## Mecanisme de reutilizare a codului

- În cazul aplicaţiilor complexe, creşterea eficienţei activităţii de programare impune refolosirea codului existent.
- Există trei modalităţi de reutilizare a codului:
    - compunerea
    - agregarea
    - moştenirea

- Prin compunere, obiectul, care include un alt obiect, va avea ca elemente membre toate elementele membre ale obiectului inclus alături de cele specifice lui, accesul la membrii obiectului inclus realizându-se în mod corespunzător.
- Se obişnuieşte să se spună că relaţia de compunere implică "posesia" obiectului inclus. Echivalent, relaţia de compunere este o relaţie de tip "a avea" ("has a" relationship).
- Agregarea este un caz particular de compunere în care obiectul care face referire la un alt obiect nu are controlul/responsabilitatea existenţei obiectului la care face referire (acesta nu este efectiv o parte a sa). În mod uzual, în astfel de cazuri, referirea la un alt obiect se face prin intermediul pointerilor sau referinţelor.
- Moştenirea reprezintă o altă modalitate prin care se pot reutiliza şi extinde clasele existente fără a fi necesară rescrierea integrală a codului aferent.
- Se obişnuieşte să se spună că relaţia de moştenire o relaţie de tip "un fel de" ("kind of" relationship).
- Conceptul de moştenire se realizează prin intermediul mecanismului de derivare, cu ajutorul aşa-numitelor clase derivate.
- Moştenirea poate fi simplă sau multiplă, după cum clasa derivată moşteneşte caracteristicile de la o singură clasă de bază sau de la mai multe clase de bază.
- Derivarea unei clase se defineşte ca fiind crearea unei clase noi, numită clasă derivată, prin preluarea componentelor unei/unor clase de bază şi controlul accesului la acestea.
- O clasa de bază este o clasă generală şi esenţa derivării este refolosirea unui comportament definit anterior într-o clasă de bază
- Prin moştenire se defineşte o ierarhie de clase cu ajutorul căreia să se modeleze sisteme complexe. Proiectarea unei ierarhii de clase constituie activitatea fundamentală de implementare a unei aplicaţii orientate obiect.

**Exemple Compunere**

**1.**

```
#include <iostream>
using namespace std;

class X
{
    int i;
  public:
    X(int j=0) { i = j; }
    void set(int ii) { i = ii; }
    int get() const { return i; }
    int modif() { return i = i * 47; }
};

class Y
{
    int i;
   public:
    X x; // obiect component
         //(Embedded object)
    Y() { i = 0; }
    void set(int ii) { i = ii; }
    int get() const {   return i; }
};

class Z
{
    int i;
    X x; // Embedded object
 public:
    Z(){
        i = 0;
    }
  void set(int ii)
  {
        i = ii;     x.set(ii);
  }
  int get() const
  {   return i * x.get(); }

  int modif()
  {
      return x.modif();
  }
};

int main()
{
    Y y;
    y.set(10);
    y.x.set(20);
    // Acces la obiectul component
    cout<<y.get()<<" "<<
    y.x.modif()<<endl; //x.i = 940
    Z z;
    z.set(100);
    cout<<z.get()<<" "<<z.modif()<<endl;
    // ATENTIE LA ORDINEA EVALUARII
    // Intai se apeleaza metoda modif!
    return 0;
}
```

**REZULTATE**

10  940
470000  4700

**2.**
**Fişierul Punct2D.h:**

```
#ifndef POINT2D_H
#define POINT2D_H
#include <iostream>

class Punct2D
{
private:
    int m_nX;
    int m_nY;

public:
    Punct2D() : m_nX(0), m_nY(0)       // constructor implicit, cu lista de initializare
     {}
    Punct2D(int nX, int nY) : m_nX(nX), m_nY(nY)//ctor cu parametri, cu lista de initializare
     {}

    // Supraincarcarea operatorului de iesire
    friend std::ostream& operator<<(std::ostream& out, const Punct2D &cPunct)
    {
        out << "(" << cPunct.GetX() << ", " << cPunct.GetY() << ")";
        return out;
    }
    // Functii de acces
    void SetPunct(int nX, int nY)
    {
        m_nX = nX;
        m_nY = nY;
    }
     int GetX() const { return m_nX; }
    int GetY() const { return m_nY; }
};

#endif
```

