ATENTIE ! **Scopul acestui document este de a structura si de a prezenta pe scurt notiunile discutate la curs. Invatarea materiei, exclusiv pe baza acestui material, reprezinta o abordare superficiala.**

ACADEMIA DE STUDII ECONOMICE
Facultatea de Cibernetică, Statistică şi Informatică Economică

PROGRAMAREA ORIENTATA OBIECT

SUPORT CURS

Conf. dr. Cătălin BOJA
Informatica Economica
catalin.boja@ie.ase.ro

CUPRINS

- Recapitulare
- Clase (Definire, Atribute, Constructori, Destructor, Metode, Interfata)
- Supraincarcare operatori
- Derivare clase (Ierarhii de clase, Polimorfism, Functii virtuale)
- Clase Template
- STL – Standard Template Library

BIBLIOGRAFIE

- **www.acs.ase.ro**
- **Ion Smeureanu, Marian Dardala – “Programarea orientata obiect in limbajul C++”, Editura CISON, 2002**
- **Ion Smeureanu – “Programarea in limbajul C/C++”, Editura CISON, 2001**
- **Recapitulare: Tudor Sorin, “Programarea in C/C++” – Manual de clasa XI**
- **Standardul: Bjarne Strastroup – The Creator of C++, “The C++ Programming Language”-3rd Edition, Editura Addison-Wesley, http://www.research.att.com/~bs/3rd.html**

CURS 1 – Recapitulare noţiuni C

- Pointeri
- Pointeri la functii
- Referinte
- Functii (Transferul parametrilor)
- Preprocesare

POINTERI

Sursa C/C++

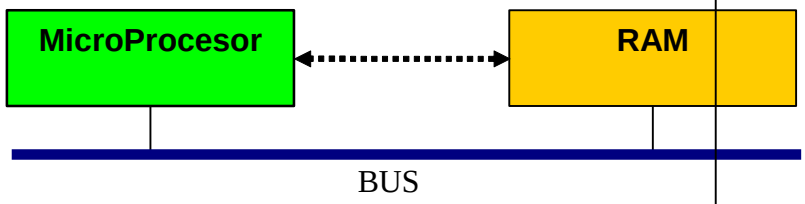
```
#include<stdio.h>

void main()
{
        char a = 7, b = 9;
        short int c;
        c = a+b;
}
```

Reprezentare ASM

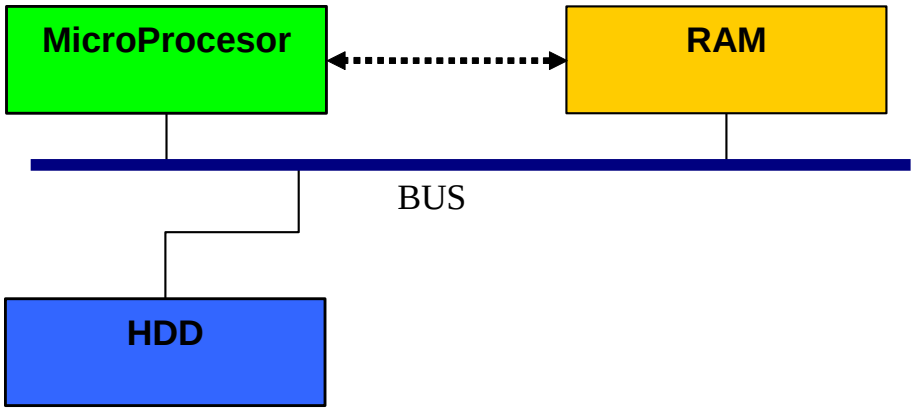
```
.model small
.stack 16
.data
        a db 7
        b db 9
        c dw ?

.code
start:
        mov AX, @data
        mov DS, AX

        mov AL,a
        add AL,b
        mov c,AX

        mov AX, 4C00h
        int 21h
end start
```

Cod Masina

```
B8 02 00 8E D8
A0 00 00 02 06
01 00 A3 02 00
B8 00 4C CD 21
00 00 00…..00 00
07 09
```

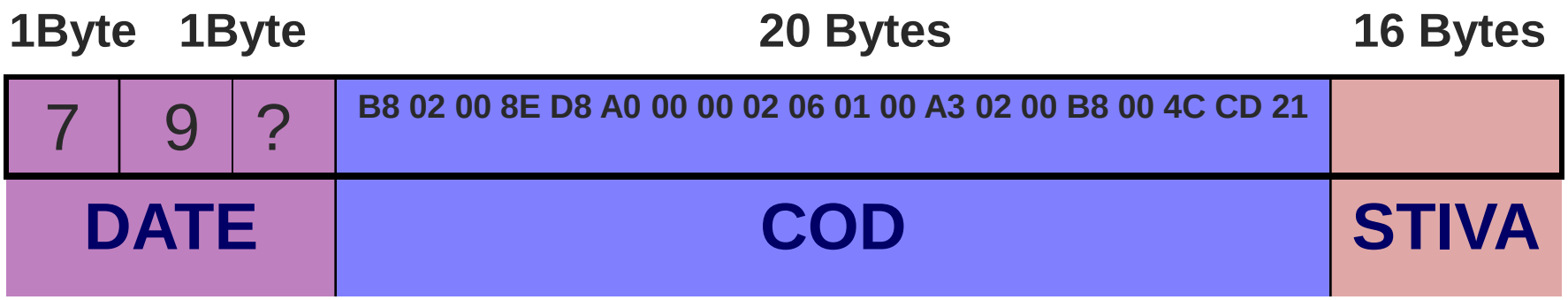


POINTERI



Sursa C/C++

```
#include<stdio.h>

void main()
{
        char a = 7, b = 9;
        short int c;
        c = a+b;
}
```



POINTERI

- date numerice utilizate pentru a gestiona valori reprezentand adrese;
- dimensiune data de arhitectura procesorului
- definire:

```
tip_data * nume_pointer;
```

- initializare:

```
nume_pointer = & nume_variabila;
```

- utilizare:

```
nume_variabila = * nume_pointer;
```

POINTERI

Exemple:

- i n t * p i ; // pointer la int
- c h a r ** p p c ; // pointer la pointer de char
- i n t * a p [1 0]; // sir de 10 pointeri la int

Valoarea 0 pentru un pointer este o valoare nula. Aceasta este asociata cu simbolul

```
#define NULL 0
```

sau cu constanta

```
const int NULL = 0;
```

POINTERI

Aritmetica pointerilor:

- pentru un pointer de tip T*, operatorii --/++ asigura deplasarea inapoi/inainte cu sizeof(T) octeti;
- pentru un pointer de tip T* pt, expresia pt + k sau pt – k este echivalenta cu deplasarea peste k * sizeof(T) octeti;
- diferenta dintre 2 pointeri din interiorul aceluiasi sir de valori reprezinta numarul de elemente dintre cele doua adrese;
- adunarea dintre 2 pointeri nu este acceptata;

POINTERI - constanti

Exemplu:

```
int * const p;                  // pointer constant la int
int const *pint;                // pointer la int constant
const int *pint2;               // pointer la int constant
const int * const pint2;        // pointer constant la int constant
```

Utilizare:

```
char* strcpy(char*p, const  char*q);
```

POINTERI

Alocare dinamica memorie

- operatorul new sau new [ ];
- rezerva memorie in Heap

Dezalocare memorie

- operatorul delete sau delete[ ];
- elibereaza memoria rezervata in Heap

REFERINTA

- reprezinta un pointer constant ce este dereferit automat la utilizare
- utilizata pentru a defini parametrii unui subprogram

```
int vb = 10 ;
int &  refvb = vb ;          // r and i now refer to the same int
int x = refvb ;              // x = 10
refvb = 20 ;                 // vb = 20
int & ref;                   //EROARE definire referinta
refvb ++;                    // vb = 21
int * pvb = & refvb;         // pvb este initializat cu adresa lui vb
```

POINTERI – la functii

- definire:

  tip_return (* nume_pointer) (lista parametrii);
- initializare:

  nume_pointer = nume_functie;
- apel functie prin pointer:

  nume_pointer (lista parametrii);

POINTERI – la functii

- f l o a t (*f p)(int *); // pointer la functie ce primeste un pointer la int si ce returneaza un float
- i n t * f (c h a r *); // functie ce primeste char* si returneaza un pointer la int
- i n t * (*f p [5]) (c h a r *); // vector de 5 pointeri la functii ce primesc char* si returneaza un pointer la int

PREPROCESARE

- Etapa ce precede compilarea
- Bazata pe simboluri definite prin #
- NU reprezintă instrucţiuni executabile
- Determina compilarea condiţionata a unor instrucţiuni
- Substituire simbolica
- Tipul enumerativ
- Macrodefinitii

PREPROCESARE

Substituire simbolica:

- bazata pe directiva #define

```
#define NMAX 1000
#define then
#define BEGIN {
#define END }

void main()
BEGIN
int vb = 10;
int vector[NMAX];
if(vb < NMAX) then printf(“mai mic”);
else printf(“mai mare”);
END
```

PREPROCESARE

Substituire simbolica:

- valabilitate simbol:
  - sfarsit sursa;
  - redefinire simbol;
  - invalidare simbol:

```
#define NMAX 1000
….
#define NMAX 10
…
#undef NMAX
```

PREPROCESARE

Tipul enumerativ:

enum denumire {llista simboluri} lista variabile

- valorile sunt in secventa
- se poate preciza explicit valoarea fiecarui simbol

enum rechizite {carte , caiet , creion = 4, pix = 6, creta}

PREPROCESARE

Macrodefinitii:

```
#define nume_macro(lista simboluri) expresie
```

Exemplu:

```
#define PATRAT(X) X*X
#define ABS(X) (X) < 0 ? – (X) : (X)
```

PREPROCESARE

Macrodefinitii generatoare de functii:

```
#define SUMA_GEN(TIP) TIP suma(TIP vb2, TIP vb2) \
        { return vb1 + vb2; }
```

Compilare conditionata:

```
#if expresie_1
                secventa_1
#elif expresie_2
                secventa_2
…
#else
                secventa_n
#endif
```

PREPROCESARE

Compilare conditionata:

```
#ifdef nume_macro
        …
#else
        …
#endif
```

sau

```
#ifndef nume_macro
        …
#endif
```

PREPROCESARE

Operatorii # si ##:

- sunt utilizati impreuna cu #define
- operatorul # (de insiruire) transforma argumentul intr-un sir cu “”;

```
#define macro1(s) # s
```

- operatorul ## (de inserare) concateneaza 2 elemente

```
#define macro2(s1, s2)  s1 ## s2
```

ELEMENTE NOI - C++

- lucru cu consola
  - citire de la consola: cin >> nume_variabila
  - afisare la consola: cout << nume_variabila
- alocare spatiu dinamic (HEAP)
  - alocare spatiu: nume_pointer = new tip_data[nr_elemente]
  - dezalocare spatiu: delete [] nume_pointer
- referinta &
  - definire parametrii iesire pentru functii: void Interschimbare(int &a, int &b)

CLASE

- reprezinta structuri de date ce incorporeaza date si functii;
- permit dezvoltarea de noi tipuri de date – ADT (Abstract Data Types);
- permit gestiunea programelor foarte mari;
- faciliteaza reutilizarea codului;
- permit implementarea conceptelor POO – incapsulare, polimorfism (“o interfata, metode multiple”), mostenire

CLASE

- fiecare obiect contine date (atribute/campuri) definite in clasa;
- clasa defineste o serie de functii (metode/operatii) ce pot fi aplicate obiectelor; acestea definesc interfata obiectului;
- datele sunt ascunse in obiect si pot fi accesate numai prin functii definite in clasa – incapsulare;
- obiectele sunt create prin instantierea clasei;
- prin abstractizare (definire clasa) se decide ce atribute si ce metode sunt suportate de obiecte;
- starea obiectului este definita de atributele sale;
- comportamentul obiectului este definit de metodele sale;
- termenul de passing a message catre un obiect este echivalent cu invocarea metodei;

CLASE

Sintaxa definire:

```
class Nume_Clasa
{
   tip_acces:
        atribute;
        functii membre;
   tip_acces:
        atribute;
        functii membre;
};
```

CLASE

tip_acces:

- descrie tipul de acces la atributele si metodele clasei;
- zona de acoperire se incheie cu definirea unui alt tip de acces sau cu terminarea clasei;

```
class Test
{
public:
      …
private:
      …
public:
      …
}
```

CLASE

tip_acces:

- private
  - implicit pus de compilator la inceputul clasei;
  - permite accesul doar din interiorul clasei;
- protected
  - are utilizare in cadrul ierarhiilor de clase obtinute prin derivare;
  - permite accesul din interiorul clasei si din interiorul; claselor derivate;
- public
  - permite accesul din interiorul clasei si din afara ei;

CLASE

atribute:

- definesc starea obiectului;
- sunt initializate prin instantierea obiectului;
- prin prisma incapsularii, sunt definite in zona privata si sunt accesate prin intermediul metodelor publice;
- definesc spatiul de memorie ocupat de o instanta a clasei (exceptie: atributele statice)
- tipuri particulare: constante, statice;

CLASE

atribute constante:

- NU este permisa modificarea valorii odata ce au fost initializate;
- sintaxa:

```
class Test
{
   public:
            const int atribut_1;
            const char atribut_2;
}
```

CLASE

atribute constante:

- initializare doar prin lista de initializari a constructorului:

```
class Test
{
   public:
            Test( …, int val_at_1):atribut_1(val_at_1),
   atribut_2(5)
            {
                     …
            }
};
```

CLASE

atribute statice:

- definesc atribute ce nu apartin unui obiect;
- sunt folosite de toate obiectele clasei;
- reprezinta “variabile globale” ce apartin unei clase de obiecte;
- declararea atributului static NU reprezinta o definire de date (este doar o descriere);
- ATENTIE la intializare (in functie de scopul utilizarii atributului static)

CLASE

atribute statice:

- sintaxa:

```
class Test
{
   public:
            static int vb_1;
            static char vb_2;
};
```

CLASE

atribute statice:

- definirea se realizeaza in zona globala folosind specificatorul de clasa (Nume_clasa ::)
- sintaxa definire:

```
class Test
{
   public:
            static int vb_1;
};
int Test:: vb_1;
```

CLASE

Pointerul THIS:

- pentru o clasa Test, acest pointer este de tipul Test *;
- reprezinta adresa obiectului care apeleaza metoda membra a clasei;
- toate functiile membre clasei primesc implicit acest pointer;
- se plaseaza pe prima pozitie in lista de parametrii a metodei;

CLASE

functii membre:

- definesc interfata obiectului;
- permit accesul la atributele obiectului – incapsulare;
- definesc comportamentul obiectului;
- categorie speciala de functii: constructor, destructor, constructor de copiere;
- tipuri particulare: statice, inline;

CLASE

functii membre:

- corpul functiilor poate fi definit in clasa

```
class Test {
        void Metoda( ) { …};
};
```

- corpul functiilor poate fi definit in afara clasei folosind specificatorul de clasa ::

```
class Test {
        void Metoda( );
};
void Test:: Metoda( ){…};
```

CLASE

functii constructor:

- rol principal: alocarea spatiului aferent unui obiect;
- rol secundar: initializarea atributelor obiectului;
- tipuri:
    - implicit
    - cu parametrii
    - cu parametrii cu valori implicite

CLASE

functii constructor:

- au denumire identica cu a clasei;
- NU au tip returnat explicit deoarece returneaza implicit adresa zonei de memorie rezervata obiectului construit;
- sunt definite pe zona publica a clasei;
- forma implicita este generata de compilator daca nu este definita de programator;

CLASE

functii constructor:

- sintaxa:

```
class Nume_clasa {
public:
        Nume_clasa( ){…}
};
```

- apel:

```
void main () {
Nume_clasa obiect_1;
Nume_clasa obiect_2( parametrii constructor)
}
```

```
class Test {
private:
        int atribut_1;
public:
        …
};
```

CLASE

constructor implicit:

```
Test ( ) { atribut_1 = 0; }
```

constructor cu parametrii

```
Test ( int val ) { atribut_1 = val ; }
Test ( int val ): atribut_1(val ) {}
```

CLASE

constructor cu parametrii cu valori implicite:

```
Test ( int val  = 0) { atribut_1 = val ; }
```

sau utilizand lista de initializari a constructorului

```
Test ( int val  = 0) { atribut_1 = val ; }
```

ATENTIE. Acest tip de constructor inlocuieste formele anterioare.

