



Programare orientată pe obiecte

- suport de curs -

Andrei Păun

Anca Dobrovăț

An universitar 2019 – 2020

Semestrul II

Seriile 13, 14 și 21

Curs 6

24-25/3/2020



Cuprins

1. Tratarea excepțiilor în C++.
2. Proiectarea descendentă a claselor. Mostenirea în C++.
 - 2.1 Controlul accesului la clasa de bază.
 - 2.2 Constructori, destructori și moștenire.
 - 2.3 Redefinirea membrilor unei clase de bază într-o clasă derivată.
 - 2.4 Declarații de acces.

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



1. Tratarea exceptiilor in C++

- automatizarea procesarii erorilor
- try, catch throw
- block try arunca exceptie cu throw care este prinsa cu catch
- dupa ce este prinsa se termina executia din blocul catch si se da controlul “mai sus, nu se revine la locul unde s-a facut throw (nu e apel de functie).



1. Tratarea exceptiilor in C++

```
try {  
    // try block  
}  
catch (type1 arg) {  
    // catch block  
}  
catch (type2 arg) {  
    // catch block  
}  
catch (type3 arg) {  
    // catch block  
}...  
catch (typeN arg) {  
    // catch block  
}
```

- tipul argumentului arg din catch arata care bloc catch este executat
- daca nu este generata exceptie, nu se executa nici un bloc catch
- instructiunile catch sunt verificate in ordinea in care sunt scrise, primul de tipul erorii este folosit



1. Tratarea exceptiilor in C++

Observatii:

- daca se face throw si nu exista un bloc try din care a fost aruncata exceptia sau o functie apelata dintr-un bloc try: eroare
- daca nu exista un catch care sa fie asociat cu throw-ul respectiv (tipuri de date egale) atunci programul se termina prin terminate()
- terminate() poate sa fie redefinita sa faca altceva



1. Tratarea exceptiilor in C++

```
class TestTry {  
    int *v, n;           Semnalarea unei posibile erori la alocarea de memorie: bad_alloc  
    public:  
    TestTry(int a) {  
        try {  
            v = new int[a];  
        }  
        catch (bad_alloc Nume_Var) {  
            cout << "Allocation Failure\n";  
            exit(EXIT_FAILURE);  
        }  
        n = a;  
    }  
};  
  
int main() {  
    TestTry T(4);  
}
```



1. Tratarea exceptiilor in C++

```
class TestTry {  
    int *v, n;  
    public:  
    TestTry(int a) { ... }  
    void Test_Throw_ok () {  
        try {  
            throw 10;  
        }  
        catch (int x) {  
            cout << "Exceptie 10\n";  
        }  
    }  
};  
  
int main() {  
    TestTry T(4);  
    T.Test_Throw_ok();  
}
```

Tipul aruncat coincide cu tipul parametrului blocului catch

Exceptia este prinsa; se afiseaza
expresia din blocul catch



1. Tratarea exceptiilor in C++

Tipul aruncat nu coincide cu tipul parametrului blocului catch

```
class TestTry {  
    int *v, n;  
    public:  
    TestTry(int a) { ... }  
    void Test_Throw_NOTok ()  
        try { throw 10; }  
        catch (char x) {  
            cout << "Exceptie 10\n";  
        }  
}  
};  
  
int main() {  
    TestTry T(4);  
    T.Test_Throw_NOTok();  
}
```

Exceptia nu este prinsa



1. Tratarea exceptiilor in C++

Aruncarea unei exceptii dintr-o functie (throw in functie)

```
class TestTry {  
    int *v, n;  
    public:  
    TestTry(int a) { ... }  
    void Test_Throw_Functie() {  
        try {  
            Test(5);  
            Test(200);  
            Test(-300);  
            Test(22);  
        }  
        catch (int x) {  
            cout << "Exceptie pe valoarea " << x << "\n";  
        }  
    }  
};
```

```
void Test(int x)  
{  
    cout << "In functie x = " << x << "\n";  
    if (x < 0) throw x;  
}
```

```
int main() {  
    TestTry T(4);  
    T.Test_Throw_Functie();  
}
```

- In functie x = 5
- In functie x = 200
- In functie x = -300
- Exceptie pe valoarea -300



1. Tratarea exceptiilor in C++

Try-catch local, in functie, se continua executia programului

```
class TestTry {  
    int *v, n;  
    public:  
    TestTry(int a) { ... }  
    void Test_Try_Local()  
    {  
        int x;  
        x = -25;  
        Try_in_functie(x);  
        x = 13;  
        Try_in_functie(x);  
        n = x;  
        cout << n;  
    }  
};
```

```
void Try_in_functie(int x)  
{  
    try  
    {  
        if (x < 0) throw x;  
    }  
    catch(int x)  
    {  
        cout << "Exceptie pe valoarea " << x << "\n";  
    }  
}  
  
int main() {  
    TestTry T(4);  
    T.Test_Try_Local();  
}
```

•Exceptie pe valoarea -25

•13



1. Tratarea exceptiilor in C++

Exceptii multiple; catch (...)

```
void Exceptii_multiple(int x){  
    try{  
        if (x < 0) throw x; //int  
        if (x == 0) throw 'A'; //char  
        if (x > 0) throw 12.34; //double  
    }  
    catch(...) {  
        cout << "Catch macar una!\n";  
    }  
}
```

```
int main(){  
    Exceptii_multiple(-52);  
    Exceptii_multiple(0);  
    Exceptii_multiple(34);  
}
```



