Facultatea de Matematica si Informatica
Universitatea din Bucuresti

Programare orientata pe obiecte

- suport de curs -

Dobrovat Anca - Madalina

An universitar 2019 – 2020
Semestrul I
Seria 25

Curs 7

13/11/2019

Agenda cursului

1. Proiectarea descendenta a claselor. Mostenirea in C++.

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

2. Polimorfims la executie prin funcții virtuale în C++.
   - Parametrizarea metodelor (polimorfism la executie).
     - Funcții virtuale în C++.
     - Clase abstracte.
     - Overloading pe functii virtuale
     - Destructori si virtualizare

1. Mostenirea in C++

- important in C++ - reutilizare de cod;
- reutilizare de cod prin creare de noi clase (nu se doreste crearea de clase de la zero);
- 2 modalitati (compunere si mostenire);
- “compunere” - noua clasa “este compusa” din obiecte reprezentand instante ale claselor deja create;
- “mostenire” - se creeaza un nou tip al unei clase deja existente.

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

1. Mostenirea in C++

Exemplu: compunere

```
class X {   int i;
public:
  X() { i = 0; }
};

class Y {     int i;
public:
  X x; // Embedded object
  Y() { i = 0; }
  void f(int ii) { i = ii; }
};

int main() {
  Y y;
  y.f(47);
  y.x.set(37); // Access the embedded object
}
```

1. Mostenirea in C++

C++ permite mostenirea ceea ce înseamnă că putem deriva o clasa din alta clasa de baza 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 baza se mai numeste clasa parinte sau superclasa, iar clasa derivata se mai numeste subclasa sau clasa copil.

1. Mostenirea in C++

Exemplu: mostenire

```
class X {   int i;
public:
  X() { i = 0; }
  void set(int ii) { i = ii; }
  int read() const { return i; }
  int permute() { return i = i * 47; }
};

class Y : public X {
  int i; // Different from X's i
public:
  Y() { i = 0; }
  int change() {   i = permute(); // Different name call
    return i;
  }
  void set(int ii) {    i = ii;
    X::set(ii); // Same-name function call  }
};
```

```
int main() {
  cout << sizeof(X) << sizeof(Y);
  Y D;
  D.change();
  // X function interface comes through:
  D.read();
  D.permute();
  // Redefined functions hide base
versions:
  D.set(12);
}
```

1. Mostenirea in C++

Initializare de obiecte

Foarte important in C++: garantarea initializarii corecte => trebuie sa fie asigurata si la compozitie si mostenire.
La crearea unui obiect, compilatorul trebuie sa garanteze apelul TUTUROR subobiectelor.

Problema: - cazul subobiectelor care nu au constructori impliciti sau schimbarea valorii unui argument default in constructor.

De ce? - constructorul noii clase nu are permisiunea sa acceseze datele **private** ale subobiectelor, deci nu le pot initializa direct.

Rezolvare: - o sintaxa speciala: ***lista de intializare pentru constructori***.

1. Mostenirea in C++

Exemple: lista de initializare pentru constructori

```
class Bar {
      int x;
public:
      Bar(int i) {x = i;}
};

class MyType: public Bar {
public:
      MyType(int);
};
```

MyType::MyType(int i) : Bar(i) { … }

1. Mostenirea in C++

Exemple: lista de initializare pentru constructori

```
class Alta_clasa {  int a;
public:  Alta_clasa(int i) {a = i;}   };

class Bar {  int x;
public:Bar(int i) {x = i;}   };

class MyType2: public Bar {
    Alta_clasa m; // obiect m = subobiect in cadrul clasei MyType2
public:
    MyType2(int);
};
```

