Facultatea de Matematică și Informatică
Universitatea din București

Programare orientată pe obiecte

- suport de curs -

Anca Dobrovăț
Andrei Păun

An universitar 2024 – 2025
Semestrul I
Seriile 13, 14 și 15

Curs 5

Cuprins

Proiectarea descendenta a claselor. Moştenirea în C++.

- Controlul accesului la clasa de bază.
- Constructori, destructori şi moştenire.
- Redefinirea membrilor unei clase de bază într-o clasă derivată.
- Declarații de acces.

Obs: în acest curs, exemplele vor fi luate, în principal, din cartea lui B. Eckel - Thinking in C++.

Moștenirea în C++

- important în C++ - reutilizare de cod;
- reutilizare de cod prin creare de noi clase (nu se dorește crearea de clase de la zero);
- 2 modalități (compunere și moștenire);
- “compunere” - noua clasă “este compusă” din obiecte reprezentând instanțe ale claselor deja create;
- “moștenire” - se creează un nou tip al unei clase deja existente.

Moștenirea în C++

Exemplu: compunere

```
class Proiector { float scara;
public:
    Proiector() { scara = 1; }
    void set(int sc) { scara = sc; }
};
class Sala { int numar; Proiector proiector;   // compunere
public:
   Sala() { numar = 118; }
   const Proiector& getProiector() { return proiector; }
   void f(int nr) { numar = nr; }
};
int main() {
    Sala sala;
    sala.f(47);
    sala.getProiector().set(37); // Access the embedded object
}
```

Moștenirea în C++

C++ permite moștenirea, ceea ce înseamnă că putem deriva o clasă din altă clasă de bază sau din mai multe clase.

Prin derivare se obțin clase noi, numite clase derivate, care moștenesc proprietățile unei clase deja definite, numită clasă de bază.

Clasele derivate conțin toți membrii clasei de bază, la care se adaugă noi membri, date și funcții membre.

Dintr-o clasă de bază se poate deriva o clasă care, la rândul său, să servească drept clasă de bază pentru derivarea altora. Prin această succesiune se obține o **ierarhie de clase**.

Se pot defini clase derivate care au la bază mai multe clase, înglobând proprietățile tuturor claselor de bază, procedeu ce poartă denumirea de **moștenire multiplă.**

Moștenirea în C++

C++ permite moștenirea, ceea ce înseamnă că putem deriva o clasă din altă clasă de bază sau din mai multe clase.

Sintaxa:

class Clasa_Derivata : [modificatori de acces] Clasa_de_Baza { .... } ;

sau

class Clasa_Derivata : [modificatori de acces] Clasa_de_Baza1, [modificatori de acces] Clasa_de_Baza2, [modificatori de acces] Clasa_de_Baza3 .....

Clasa de bază se mai numește clasă părinte sau superclasă, iar clasa derivată se mai numește subclasă sau clasă copil.

Moștenirea multiplă: foarte utilă în unele situații, însă inclusă din motive istorice. Majoritatea limbajelor apărute ulterior au eliminat-o din cauza complexității.

Moștenirea în C++

Exemplu: moștenire

```
class Dispozitiv {   bool in_use;
   public:
      Dispozitiv() { in_use = false; }
      void start() { in_use = true; }
      bool status() const { return in_use; }
      void stop() { in_use = false; }
};
class Proiector : public Dispozitiv {
   int scara;
public:
   Proiector() { scara = 1; }
   void foloseste() {
      start(); // Different name call
      std::cout << "proiecteaza lectie\n";
   }
bool status() {
    return Dispozitiv::status() &&
          scara > 0; // Same-name function call
      }
};
```

```
int main() {
    std::cout << sizeof(Dispozitiv) << ' ' << sizeof(Proiector);

    Proiector pr;
     pr.start(); // funcție din interfața clasei Dispozitiv
     pr.stop();
     pr.foloseste();
     std::cout << pr.status() ? "on" : "off"; // Redefined
functions hide base versions:
}
```

Moștenirea în C++

Moștenire vs. Compunere

Moștenirea este asemănătoare cu procesul de includere a obiectelor în obiecte (procedeu ce poartă denumirea de compunere), dar există câteva elemente caracteristice moștenirii:

- implementarea poate fi comună mai multor clase;

- clasele pot fi extinse, fără a recompila codul care depinde de clasa de bază inițială;

- funcţiile ce utilizează obiecte din clasa de bază pot utiliza și obiecte din clasele derivate din această clasă.

