Circle: [22, 8]; Raza = 3.50
Cylinder: [10, 10]; Raza = 3.30; Inaltimea = 10.00

Apeluri de functii virtuale prin pointeri la clasa de baza
Point: [7, 11]
Aria = 0.00
Volumul = 0.00

Circle: [22, 8]; Raza = 3.50
Aria = 38.48
Volumul = 0.00

Cylinder: [10, 10]; Raza = 3.30; Inaltimea = 10.00
Aria = 275.77
Volumul = 342.12
```

```
Apeluri de functii virtuale prin referinte la clasa de baza
Point: [7, 11]
Aria = 0.00
Volumul = 0.00

Circle: [22, 8]; Raza = 3.50
Aria = 38.48
Volumul = 0.00

Cylinder: [10, 10]; Raza = 3.30; Inaltimea = 10.00
Aria = 275.77
Volumul = 342.12
```

În programul de mai sus, `arrayOfShapes` este un tablou de pointeri la `Shape`. Elementele sale sunt inițializate cu pointeri la `Point`, `Circle` şi `Cylinder` care sunt clase derivate din `Shape`. Ciclul for parcurge elementele tabloului care sunt transmise pe rând ca parametri funcției `virtualViaPointer`:

```
virtualViaPointer(arrayOfShapes[i]);
```

Paramentrul funcției este recepționat în pointerul baseClassPtr de tip `const Shape*` şi, prin intermediul său, se fac apelurile de funcții virtuale:

```
baseClassPtr->printShapeName()
baseClassPtr->print()
baseClassPtr->area()
baseClassPtr->volume()
```

Fiecare dintre aceste apeluri invocă funcția virtuală a obiectului către care `baesClassPtr` pointează în momentul apelului, obiect al cărui tip nu poate fi determinat la compilare.

Asemănător, funcția `virtualViaReference` este invocată pe rând cu trei parametri care reprezintă referințe la cele trei obiecte ale căror adrese sunt stocate în `arrayOfShapes`:

```
virtualViaReference(*arrayOfShapes[i]);
```

Funcția `virtualViaReference` încarcă valoarea parametrului în `baseClassRef` de tip `const Shape&` şi apelează funcțiile virtuale astfel:

```
baseClassRef.printShapeName()
baseClassRef.print()
baseClassRef.area()
baseClassRef.volume()
```