MyType2::MyType2(int i) : Bar(i), m(i+1) { … }

1. Mostenirea in C++

Exemple: "pseudo - constructori" pentru tipuri de baza

- membrii de tipuri predefinite nu au constructori;
- solutie: C++ permite tratarea tipurilor predefinite asemanator unei clase cu o singura data membra si care are un constructor parametrizat.

```
class X {
        int i;
        float f;
        char c;
        char* s;
public:
        X() : i(7), f(1.4), c('x'), s("howdy") {  }
};
int main() {
  X x;
  int i(100);  // Applied to ordinary definition
  int* ip = new int(47);  }
```

1. Mostenirea in C++

Exemple: compozitie si mostenire

```
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);
}
```

1. Mostenirea in C++

Constructorii clasei derivate

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

Declaraţia obiectului derivat trebuie să conţină valorile de iniţializare, atât pentru elementele specifice, cât şi pentru obiectul clasei de bază.

Această specificare se ataşează la antetul funcţiei constructor a clasei derivate.

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

1. Mostenirea in 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;    }
};
```

1. Mostenirea in C++

Constructorii clasei derivate

Constructorul de copiere

Se pot distinge mai multe situaţii.

1) Dacă ambele clase, atât clasa derivată cât şi 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.

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

1. Mostenirea in 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;    }
};
```

1. Mostenirea in C++

Ordinea chemarii constructorilor si destructorilor

- constructorii sunt chemati in ordinea definirii obiectelor ca membri ai clasei si in ordinea mostenirii:

- la fiecare nivel se apeleaza intai constructorul de la mostenire, apoi constructorii din obiectele membru in clasa respectiva (care sunt chemati in ordinea definirii) si la final;

- se merge pe urmatorul nivel in ordinea mostenirii;

- destructorii sunt chemati in ordinea inversa a constructorilor

1. Mostenirea in C++

Ordinea chemarii constructorilor si destructorilor

```
class cls {    int x;
  public: cls(int i=0) {cout << "Inside constructor 1" << endl; x=i; }
             ~cls(){ cout << "Inside destructor 1" << endl;} };

class clss { int x;
                cls xx;
  public: clss(int i=0) {cout << "Inside constructor 2" << endl; x=i; }
             ~clss(){ cout << "Inside destructor 2" << endl;} };

class clss2 { int x;
                    clss xx;
                    cls xxx;
  public: clss2(int i=0) {cout << "Inside constructor 3" << endl; x=i; }
             ~clss2(){ cout << "Inside destructor 3" << endl;} };

int main() {   clss2 s;   }
```

1. Mostenirea in C++

Ordinea chemarii constructorilor si destructorilor

```
Inside constructor 1
Inside constructor 2
Inside constructor 1
Inside constructor 3
Inside destructor 3
Inside destructor 1
Inside destructor 2
Inside destructor 1
```

1. Mostenirea in C++

Ordinea chemarii constructorilor si 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;}
```

1. Mostenirea in C++

Ordinea chemarii constructorilor si 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
```

1. Mostenirea in C++

Redefinirea funcţiilor membre

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

Este permisă supradefinirea funcţiilor membre clasei de bază cu funcţii membre ale clasei derivate.

- 2 modalitati de a redefini o functie membra:
  - cu acelasi antet ca in clasa de baza (“redefining” - in cazul functiilor oarecare / “overloading” - in cazul functiilor virtuale);
  - cu schimbarea listei de argumente sau a tipului returnat.

1. Mostenirea in C++

Redefinirea funcţiilor membre

Exemplu: - pastrarea 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”
}
```

1. Mostenirea in C++

Redefinirea funcţiilor membre
Exemplu: - nepastrarea 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( )
   d.afis(3); }
```

In general, redefinirea unei functii din clasa de baza, toate celelalte versiuni sunt automat ascunse.