Moștenirea în C++

Inițializare de obiecte

Foarte important în C++: garantarea inițializării corecte => trebuie să fie asigurată și la compunere, și la moștenire.
La crearea unui obiect, compilatorul trebuie să garanteze apelul TUTUROR sub-obiectelor.

Problemă: - cazul sub-obiectelor care nu au constructori impliciți sau schimbarea valorii unui argument default în constructor.

De ce? - constructorul noii clase nu are permisiunea să acceseze datele **private** ale sub-obiectelor, deci nu le poate inițializa direct.

Rezolvare: - o sintaxă specială: ***lista de inițializare pentru constructori***.

Moștenirea în C++

Exemplu: lista de inițializare pentru constructori

```
class Dispozitiv {
      int nr_butoane;
      public:
         Dispozitiv(int nr) {nr_butoane = nr;}
 };

class Proiector: public Dispozitiv {
      public:
         Proiector(int);
};
Proiector::Proiector (int nr) : Dispozitiv (nr) { … }
```

Moștenirea în C++

Exemplu 2: lista de inițializare pentru constructori

```
class Telecomanda {
        int nr_baterii;
        public:  Telecomanda(int nr) {nr_baterii = nr;}
 };
class Dispozitiv {
        int nr_butoane;
        public:  Dispozitiv(int nr) {nr_butoane = nr;}
 };
class Proiector: public Dispozitiv {
        Telecomanda telecomanda; // obiect telecomanda = subobiect în cadrul clasei Proiector
        Public:  Proiector(int, int);
};
```

```
Proiector::Proiector(int i, int j) :
        Dispozitiv (i), telecomanda (j) { … }
```

Moștenirea în C++

Chestiune specială: "pseudo - constructori" pentru tipuri primitive

- membrii de tipuri primitive (int, char, double) nu au constructori;

- soluție: C++ permite tratarea tipurilor predefinite asemănător unei clase cu o singură dată membră și care are un constructor parametrizat.
- Obs: tipurile din STL sunt clase, nu tipuri primitive:
  - Exemple: std::string, std::vector

Moștenirea în C++

```
class Proiector {
    int nr_butoane;   // sau int butoane{7}; // sau int butoane = 7;
    float scara;
    char categorie;
public:
    Proiector() : nr_butoane(7), scara(1.4), categorie('a') {}
};

int main() {
   Proiector pr;
   int i(100); // Applied to ordinary definition
   int* ip = new int(47);  delete ip;
}
```

Moștenirea în C++

Exemplu: compunere și moștenire

```
class A { int i;
public:
   A(int ii) : i(ii) {}
   ~A() {}
   void f() const {}
};

class B { int i;
public:
   B(int ii) : i(ii) {}
   ~B() {}
   void f() const {}
};

class C : public B {
   A a;
public:
   C(int ii) : B(ii), a(ii) {}
   ~C() {} // Calls ~A() and ~B()
   void f() const
        { // Redefinition
            a.f();
            B::f();
        }
};
int main() {
   C c(47);
}
```

Moștenirea în C++

Constructorii clasei derivate

Pentru crearea unui obiect al unei clase derivate, **se creează inițial un obiect al clasei de bază** prin apelul constructorului acesteia, apoi se adaugă elementele specifice clasei derivate prin apelul constructorului clasei derivate.

Declarația obiectului derivat trebuie să conțină valorile de inițializare, **atât pentru elementele specifice, cât și pentru obiectul clasei de bază**.

Această specificare se atașează la antetul funcției constructor a clasei derivate.

În situația în care clasele de bază au definit **constructor implicit** sau **constructor cu parametri impliciți**, nu se impune specificarea parametrilor care se transferă către obiectul clasei de bază.

Moștenirea în C++

Constructorii clasei derivate *Constructorul parametrizat*

```
class Forma {
protected:
    int h;
public:
    Forma(int a = 0) : h(a) {}
};

class Cerc: public Forma {
protected:
    float raza;
public:
    Cerc(int h=0, float r=0) : Forma(h), raza(r) {}
};
```

Problema cu cercul și elipsa

Moștenirea în C++

Constructorii clasei derivate

Constructorul de copiere

Se pot distinge mai multe situații.

1) Dacă ambele clase, atât clasa derivată, cât și clasa de bază, nu au definit constructor de copiere, se apelează constructorul implicit creat de compilator. Copierea se face membru cu membru.

2) Dacă clasa de bază are constructorul de copiere definit, dar clasa derivată nu, pentru clasa derivată compilatorul creează un constructor implicit care apelează constructorul de copiere al clasei de bază. (poate fi considerată un caz particular al primei situații, deoarece și partea de bază poate fi privită ca un fel de membru, iar la copiere se apelează cc pentru fiecare membru).

