Programare orientata-obiect (POO) utilizand C++

Polimorfism

Dorel Lucanu

Cuprins

- Ce este polimorfism
- tipuri de polimorfism
  - polimorfism ad-hoc
    - coercitiv (*coercion*)
    - supraincarcare (overloading)
  - polimorfism universal
    - parametric
    - subtip (incluziune)
- cum se realizeaza toate acestea in C++
- aplicatie

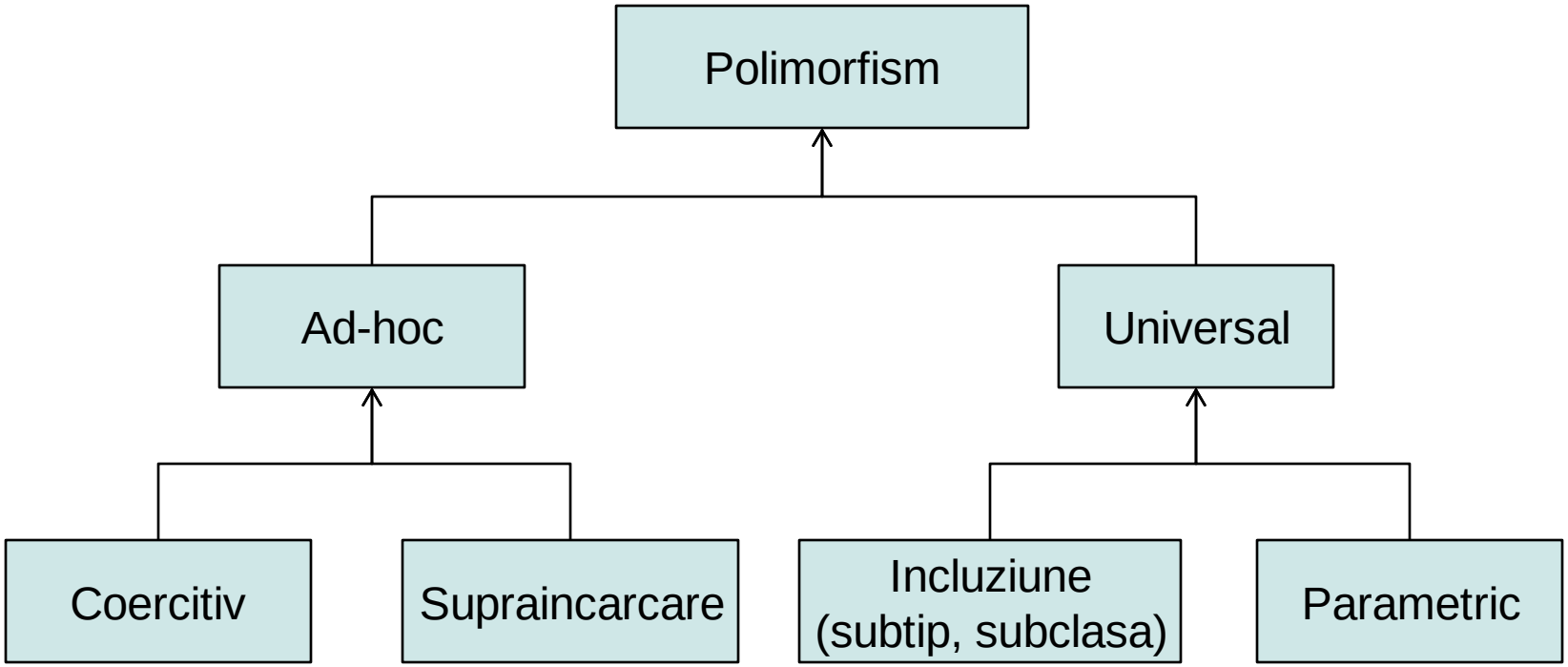
Definitie

- vine din greaca: “mai multe forme”
- in sens larg: aceeasi constructie sintactica cu mai multe intelesuri, alegandu-se intelesul corect in functie de “context”
- in POO: obiecte diferite raspund la acelasi mesaj executand metode adecvate
- polimorfismul a fost definit pentru prima data pentru limbajele de programare de catre Strachey in 1967
- Strachey a definit polimorfismul “ad-hoc” si cel “parametric”:

"Ad-Hoc polymorphism is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type. Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure."