CLASE

constructor cu un parametru – caz special

```
class Test {
private:
  int vb;
public:
  Test2(int z) {vb = z;}
};
void main() {
Test t = 34;
}
```

CLASE

functii destructor:

- rol principal: dezalocarea spatiului aferent unui obiect;
- au denumire identica cu a clasei; pentru a se deosebi de constructor, numele lor este prefixat de ~;
- NU au tip returnat explicit deoarece returneaza implicit void;

CLASE

functii destructor:

- sunt definite pe zona publica a clasei;
- forma implicita este generata de compilator daca nu este definita de programator;
- sunt apelate implicit inainte de terminarea programului pentru toate obiectele definite;
- pentru obiecte locale sunt executate in ordine inversa fata de cele constructor;

CLASE

- sintaxa:

```
class Nume_clasa {
public:
        ~Nume_clasa( ){…}
};
```

- apel implicit:

```
void main () {
Nume_clasa obiect_1;
}
```

CLASE

functii destructor:

ATENTIE ! Pentru atributele alocate dinamic in functiile constructor este OBLIGATORIU dezalocarea lor in destructor. In caz contrar programul genereaza memory leaks.

CLASE

metode statice:

- definesc functii ce nu apartin unui obiect;
- sunt folosite de toate obiectele clasei;
- reprezinta “functii globale” ce apartin unei clase de obiecte;
- au acces DOAR la alti membrii statici ai clasei;
- sunt apelate prin specificatorul de clasa ::
- NU primesc in lista de parametrii pointerul THIS;

CLASE

metode statice:

- sintaxa:

```
class Nume_clasa {
public:
        static void Metoda_1( ){…}
};

void main( ) {
        Nume_clasa::Metoda_1( );
}
```

CLASE

metode inline:

- functii scurte care nu sunt apelate;
- la compilare, apelul functiei inline este inlocuit de codul ei, similar functiilor macro;
- permit executia rapida a codului prin evitarea efortului necesar unui apel de functie;
- contribuie la cresterea dimensiunii codului executabil;

CLASE

metode inline:

- implicit metodele al caror corp este definit in clasa sunt considerate inline (NU este o regula, depinzand foarte mult de compilator);
- explicit, o metoda este definita ca inline este anuntata prin cuvantul cheie inline;

```
class Test {
            void Metoda( );
};
inline void Test:: Metoda( ){…};
```

CLASE

metode “accesor”:

- permit accesul (citire / scriere) la atributele private ale clasei;
- presupun validarea datelor de intrare;
- sunt definite in zona publica;
- neoficial, metodele de citire sunt prefixate cu get iar cele de modificare sunt prefixate cu set;

CLASE

metode “accesor”:

```
class Nume_clasa {
private:
        int Atribut_1;
public:
        int Get_Atribut_1( ) { return Atribut_1;}
        void Set_Atribut_1(int val) {
                //validare val
                Atribut_1 = val;
        }
};
```

CLASE

trimiterea parametrilor in/din functii:

- prin valoare (ATENTIE la constructorul de copiere si la operatorul =)

```
class Nume_clasa {
...
};
Nume_clasa Metoda1 (Nume_clasa obiect);
```

- prin referinta (ATENTIE la modificari + return) ;

```
void Metoda2 (Nume_clasa & obiect);
```

- prin pointer (ATENTIE la modificari + return) ;

```
void Metoda3 (Nume_clasa * obiect);
```

CLASE

constructor de copiere:

- rol principal: alocarea spatiului aferent unui obiect si initializarea acestuia cu valorile unui obiect existent;
- are forma implicita pusa de compilator ce copiaza bit cu bit valoarea obiectului existent in zona de memorie a obiectului creat;
- este apelat automat in toate situatiile de definire + initializare obiect nou;

CLASE

constructor de copiere:

- sintaxa:

```
class Nume_clasa {
        public:
    Nume_clasa(Nume_clasa & ob_existent){…}
            };
```

- apel explicit:

```
void main () {
Nume_clasa obiect_1(…);
Nume_clasa obiect_2 = obiect_1;
}
```

CLASE

constructor de copiere:

- apel implicit: compilatorul apeleaza automat constructorul de copiere pentru a copia pe stiva subprogramului valorile obiectelor din lista de parametrii (daca sunt trimise prin valoare);
- apel implicit: compilatorul apeleaza automat constructorul de copiere pentru a copia pe stiva programului apelator valoarea obiectului returnat de subprogram (daca este returnat prin valoare);

CLASE

constructor de copiere:

- apel implicit :

```
class Test {
        public:
    Test (Test & ob_existent){…}
    void Metoda1(Test ob1, Test *ob2) {…}
    Test Metoda2(Test ob1) {…}
            };
void main () {
Test obiect_1, obiect_2, obiect_3, obiect_4;
obiect_1.Metoda1(obiect_2, obiect_3);
obiect_4 = obiect_1.Metoda1(obiect_2);
}
```

apel implicit constructor de copiere

CLASE

operator =

- rol principal: copiaza bit cu bit valoarea zonei de memorie sursa in zona de memorie a destinatiei (cele doua zone sunt identice ca structura si tip);
- in cazul obiectelor, copiaza valoarea obiectului sursa in obiectul destinatie

CLASE

operator =

- apel explicit :

```
class Nume_clasa {
        …
};

void main () {
Nume_clasa obiect_1(…);
Nume_clasa obiect_2(…);
obiect_2 = obiect_1;
}
```

CLASE

operator =

- supraincarcare obligatorie prin functie membra

```
class Nume_clasa {
        Nume_clasa operator = (Nume_clasa obiect)
        {
                //copiere din obiect in this;
        }
};

void main () {
Nume_clasa obiect_1(…);
Nume_clasa obiect_2(…);
obiect_2 = obiect_1;
}
```

CLASE

clase incluse:

- sunt definite clase in interiorul altor clase;

```
class Nume_clasa_parinte {
        …
        class Nume_clasa_copil {…};
};
```

- declaratia este vizibila doar in interiorul clasei parinte
- accesul la clasa copil este posibila doar prin specificatorul clasei parinte

```
Nume_clasa_parinte:: Nume_clasa_copil test;
```

CLASE

clase prietene (friend):

- se permite accesul pe zona privata sau protected din afara clasei (din interiorul clasei prietene);
- clasa prietena se anunta in clasa protejata prin atributul friend

```
class Nume_clasa_1 {
        …
        friend class Nume_clasa_2;
    };
class Nume_clasa_2 {
        …
    };
```

CLASE

pointeri de membrii (atribute):

- indica “adresa” unui atribut in cadrul obiectului – offset (deplasament);
- sintaxa definire:

```
tip_atribut Nume_clasa:: * nume_pointer_atribut ;
```

- initializare:

```
nume_pointer_atribut = & Nume_clasa:: nume_atribut ;
```

- utilizare:

```
Nume_clasa obiect, *pobiect = & obiect;

tip_atribut variabila = obiect.* nume_pointer_atribut
tip_atribut variabila = pobiect->* nume_pointer_atribut
```

CLASE

pointeri de membrii (metode):

- indica “adresa” unei metode in cadrul listei de functii a clasei – offset (deplasament);
- sintaxa definire:

```
tip_returnat (Nume_clasa:: * nume_pointer_metoda) (parametrii) ;
```

- initializare:

```
nume_pointer_metoda = & Nume_clasa:: nume_functie_membra ;
```

- utilizare:

```
Nume_clasa obiect, *pobiect = & obiect;

tip_returnat variabila = (obiect.* nume_pointer_metoda)(parametrii)

tip_returnat variabila = (pobiect->*nume_pointer_metoda)(parametrii)
```

CLASE - Supraincarcare

supraincarcare functii (overloading):

- **implementeaza conceptul de polimorfism (acelasi lucru, mai multe interpretari)**
- **atribuirea unui simbol (nume functie) mai multe semnificatii;**
- **diferenta se face in functie de semnatura functiei = numarul si tipul parametrilor;**
- **tipul returnat NU reprezinta criteriu de selectie la apel**

```
int suma(int a, int b)
{
        return a+b;
}
```

eroare compilare ⟷ **situatie ambigua**

```
double suma(int a, int b)
{
        return a+b;
}
```

CLASE - Supraincarcare

supraincarcare functii (overloading):

- etape identificare forma functie:
  1. identificare forma exacta;
  2. aplicare conversii nedegradante asupra parametrilor;
  3. aplicare conversii degradante asupra parametrilor;
  4. aplicare conversii definite explicit de programator prin supraincarcarea operatorului cast;
  5. generare eroare ambiguitate : overloaded function differs only by return type from …

CLASE - Supraincarcare

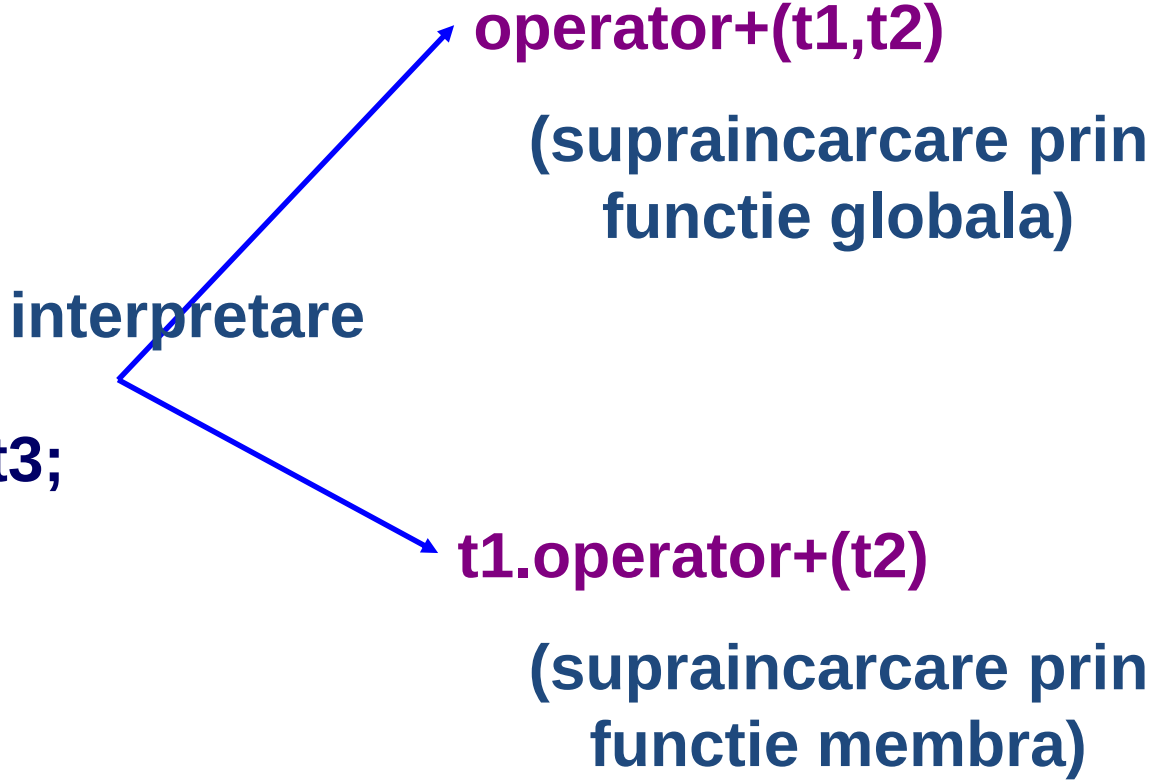
```
int suma(int a, int b)
{
        return a+b;
}

void main()
{
//identificare forma exacta
        int rez1 = suma(5,4);
//identificare forma functie prin conversii
nedegradante
        int rez2 = suma('0',5);
//identificare forma functie prin conversii degradante
        int rez3 = suma(4.6, 5);
}
```

CLASE - Supraincarcare

supraincarcare operatori:

- sunt implementati prin functii:

```
class Test{
        …
};
void main()
{
        Test t1, t2, t3;
        t1 = t2 + t3;
}
```



CLASE - Supraincarcare

restrictii supraincarcare operatori:

- NU schimba precedenta operatorilor;
- NU schimba asociativitatea;
- conserva cardinalitatea (numarul parametrilor)
- NU creaza operatori noi;
- formele supraincarcate nu se compun automat;
- NU se supraincarca . .* :: ?:

CLASE - Supraincarcare

restrictii supraincarcare operatori:

- supraincarcarea se realizeaza prin functii membre sau functii globale

EXCEPTII:

- functie membra: ( ) [ ] -> =
- functie globale: new delete

- NU garanteaza comutativitatea;
- formele post si pre sunt supraincarcate diferit;

CLASE - Supraincarcare

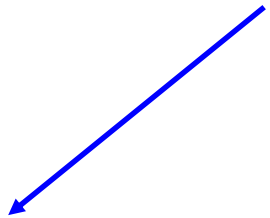
supraincarcarea prin functii membre sau functii globale ?