1. Mostenirea in C++

Redefinirea funcţiilor membre

Care este efectul codului urmator?

```
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;  }
};
```

```
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;
  }
};
```

1. Mostenirea in C++

Redefinirea funcţiilor membre

Care este efectul codului urmator?

```
class Base {  public:
  int f() const;       int f(string) const;     void g();  };

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

class Derived2 : public Base {
public:  // Redefinition:  int f() const;  };

class Derived3 : public Base {
public:  // Change return type:  void f() const;  };

class Derived4 : public Base {
public:  // Change argument list:  int f(int) const ; };
```

```
int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // string version hidden
  Derived3 d3;
//!  x = d3.f(); // return int version
hidden
  Derived4 d4;
//!  x = d4.f(); // f() version hidden
  x = d4.f(1);
}
```

1. Mostenirea in C++

Redefinirea funcţiilor membre

Observatie:

Schimbarea interfetei clasei de baza prin modificarea tipului returnat sau a signaturii unei functii, inseamna, de fapt, utilizarea clasei in alt mod.

Scopul principal al mostenirii: polimorfismul.

Schimbarea signaturii sau a tipului returnat = schimbarea interfetei = contravine exact polimorfismului (un aspect esential este pastrarea interfetei clasei de baza).

1. Mostenirea in C++

Redefinirea funcţiilor membre

Particularitati la functii

- constructorii si destructorii nu sunt mosteniti (se redefiniesc noi constr. si destr. pentru clasa derivata)

- similar operatorul = (un fel de constructor)

1. Mostenirea in C++

Functii care nu se mostenesc 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;  }
};
```

1. Mostenirea in C++

Mostenirea si functiile statice

Functiile membre statice se comporta exact ca si functiile nemembre:

1. Se mostenesc in clasa derivata.

2. Redefinirea unei functii membre statice duce la ascunderea celorlalte supraincarcari.

3. Schimbarea signaturii unei functii din clasa de baza duce la ascunderea celorlalte versiuni ale functiei.

Dar: O functie membra statica nu poate fi virtuala.

1. Mostenirea in C++

Mostenirea si functiile 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();
}
```

1. Mostenirea in C++

Modificatorii de acces la mostenire

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

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

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

Dacă modificatorul de acces la mostenire este **public**, membrii din clasa de baza isi pastreaza tipul de acces si in derivata.

Dacă modificatorul de acces la mostenire este **private**, toti membrii din clasa de baza vor avea tipul de acces “private” in derivata, indiferent de tipul avut in baza.

Dacă modificatorul de acces la mostenire este **protected**, membrii “publici” din clasa de baza devin “protected” in clasa derivata, restul nu se modifica.

1. Mostenirea in C++

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

class Derivata : public Baza{  };

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

Obs. Functia f( ) este accesibila din derivata;

- modificatorul de acces la mostenire este "public", deci f( ) ramane "public" si in Derivata, deci este accesibil la apelul din main().

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

class Derivata : private/protected Baza{  };

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

- Daca modificatorul de acces la mostenire este "private", f( ) devine private in Derivata, deci inaccesibila in main.
- Daca modificatorul de acces la mostenire este "protected", f( ) devine protected in Derivata, deci inaccesibila in main.

1. Mostenirea in C++

Mostenirea cu specificatorul "private"

- inclusa in limbaj pentru completitudine;
- este mai bine a se utiliza compunerea in locul mostenirii **private**;
- toti membrii **private** din clasa de baza sunt ascunsi in clasa derivata, deci inaccesibili;
- toti membrii **public** si **protected** devin **private**, dar sunt accesibile in clasa derivata;
- un obiect obtinut printr-o astfel de derivare se trateaza diferit fata de cel din clasa de baza, e similar cu definirea unui obiect de tip baza in interiorul clasei noi (fara mostenire).
- daca in clasa de baza o componenta era **public**, iar mostenirea se face cu specificatorul **private**, se poate reveni la public utilizand: ***using Baza::nume_functie_membra***