Clasificarea Cardelli-Wegner 1/5

- Cardelli si Wegner a rafinat definitia lui Strachey



Polimorfism ad-hoc

- functiile lucreaza peste o multime finita de tipuri, care potential nu sunt relationate
- “dispare” la o privire mai atenta
  - coercitiv: compilatorul converteste dintr-un tip in altul pentru a evita erori
    - poate fi
      - implicit sau
      - explicit (casting)

      sau, clasificat dupa alt criteriu,
      - de restringere (narrowing)
      - de largire (widening)
  - supraincare: aceeasi functie are mai multe implementari

Polimorfism coercitiv: exemplu simplu

```
void print (int x) const
{
  cout << "Valoare (" << x << ") " << endl;
}

int main( )
{
  print(15);
  print(16.8); // narrowing
  print('\n'); // widening
}
```

Polimorfism coercitiv: exemplu cu clase

```
class Drept {
public:
  int arie() const {
    return lung * latime;
  }
private:
  int lung, latime;
}
```

```
class Patrat {
public:
  Patrat(const Drept& d)
  { latura = d.lung; }
  int arie() const {
    return latura * latura;
  }
private:
  int latura;
}
```

constructor de conversie

```
void printArie(const Patrat& p)
{
  return p.arie();
}
```

apel prin referinta

```
Drept drpt;
Patrat ptrt;
printArie(drpt);
printArie(ptrt);
```

conversie

Polimorfism coercitiv: suprascriere

```
class Patrat {
public:
  int arie() const {
    return lat * lat;
  }
private:
  int lat;
}
```

```
class Drept : public Patrat {
public:
  int arie() const {
    return lung * lat;
  }
private:
  int lung;
}
```

```
void printArie(const Patrat& p)
{
  cout << p.arie();
}
```

apel prin referinta

```
Drept drpt;
Patrat ptrt;
printArie(drpt);
printArie(ptrt);
```

upcast

Polimorfism prin supraincarcare: exemplu

```
string s = "exemplu";
int n = 123;
double z = 3.1415

cout << s;
cout << n;
cout << z;
```

- alte exemple: `+  / *`

Supraincarcare cu tipuri diferite

- interschimbarea a doua "**int**"-uri

```
void swap(int& x, int& y) {
  int aux = x;
  x = y;
  y = aux;
}
```

- interschimbarea a doua "**double**"-uri

```
void swap(double& x, double& y) {
  double aux = x;
  x = y;
  y = aux;
}
```

- utilizare

```
int m = 5; n = 8;
swap(m, n);
```

```
double a = 5.71, b = 7.51;
swap(a, b);
```

Supraincarcare cu numar diferiti de parametri

```
int sqrSum(int x, int y) {
  return x*x + y*y;
}

int sqrSum(int x, int y, int z) {
  return x*x + y*y + z*z;
}

int a = 2, b = 3, c = 4;
cout << sqr(a, b) << endl;  // 13
cout << sqr(a, b, c) << endl;  // 29
```

Operatori supraincarcati de utilizator

```
class Data {
public:
  friend ostream& operator << (ostream&,
                               const Data&);
 ...
private:
  int zi, luna, an;
}
ostream& operator << (ostream& o,
                      const Data& d)
{
  o << d.zi << ' ' << d.luna << ' ' << d.an;
  return o;
}
```

supraincarcarea operatorului de scriere intr-un flux

o clasa C++ poate avea **prieteni** (clase, functii, operatori ...)
prietenii au acces la membrii privati
... deci atentie cum alegeti prietenii

fiind prieten, operatorul are acces la membrii privati (atentie la prieteni!)

Supraincarcarea op. de incrementare 1/2

- data de maine poate fi obtinuta cu operatorul de incrementare

```
class Data {
  //...
public:
  Data  operator ++(int); // postfix
public:
  Data& operator ++();    // prefix
   //...
};
```

Supraincarcarea op. de incrementare 2/2

- postfixat

```
Data Data::operator ++(int)  {
   Data oldValue = *this;
   zi++;

   if ...
   ...
   return oldValue;
}
```

- prefixat

```
Data& Data::operator ++()  {
   zi++;
   if ...
   ...
   return *this;
}
```

Polimorfism universal

- mai este numit si polimorfismul adevarat (true polymorphism) – nu “dispare” la o privire mai atenta
- nu exista restrictii asupra numarului de tipuri acceptabile
- se caracterizeaza prin unformitate: aceeasi functie “lucreaza” peste diferite tipuri
  - parametric: lucreaza peste functii si clase care partajeaza o anumita structura comuna peste un numar potential infinit de tipuri nerelationate
    - este implementat la momentul compilarii
    - in C++ este realizat prin definitii parametrizate
  - incluziune (subtip, subclasa):
    - o metoda apelata pentru un obiect, care poate avea tipuri diferite in timpul executiei, poate avea definitii (comportari)diferite
    - in C++ este realizat prin mostenire, functii virtuale, apel prin referinta sau pointer