- verificare exceptie ?
- verificare tip primul parametru:
  - daca are tip diferit de cel al clasei analizate atunci supraincarc prin functie globala
  - daca are tip identic cu cel al clasei analizate atunci aleg functie membra sau functie globala

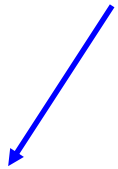
CLASE - Supraincarcare

ATENTIE

Operatorii supraincarcati prin functii membre primesc pe prima pozitie ca parametru pointerul this



```
class Test{
        Test operator+(Test t, int vb){
        …
        }
};
```



operator + cu 3 parametrii !!!!!

CLASE - Supraincarcare

ATENTIE

Trebuie acordata atentie la alegerea tipului returnat:

- daca operatorul se apeleaza in cascada;
- daca returneaza referinte de obiecte sa nu fie ale unor obiecte temporare;
- daca returneaza valori de obiecte, atentie la apelurile constructorului de copiere;

CLASE - Supraincarcare

supraincarcare operatorii << si >> (iostream):

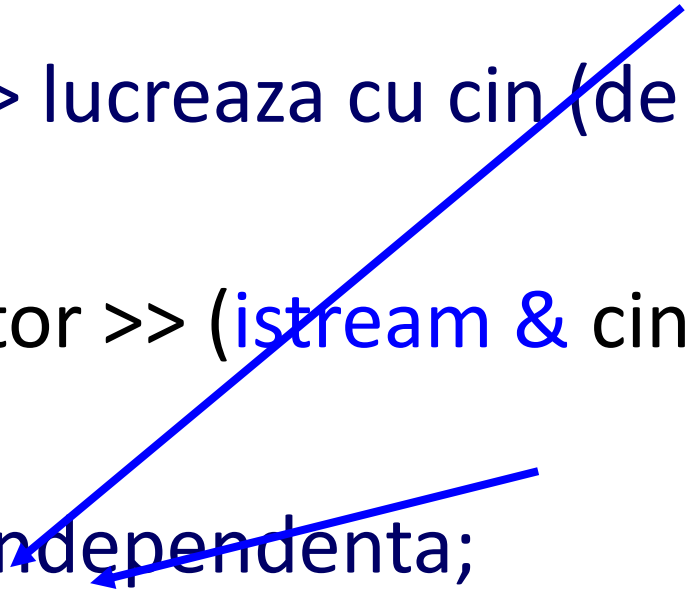
- operatorul << lucreaza cu cout (de tip ostream &);

ostream & operator << (ostream & cout, tip_data)

- operatorul >> lucreaza cu cin (de tip istream&)

istream & operator >> (istream & cin, tip_data &)

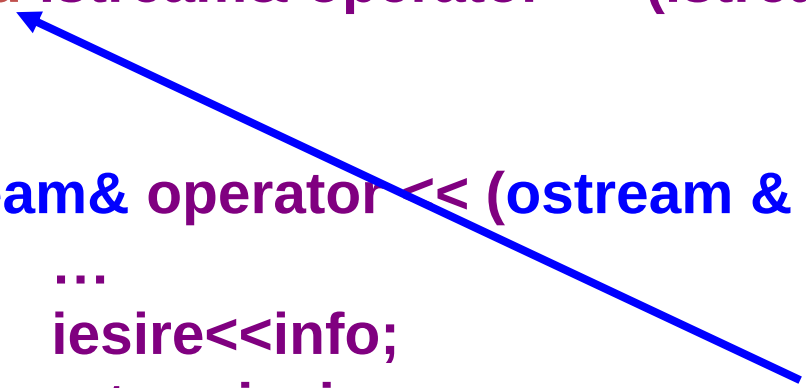
- prin functie independenta;



CLASE - Supraincarcare

supraincarcare operatorI << si >> (iostream):

```
class Test{
        int info;
friend ostream& operator << (ostream &, Test);
friend istream& operator >> (istream &, Test &);
};

ostream& operator << (ostream & iesire, Test t){
        …
        iesire<<info;
        return iesire;
}
istream& operator >> (istream & intrare, Test & t){
        …
        intrare>>info;
        return intrare;
}
```



este friend ca sa aiba acces pe zona privata
NU ESTE OBLIGATORIU !

CLASE - Supraincarcare

supraincarcare operatorii unari ++ si --:

- 2 forme: prefixata si postfixata;
- prin functie membra sau independenta;

```
int vb1 = 10;
int vb2 = vb1++;          -> vb2 = 10 si vb1 = 11;
int vb3 = 10;
int vb4 = ++vb3           -> vb4 = 11 si vb3 = 11
```

CLASE - Supraincarcare

supraincarcare operatorI unari ++ si --:

```
class Test{
        ...
        Test & operator++ ( ) {
                //prelucrari
                return *this;
        }
        friend Test operator++(Test &, int);
};
        Test operator++ (Test &t, int) {
                Test copie = t;
                //prelucrari
                return copie;
        }
```

forma **prefixata** prin functie membra

forma **postfixata** prin functie independenta

este **friend** ca sa aiba acces pe zona privata
NU ESTE OBLIGATORIU !

CLASE - Supraincarcare

supraincarcare operatorii binari +, -, *, /:

- au intotdeauna 2 parametri;
- comutativitatea operatiei matematice nu are sens in C++ (trebuie definita explicit)
- prin functie membra sau independenta in functie de forma operatorului;

CLASE - Supraincarcare

supraincarcare operatori binari +, -, *, /...

- pentru forma obiect + [obiect / alt tip]:
  - **prin functie membra**:

```
class Test{
      …
      int operator+ (int vb) {…}
      };
void main()
{
      Test t;
      int rez = t + 5;
}
```

```
int rez = 5 + t;
```

CLASE - Supraincarcare

supraincarcare operatorI binarI +, -, *, /:

- pentru forma obiect + obiect / alt tip:
  - **prin functie independenta [si friend]:**

```
class Test{
    …
    friend int operator+ (Test,int);
    };

int operator+ (Test t, int vb){…}
```

CLASE - Supraincarcare

supraincarcare operatorI binarI +, -, *, /:

- pentru forma alt tip + obiect:
  - **doar prin functie independenta [si friend]:**

```
class Test{
    …
    friend int operator+ (int, Test);
    };

int operator+ (int vb, Test t){…}
```

CLASE - Supraincarcare

supraincarcare operatori binari +=, -=, *=, /=:

- au intotdeauna 2 parametri;
- prin functie membra sau independenta;

```
class Test{
    …
    friend int operator+= (Test,int);
    };

int operator+= (Test t, int vb){…}
```

CLASE - Supraincarcare

supraincarcare operator [ ]

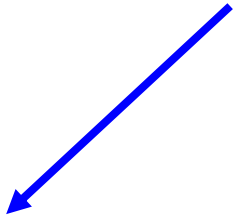
- are intotdeauna 2 parametri;
- doar prin functie membra;
- este folosit pentru a permite acces in citire / scriere pe elementele unui sir de valori din zona privata a obiectului;
- poate fi apelat in cascada;
- indexul nu este obligatoriu de tip numeric;

CLASE - Supraincarcare

supraincarcare operator [ ] :

```
class Test{
        int *valori;
        int nr_valori;
        …
        int operator[ ] (int);
        };

int Test::operator[ ] (int index){
        if (index >=0 && index < nr_valori)
                return valori[index];
        else return -1;
}
void main(){
        Test t;
        int vb = t[5];
        t[3] = 10;          ATENTIE - EROARE !
}
```

forma care asigura doar citirea datelor !!!



CLASE - Supraincarcare

supraincarcare operator [ ] :

```
class Test{
        int *valori;
        int nr_valori;
        …
        int& operator[ ] (int);
        };

int& Test::operator[ ] (int index){
        static int eroare;
        if (index >=0 && index < nr_valori)
                return valori[index];
        else return eroare;
}
void main(){
        Test t;
        int vb = t[5];
        t[3] = 10;
}
```

forma care asigura citirea / modificarea datelor !!!

CLASE - Supraincarcare

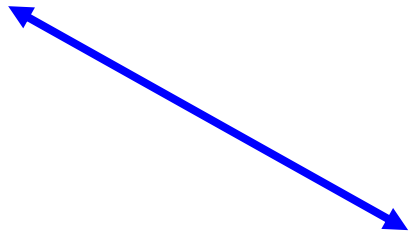
supraincarcare operator cast:

- are intotdeauna 1 parametru;
- numele castului reprezinta tipul returnat;
- nu are tip returnat explicit;
- prin functie membra;
- folosit la conversia intre diferite tiouri de date;
- ATENTIE la apelurile implicite puse de compilator pentru a determina semnatura unei functii;

CLASE - Supraincarcare

supraincarcare operator cast:

```
class Test{
        int valoare;
        …
        int operator int () { return valoare;}
        };
void main(){
        Test t;
        int vb = t;   //echivalent cu vb = t.valoare;
}
```



CLASE - Supraincarcare

supraincarcare operator !:

- are un parametru;
- prin functie membra sau independenta;

```
class Test{
        int valoare;
        …
        void operator ! () {valoare*=-1;}
        };
void main(){
        Test t;
        !t;
}
```

CLASE - Supraincarcare

supraincarcare operator ,:

- are doi parametri;
- prin functie membra sau independenta;
- in mod uzual returneaza valoarea ultimului parametru;

```
class Test{
        int valoare;
        …
        Test& operator ! (Test& t) {return t;}
        };
void main(){
        Test t1,t2, t3,t4;
        t4 = (t1,t2,t3);     //echivalent cu t4 = t3;
}
```

CLASE - Supraincarcare

supraincarcare operator functie:

- are numar variabil de parametri;
- prin functie membra;
- nu creaza o noua cale de apelare a unei functii;
- se creaza o functie operator care poate primi un numar arbitrar de parametri;

CLASE - Supraincarcare

supraincarcare operator functie:

```
class Test{
        int valoare;
        …
        int operator () (int i, int j) {
                valoare = i + j;
                return valoare;}
        };
void main(){
        Test t;
        t(3,4);
        int vb = 10 + t(5,10);
}
```

CLASE - Supraincarcare

supraincarcare operator ->;

- are intotdeauna 1 parametru;
- obligatoriu prin functie membra;
- intoarce pointer spre un obiect asupra caruia opereaza;

```
class Test{
        ...
        Test * operator-> ( ) {return *this;}
};
```

CLASE

Conversii intre obiecte de diferite tipuri:

- supraincarcarea constructorului clasei rezultat;
- supradefinirea operatorului cast al clasei sursa;
- apelurile implicite ale constructorului clasei rezultat sunt eliminate prin atributul explicit pus la definirea acestuia;

CLASE

Pointeri constanti de obiecte si pointeri de obiecte constante:

- definirea se face prin pozitionarea atributului const in raport cu tipul si numele pointerului;

```
class Test{
        …
        void Metoda const ( ) {…}
};
…
Test * const pConstantTest;
const Test * pTestConstant1;
Test const * pTestConstant2;
```

Obiectul referit prin *this* este constant

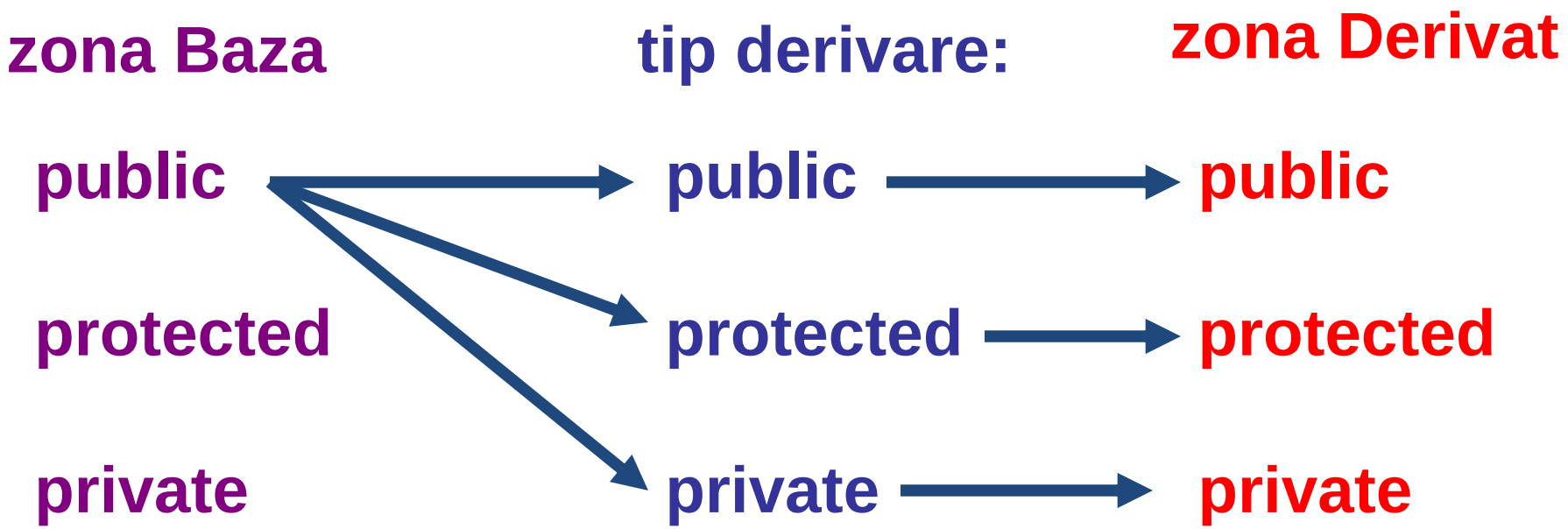
CLASE – Derivare / Mostenire

- REUTILIZARE COD;
- dezvoltarea de noi entitati (clase) pornind de la cele existente
- Derivare – clasa existenta se deriveaza intr-o noua clasa;
- Mostenire – clasa nou definita mosteneste atributele + metodele clasei derivate (clasei de baza);

```
class Baza{
};
class Derivat : tip derivare Baza{
};
```