1. Mostenirea in C++

Mostenirea 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 function
}
```

1. Mostenirea in C++

Mostenirea cu specificatorul "protected"

- sectiuni definite ca protected sunt similare ca definire cu private (sunt ascunse de restul programului), cu exceptia claselor derivate;

- good practice: cel mai bine este ca variabilele de instanta sa fie PRIVATE si functii care le modifica sa fie protected;

- Sintaxa: ***class derivata1: protected baza {…};***

- toti membrii publici si protected din baza devin protected in derivata;

- nu se prea foloseste, inclusa in limbaj pentru completitudine.

1. Mostenirea in C++

Mostenirea cu specificatorul "protected"

```
class Base {   int i;
protected:
  int read() const { return i; }
  void set(int ii) { i = ii; }
public:
  Base(int ii = 0) : i(ii) {}
  int value(int m) const { return m*i; }
};

class Derived : public Base {   int j;
public:
  Derived(int jj = 0) : j(jj) {}
  void change(int x) { set(x); }
};
```

```
int main() {
  Derived d;
  d.change(10);
}
```

1. Mostenirea in C++

Mostenirea cu specificatorul "protected"

Exemplu:

- mostenire: derivata1 din baza (public) si derivata2 din derivata1 (public)
- daca in baza avem zone "protected" ele sunt transmise si in derivata1,2 tot ca protected

- mostenire derivata1 din baza (private) atunci zonele protected devin private in derivata1 si neaccesibile in derivata2.

1. Mostenirea in C++

Mostenire multipla (MM)

- putine limbaje au MM;
- mostenirea multipla e complicata: ambiguitate;
- nu e nevoie de MM (se simuleaza cu mostenire simpla);
- se mosteneste in acelasi timp din mai multe clase;

Sintaxa: **class derivata: public Baza1, public Baza2 { … };**

1. Mostenirea in C++

Mostenire multipla (MM)

Exemplu:

```
class Imprimanta {  };
class Scaner {  };
class Multifunctionala: public Imprimanta, public Scaner {   };
```

Ce ar putea crea probleme in cazul urmator?

```
class Produs{  };
class Perisabil: public Produs{  };
class Promotie: public Produs{  };
class Perisabil_la_promotie: public Perisabil, public Promotie {  };
```

In **Perisabil_la_promotie** avem de doua ori variabilele din baza!!

1. Mostenirea in C++

Mostenire multipla (MM)

```
class base {   public:       int i; };
class derived1 : public base {  public:       int j; };
class derived2 : public base {   public:        int k; };
class derived3 : public derived1, public derived2 {public:     int sum; };

int main() {
derived3 ob;
ob.i = 10; // this is ambiguous, which i???  // expl ob.derived1::i
ob.j = 20;
ob.k = 30;
ob.sum = ob.i + ob.j + ob.k; // i ambiguous here, too
cout << ob.i << " "; // also ambiguous, which i?
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

1. Mostenirea in C++

Mostenire multipla (MM)

- dar daca avem nevoie doar de o copie lui i?

- nu vrem sa consumam spatiu in memorie;

- **folosim mostenire virtuala**:

```
class base {   public:       int i; };
class derived1 : virtual public base {  public:       int j; };
class derived2 : virtual public base {   public:        int k; };
class derived3 : public derived1, public derived2 {public:     int sum; };
```

Daca avem mostenire de doua sau mai multe ori dintr-o clasa de baza (fiecare mostenire trebuie sa fie virtuala) atunci compilatorul aloca spatiu pentru o singura copie
In clasele derived1 si 2 mostenirea e la fel ca mai inainte (nici un efect pentru virtual in acel caz)

2. Polimorfismul la executie prin functii virtuale

Putem avea si functii virtuale

- Functiile virtuale si felul lor de folosire: componenta IMPORTANTA a limbajului OOP.

- folosit pentru polimorfism la executie ---> cod mai bine organizat cu polimorfism.

- Codul poate “creste” fara schimbari semnificative: programe extensibile

- functii virtuale sunt definite in baza si redefinite in clasa derivata

- pointer de tip baza care arata catre obiect de tip derivat si cheama o functie virtuala definita in baza si in derivata executa FUNCTIA DIN CLASA DERIVATA.