1. Tratarea exceptiilor in C++

- aruncarea de erori din clase de baza si derivate
- un catch pentru tipul de baza va fi executat pentru un obiect aruncat de tipul derivat
- sa se puna catch-ul pe tipul derivat primul si apoi catchul pe tipul de baza

```
class B { };  
class D: public B { };  
int main()  
{  
    D derived;  
    try {    throw derived;  }  
  
    catch(B b) {    cout << "Caught a base class.\n";  }  
  
    catch(D d) {    cout << "This won't execute.\n";  }  
    return 0;  
}
```



1. Tratarea exceptiilor in C++

Observatii:

void Xhandler(int test) throw(int, char, double)

- se poate specifica ce exceptii arunca o functie
- se restrictioneaza tipurile de exceptii care se pot arunca din functie
- un alt tip nespecificat termina programul:
 - apel la `unexpected()` care apeleaza `abort()`
 - se poate redefini
- re-aruncarea unei exceptii: `throw; // fara exceptie din catch`



1. Tratarea exceptiilor in C++

XVIII. Spuneți dacă programul de mai jos este corect. În caz afirmativ, spuneți ce afișează pentru o valoare întreagă citită egală cu 7, în caz negativ spuneți de ce nu este corect.

```
#include <iostream.h>
float f(int y)
{ try
  { if (y%2) throw y/2;
  }
  catch (int i)
  { if (i%2) throw;
    cout<<"Numarul " <<
  }
  return y/2;
}
int main()
{ int x;
  try
  { cout<<"Da-mi un nu
    cin>>x;
    if (x) f(x);
    cout<<"Numarul " <<
  }
  catch (int i)
  { cout<<"Numarul " <<
  }
  return 0;
}
```

```
Da-mi un numar intreg: -2    Numarul -2 nu e bun!
Da-mi un numar intreg: -1    Numarul 0 nu e bun!
Numarul -1 nu e bun!
Da-mi un numar intreg: 0     Numarul 0 nu e bun!
Da-mi un numar intreg: 1     Numarul 0 nu e bun!
Numarul 1 nu e bun!
Da-mi un numar intreg: 2     Numarul 2 nu e bun!
Da-mi un numar intreg: 3     Numarul 1 e bun!
Da-mi un numar intreg: 4     Numarul 4 nu e bun!
Da-mi un numar intreg: 5     Numarul 2 nu e bun!
Numarul 5 nu e bun!
Da-mi un numar intreg: 6     Numarul 6 nu e bun!
Da-mi un numar intreg: 7     Numarul 3 e bun!
Da-mi un numar intreg: 8     Numarul 8 nu e bun!
Da-mi un numar intreg: 9     Numarul 4 nu e bun!
Numarul 9 nu e bun!
Da-mi un numar intreg: 10    Numarul 10 nu e bun!
```



2. 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.



2. Mostenirea in C++

Exemplu: compunere

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

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
}
```




2. Mostenirea in C++

C++ permite mostenirea ceea ce înseamnă că putem deriva o clasa din alta clasa de baza 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 membrii, 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**.



2. 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.



2. 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);
}
```



2. Mostenirea in C++

Mostenire 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:

- codul poate fi comun mai multor clase;
- clasele pot fi extinse, fără a recompila clasele originare;
- funcțiile ce utilizează obiecte din clasa de bază pot utiliza și obiecte din clasele derivate din această clasă.



2. Mostenirea in C++

Modificatorii de acces la mostenire

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

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

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

Dacă lipsește modificatorul de acces, atunci e considerat implicit private.

Funcțiile membre din clasa derivată au acces doar la membrii publici și protected din clasa de bază.



2. 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.



2. Mostenirea in C++

```
class Baza { int a;
```

```
public:
```

```
    void f() {cout<<a;} /// a este privat dar accesibil in clasa
```

```
private:
```

```
    void g() {cout<<a;}  
};
```

```
class Derivata1 : protected Baza{
```

```
public:
```

```
    void h() {cout<<a;} /// a este privat, inaccesibil  
};
```

```
class Derivata2: public Derivata1 {
```

```
public:
```

```
    void z(){cout<<a;}  
};
```

```
int main(){
```

```
    Baza ob1;
```

```
    /// cout<<ob1.a; /* a este privat deci am  
    acces la el doar din Baza */  
}
```



2. 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 initializare pentru constructori*.



2. 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) { ... }
```



2. 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) { ... }
```



2. 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.



2. Mostenirea in C++

```
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);  
}
```



2. 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);  
}
```



2. Mostenirea in 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 implicați, nu se impune specificarea parametrilor care se transferă către obiectul clasei de bază.



2. 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; }  
};
```



2. Mostenirea in 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.
- 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.



2. 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;    }  
};
```



2. 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



2. Mostenirea in C++

Ordinea chemarii constructorilor si 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;  
}
```



2. Mostenirea in C++

Operatorul=

```
class Forma {  
    protected:  
        int h;  
    public:  
        Forma& operator=(const Forma& );  
};  
class Cerc: public Forma {  
    protected:  
        float raza;  
    public:  
        Cerc& operator=(const Cerc& );  
};
```

```
Forma& Forma::operator=(const Forma& ob) {  
    if (this!=&ob) { h = ob.h; }  
    return *this;  
}  
  
Cerc& Cerc::operator=(const Cerc& ob) {  
    if (this != &ob)  
    {  
        this->Forma::operator=(ob);  
    }  
    return *this;  
}
```



2. Mostenirea in C++

Redefinirea funcțiilor membre

```
class Forma {  
protected:  
    int h;  
public:  
    void afis() { cout<<h<<" "; }  
};
```

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

```
class Cerc: public Forma {  
protected:  
    float raza;  
public:  
    void afis() {  
        Forma::afis();  
        cout<<raza; }  
};
```

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



2. Mostenirea in 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 7:

Funcții virtuale în C++.

- Parametrizarea metodelor (polimorfism la executie).
- Funcții virtuale în C++. Clase abstracte.
- Destructori virtuali.