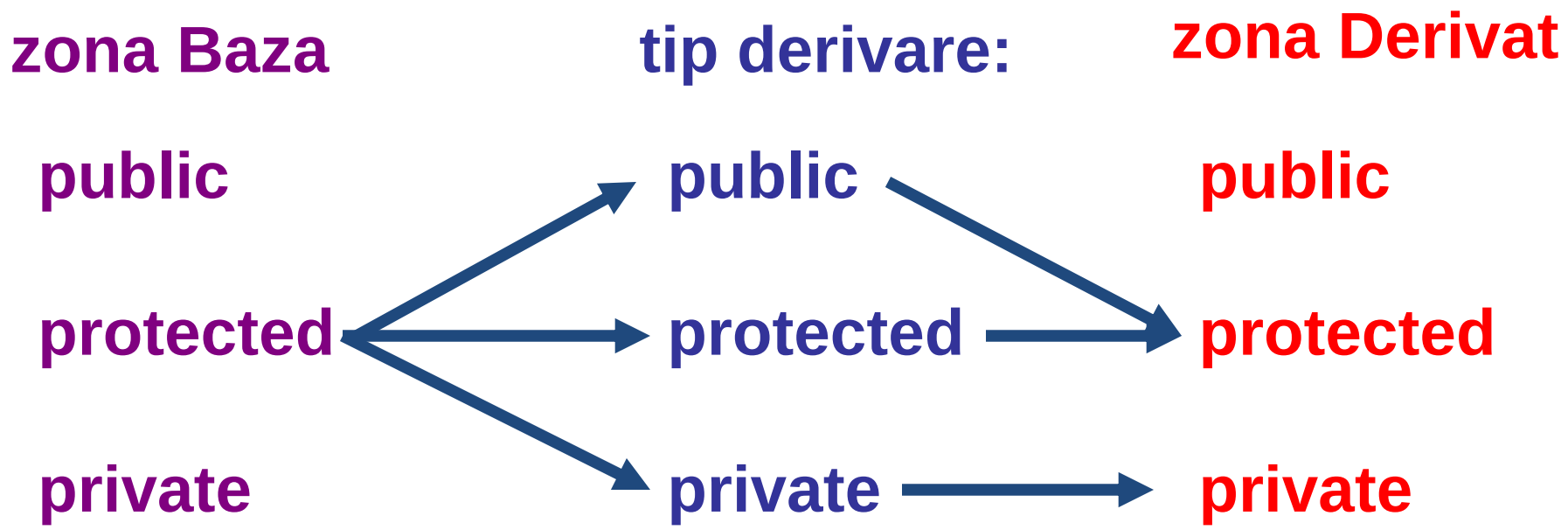
CLASE – Derivare / Mostenire

- prin derivare NU se elimina restrictiile de acces din clasa de baza;



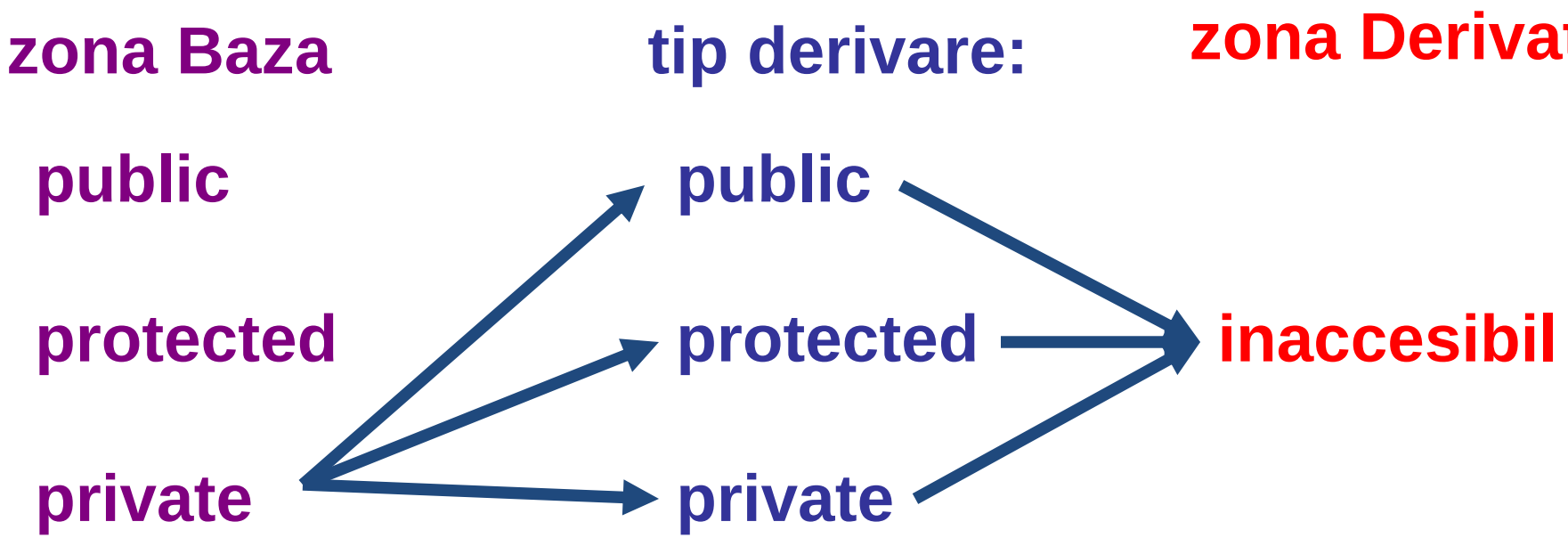
CLASE – Derivare / Mostenire

- prin derivare NU se elimina restrictiile de acces din clasa de baza;



CLASE – Derivare / Mostenire

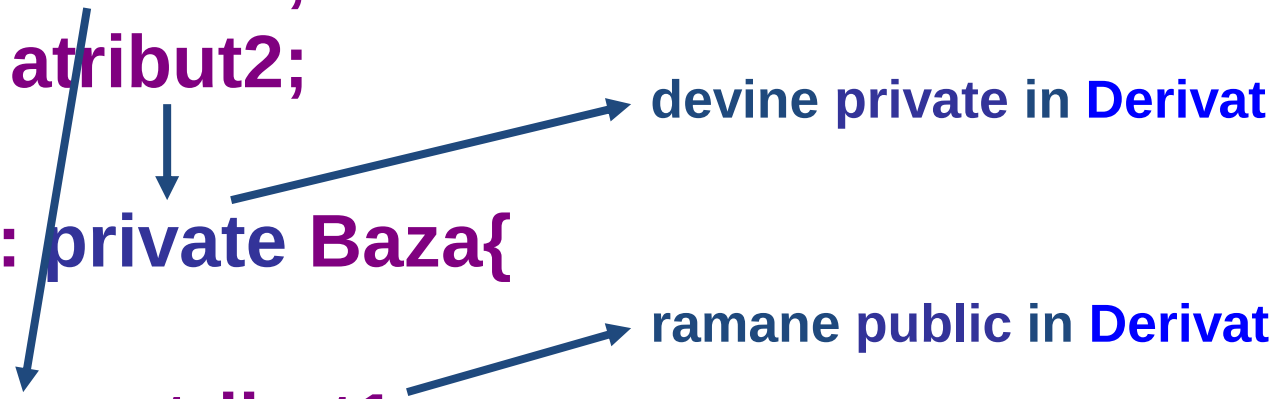
- prin derivare NU se elimina restrictiile de acces din clasa de baza;



CLASE – Derivare / Mostenire

- exceptii de la tip derivare (protected sau private) pentru anumite atribute sau metode (publice in clasa de baza): publicizare

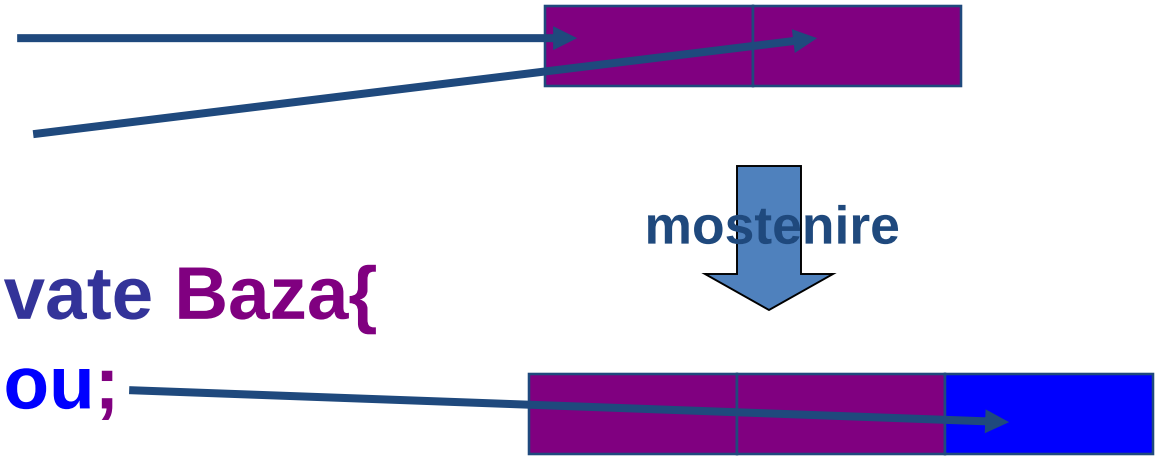
```
class Baza{
        public:
                int atribut1;
                int atribut2;
};
class Derivat : private Baza{
        public:
                Baza::atribut1;
};
```



CLASE – Derivare / Mostenire

- prin derivare noua clasa primeste de la clasa de baza toate metodele + atributele

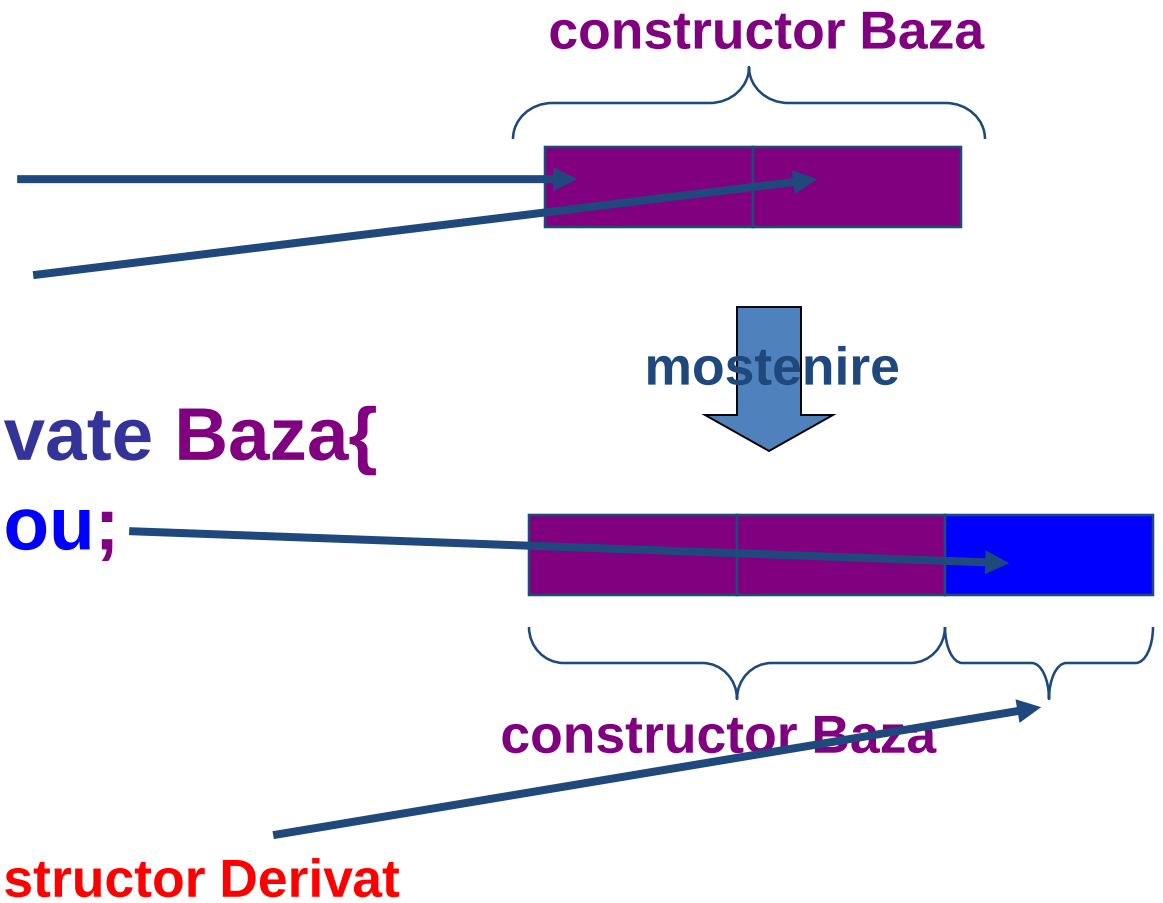
```
class Baza{
        int atribut1;
        int atribut2;
};
class Derivat : private Baza{
        int atribut_nou;
};
```



CLASE – Derivare / Mostenire

- fiecare constructor este responsabil strict de zona clasei pe care o reprezinta

```
class Baza{
        int atribut1;
        int atribut2;
};
class Derivat : private Baza{
        int atribut_nou;
};
```



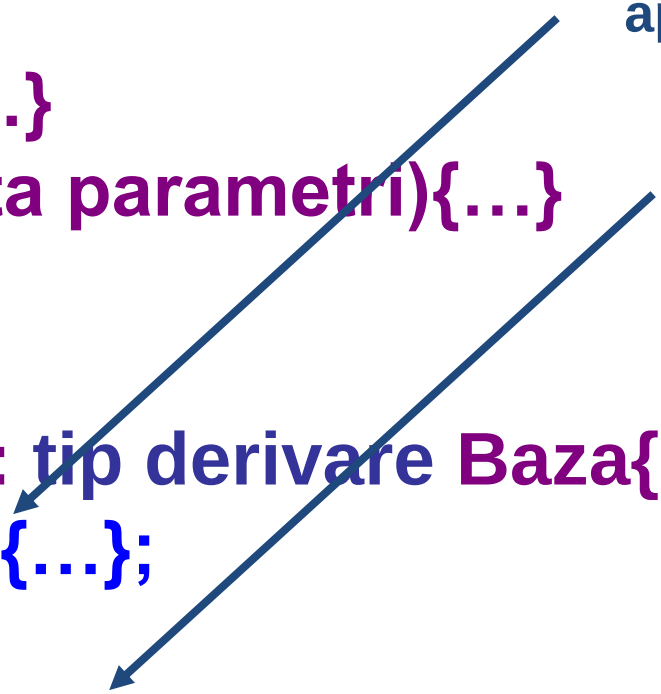
CLASE – Derivare / Mostenire

- constructie obiect derivat = CONSTRUCTOR BAZA + CONSTRUCTOR DERIVAT

```
class Baza{
        Baza(){…}
        Baza(lista parametri){…}
};

class Derivat : tip derivare Baza{
        Derivat(){…};
        SAU
        Derivat() : Baza(lista parametri) {…}
};
```

apel implicit Baza()

apel explicit :Baza(lista parametri)



CLASE – Derivare / Mostenire

- distrugere obiect derivat = DESTRUCTOR DERIVAT + DESTRUCTOR BAZA

ATENTIE ! Fiecare destructor trebuie sa se concentreze strict pe ceea ce au facut constructorii clasei.

```
class Baza{
        int * spatiu();
        ~Baza(){delete [ ]spatiu;}
};
class Derivat : tip derivare Baza{
        ~Derivat(){delete [ ]spatiu};
};
```

2 – dezalocare atribut !!!!!!!! (mai este alocat ???)