3) Dacă se definește constructor de copiere pentru clasa derivată, acestuia îi revine în totalitate sarcina transferării valorilor corespunzătoare membrilor ce aparțin clasei de bază.

Moștenirea în C++

Constructorii clasei derivate

Constructorul de copiere

```
class Forma {
protected:     int h;
public:
   Forma(const Forma& ob), h(ob.h)  { }
};

class Cerc: public Forma  {
protected:
   float raza;
public:
   Cerc(const Cerc&ob):Forma(ob), raza(ob.raza) { }
};
```

Moștenirea în C++

Ordinea apelării constructorilor și destructorilor

- constructorii sunt apelați în ordinea definirii obiectelor ca membri ai clasei şi în ordinea moștenirii:

- la fiecare nivel **se apelează:**
  - **întâi constructorul de la moștenire**,
  - apoi **constructorii din obiectele membru** în clasa respectivă (care sunt apelați în ordinea definirii)
  - și la final se merge pe următorul nivel în ordinea moștenirii;

  - destructorii sunt executați în ordinea inversă a constructorilor

Moștenirea în C++

Ordinea apelării constructorilor și destructorilor

```
class A{
public:
    A(){cout<<"A ";}
    ~A(){cout<<"~A ";}
};

class C{
public:
    C(){cout<<"C ";}
    ~C(){cout<<"~C ";}
};
```

Ordine: C B A D ~D ~A ~B ~C

```
class B{
    C ob;
public:
    B(){cout<<"B ";}
    ~B(){cout<<"~B ";}
};
class D: public B{
    A ob;
public:
    D(){cout<<"D ";}
    ~D(){cout<<"~D ";}
};
int main() {
    D s;
}
```

Moștenirea în C++

- ***Ordinea chemării constructorilor și destructorilor***

```
#define CLASS(ID) class ID { \
public: \
  ID(int)
     { cout << #ID " constructor\n"; } \
  ~ID()
     { cout << #ID " destructor\n"; } \
};

CLASS(Base1);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);
```

```
class Derived1 : public Base1 {
  Member1 m1;
  Member2 m2;
public:
  Derived1(int) : m2(1), m1(2), Base1(3)
     {  cout << "Derived1 constructor\n";  }
  ~Derived1() { cout << "Derived1 destructor\n";  }
};

class Derived2 : public Derived1 {
  Member3 m3;
  Member4 m4;
public:
  Derived2() : m3(1), Derived1(2), m4(3)
     {  cout << "Derived2 constructor\n";  }
  ~Derived2() { cout << "Derived2 destructor\n";  }
};
int main() {  Derived2 d2;}
```

- ***Ordinea chemării constructorilor și destructorilor***

```
#define CLASS(ID) class ID { \
public: \
  ID(int)
     { cout << #ID " constructor\n"; } \
  ~ID()
     { cout << #ID " destructor\n"; } \
};

CLASS(Base1);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);
```

Se va afisa:

```
Base1 constructor
Member1 constructor
Member2 constructor
Derived1 constructor
Member3 constructor
Member4 constructor
Derived2 constructor
Derived2 destructor
Member4 destructor
Member3 destructor
Derived1 destructor
Member2 destructor
Member1 destructor
Base1 destructor
```

Redefinirea funcțiilor membre

Clasa derivată are acces la toți membrii cu acces **protected** sau **public** ai clasei de bază.

Este permisă supradefinirea funcțiilor membre clasei de bază cu funcții
membre ale clasei derivate.

- 2 modalități de a redefini o funcţie membră:
  - **cu același antet ca în clasa de bază** (“redefining” - în cazul funcțiilor oarecare / “overriding” - în cazul funcțiilor virtuale);
  - **cu schimbarea listei de argumente sau a tipului returnat**.

Redefinirea funcțiilor membre

Exemplu: - păstrarea antetului/tipului returnat

```
class Baza {
public:
  void afis( )  {      cout<<"Baza\n";   }
};

class Derivata : public Baza  {
public:
  void afis( )  {      Baza::afis();      cout<<"si Derivata\n";   }
};

int main( ) {
Derivata d;
d.afis( ); // se afiseaza “Baza si Derivata”
}
```

Redefinirea funcţiilor membre

Exemplu: - nepăstrarea antetului/tipului returnat

```
class Baza {
public:
   void afis( )   {       cout<<"Baza\n";    }
};

class Derivata : public Baza {
public:
   void afis (int x)    {
       Baza::afis();
       cout<<"si Derivata\n";    }
};

int main( ) {
    Derivata d;
    d.afis(); //nu exista Derivata::afis( )
```

Obs: la redefinirea unei funcţii din clasa de baza, toate celelalte versiuni sunt automat ascunse!