- Poate fi vazuta si ca exemplu de separare dintre interfata si implementare

2. Polimorfismul la executie prin functii virtuale

Decuplare in privinta tipurilor.

Upcasting - Tipul derivat poate lua locul tipului de baza (foarte important pentru procesarea mai multor tipuri prin acelasi cod).

Functii virtuale: ne lasa sa chemam functiile pentru tipul derivat.

Problema cand facem apel la functie prin pointer (tipul pointerului ne da functia apelata)

2. Polimorfismul la executie prin functii virtuale

```
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {  public:
  void play(note) const {
    cout << "Instrument::play" << endl;  }
};

class Wind : public Instrument {
public:  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;  }
};

void tune(Instrument& i) {
  i.play(middleC); }

int main() {
  Wind flute;
  tune(flute); // Upcasting ===> se afiseaza Instrument::play
}
```

2. Polimorfismul la executie prin functii virtuale

In C ---> early binding la apel de functii - se face la compilare.

In C++ ---> putem defini late binding prin functii virtuale (late, dynamic, runtime binding) - se face apel de functie bazat pe tipul obiectului, la rulare (nu se poate face la compilare).

Late binding ===> prin pointeri!

Late binding pentru o functie: se scrie virtual inainte de definirea functiei.

Pentru clasa de baza: nu se schimba nimic!

Pentru clasa derivata: late binding insemna ca un obiect derivat folosit in locul obiectului de baza isi va folosi functia sa, nu cea din baza (din cauza de late binding).

Utilitate: putem extinde codul precedent fara schimbari in codul deja scris.

2. Polimorfismul la executie prin functii virtuale

```
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {  public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;  }
};

class Wind : public Instrument {
public:  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;  }
};

void tune(Instrument& i) {
  i.play(middleC); }

int main() {
  Wind flute;
  tune(flute); // Upcasting ===> se afiseaza Instrument::play
}
```

2. Polimorfismul la executie prin functii virtuale

```
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {  public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;  }
};

class Wind : public Instrument {
public:  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;  }
};

void tune(Instrument& i) {
  i.play(middleC); }

int main() {
  Wind flute;
  tune(flute); // Upcasting ===> se afiseaza Instrument::play
}
```

2. Polimorfismul la executie prin functii virtuale

Cum se face late binding

Tipul obiectului este tinut in obiect pentru clasele cu functii virtuale.

Late binding se face (uzual) cu o tabela de pointeri: vptr catre functii.

In tabela sunt adresele functiilor clasei respective (functiile virtuale sunt din clasa, celelalte pot fi mostenite, etc.).

Fiecare obiect din clasa are pointerul acesta in componenta.

La apel de functie membru se merge la obiect, se apeleaza functia prin vptr.

Vptr este initializat in constructor (automat).

2. Polimorfismul la executie prin functii virtuale

Cum se face late binding

```
class NoVirtual {  int a;
public:
  void x() const {}
  int i() const { return 1; } };

class OneVirtual {  int a;
public:
  virtual void x() const {}
  int i() const { return 1; } };

class TwoVirtuals {  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; } };
```

```
int main() {
  cout << "int: " << sizeof(int) <<
endl;
  cout << "NoVirtual: "
       << sizeof(NoVirtual) << endl;
  cout << "void* : " << sizeof(void*)
<< endl;
  cout << "OneVirtual: "
       << sizeof(OneVirtual) << endl;
  cout << "TwoVirtuals: "
       << sizeof(TwoVirtuals) << endl;
}
```

2. Polimorfismul la executie prin functii virtuale

Cum se face late binding

```
class Pet { public:
  virtual string speak() const { return " "; } };

class Dog : public Pet { public:
  string speak() const { return "Bark!"; } };

int main() {
  Dog ralph;
  Pet* p1 = &ralph;
  Pet& p2 = ralph;
  Pet p3;
  // Late binding for both:
  cout << "p1->speak() = " << p1->speak() <<endl;
  cout << "p2.speak() = " << p2.speak() << endl;
  // Early binding (probably):
  cout << "p3.speak() = " << p3.speak() << endl;
}
```

2. Polimorfismul la executie prin functii virtuale

Daca functiile virtuale sunt asa de importante de ce nu sunt toate functiile definite virtuale (din oficiu)?