**Fişierul Creatura.h:**

```
#ifndef CREATURE_H
#define CREATURE_H

#include <iostream>
#include <string>
#include "Punct2D.h"

class Creatura
{
private:
    std::string m_strNume;   // exemplu de utilizare a clasei string
    Punct2D m_cLocatie;

public:
    Creatura(std::string strNume, const Punct2D &cLocatie)
        : m_strNume(strNume), m_cLocatie(cLocatie)
    {
    }

    friend std::ostream& operator<<(std::ostream& out, const Creatura &cCreatura)
    {
        out << cCreatura.m_strNume.c_str() << " se afla la " << cCreatura.m_cLocatie;
        return out;
    }

    void MutaLa(int nX, int nY)
    {
        m_cLocatie.SetPunct(nX, nY);
    }
};

#endif
```

**Fişierul main.cpp**

```
#include <string>
#include <iostream>
#include "Creatura.h"

using namespace std;

int main()
{
    cout << "Numele creaturii: ";
    string cNume;
    cin >> cNume;
    Creatura cCreatura(cNume, Punct2D(4, 7));

    while (1)
    {
        cout << cCreatura << endl;
        cout << "Introduceti locatia X pentru creatura (-1 pt. a incheia): ";
        int nX=0;
        cin >> nX;
        if (nX == -1)
            break;

        cout << "Introduceti locatia Y pentru creatura (-1 pt. a incheia): ";
        int nY=0;
        cin >> nY;
        if (nY == -1)
            break;

        cCreatura.MutaLa(nX, nY);
        }

    return 0;
}
```

//Exemplu Agregare

```
#include <string>
using namespace std;

class Profesor
{
private:
    string m_strNume;
public:
    Profesor(string strNume)
        : m_strNume(strNume)
    {
    }

    string GetNume() { return m_strNume;
}
};

class Departament
{
private:
    Profesor *m_pcProfesor;
//Se presupune ca exista DOAR un prof.

public:
    Departament(Profesor
*pcProfesor=NULL)
        : m_pcProfesor(pcProfesor)
    {
    }
};

int main()
{
// Se creeaza un prof. in afara
// Departamentului

  Profesor *pProfesor = new
  Profesor("Popescu"); // creeaza prof.
  {
/* Se creeaza un Departament si se
foloseste constructorul cu parametrii
pentru a transmite prof. */

        Departament cDept(pProfesor);

   }// cDept iese din domeniu si este
    // distrus

    // pProfesor inca exista!

  delete pProfesor;

  return 0;
}
```

In concluzie, caracteristicile specifice  compunerii şi respective agregării sunt:

In cazul compunerii, clasa ce utilizează acest mecanism :

- Tipic utilizează variabile obişnuite
- Poate folosi pointeri dacă în clasa respectivă se controlează alocarea/dealocarea
- Este responsabilă pentru crearea/distrugerea subclaselor

In cazul agregării, clasa ce utilizează acest mecanism :

- Tipic utilizează variabile de tip pointer ce pointează spre obiecte create în afara clasei
- Poate folosi şi referinţe pentru a referi obiecte create în afara clasei
- Nu este responsabilă pentru crearea/distrugerea subclaselor

- **Moştenire. Derivarea claselor.**
- Sintaxa derivării în C++ este :

```
class D : [modificator acces] B1, [modificator acces] B2, ……,[modificator acces] Bn
{
  ……..
};
```

unde : modificator acces ::= [***virtual***][ ***private***| ***protected***| ***public***]
D este clasa derivată, iar B1, B2, …. Bn, sunt clasele de bază

- Dacă nu se specifică nici un modificator de acces, moştenirea se presupune ***private***
- Derivarea se poate face în mai multe moduri prin intermediul modificatorilor de acces :

| **SPECIFICATOR DE PROTECŢIE ÎN CLASA DE BAZĂ** | **MODIFICATOR DE ACCES LA DERIVARE** | **TIP DE ACCES ÎN CLASA DERIVATĂ** |
|---|---|---|
| ***private***<br>***protected***<br>***public*** | ***public***<br>***public***<br>***public*** | **INACCESIBIL**<br>**PROTEJAT**<br>**PUBLIC** |
| ***private***<br>***protected***<br>***public*** | ***protected***<br>***protected***<br>***protected*** | **INACCESIBIL**<br>**PROTEJAT**<br>**PROTEJAT** |
| ***private***<br>***protected***<br>***public*** | ***private***<br>***private***<br>***private*** | **INACCESIBIL**<br>**PRIVAT**<br>**PRIVAT** |