Redefinirea funcțiilor membre

Care este efectul codului următor?

```
class Base {
public:
  int f() const { cout << "Base::f()\n";  return 1;  }
  int f(string) const { return 1; }
  void g() { }
};

class Derived1 : public Base {
public:    void g() const {}
};

class Derived2 : public Base {
public:
  // Redefinition:
  int f() const { cout << "Derived2::f()\n"; return 2;  }
```

```
int main() {
  string s("hello");

  Derived1 d1;
  int x = d1.f();
  cout<<d1.f(s);

  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // string version hidden
```

Redefinirea funcțiilor membre

Care este efectul codului următor?

```
class Base {
public:
  int f() const { cout << "Base::f()\n";  return 1;  }
  int f(string) const { return 1; }
  void g() { }
};
class Derived3 : public Base {
public:
  // Change return type:
  void f() const { cout << "Derived3::f()\n"; }
};
class Derived4 : public Base {
public:
  // Change argument list:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
```

```
int main() {

  Derived3 d3;
//!  x = d3.f(); // return int version hidden

  Derived4 d4;
//!  x = d4.f(); // f() version hidden
  x = d4.f(1);
}
```

Redefinirea funcțiilor membre

Obs:

Schimbarea interfeței clasei de bază prin modificarea tipului returnat sau a signaturii unei funcții, înseamnă, de fapt, utilizarea clasei în alt mod.

Scopul principal al moștenirii: polimorfismul.

Schimbarea signaturii sau a tipului returnat = schimbarea interfeței = contravine exact polimorfismului (un aspect esențial este păstrarea interfeței clasei de bază).

Redefinirea funcțiilor membre

Particularități la funcții

constructorii și destructorii nu sunt moșteniți până în C++11 (se redefinesc noi constr. și destr. pentru clasa derivată)

similar operatorul = (un fel de constructor care poate refolosi resurse alocate anterior – obiectul există deja)

Funcții care nu se moștenesc automat

Operatorul=

```
class Forma {
protected:     int h;
public:
   Forma& operator=(const Forma& ob)    {
      if (this!=&ob)        {          h = ob.h;       }
      return *this;    }
};

class Cerc: public Forma  {
protected:
   float raza;
public:
   Cerc& operator=(const Cerc& ob)     {
      if (this != &ob)        {          this->Forma::operator=(ob);       }
      return *this;  }
```

Moștenirea și funcțiile statice

Funcțiile membre statice se comportă exact ca și funcțiile membre nestatice:

Se moștenesc în clasa derivată.

Redefinirea unei funcții membre statice duce la ascunderea celorlalte supraîncărcări.

Schimbarea signaturii unei funcții din clasa de bază duce la ascunderea celorlalte versiuni ale funcției.

Dar: O funcție membră statică nu poate fi virtuală. (detalii mai târziu)

Moștenirea și funcțiile statice

```
class Base {
public:
   static void f()    {      cout << "Base::f()\n";    }
   static void g()    {      cout << "Base::f()\n";    }
 };

class Derived : public Base {
public:    // Change argument list:
   static void f(int x)    {      cout << "Derived::f(x)\n";    }
};

int main() {
   int x;
   Derived::f(); // ascunsa de supradefinirea f(x)
   Derived::f(x);
   Derived::g();
```

Moștenirea în C++

Modificatorii de acces la moștenire

class A : ***public*** B { /* declarații */};

class A : ***protected*** B { /* declarații */};

class A : ***private*** B { /* declarații */};

Dacă modificatorul de acces la moștenire este **public**, membrii din clasa de bază își păstrează tipul de acces și în derivată.

Dacă modificatorul de acces la moștenire este **private**, toți membrii din clasa de bază vor avea tipul de acces "private" în derivată, indiferent de tipul avut în bază.

Dacă modificatorul de acces la moștenire este **protected**, membrii "publici" din clasa de bază devin "protected" în clasa derivată, restul nu se modifică.