Deoarece “costa” in viteza programului.

In Java sunt “default”, dar Java e mai lent.

Nu mai putem avea functii inline (ne trebuie adresa functiei pentru VPTR).

2. Polimorfismul la executie prin functii virtuale

Clase abstracte si functii virtuale pure

Clasa abstracta = clasa care are cel putin o functie virtuala PURA

Necesitate: clase care dau doar interfata (nu vrem obiecte din clasa abstracta ci upcasting la ea).

Eroare la instantierea unei clase abstracte (nu se pot defini obiecte de tipul respectiv).

Permisa utilizarea de pointeri si referinte catre clasa abstracta (pentru upcasting).

Nu pot fi trimise catre functii (prin valoare).

2. Polimorfismul la executie prin functii virtuale

Functii virtuale pure

Sintaxa: **virtual** tip_returnat nume_functie(lista_parametri) =0;
Ex: virtual int pura(int i)=0;

Obs: La mostenire, daca in clasa derivata nu se defineste functia pura, clasa derivata este si ea clasa abstracta ---> nu trebuie definita functie care nu se executa niciodata

UTILIZARE IMPORTANTA: prevenirea “object slicing”.

2. Polimorfismul la executie prin functii virtuale

Clase abstracte si functii virtuale pure

```
class Pet {  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " + favoriteActivity;
  }
};
```

```
void describe(Pet p) { // Slicing
  cout << p.description() <<
endl;
}

int main() {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
}
```

2. Polimorfismul la executie prin functii virtuale

Overload pe functii virtuale

Obs. Nu e posibil overload prin schimbarea tipului param. de intoarcere (e posibil pentru ne-virtuale)

De ce. Pentru ca se vrea sa se garanteze ca se poate chema baza prin apelul respectiv.

Exceptie: pointer catre baza intors in baza, pointer catre derivata in derivata

2. Polimorfismul la executie prin functii virtuale

Overload pe functii virtuale

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

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

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

```
int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // string version
hidden
```

2. Polimorfismul la executie prin functii virtuale

Overload pe functii virtuale

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

class Derived3 : public Base {public:
//! 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() {
  string s("hello");
  Derived4 d4;
  x = d4.f(1);
//!  x = d4.f(); // f() version hidden
//!  d4.f(s); // string version hidden
  Base& br = d4; // Upcast
//!  br.f(1); // Derived version unavailable
  br.f();   br.f(s); // Base version available}
```

2. Polimorfismul la executie prin functii virtuale

Constructori si virtualizare

OBS. NU putem avea constructori virtuali.

In general pentru functiile virtuale se utilizeaza late binding, dar in utilizarea functiilor virtuale in constructori, varianta locala este folosita (early binding)

De ce?

Pentru ca functia virtuala din clasa derivata ar putea crede ca obiectul e initializat deja

Pentru ca la nivel de compilator in acel moment doar VPTR local este cunoscut

2. Polimorfismul la executie prin functii virtuale

Destructori si virtualizare

Este uzual sa se intalneasca.

Se cheama in ordine inversa decat constructorii.

Daca vrem sa eliminam portiuni alocate dinamic si pentru clasa derivata dar facem upcasting trebuie sa folosim destructori virtuali.

2. Polimorfismul la executie prin functii virtuale

Destructori si virtualizare