- Specificatorul de protecţie ***protected*** folosit în definiţia unei clase permite accesarea în clasa derivată a unor membri care nu fac parte din secţiunea publică a clasei de bază.
- În general, modificatorul de acces ***protected*** nu este folosit ca modificator de acces în procesul de derivare a unei clase.
- Se observă că, indiferent de specificatorul de acces (***public***, ***protected*** sau ***private***) folosit, membrii din secţiunea ***private*** a clasei de bază nu pot fi direct accesaţi în clasa derivată. Accesarea lor se face exclusiv prin funcţiile membru publice sau protejate moştenite de la clasa de bază.

- Constructorii, destructorul şi metoda care supraîncarcă operatorul de atribuire (=) **NU SE MOŞTENESC**.
- Evident, deoarece funcţiile ***friend*** nu aparţin unei clase, nici acestea nu se vor moşteni!
- La construcţia unui obiect dintr-o clasă derivată se construieşte mai întâi partea corespunzătoare moştenită din clasa de bază, în timp ce, la distrugerea unui obiect dintr-o clasă derivată se distruge mai întâi partea proprie şi apoi cea corespunzătoare moştenirii din clasa de bază.

Exemplul 1:

```
#include <iostream>
using namespace std;

class Baza
{
     public:
        Baza()
        {
            cout<<"Constructor Baza"<<endl;
        }
        ~Baza()
        {
            cout<<"Destructor Baza"<<endl;
        }
};

class Derivat:public Baza
{
     public:
        Derivat()
        {
            cout<<"Constructor Derivat"<<
            endl;
        }
        ~Derivat()
        {
            cout<<"Destructor Derivat"<<
            endl;
        }
};
```

```
int main()
{
     Derivat ob_derivat;

     return 0;

}
```

**REZULTATE**

Constructor Baza
Constructor Derivat
Destructor Derivat
Destructor Baza

Exemplul 2 :

```
#include <iostream>
using namespace std;

class Baza
{
        int    i;
   protected:
        int    j;
   public:
        Baza(){i=0;j=0;}
        void Setij(int x, int y)
        {
             i=x; j=y;
        }
        int Reti(){return i;}
        int Retj(){return j;}
};

class Derivat_pb:public Baza
{
        int k;
   public:
        Derivat_pb(int kd)
         {
              k=kd;
         }
         int Retk()
         {
// i=99; access to private member Baza::i
// is not allowed
              k=Reti();
              j=88;
              return k;
         }
};

class Derivat_pv:private Baza
{
        int k;
   public:
         Derivat_pv(int kd)
         {
              k=kd;
         }
         int Retk()
         {
            k=Reti();
            j=99;  //data cu acces protected
            return k;
         }
};

int main()
{
     Derivat_pb obj_deriv(3) ;
     obj_deriv.Setij(-3,-99);
     cout<<"obj_deriv i="
     <<obj_deriv.Reti()<<
     "  k="<<obj_deriv.Retk()<<endl;
     Derivat_pv obj_deriv_pv(3) ;
/* obj_deriv_pv.Setij(-3); // cannot
acces Baza:Seti through a private base
class*/
     cout<<"k obj_deriv_pv ="<<
     obj_deriv_pv.Retk()<<endl;

     return 0;
}
```