Clase parametrizate : celule de memorie 1/3

```
template <class Elt>
class Cell {
public:
      Cell();
      ~Cell();
      Elt getVal();
      void setVal(Elt);
private:
      Elt* val;
};

template <class Elt>
Cell<Elt>::Cell() {
      val = new Elt;
}
```

Clase parametrizate : celule de memorie 2/3

```
template <class Elt>
Cell<Elt>::~Cell()
{
  delete val;
}

template <class Elt>
Elt Cell<Elt>::getVal()
{
  return *val;
}

//. . .
```

Clase parametrizate : celule de memorie 3/3

```
template <class Elt>
void printCell(string name, Cell<Elt> cell) {
  cout << name + " = " << cell.getVal() << endl;
}
```

```
Cell<int> x;
x.setVal(100);
Cell<char> c;
c.setVal('A');
printCell("x", x);
printCell("c", c);
```

```
typedef Cell<int> Int;
typedef Cell<char> Char;
```

Parametrizare prin incluziune (subtip, subclasa)

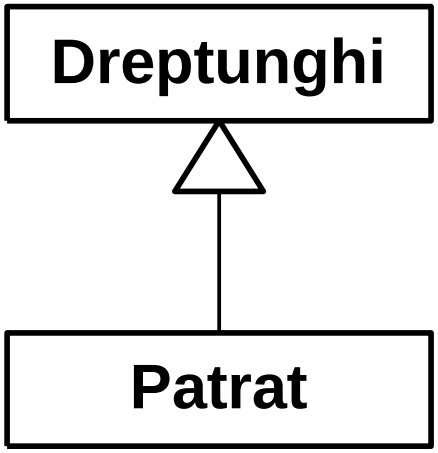
- in C++ se realizeaza cu ajutorul urmatoarelor trei elemente:
  - mostenire
  - functii virtuale
  - apel prin referinta sau apel prin pointer

Mostenire

- **mostenirea** = mecanismul prin care elementele specifice (specializate) încorporează structura şi comportarea elementelor generale **(reutilizare).**
- **principiul substituirii (B. Liskov, 1987):** obiectele clasei specializate (copil) pot fi utilizate oriunde apar obiecte ale clasei generale (părinte) fara a altera proprietatile dorite (dar nu şi reciproc)**;**
- mostenirea este sinonima cu relatia de **subtip**

  Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S, where S is a subtype of T.
- relatia de mostenire este tranzitiva

Principiul substituirii - contraexemplu



- clasa **`Patrat`** are invariantul **`lungimea = latimea`**
- o operatie **`maresteLungimea() a clasei Dreptunghi`** nu pstreaza acest invariant
- deci un patrat nu poate fi utilizat in locul unui dreptunghi totdeauna

Functii virtuale si apel prin referinta

```
class Patrat {
public:
  virtual int arie()
   const {
     return lat * lat;
   }
private:
  int lat;
}
```

```
class Drept : public Patrat {
public:
  int arie() const {
     return lung * lat;
  }
private:
  int lung;
}
```

```
void printArie(const Patrat& p)
{
  return p.arie();
}
```

apel prin referinta

apel polimorfic

```
Drept drpt;
Patrat ptrt;
printArie(drpt);
printArie(ptrt);
```

Interfete versus clase abstracte 1/4

- Mostenirea nu este destul de flexibila pentru a beneficia pe deplin de avantajele polimorfismului
- Intefetele au menirea de a aduce plusul de flexibilitate de care e nevoie
- Reamintim ca interfata unui obiect este formata din totalitatea metodelor si proprietatilor publice
- Acestea pot fi organizate intr-o structura asemanatoare clasei, dar fara sa includa implementari ale metodelor sau membri privati
- o interfata include numai “semnaturile” (signatures) metodelor: tipul rezultatului, numele metodei, numarul si tipurile argumentelor
- se pot crea ierarhii de interfete
- interfetele sunt implementate de clase

Interfete versus clase abstracte 2/4

- C++ nu include posibilitatea de a declara interfete
- In locul lor se pot utiliza clase abstracte
- Ca si interfetele, clasele abstracte nu pot fi instantiate
- Spre deosebire de interfete, o clasa abstracta poate include implementari partiale de metode si membri privati
- O clasa abstracta, vazuta ca o interfata, este "implementata" prin mostenire
- In C++, o clasa este abstracta daca include cel putin o metoda virtuala pura, care nu are implementare
- Implementarea unei metode virtuale pure se face intr-o clasa derivata
- Pot exista ierarhii de clase derivate