```
class Base1 {public:  ~Base1() { cout << "~Base1()\n"; } };

class Derived1 : public Base1 {public:  ~Derived1() { cout << "~Derived1()\n"; } };

class Base2 {public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {public:  ~Derived2() { cout << "~Derived2()\n"; } };

int main() {
  Base1* bp = new Derived1;
  delete bp; // Afis: ~Base1()
  Base2* b2p = new Derived2;
  delete b2p; // Afis: ~Derived2()  ~Base2()
}
```

2. Polimorfismul la executie prin functii virtuale

Destructori virtuali puri

Utilizare: recomandat sa fie utilizat daca mai sunt si alte functii virtuale.
Restrictie: trebuiesc definiti in clasa (chiar daca este abstracta).
La mostenire nu mai trebuiesc redefiniti (se construieste un destructor din oficiu)
De ce? Pentru a preveni instantierea clasei.
Obs. Nu are nici un efect daca nu se face upcasting.

```
class AbstractBase {
public:
  virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};
// No overriding of destructor necessary?
int main() { Derived d; }
```

2. Polimorfismul la executie prin functii virtuale

Functii virtuale in destructori

La apel de functie virtuala din functii normale se apeleaza conform VPTR
In destructori se face early binding! (apeluri locale)
De ce? Pentru ca acel apel poate sa se bazeze pe portiuni deja distruse din obiect

```
class Base { public:
  virtual ~Base() {  cout << "Base1()\n";     f();   }
  virtual void f() { cout << "Base::f()\n"; }
};
class Derived : public Base { public:
  ~Derived() { cout << "~Derived()\n"; }
  void f() { cout << "Derived::f()\n"; }
};

int main() {
  Base* bp = new Derived; // Afis: ~Derived() Base1() Base::f()
  delete bp;
}
```

2. Polimorfismul la executie prin functii virtuale

Downcasting

Folosit in ierarhii polimorfice (cu functii virtuale)

Problema: upcasting e sigur pentru ca respectivele functii trebuie sa fie definite in baza, downcasting e problematic

Explicit cast prin: **dynamic_cast**

Daca stim cu siguranta tipul obiectului putem folosi "static_cast".
Static_cast intoarce pointer catre obiectul care satiface cerintele sau 0.

Foloseste tabelele VTABLE pentru determinarea tipului

2. Polimorfismul la executie prin functii virtuale

Downcasting

```
class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast
  Dog* d1 = dynamic_cast<Dog*>(b);  // Afis - 0; Try to cast it to Dog*:
  Cat* d2 = dynamic_cast<Cat*>(b);// Try to cast it to Cat*:
  // b si d2 retin aceeasi adresa
  cout << "d1 = " << d1 << endl;
  cout << "d2 = " << d2 << endl;
cout << "b  = " << b << endl;
}
```

2. Polimorfismul la executie prin functii virtuale

Downcasting

```
class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
  Circle c;
  Shape* s = &c; // Upcast: normal and OK
  // More explicit but unnecessary:
  s = static_cast<Shape*>(&c);
  // (Since upcasting is such a safe and
common
  // operation, the cast becomes cluttering)
  Circle* cp = 0;
  Square* sp = 0;
// Static Navigation of class hierarchies
requires extra type information:
if(typeid(s) == typeid(cp)) // C++ RTTI
    cp = static_cast<Circle*>(s);
  if(typeid(s) == typeid(sp))
    sp = static_cast<Square*>(s);
  if(cp != 0)
    cout << "It's a circle!" << endl;
  if(sp != 0)
    cout << "It's a square!" << endl;
  // Static navigation is ONLY an
efficiency hack;
  // dynamic_cast is always safer.
However:
  // Other* op = static_cast<Other*>(s);
  // Conveniently gives an error
message, while
  Other* op2 = (Other*)s;
  // does not
}
```

Perspective

Cursul 9:

Controlul tipului în timpul rulării programului în C++.

1 Mecanisme de tip RTTI (Run Time Type Identification).
2 Moştenire multiplă şi identificatori de tip (dynamic_cast, typeid).