**REZULTATE**

obj_deriv i=-3  k=-3
k obj_deriv_pv =0

## Exemplul 5. Reutilizarea codului prin derivare

**A.**

```
#include <iostream>
#include <process.h>
using namespace std;

class Stack
{
   protected:
      enum {SIZE=20};
      int st[SIZE];
      int top;
   public:
      Stack()
         { top = -1; }
      void push(int var)
         { st[++top] = var; }
      int pop()
        { return st[top--]; }
};

class Stack_d: public Stack
{
   public:
   void push(int var)
   {
      if(top >= SIZE-1)
         { cout <<
 "Error: stack overflow"; exit(1); }
      Stack::push(var);
   }
   int pop()
   {
      if(top<0)
      {
 cout << "Error: stack underflow";
exit(1);
      }
        return Stack::pop();
      }
};

void main()
 {
   Stack_d s;
   s.push(11);
   s.push(12);
   s.push(13);
   cout << s.pop() << endl;
   cout << s.pop() << endl;
   cout << s.pop() << endl;
   cout << s.pop() << endl;  // oops
}
```

**B.**

```
#include <iostream>
#include <process.h>
using namespace std;

class Stack
{
   protected:
      enum {SIZE=20};
   private:
      int st[SIZE];
      int top;
   protected:
      int get_top()
      {
          return top;
      }
   public:
      Stack()
         { top = -1; }
      void push(int var)
         { st[++top] = var; }
      int pop()
        { return st[top--]; }
};
class Stack_d: public Stack
{
   public:
   void push(int var)    {
      if(get_top() >= SIZE-1)
         { cout <<
"Error: stack overflow"; exit(1); }
      Stack::push(var);
   }
   int pop()    {
      if(get_top()<0)
      {
     cout <<
"Error: stack underflow";;exit(1);
       }
        return Stack::pop();
      }
};
void main()
{
   Stack_d s;
 s.push(11); s.push(12); s.push(13);
   cout << s.pop() << endl;
   cout << s.pop() << endl;
   cout << s.pop() << endl;
   cout << s.pop() << endl; // oops
}
```

- În general există 4 combinaţii posibile între modalităţile de definire a constructorilor în clasa de bază şi în clasa derivată :

1. Clasa de bază nu are constructori şi clasa derivată nu are constructori definiţi explicit
   - În acest caz, în clasa derivată se pot instanţia doar obiecte neiniţializate, compilatorul generează automat constructori (implicit impliciţi!)

2. Clasa de bază are constructori şi clasa derivată nu are constructori definiţi explicit
   - În acest caz, clasa de bază trebuie să aibă definit constructorul implicit, iar în clasa derivată se pot instanţia doar obiecte neiniţializate, compilatorul generează automat constructor implicit pentru clasa derivată, care va apela automat constructorul implicit al clasei de bază.

3. Clasa de bază nu are constructori definiţi explicit şi clasa derivată are constructori
   - În acest caz compilatorul generează automat constructor implicit pentru clasa de bază, doar membrii clasei de bază **accesibili în clasa derivată** vor putea fi iniţializaţi

4. Atât clasa de bază cât şi clasa derivată au constructori definiţi explicit
   - În acest caz, se pot transmite argumente constructorilor claselor de bază (folosind o aşa-numită listă de iniţializare), cu condiţia ca aceştia să fie publici, cu următoarea sintaxă :

```
class B1; class B2; ..... class Bn;

class Deriv: public B1, public B2, ..., public Bn
{
........
public:
  Deriv(lista_argumente); // lista_argumente= lista_1 U lista_2 U ....  U lista_n
.............
};

Deriv:: Deriv(lista_argumente): B1(lista_1), B2(lista_2),.... , Bn(lista_n)
{
 ..........
}
```

- O sintaxă similară celei corespunzătoare cazului 4 (cu listă de iniţializare) se poate folosi şi în cazul compunerii **şi se poate extinde şi pentru datele membru de tipuri predefinite**

  Exemplu:

```
class Element
{
.......
public:
  Element(lista_el);
......
};

class Comp
{
  Element el;
.......
public:
  Comp(lista_comp);
......
};
```

Comp:: Comp(lista_comp) : el(lista_el) // **Lista de initializare se descrie DOAR IN ANTET**

```
{
..........
}
```

unde elementele din *lista_el* sunt incluse în *lista_comp*

Observaţie : Folosirea listei de iniţializare determină obţinerea unui cod mai eficient