1- dezalocare atribut mostenit

CLASE – Derivare / Mostenire

- metode care nu se mostenesc integral:

operatorul = si Constructor Copiere

```
class Baza{
        int atribut1;int atribut2;
        Baza& operator=(Baza& b){…}
        Baza(Baza& b) {…}
};
class Derivat : private Baza{
        int atribut_nou;
};
```

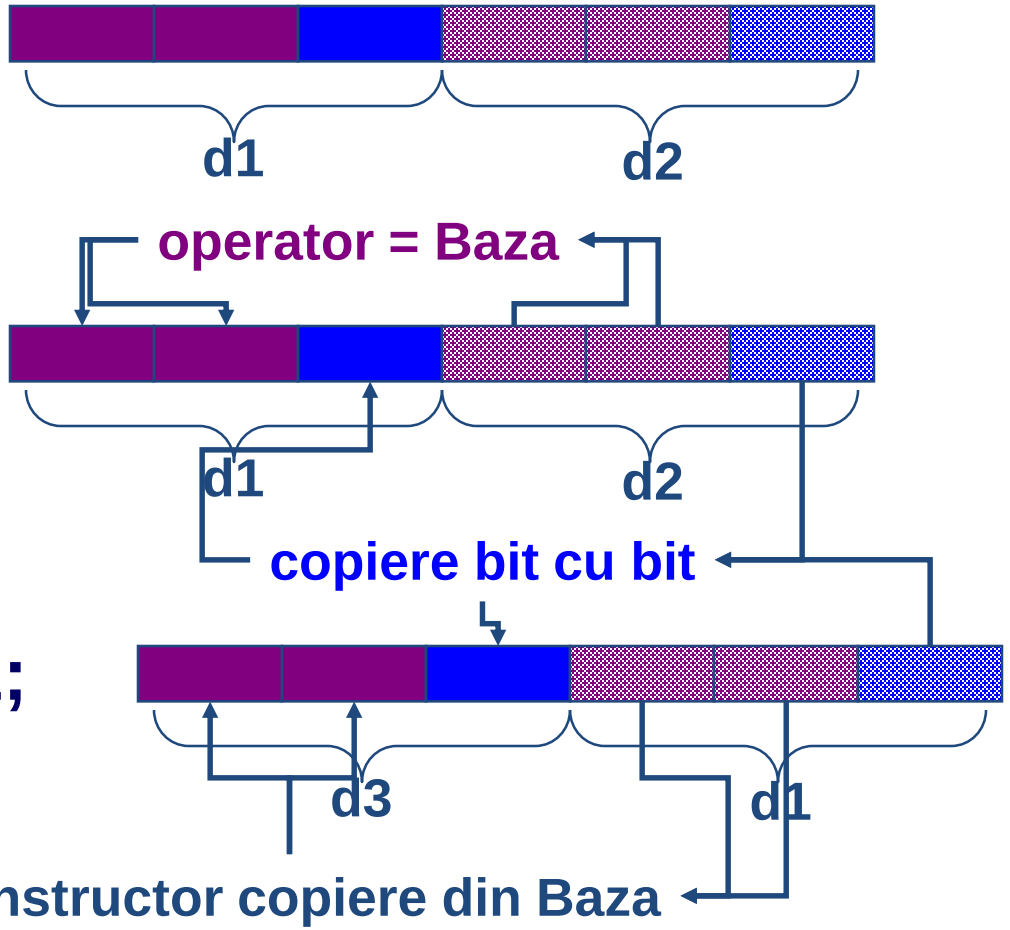
CLASE – Derivare / Mostenire

- metode care NU se mostenesc integral:

operatorul = si Constructor Copiere

```
void main(){
        Derivat d1;
        Derivat d2;

        d1 = d2;


        Derivat d3 = d1;
}
```

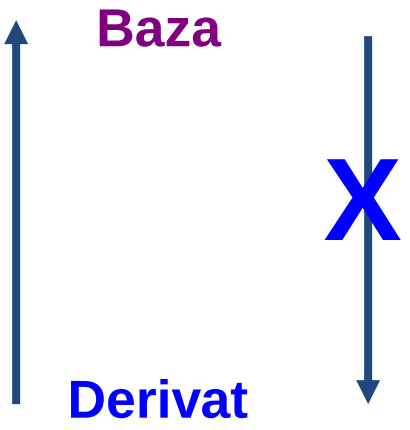
constructor Derivat



CLASE – Derivare / Mostenire

- UPCASTING – este permisa transformarea implicita a obiectelor sau pointerilor de tip derivat in obiecte sau pointeri de tip baza

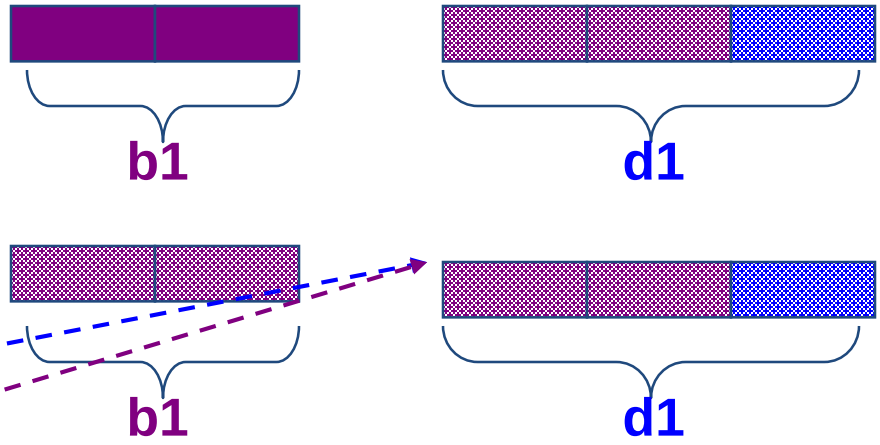
```
class Baza{
        …
};

class Derivat : public Baza{
        …
};
```



CLASE – Derivare / Mostenire

- UPCASTING

```
void main(){
        Derivat d1, *pd1;
        Baza b1, *pb1;

        b1 = d1;

        pd1 = &d1;
        pb1 = pd1;
}
```



CLASE – Derivare / Mostenire

- pot fi definite functii cu acelasi header in clasa de baza si in clasa derivata

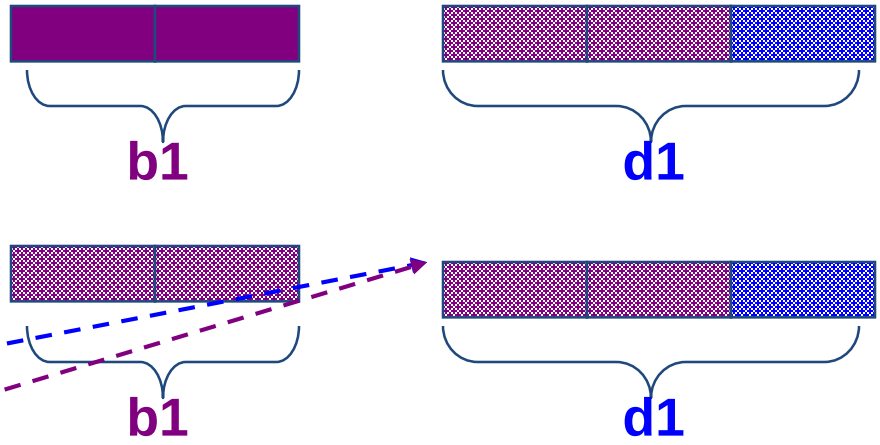
```
class Baza{
        int Metoda1(int a){…}
};
class Derivat : private Baza{
        int atribut_nou;
        int Metoda1(int a){…}
};
```

```
void main(){
Derivat d1;

d1.Metoda1(5);

d1.Baza::Metoda1(5);
}
```

CLASE – Derivare / Mostenire

- UPCASTING + redefinire metode

```
void main(){
        Derivat d1, *pd1;
        Baza b1, *pb1;

        b1 = d1;

        pd1 = &d1;
        pb1 = pd1;

        b1.Metoda1(5);
        pb1->Metoda1(5);
}
```

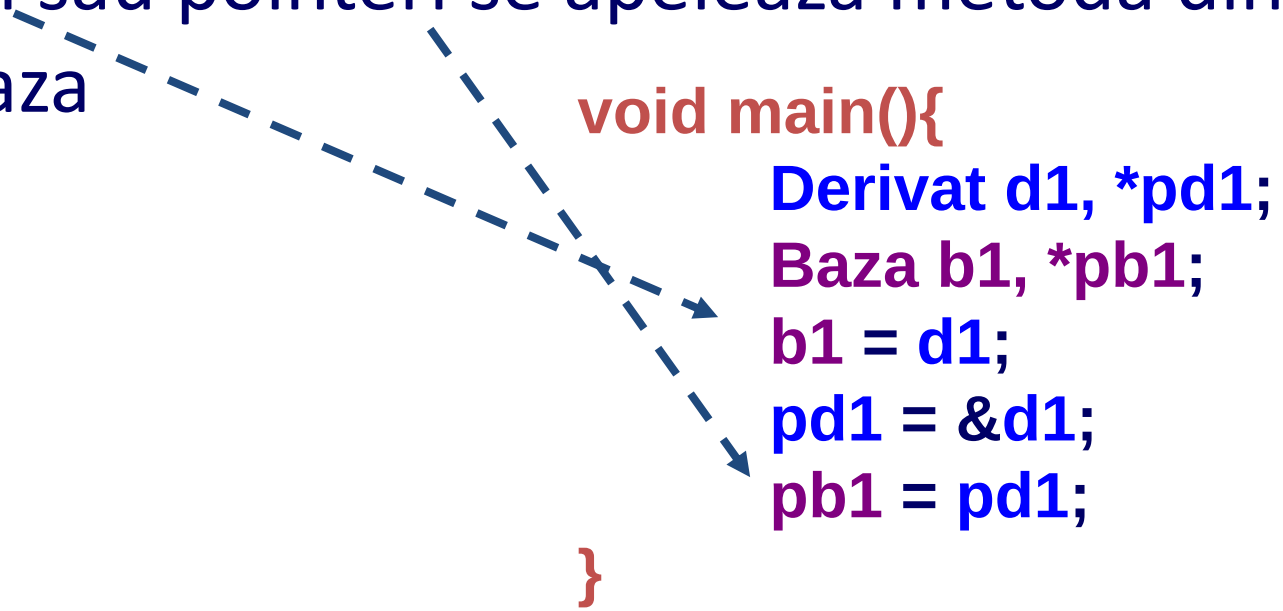


INTODEAUNA forma metodei din Baza;

CLASE – Derivare / Mostenire

UPCASTING + redefinire metode

- versiunea functiei se stabileste de la compilare (early binding)
- indiferent daca se realizeaza UPCASTING prin valori sau pointeri se apeleaza metoda din clasa de baza



```
void main(){
        Derivat d1, *pd1;
        Baza b1, *pb1;
        b1 = d1;
        pd1 = &d1;
        pb1 = pd1;
}
```

CLASE – Derivare / Mostenire

functii VIRTUALE:

- permit redefinirea (overriding) functiei din clasa de baza in clasa derivata

```
class Baza{
virtual int Metoda1(int a){…}
};
```

```
class Derivat : private Baza{
    int atribut_nou;
    int Metoda1(int a){…}
    };
```

CLASE – Derivare / Mostenire

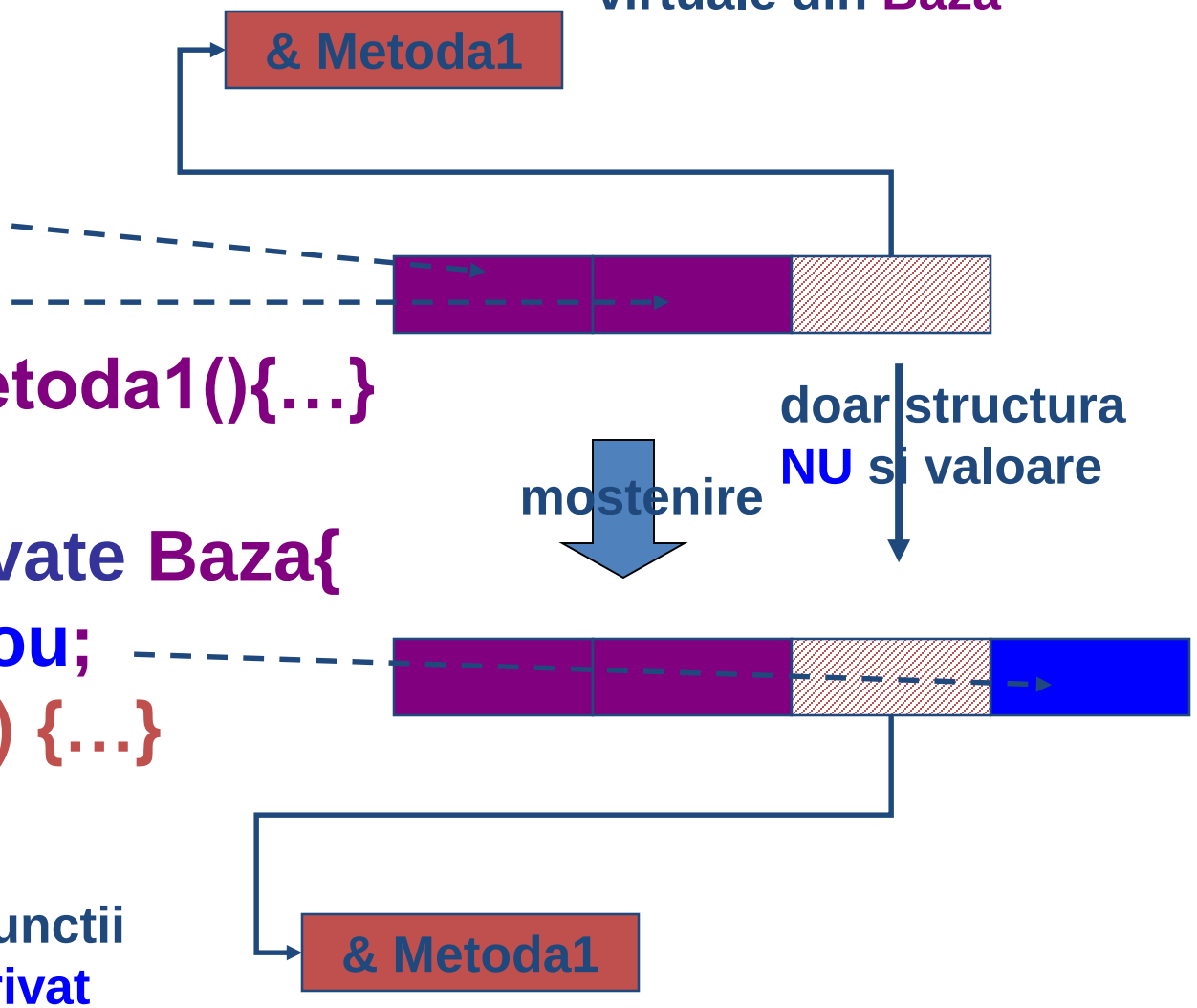
functii VIRTUALE:

- versiunea functiei se stabileste la momentul executiei (late binding)
- fiecare clasa contine o tabela de pointeri la functii virtuale;
- fiecare obiect primeste un pointer la tabela de pointeri la functii virtuale
- daca se realizeaza UPCASTING prin pointeri (NU si prin valori) se apeleaza metoda din clasa derivata

CLASE – Derivare / Mostenire

functii VIRTUALE:

```
class Baza{
        int atribut1;
        int atribut2;
        virtual int Metoda1(){…}
};
class Derivat : private Baza{
        int atribut_nou;
        int Metoda1() {…}
};
```



CLASE – Derivare / Mostenire

- UPCASTING + functii VIRTUALE:

```
void main(){
        Derivat d1, *pd1;
        Baza b1, *pb1;

        b1 = d1;
        b1.Metoda1(5);

        pd1 = &d1;
        pb1 = pd1;

        pb1->Metoda1(5);
}
```

INTODEAUNA forma metodei din Baza;

forma metodei din Derivat pentru Metoda1 virtuala;

CLASE – Derivare / Mostenire

functii VIRTUALE:

- natura virtuala a unei functii se mosteneste
- functia de pe ultimul nivel unde a fost redefinita raspunde pentru subierarhia ei;
- functia devine si ramane virtuala de la prima definire a ei din ierarhie a care este anuntata ca fiind virtuala
- ATENTIE la destructori virtuali

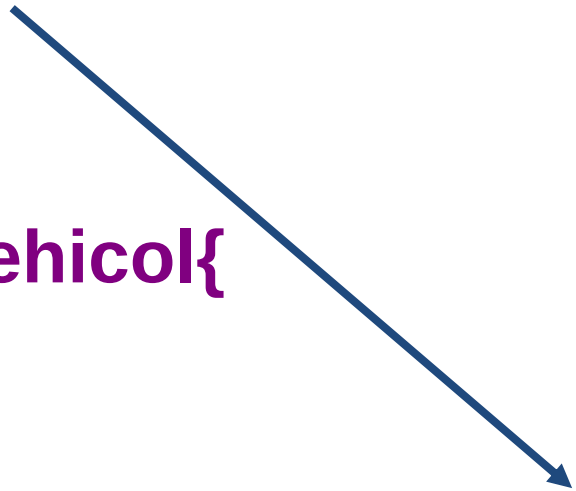
CLASE – Derivare / Mostenire

POLIMORFISM (**acelasi lucru, mai multe interpretari**) :

- SUPRAINCARCAREA (OVERLOADING) de functii in cadrul unei clase
- SUPRADEFINIRE (REDEFINIRE) (OVERRIDING) de functii virtuale in clasele derivate

CLASE – Derivare / Mostenire

- Mostenire vs Includere



se implementeaza cand intre clasa derivata si clasa de baza exista relatia *is a*;

```
class Vehicol{
        …
};

class Automobil : public Vehicol{
        …
};
```

CLASE – Derivare / Mostenire

- Mostenire vs Includere

```
class Motor{
        …
};

class Automobil{
        Motor motor;
};
```

se implementeaza cand intre clasa principala si cea inclusa exista o relatie *has a*;

CLASE – Derivare / Mostenire

Proprietati mosteniri multiple:

- constructorii se apeleaza in ordinea derivarii;
- destructorii se apeleaza in ordine inversa derivarii;
- ambiguitati la adresarea membrilor mosteniti care se numesc la fel in clasele de baza
- ambiguitati la mostenirile din clase de baza cu o baza comuna

CLASE – Derivare / Mostenire

- Mostenirii multiple – derivare din mai multe clase de baza

```
class Baza1{
};
class Baza2{
};

class Derivat : tip derivare Baza1, tip derivare Baza2{
};
```

CLASE – Derivare / Mostenire

functii VIRTUALE PURE:

- functii virtuale ce nu au corp definit in clasa in care sunt anuntate
- sunt definite prin expresia

virtual tip returnat nume_functie(parametrii) = 0;

- IMPUN redefinirea (overriding) functiei in clasa derivata (daca nu se doreste abstractizarea clasei derivat)

CLASE – Derivare / Mostenire

functii VIRTUALE PURE:

```
class Baza_abstracta{
virtual int Metoda1(int a) = 0
};

class Derivat : public Baza{
int Metoda1(int a){…}
};
```

CLASE abstracte

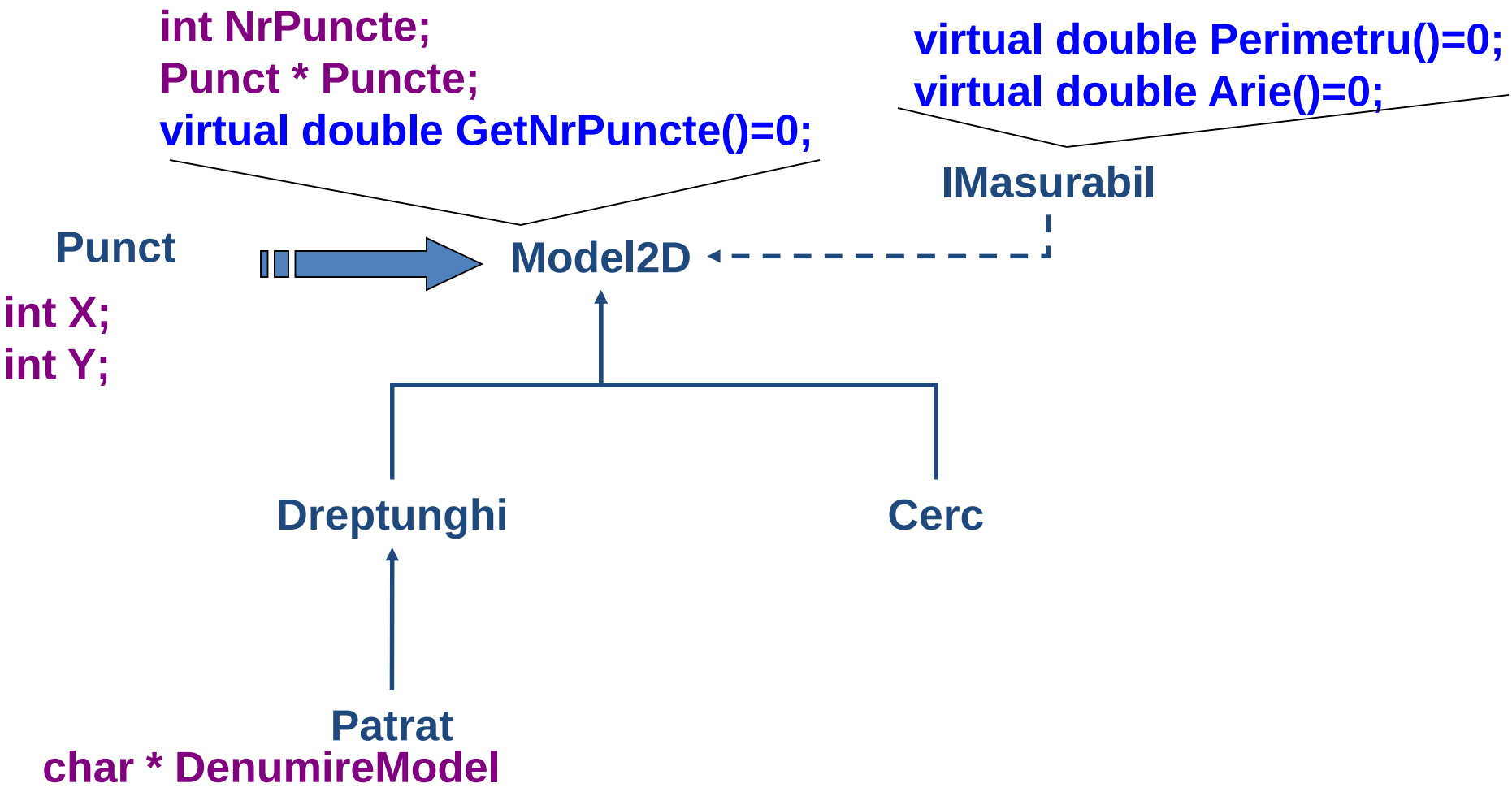
- clase ce contin minim o functie virtuala pura;
- rol de interfata pentru clase care trebuie sa defineasca o serie de metode comune
- un contract intre proprietarii mai multor clase prin care se impune definirea unor serii de metode comune;
- contractul se incheie prin derivarea din clasa abstracta;

CLASE abstracte

- NU este permisa instantierea claselor abstracte;
- utilizate ca suport pentru derivare

```
class Baza_abstracta{
int atribut1;
virtual int Metoda1(int a) = 0
};
void main(){
Baza_abstracta ba;
}
```

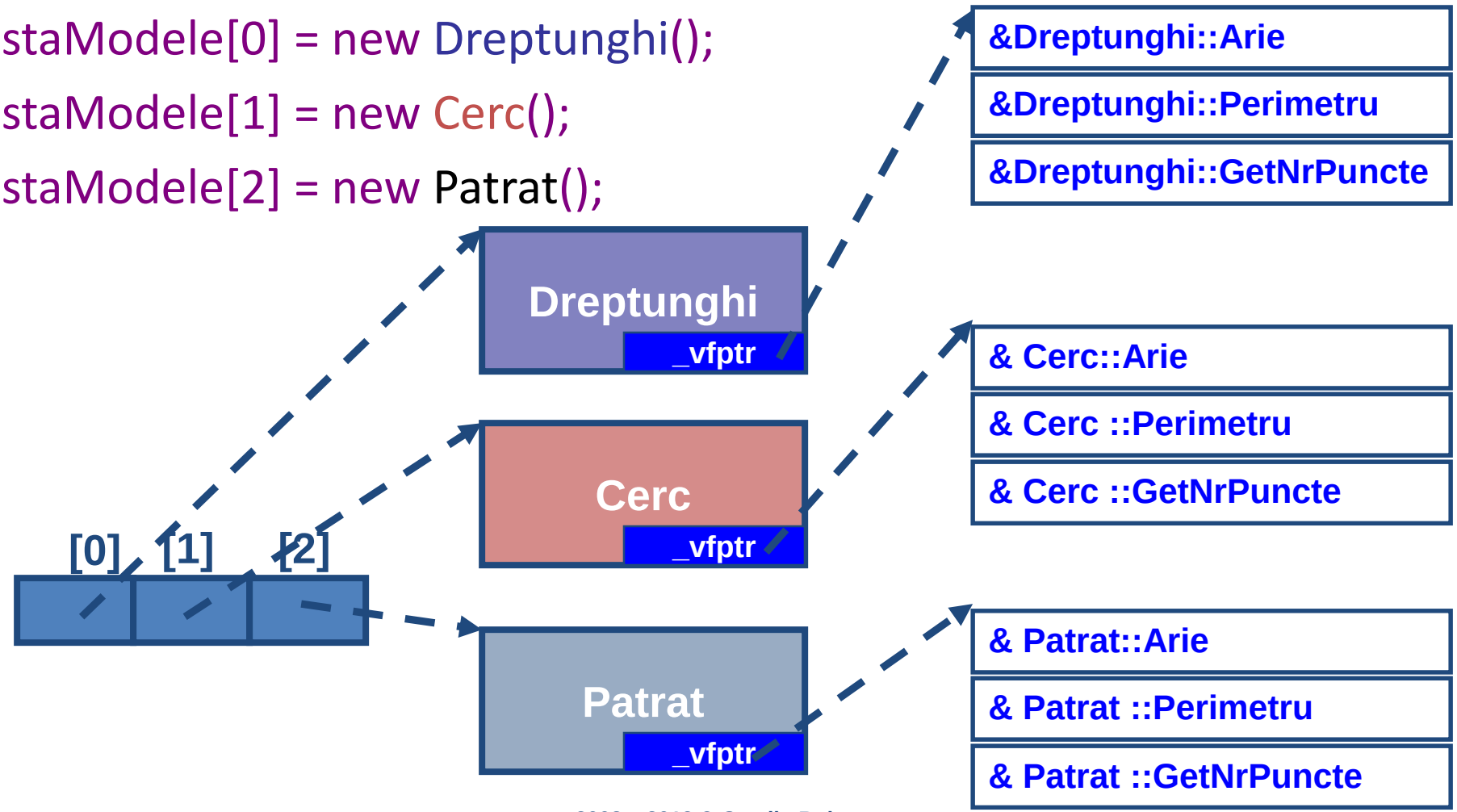
Exemplu ierarhie: CLASE abstracte



CLASE abstracte

Exemplu utilizare ierarhie:

```
Model2D * ListaModele[3];
ListaModele[0] = new Dreptunghi();
ListaModele[1] = new Cerc();
ListaModele[2] = new Patrat();
```



CLASE – Domenii de nume

- reprezinta o modalitate de grupare a variabilelor globale, claselor si functiilor globale
- permite definirea de clase, variabile, functii identice ca nume dar in spatii de nume diferite
- faciliteaza dezvoltarea distribuita de cod deoarece programatorii nu impun restrictii de nume intre ei
- cuvant cheie namespace

CLASE – Domenii de nume

- definire:

```
namespace Colectie1{
   //definire clase, variabile, functii
};
```