Moștenirea în C++

```
class Baza {
public:    void f() {cout<<"B";}
};

class Derivata : public Baza{  };

int main()
{
    Derivata ob1;
    ob1.f( );
}
```

Obs. Funcţia f( ) este accesibilă din derivată;

- modificatorul de acces la moștenire este "public", deci f( ) rămâne "public" și in Derivata, deci este accesibil la apelul din main().

```
class Baza {
public:    void f() {cout<<"B";}
};

class Derivata : private sau protected Baza{  };

int main()
{
    Derivata ob1;
    ob1.f( ); // inaccesibil
}
```

- Dacă modificatorul de acces la moștenire este "private", f( ) devine private în Derivata, deci inaccesibilă în main.
- Dacă modificatorul de acces la moștenire este "protected", f( ) devine protected în Derivata, deci inaccesibilă în main

Moștenirea în C++

Moștenirea cu specificatorul "private"

- inclusă în limbaj pentru completitudine;
- este mai bine a se utiliza compunerea în locul moștenirii private;
- toți membrii private din clasa de bază sunt ascunși în clasa derivată, deci inaccesibili;
- toți membrii public și protected devin private, dar sunt accesibili în clasa derivată;
- un obiect obținut printr-o astfel de derivare se tratează diferit față de cel din clasa de bază, e similar cu definirea unui obiect de tip bază în interiorul clasei noi (fără moștenire).
- dacă în clasa de bază o componentă era public, iar moștenirea se face cu specificatorul private, se poate reveni la public utilizând:
- ***using Baza::nume_componenta***

Moștenirea cu specificatorul "private"

```
class Pet {
public:
  char eat() const { return 'a'; }
  int speak() const { return 2; }
  float sleep() const { return 3.0; }
  float sleep( int) const { return 4.0; }
};

class Goldfish : Pet { // Private inheritance
public:
  using Pet::eat; // Name publicizes member
  using Pet::sleep; // Both overloaded members exposed
};
```

```
int main() {
  Goldfish bob;
  bob.eat();
  bob.sleep();
  bob.sleep(1);
//! bob.speak();// Error: private member
funcţion
}
```

Moștenirea cu specificatorul "protected"

- secțiuni definite ca protected sunt similare ca definire cu private (sunt ascunse de restul programului), cu excepția claselor derivate;
- good practice: cel mai bine este ca variabilele de instanță să fie PRIVATE și funcții care le modifică să fie protected;
- Sintaxă: ***class derivată: protected baza {…};***
- toți membrii publici și protected din baza devin protected în derivată;
- nu se prea folosește, inclusă în limbaj pentru completitudine.

Moștenirea în C++

Moștenirea cu specificatorul "public"

```
class Base {
protected:
 int i;
public:
  Base() : i(7) {}
};

class Derived1 : public Base { };

class Derived2 : public Derived1 {
public:
void set(int x) { i = x); }
};
```

```
int main() {
    Derived2;
    d.set(10);
}
```

- dacă în baza avem zone "protected" ele sunt transmise și în derivata1,2 tot ca protected, deci i e accesibil în funcția set()

Moștenirea în C++

Moștenirea cu specificatorul "protected"

```
class Base {
protected:
 int i;
public:
  Base() : i(7) {}
};

class Derived1 : protected Base {  };

class Derived2 : public Derived1 {
public:
void set(int x) { i = x); }
};
```

```
int main() {
    Derived2;
    d.set(10);
}
```

- dacă în baza avem zone "protected" ele sunt transmise și în derivata1,2 tot ca protected, deci i e accesibil în funcția set()

Moștenirea în C++

Moștenirea cu specificatorul "private"

```
class Base {
protected:
 int i;
public:
  Base() : i(7) {}
};

class Derived1 : private Base {  };

class Derived2 : public Derived1 {
public:
void set(int x) { i = x); }
};
```

```
int main() {
    Derived2;
    d.set(10);
}
```

- moștenire derivata1 din baza (private) atunci zonele protected devin private în derivata1 și neaccesibile în derivata2.

Moștenirea în C++

Compatibilitatea între o clasă derivată și clasa de bază. Conversii de tip

Deoarece clasa derivată moștenește proprietățile clasei de bază, între tipul clasă derivată și tipul clasă de bază se admite o anumită compatibilitate.

Compatibilitatea este valabilă numai pentru clase **derivate cu acces public** la clasa de bază și numai în sensul de la clasa derivată spre cea de bază, nu și invers.

Compatibilitatea se manifestă sub forma unor **conversii implicite de tip**:

- dintr-un obiect derivat într-un obiect de bază;
- dintr-un pointer sau referință la un obiect din clasa derivată într-un pointer sau referință la un obiect al clasei de bază.

Perspective

Cursul 6:

Funcții virtuale în C++.

- Parametrizarea metodelor (polimorfism la execuție).

- Funcții virtuale în C++. Clase abstracte.

- Destructori virtuali.