Exemplul 1

```
#include <iostream>
#include <string.h>
using namespace std;

class Sir
{
    char *psir;
    int lung;
   public:
    Sir(char * s="");
    Sir(const Sir&);
    Sir& operator=(const Sir& sursa) ;
    ~Sir(){ delete []psir;}
};

Sir:: Sir (char *sursa)
{
    lung=strlen(sursa);
    psir=new char[lung+1];
    strcpy(psir, sursa);
    cout<<"Constructor sursa="
          <<sursa<<endl;
}

Sir ::Sir(const Sir& sursa)
{
    lung=sursa.lung;
    psir=new char[lung+1];
    strcpy(psir, sursa.psir);
    cout<<"Constructor copiere "
      << sursa.psir<<endl;
}

Sir& Sir::operator=(const Sir& sursa)
{
    if(this != &sursa)
    {
       delete []psir;
       lung=sursa.lung;
       psir=new char[lung+1];
       strcpy(psir, sursa.psir);
    }
    cout<<"Operator = Sir"
          <<sursa.psir<<endl;
    return *this;
}
```

```
class Persoana
{
  Sir  nume;
  Sir  prenume;
 public:
  Persoana(const Sir& n,const Sir& p);
};
/*
Persoana:: Persoana(const Sir& n,
const Sir& p)
{
    cout<<"Constr. clasic\n";
    nume=n; prenume=p;

} */

Persoana:: Persoana(const Sir& n,
const Sir& p):  nume(n), prenume(p)
// Lista de initializare se descrie
// DOAR IN ANTET
{
     cout<<"Constr. lista de
initializare\n";
}

int main(void)
{
    Persoana p1("Popescu", "Ion");
    return 0;
}
```

**REZULTATE**

CLASIC
Constructor sursa=Ion
Constructor sursa=Popescu
Constructor sursa=
Constructor sursa=
Constr. clasic
Operator = SirPopescu
Operator = SirIon

LISTA DE INITIALIZARE
Constructor sursa=Ion
Constructor sursa=Popescu
Constructor copiere Popescu
Constructor copiere Ion
Constr. lista de initializare

**Comportamentul constructorului de copiere şi al metodei care supraîncarcă operatorul de atribuire în contextul moştenirii**

- Pentru a caracteriza constructorii de copiere în contextul moştenirii, trebuie considerate următoarele situaţii:
    1. Dacă nici clasele de bază, nici clasa derivată nu au definiţi explicit constructori de copiere, copierea obiectelor se realizează bit cu bit prin cod sintetizat implicit de către compilator
    2. În cazul în care clasa derivată nu are constructor de copiere şi există clase de bază ce au definiţi explicit constructori de copiere, se sintetizează automat constructor de copiere pentru clasa derivată, ce realizează copierea bit cu bit a datelor membru specifice clasei derivate şi pentru datele moştenite din clasele de bază care nu au definiţi explicit constructori de copiere, realizând apelul adecvat pentru constructorii de copiere ai claselor de bază ce au definiţi explicit astfel de constructori.
    3. În cazul în care clasa derivată are definit explicit constructor de copiere şi există clase de bază ce au definiţi explicit constructori de copiere NU SE PRESUPUNE APELUL AUTOMAT AL CONSTRUCTORILOR DE COPIERE AI CLASELOR DE BAZĂ. APELUL CONSTRUCTORULUI DE COPIERE AL UNEI CLASE DE BAZĂ TREBUIE MENŢIONAT EXPLICIT, ALTFEL ELEMENTELE MOŞTENITE DIN CLASELE DE BAZĂ SE CONSTRUIESC FOLOSIND CONSTRUCTORUL IMPLICIT.

- În cazul metodei care supraîncarcă operatorul de atribuire, atunci când în clasa derivată se supraîncarcă operatorul de atribuire , trebuie făcut apel explicit la metoda ce supraîncarcă operatorul de atribuire pentru fiecare din clasele de bază ce au definită o astfel de metodă, altfel NU SE REALIZEAZĂ ATRIBUIRILE AFERENTE PARŢILOR MOŞTENITE DIN ACELE CLASE.

- Precauţiile menţionate trebuie avute în vedere şi pentru cazul obiectelor înglobate (embedded sau nested).