- definire alias:

```
namespace Colectia = Colectia1;
```

- adaugare de elemente:

```
namespace Colectie1{
   int vb1;
};
namespace Colectie1{
   int vb2;
};
```

CLASE – Domenii de nume

- utilizare - prin operatorul de rezolutie:

```
namespace Colectie1{
   int vb1;
};
namespace Colectie2{
   int vb2;
};
void main(){
   Colectie1::vb1 = 10;
   Colectie2::vb2 = 20;
}
```

CLASE – Domenii de nume

- utilizare - prin directiva using namespace:

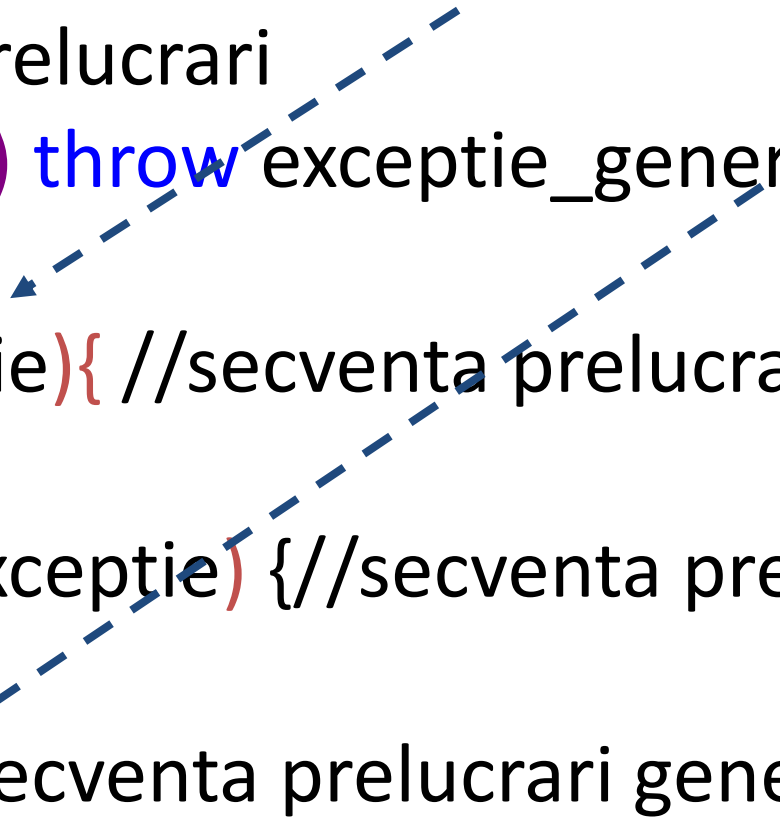
```
namespace Colectie1{
   int vb1;};
namespace Colectie2{
   int vb2;};
void main(){
   using namespace Colectie1;
   vb1 = 10;
   Colectie2::vb2 = 20;
}
```

CLASE – Gestiune exceptii

- exceptie – situatie in care prelucrarea anumitor date de intrare nu este gestionata sau nu este posibila (ex: impartire la 0, citire in afara unui masiv)
- permite gestiunea situatiilor exceptionale care conduc la terminarea imediata a programului
- necesar pentru a realiza programe robuste si fiabile
- implementat prin try, catch si throw
- permite gestiunea erorilor de sistem si a erorilor definite de programator

CLASE – Gestiune exceptii

```
try {
//secventa prelucrari
if(conditie_1) throw exceptie;
//secventa prelucrari
if(conditie_2) throw exceptie_generala;
}
catch(exceptie){ //secventa prelucrari specifice}

catch(alta_exceptie) {//secventa prelucrari specifice}

catch(…){ //secventa prelucrari generale}
```



CLASE – Gestiune exceptii

blocul try{…}

- contine secventa de prelucrari care genereaza exceptii prin throw;
- are asociat minim un bloc catch
- intre blocul try si blocurile catch asociate nu exista alte instructiuni

blocul catch(tip_exceptie exceptie)

- gestioneaza o exceptie de tipul anuntat

blocul catch(…)

- gestioneaza toate tipurile de exceptii

CLASE – Gestiune exceptii

Blocurile catch sunt definite in ordine crescatoare a generalitatii exceptiilor tratate

```
try { … }
catch(exceptie_tip_1){…}
catch(exceptie_tip_2){…}
…
catch(…){…}
```

CLASE – Gestiune exceptii

catch(...){...} poate fi inlocuita de functia standard apelata la tratarea unei exceptii necaptate – terminate( )

```
void functie_terminate(){
cout << "functie_terminate()";
exit(-1);
}
set_terminate( functie_terminate );
```

CLASE – Gestiune exceptii

DEFINIRE METODE (convenție, NU regula) ce arunca exceptii:

- functia anunta prin header ce exceptii genereaza

```
void functie() throw(exception, DivideByZero){ …}
```

- functia poate genera orice tip de exceptie

```
void functie(){ …}
```

- functia nu genereaza exceptii

```
void functie() throw(){ …}
```

CLASE – Gestiune exceptii

UTILITATE:

- permite separarea prelucrarilor de gestiunea erorilor;
- o noua metoda de a anunta executia cu succes sau nu a unei functii (in detrimentul variabilelor globale)
- IMPORTANTA pentru gestiunea erorilor in constructori

CLASE – Gestiune exceptii

UTILITATE:

- functia anunta o exceptie iar programul apelator:
  - rezolva problema
  - decide reapelarea functiei sau continuarea programului
  - genereaza alta rezultate
  - termina programul intr-un mod “normal” (dezaloca memoria, salveaza rezultate partiale);
  - rezolva partial problema si arunca o noua exceptie pentru un context superior.

CLASE – Gestiune exceptii

DEZAVANTAJE:

- poate complica codul;
- in C++ reprezinta o alternativa la tratarea erorilor local (in interiorul functiei)
- sunt ineficiente din punctul de vedere al executiei programului
- captarea exceptiilor prin valoare;
- nu pentru evenimente asincrone (in C++ exceptia si handler-ul ei sunt prezente in acelasi apel (call stack)
- IMPORTANT generearea exceptiilor in functiile destructor si constructor;

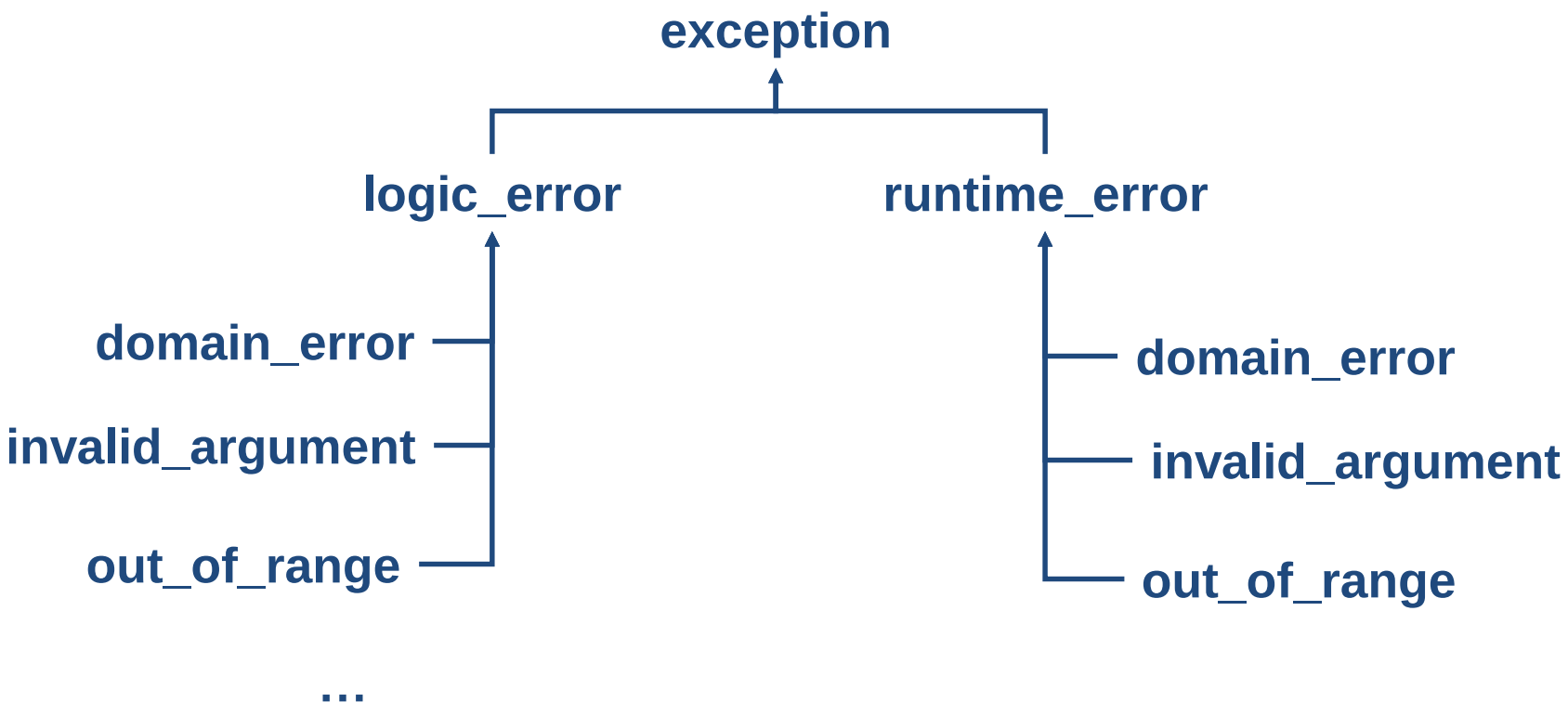
CLASE – Gestiune exceptii

DEZAVANTAJE:

- generarea de exceptii in constructor intrerupe executia acestuia si obiectul nu mai este construit (NU se mai apeleaza destructorul) insa memoria alocata dinamic pana la throw genereaza memory leak
- generarea de exceptii in destructor intrerupe executia acestuia si pot fi generate memory leaks

CLASE – Gestiune exceptii

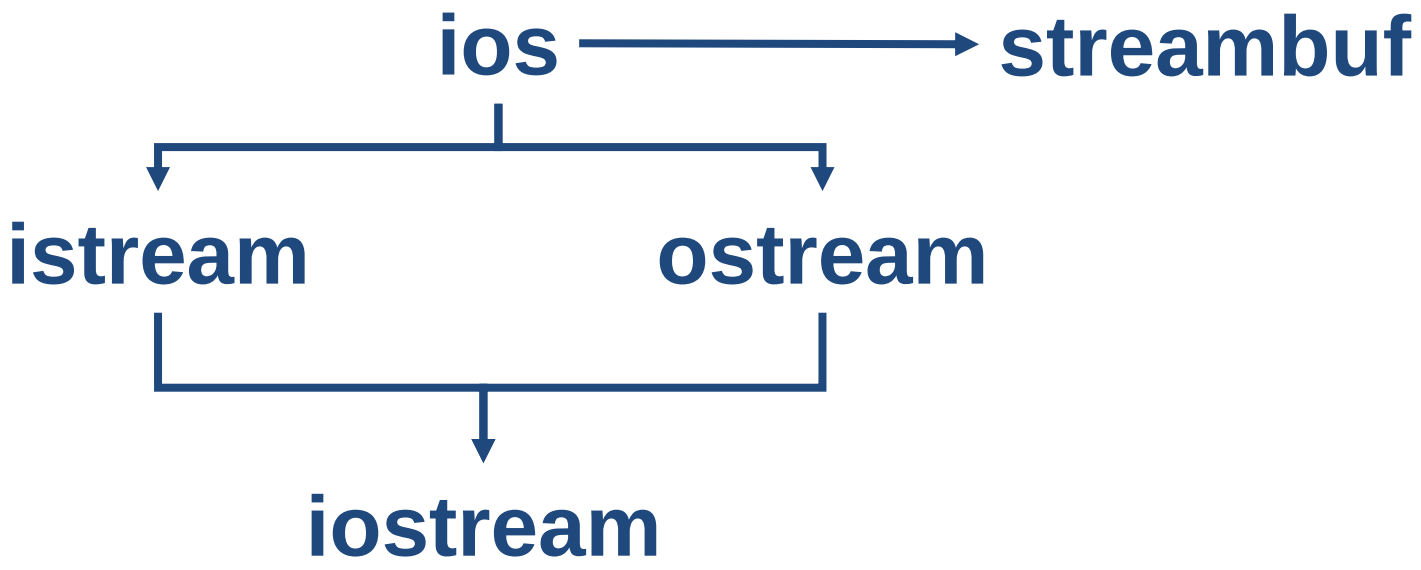
Ierarhie clase C++ pentru exceptii standard:



CLASE – Stream-uri

- STREAM – obiect ce permite gestiunea si manipularea siruri de baiti
- utilizate in mod obisnuit cu forma supraincarcata a operatorului << (inserter) si a operatorului >> (extracter)
- obiecte standard: cout de tip ostream si cin de tip istream, cerr (asociat stream-ului standard de erori)

CLASE – Stream-uri standard



CLASE – Stream-uri standard

Formatarea datelor din stream-uri:

- metode ale obiectelor cin si cout (de exemplu width, fill)
- manipulatori (iomanip.h)
- flag-uri (biti) de formatare din clasa ios (metoda ios::setiosflags)
- flag-uri (biti) de formatare ale obiectelor cin si cout (metoda setf)

CLASE – Stream-uri standard

Manipulatori:

- dec
- hex
- oct
- setprecision(int)
- endl
- ends
- ws
- flush
- setbase(int)
- setfill()
- setw(int)
- setiosflags(long)
- resetiosflags(long)

CLASE – Stream-uri standard

Flag-uri formatare (setiosflags si resetiosflags):

- ios::left
- ios::right
- ios::internal
- ios::dec
- ios::hex
- ios::showpos
- ios::showbase
- ios::scientific
- ios::fixed
- ios::showpoint
- ios::skipws
- ios::stdio
- ios::uppercase
- ios::unitbuf

CLASE – Stream-uri standard

Flag-uri formatare (long ios::setf(long val, long ind)):

- ios::basefield
- ios::floatfield
- ios::adjustfield

- ios::dec
- ios::hex
- ios::oct
- ios:fixed
- ios::scientific
- ios::left
- ios::right
- ios::internal

CLASE – Stream-uri standard

Citire string-uri cu cin:

- asemenea functiei scanf considera terminator de sir de caractere si spatiul
- pentru citiri speciale se folosesc metodele get si getline ale obiectului cin;

cin.getline(char* adr_buffer, int nr_bytes, int delimitator) – extrage delimitatorul din input

cin.get(char* adr_buffer, int nr_bytes, int delimitator)