Interfete versus clase abstracte 3/4

```
class AbstractBase {
/* no private data */
public:
virtual void a(...) = 0;
virtual int b(...) = 0;
...
};
class ConcreteSubclass : public AbstractBase {
public:
virtual void a(...) { ... }
virtual int b(...);
...
};
```

metode virtuale pure

implementare inline

implementare in alt fisier

Interfete versus clase abstracte 4/4

```
class Figura {
public:
   virtual double arie() = 0;
}
```

```
class Drept : public Figura
{
public:
  virtual double arie() {
    return lat * lung;
  }
private:
 double lat, lung;
}
```

```
class Cerc : public Figura {
public:
 virtual double arie() {
    return pi * raza * raza;
  }
private:
 double raza;
}
```

Aplicatie

- sa realizam o aplicatie simpla care gestioneaza o colectie de figuri
- actiuni: adauga o figura (dreptunghi sau cerc), afiseaza figurile
- cerinte: aplicatia va include o interfata textuala, in care vor fi utilizate obiecte de tip *display-box, edit-box, menu*
- aplicatia trebuie sa fie relativ simplu de extins

Elemente de interfata: display-box 1/2

```
template <class T>
class DisplayBox
{
private:
   string label;
private:
   T value;
public:
   DisplayBox(char *newLabel = "")
   {
       label = newLabel;
   }
public:
   void setLabel(string newLabel)
   {
       label = newLabel;
   }
```

Elemente de interfata: display-box 2/2

```
public:
   void setValue(T newValue);
public:
   void display();
};

template <class T>
void DisplayBox<T>::setValue(T newValue)
   {
       value = newValue;
   }

template <class T>
void DisplayBox<T>::display()
   {
       cout << label.c_str() << ":   " << value <<
   endl;
   }
```

Elemente de interfata: edit-box 1/2

```
template <class T>
class EditBox
{
private:
   string label;
private:
   T value;
public:
   EditBox(char *newLabel = "")
   {
       label = newLabel;
   }
public:
   void setLabel(string newLabel)
   {
       label = newLabel;
   }
```

Elemente de interfata: edit-box 2/2

```
public:
  T getValue()
  {
     T buffer;
     cout << label.c_str() << ":  ";
     cin >> buffer;
     return buffer;
  }
};
```

Elemente de interfata: menu 1/2

```
class Menu
{
private:
   string options[25];
private:
   int size;
public:
   Menu::Menu()
   {
       size = 0;
   }
public:
   void addOption(char *newOption)
   {
       options[size++] = string(newOption);
   }
```

Elemente de interfata: menu 2/2

```
public:
   void display()
   {
       int i;
       cout << "Menu:" << endl;
       for (i = 0; i < size; ++i)
             cout << options[i].c_str() << endl;
   }
public:
   int getOption()
   {
       int option;
       cout << "Option: ";
       cin >> option;
       return option;
   }
};
```

Ierarhia Figura – revizuita 1/2

```
class Figura {
public:
  virtual double arie() const = 0;
  virtual void print() = 0;
};
```

Ierarhia Figura – revizuita 2/2

```
class Drept : public Figura
{
public:
  Drept(double oLat = 0.0, double oLung = 0.0) {
    lat = oLat;
    lung = oLung;
  }
  virtual double arie() const {
    return lat * lung;
  }
  virtual void print() {
    cout << "Dreptughi" << endl;
    cout << "  Latime = " << this->lat << endl;
    cout << "  Lungime = " << this->lung << endl;
  }
private:
  double lat, lung;
};
```

Main() 1/2

```
Menu main;
 Figura* fig[100];
 int nFig = 0;
 EditBox<double> dbl;
 int option;
 double lung, lat;
 main.addOption("1. Adauga Dreptunghi.");
 main.addOption("2. Adauga Cerc.");
 main.addOption("3. Afiseaza.");
 main.addOption("0. Terminat.");
 main.display();
```

Main() 2/2

```
while ((option = main.getOption()) >0 ) {
     switch (option)
     {
          case 1 :
                dbl.setLabel("Latime");
                lat = dbl.getValue();
                dbl.setLabel("Lungime");
                lung = dbl.getValue();
                fig[nFig] = new Drept(lat, lung);
                break;
          case 2: /*...*/ break;
          case 3:
                for (int i = 0; i < nFig; ++i)
                      fig[i]->print();
     }
     ++nFig;
     main.display();
   }
```