**Exemplul 2**

```
#include <iostream>
using namespace std;

class Parinte {
  int i;
public:
  Parinte() : i(0)  {
      cout << "Constr Implic, Parinte\n";
  }
  Parinte(int ii) : i(ii) {
      cout << "Parinte" <<" ii="<<ii<<"\n";
  }
  Parinte(const Parinte& b) : i(b.i) {
        cout << "Constr. copiere Parinte\n";
  }
   Parinte& operator=(const Parinte& p)
  {
    i=p.i;return *this;
  }
  ~Parinte() {
        cout << "Destructor Parinte()\n";
  }
  friend ostream& operator<<(ostream& os, const Parinte& b) {
    return os << "Parinte: " << b.i << endl;
  }
};

class Clasa {
   int i;
 public:
  Clasa() : i(0) {
        cout << "Constr Implic, Clasa " <<"\n";
  }
  Clasa(int ii) : i(ii) {
    cout << "Clasa " <<"ii="<<ii<<"\n";
  }
  Clasa(const Clasa& m) : i(m.i) {
    cout << "Constr. copiere Clasa\n";
  }
   Clasa& operator=(const Clasa& c)
  {
    i=c.i; return *this;
  }

 ~Clasa() {
      cout << "Destructor Clasa()\n";
  }
  friend ostream& operator<<(ostream& os, const Clasa& m) {
    return os << "Clasa: " << m.i << endl;
  }
};
```

```
class Copil : public Parinte
{
   int i;
   Clasa m;
 public:
  Copil()
  {
          cout<<"Constr Implic, Copil\n";
  }
   Copil(int ii) : Parinte(ii), i(ii), m(ii)
   // Se apeleaza explicit Constr Parinte cu parametri
   // Daca nu se defineste explicit se sintetizeaza de catre compilator
   // Daca se scrie Copil(int ii) :  i(ii), m(ii) NU se initializeaza partea
   // mostenita din Parinte ci se apeleaza Constructorul implicit Parinte
   // De testat similar apelul constr implicit si pt. Clasa
  {
    cout << "Copil "<< "ii="<<ii<<"\n";
  }

  Copil(const Copil& c):Parinte(c),i(c.i),m(c.m)
  // Daca nu se defineste explicit se sintetizeaza de catre compilator
  // Daca se scrie Copil(const Copil& c):i(c.i),m(c.m){} nu se copie partea
  // mostenita din Parinte ci se apeleaza ctor-ul implicit al clasei Parinte
  {
    cout<<"Constr. copiere Copil\n";
  }
  ~Copil() {
      cout << "Destructor Copil()\n";
  }
  Copil& operator=(const Copil& c)// NU merge:m(c.m)
  // Daca nu se defineste explicit se sintetizeaza de catre compilator
  {
    Parinte:: operator= (c);
    m=c.m; //Fara , nu se atribuie corespunzator datele mostenite din Parinte
    i=c.i; //Fara , nu se atribuie corespunzator datele din Clasa
    return *this;
  }

  friend ostream&    operator<<(ostream& os, const Copil& c){
    return os << (Parinte&)c << c.m << "Copil: " << c.i << endl;
  }
};
 int main(void) {
  Copil c0;
  Copil c1(2);
  cout << "Valori in c1:\n" << c1;
  cout << "Apel constr. copiere " << endl;
  Copil c2(c1);
  cout << "Valori in c2:\n" << c2;
  c0=c2;
  cout << "Valori in c0:\n" << c0;
  return 0;
}
```

**CU : Copil(int ii) : Parinte(ii), i(ii), m(ii)**
**SI Parinte:: operator=(c); m=c.m;**

Constr Implic, Parinte
Constr Implic, Clasa
Constr Implic, Copil
Parinte ii=2
Clasa ii=2
Copil ii=2
Valori in c1:
Parinte: 2
Clasa: 2
Copil: 2
Apel constr. copiere
Constr. copiere Parinte
Constr. copiere Clasa
Constr. copiere Copil
Valori in c2:
Parinte: 2
Clasa: 2
Copil: 2
Valori in c0:
Parinte: 2
Clasa: 2
Copil: 2
Destructor Copil()
Destructor Clasa()
Destructor Parinte()
Destructor Copil()
Destructor Clasa()
Destructor Parinte()
Destructor Copil()
Destructor Clasa()
Destructor Parinte()

**CU :Copil(int ii) : i(ii), m(ii)**
**SI Parinte:: operator=(c); m=c.m;**

Constr Implic, Parinte
Constr Implic, Clasa
Constr Implic, Copil
Constr Implic, Parinte
Clasa ii=2
Copil ii=2
Valori in c1:
Parinte: 0
Clasa: 2
Copil: 2
Apel constr. copiere
Constr. copiere Parinte
Constr. copiere Clasa
Constr. copiere Copil
Valori in c2:
Parinte: 0
Clasa: 2
Copil: 2
Valori in c0:
Parinte: 0
Clasa: 2
Copil: 2
.....