CLASE – Stream-uri standard

Detectare erori la citire/scriere:

- erori semnalate prin setarea (valoare 1) unor flag-uri (biti) de stare ai fluxurilor:
  - failbit
  - badbit
- flag-urile sunt testate prin metodele:
  - boolean good()
  - int fail()
  - int bad()
- flag-urile sunt resetate prin metoda clear()

CLASE – Stream-uri standard

Detectare erori la citire/scriere:

```
int Cod = 0;
bool IsOk = false;
      while(!IsOk){
         cout<<"\n Cod angajat:";
         cin>>Cod;
         IsOk = intrare.good();
         if(!IsOk)
                  cerr<<"\n Valoare eronata pentru cod !";
         cin.clear();
         cin.ignore(256,'\n');
      }
```

resetare flag-uri

golire buffer input

CLASE – Stream-uri standard

Detectare erori la citire/scriere:

pt erori >> returneaza NULL

```
int valoare;
while(cout<<"Dati o valoare sau CTRL+Z:",
cin>>valoare);
```

operator , returneaza ultima valoare

CLASE – Stream-uri nestandard

Intrari / Iesiri pe fisiere nestandard:

- prin obiecte definite in fstream.h
- ifstream - lucru cu fisiere in intrare;
- ofstream - lucru cu fisiere in iesire;
- fstream - lucru cu fisiere in intrare/intrare;
- definire prin:
  - constructor cu nume:

tip_fstream ob_fis(char * nume_fisier)

  - prin metoda open:

ob_fis.open (char * nume_fisier, long mod_open)

CLASE – Stream-uri nestandard

Mod deschidere:

- ios::in - deschis in citire
- ios::out -deschis in scriere
- ios::ate -deschidere si pozitionare la sfarsit
- ios::app -deschidere pentru adaugare
- ios::trunc -deschidere si stergere continut
- ios::nocreate -nu deschide daca nu exista
- ios::noreplace -nu deschide daca el exista
- ios::binary -deschidere in mod binar

CLASE – Stream-uri nestandard

Citire / Scriere din fisiere:

- prin operatorii >> si <<;
- prin metodele get( ) (returneaza EOF, -1, pentru sfarsit de fisier) si put ( ) ce lucreaza la nivel de octet
- prin metodele generale read( ) (returneaza NULL, pentru sfarsit de fisier) si write( ) :

istream& istream::read(char * buffer, int nr_octeti)

ostream& ostream::write(char * buffer, int nr_octeti)

CLASE Stream-uri nestandard

Citire / Scriere din fisiere:

buffer in care se citeste din fisier

istream& istream::read(char * buffer, int nr_octeti)

nr octeti de citit / scris

buffer din care se scrie in fisier

ostream& ostream::write(char * buffer, int nr_octeti)

CLASE – Stream-uri nestandard

Citire din fisiere:

- testarea sfarsitului de fisier se face si prin verificarea flag-ului eofbit prin metoda

bool eof()

a unui obiect de tip fstream.

CLASE – Stream-uri nestandard

Pozitionare in fisiere:

- pentru fisiere de input (citire):

istream & istream::seekg(long streamoff, ios::seekdir)

sau

istream & istream::seekg(long streampos)

unde:

ios::seekdir poate fi:

ios::beg

ios::cur

ios::end

CLASE – Stream-uri nestandard

Pozitionare in fisiere:

- pentru fisiere de output(scriere):

ostream & ostream::seekp(long streamoff, ios::seekdir)

sau

ostream & ostream::seekp(long streampos)

unde:

ios::seekdir poate fi:

ios::beg

ios::cur

ios::end

CLASE – Stream-uri nestandard

Pozitionare in fisiere:

- pentru fisiere de output(scriere) determinarea pozitiei curente se face prin metoda long tellp() ce returneaza numarul de octeti de la inceputul fisierului
- pentru fisiere de input(citire) determinarea pozitiei curente se face prin metoda long tellg() ce returneaza numarul de octeti de la inceputul fisierului

CLASE - Streamuri standard

RECAPITULARE MODURI DE ORGANIZARE FISIERE (Anul 2 + Structuri de date):

- Organizare secventiala cu inregistrari de lungime fixa si variabila
- Acces direct
- Fisiere indexate
- Fisiere de tip invers

CLASE – Clase TEMPLATE

Functii template:

- permit cresterea gradului de generalizare prin definirea de sabloane de functii
- la definire se utilizeaza tipuri generice:

```
class T
typename T
```

- functia este instantiata de compilator la utilizare cand tipul generic este inlocuit de un tip concret

CLASE – Clase TEMPLATE

Functii template:

- definire:

```
template <typename T1, typenameT2, …>

tip_returnat nume_functie( T1 param1, T1 param2,
    T2 param3, … )
```

- initializare & utilizare:

```
nume_functie <tip_concret, tip_concret, …>
( param1, param2, param3,… )
```

CLASE – Clase TEMPLATE

Functii template:

- definire:

```
template <typename T>
T aduna (T a, T b){ return a+b;}
```

- initializare &utilizare:

```
int suma = aduna<int> (5,6);
double suma2 = aduna<double>(5.5, 6.6);
```

CLASE – Clase TEMPLATE

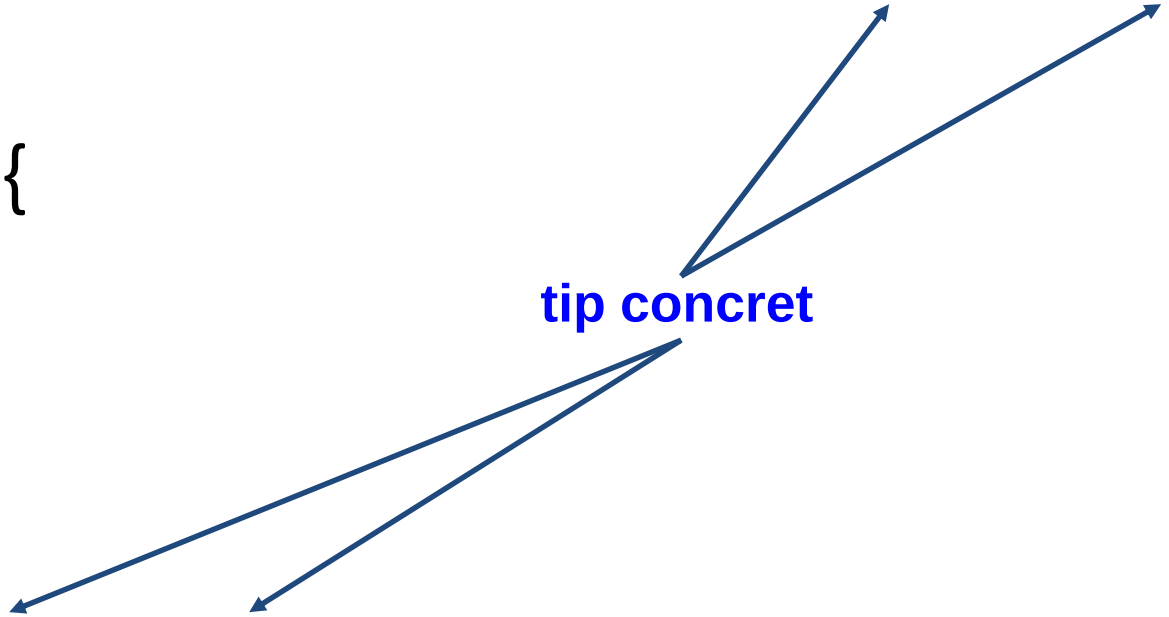
Clase template:

- reprezinta sabloane de clase, descrieri parametrizate de clasa;
- permit adaptare la tipuri de date concrete (fundamentale + utilizator)
- prin instantierea sablonului, constructorul genereaza clase concrete

CLASE – Clase TEMPLATE

Clase template:

```
template <class T1, typename T2, …, tip1 c1, tip2 c2,
        …>
class nume_clasa{
        …
        }
```




```
nume_clasa<tipc1, tipc2, …, val1, val2, …> obiect;
```

CLASE – Clase TEMPLATE

Clase template cu tip generic :

```
template <typename T>
    class Vector{
        T * valori;
        int dim;
    public:
        …
    };

    Vector<int> v1;
```

CLASE – Clase TEMPLATE

Clase template cu tip generic + constante:

```
template <typename T, int n>
        class Vector_S{
                T valori[n];
                int dim;
        public:
                …
        };

        Vector<int,10> v1;
```

CLASE – Clase TEMPLATE

Clase template cu tip generic + constante (cu valori default):

```
template <typename T=int, int n=5>
        class Vector_S{
                T valori[n];
                int dim;
        public:
                …
        };
        Vector<int,10> v1;
        Vector<> v2;
        Vector<double> v2;
```

CLASE – Clase TEMPLATE

Utilizare clase template:

Se doreste utilizarea sablonului sau a unui caz concret ?

- caz concret:

```
int compara(Vector<int> v1, Vector<int>
    v2){…}
```

- sablon:

```
template<typename T1, typename T2>
 int compara(Vector<T1> v1, Vector<T2>
    v2){…}
```

CLASE – Clase TEMPLATE

Utilizare clase template in librarii dinamice:

- NU se pot construi librarii dinamice (LIB, DLL) de sabloane;
- trebuie anuntate utilizari viitoare pentru a forta instantieri

```
template class Vector<int, 5>;
```

si pentru metode:

```
template class Vector<int, 5>::Vector(int, int)

template int compara<int, int>(int, int)
```

Specializari: CLASE – Clase TEMPLATE

- definesc situatii concrete in care metodele, functiile, clasele se comporta diferit fata de situatia generala
- au prioritate asupra abordarii generale
- se aplica de obicei unor metode:

```
tip_returnat nume_clasa<tipc>::nume_metoda (lista parametrii) { … }
```

- pot fi specializate clase template intregi:

```
template<> nume_clasa<tipc> { … }
```

Derivare: CLASE – Clase TEMPLATE

template<typename T> class bt {…};

class b {…};

- clasa template derivata din clasa template

template<typename T> class d: public bt<T> {…}

- clasa template derivata din clasa non template

template<typename T> class d: public b {…}

- clasa non template derivata din clasa template

template class d: public bt<int> {…}

CLASE – Clase TEMPLATE

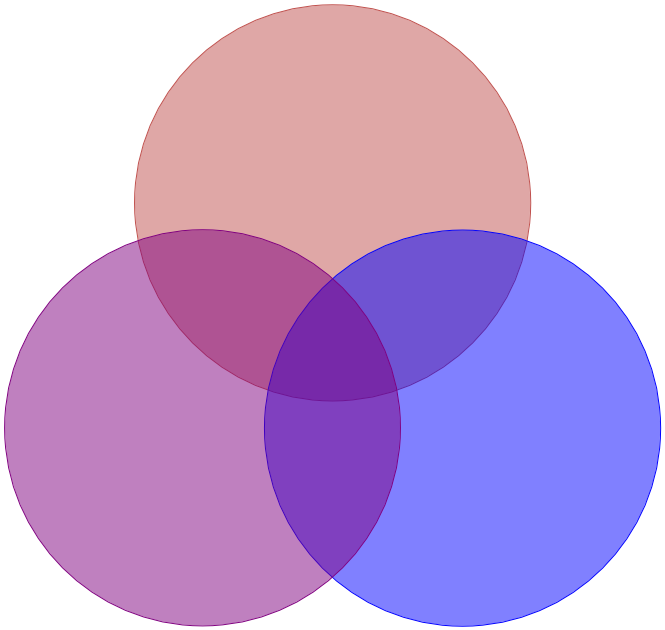
- compunere de clase template prin includere
- compunere de clase template prin parametrizare cu alta clasa template.

CLASE – STL

- reprezinta o librarie de clase template standard (standard template library)
- acopera principalele structuri de date: vector, lista, stiva, coada, tabela de dispersie (hash-table);
- pot fi utilizate fara alte modificari pentru tipuri fundamentale sau definite de programator.

CLASE – STL

CONTAINERE



ITERATORI

ALGORITMI

CONTAINERE CLASE – STL

- un obiect ce stocheaza alte obiecte si are metode pentru a le accesa;
- tipuri (fct de ordine si acces):
  - forward
  - reversible
  - random access
- tipuri (fct de aranjare):
  - sequences
  - associative containers
  - container adaptors

CLASE – STL

TIP CONTAINERE

- secventiale:
  - vector;
  - list;
  - deque;
- asociative (valoare – cheie):
  - set (multime de chei unice, sortate)
  - multiset (multime de chei, sortate)
  - map (multime valori-chei unice, sortate)
  - multimap (multime valori-chei sortate)
- adaptive:
  - stack
  - queue
  - priority_queue

CLASE – STL

ITERATORI

- forma generalizata a pointerilor;
- utilizati pentru a itera prin elementele containerelor
- interfata intre containere si algoritmi
- iteratori predefiniti:
  - ostream_iterator;
  - istream_iterator;
  - reverse_iterator;
  - insert_iterator;

CLASE – STL

ALGORITMI

- functii generice independente de tipul containerului;
- utilizate pentru a prelucra elementele containerelor
- folosesc iteratori pentru acces la elemente
- functii importante:
  - copy;
  - for_each;
  - sort;
  - find;
  - transform

RTTI

CLASE – RTTI

- Run-Time Type Identification;
- mecanism pentru a determina la Run-time tipul obiectului gestionat printr-un pointer la baza
- are sens in contextul unei ierarhii de clase + upcasting + functii virtuale

CLASE – RTTI

RTTI - typeid()

- determina tip continut pointer printr-o structura de tip type_info (typeinfo)

```
ComponenteGrafice::Model2D *pModel;
pModel = new ComponenteGrafice::Dreptunghi();
cout << typeid(*pModel).name() << endl;
```

CLASE – RTTI

RTTI - dynamic_cast<T>()

- “type-safe downcast”;
- este o functie template;
- permite conversia la tipul T pentru un pointer la obiect de baza dca continutul de la adresa data de pointer este de tip T
- evita erorile de conversie imposibile;
- returneaza T* daca este posibil, altfe NULL;

CLASE – RTTI

RTTI - dynamic_cast<T>()

```
using namespace ComponenteGrafice;
pModel = new Dreptunghi();
if(dynamic_cast<Dreptunghi*>(pModel))
{
   cout<<endl<<"Continut de tip Dreptunghi !";
   Dreptunghi oDreptunghi =
*dynamic_cast<Dreptunghi*>(pModel);
   oDreptunghi.Arie();
}
```