**CU : Copil(int ii) : Parinte(ii), i(ii), m(ii)**
**SI FARA Parinte:: operator=(c); m=c.m;**

Constr Implic, Parinte
Constr Implic, Clasa
Constr Implic, Copil
Parinte ii=2
Clasa ii=2
Copil ii=2
Valori in c1:
Parinte: 2
Clasa: 2
Copil: 2
Apel constr. copiere
Constr. copiere Parinte
Constr. copiere Clasa
Constr. copiere Copil
Valori in c2:
Parinte: 2
Clasa: 2
Copil: 2
Valori in c0:
Parinte: 0
Clasa: 0
Copil: 2

**Exemplul 3**

```
#include <iostream>
using namespace std;

class TabladeJoc
{
  public:
    TabladeJoc()  {
      cout << "TabladeJoc()\n";
    }
    TabladeJoc(const TabladeJoc&)
    {
       cout << "TabladeJoc(const
              TabladeJoc&)\n";
    }
    TabladeJoc& operator=(const
            TabladeJoc&)
    {
    cout <<
        "TabladeJoc::operator=()\n";
    return *this;
   }
   ~TabladeJoc()   {
     cout << "~TabladeJoc()\n";
   }
};

class Joc
{
  TabladeJoc tj; // Compunere
 public:
 //Se apeleaza constructorul implicit
 // TabladeJoc
  Joc()
  {
     cout << "Joc()\n";
  }

// Trebuie apelat explicit constr. de
// copiere TabladeJoc, altfel se
// apeleaza constructorul implicit:

  Joc(const Joc& j) : tj(j.tj)
  {
    cout << "Joc(const Joc&)\n";
  }

  Joc(int)
  {
          cout << "Joc(int)\n";
  }

  Joc& operator=(const Joc& j)
  {
  // Trebuie apelata explicit metoda
  // care supraincarca op = in
  // TabladeJoc
    tj = j.tj;
    cout << "Joc::operator=()\n";
    return *this;
  }

  ~Joc() { cout << "~Joc()\n"; }
};

class Sah : public Joc {};

class Dame : public Joc
{
  public:
    Dame()
    {
       cout << "Dame()\n";
    }


  Dame(const Dame& c) : Joc(c)
  {
    // Trebuie apelat explicit
    // constructorul de copiere al
    // clasei de baza altfel se
  // apeleaza constructorul implicit:
    cout << "Dame(const Dame& c)\n";
  }

Dame& operator=(const Dame& c) {
 // Trebuie aplelata explicit
 // versiunea din clasa de baza
 // pentru operator=() ALTFEL NU SE
 // FAC ATRIBUIRI PENTRU PARTEA
 // MOSTENITA DIN CLASA DE BAZA

    Joc::operator=(c);
//  (Joc&)(*this)=c;
    cout << "Dame::operator=()\n";
    return *this;
  }
   ~Dame()
  {
     cout << "~Dame()\n";
  }
};
```

```
int main()
{
  Sah s1;
  Sah s2(s1);
  s1 = s2; // Operator= sintetizat
  Dame d1, d2(d1);
  d1 = d2;
  return 0;
}
```

TabladeJoc()
Joc()
TabladeJoc(const TabladeJoc&)
Joc(const Joc&)
TabladeJoc::operator=()
Joc::operator=()
TabladeJoc()
Joc()
Dame()
TabladeJoc(const TabladeJoc&)
Joc(const Joc&)
Dame(const Dame& c)
TabladeJoc::operator=()
Joc::operator=()
Dame::operator=()
~Dame()
~Joc()
~TabladeJoc()
~Dame()
~Joc()
~TabladeJoc()
~Joc()
~TabladeJoc()
~Joc()
~TabladeJoc