

# PROGRAMARE ORIENTATĂ OBIECT

Curs 1

Structura cursului

Competențe

Bibliografie

Principalele tehnici de programare

Limbajul C++, tipuri de date

# Structura cursului

---

- ▶ Limbajul de programare C++
- ▶ Principii POO și suportul oferit de limbajul C++
  - ▶ Abstractizare
  - ▶ Moștenire
  - ▶ Polimorfism
- ▶ STL



# Competențe obținute la finalul cursului

---

- ▶ Asimilarea tehnicii de programare pe obiecte
- ▶ Limbajul de programare C++



# Evaluare

---

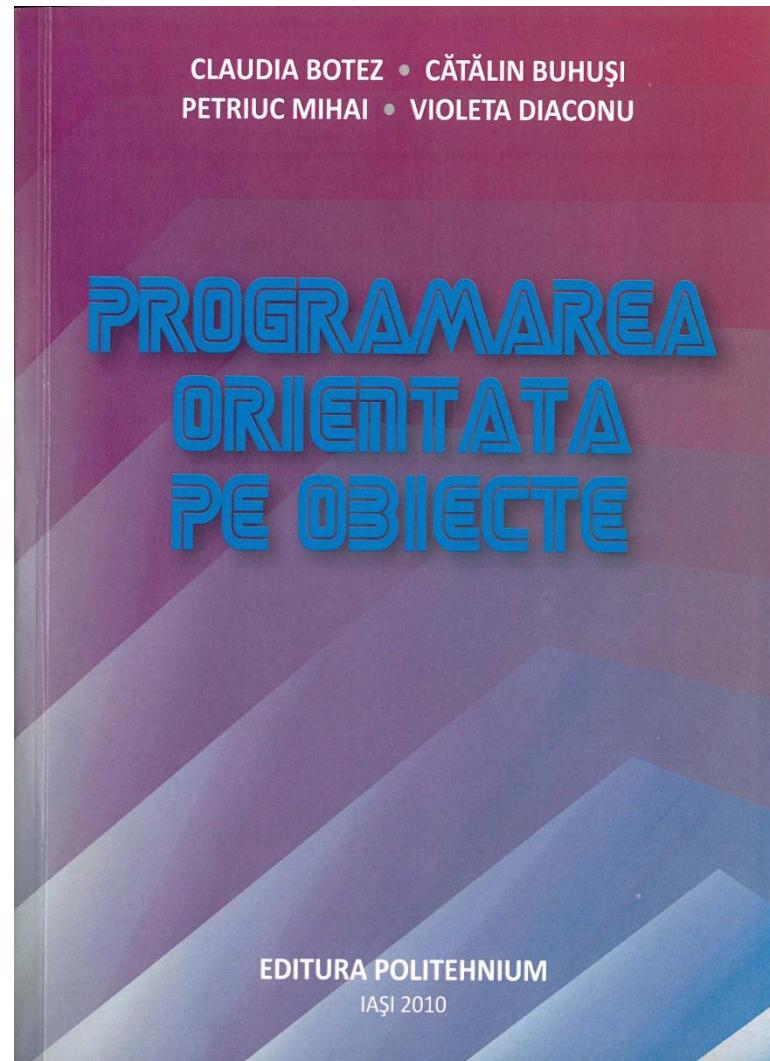
- ▶ Activitate de laborator – 10%, minim nota 5 pentru a intra in examen
  - ▶ Medie teste de laborator (2 teste anunțate) – 30%, nota obtinuta nu condiționează intrarea în examen
  - ▶ Examen (practic C++ fără Java) – 60%
  - ▶ Proiect – 1 punct în plus la nota finală
- 
- ▶ Nota finală:
    - ▶  $N = L \times 0.1 + T \times 0.3 + E \times 0.6 + (E \geq 8) \times P \times 0.1$
    - ▶ Verificare cunoștințe teoretice în caz de incertitudine (rotunjire în plus sau în minus)



# Bibliografie

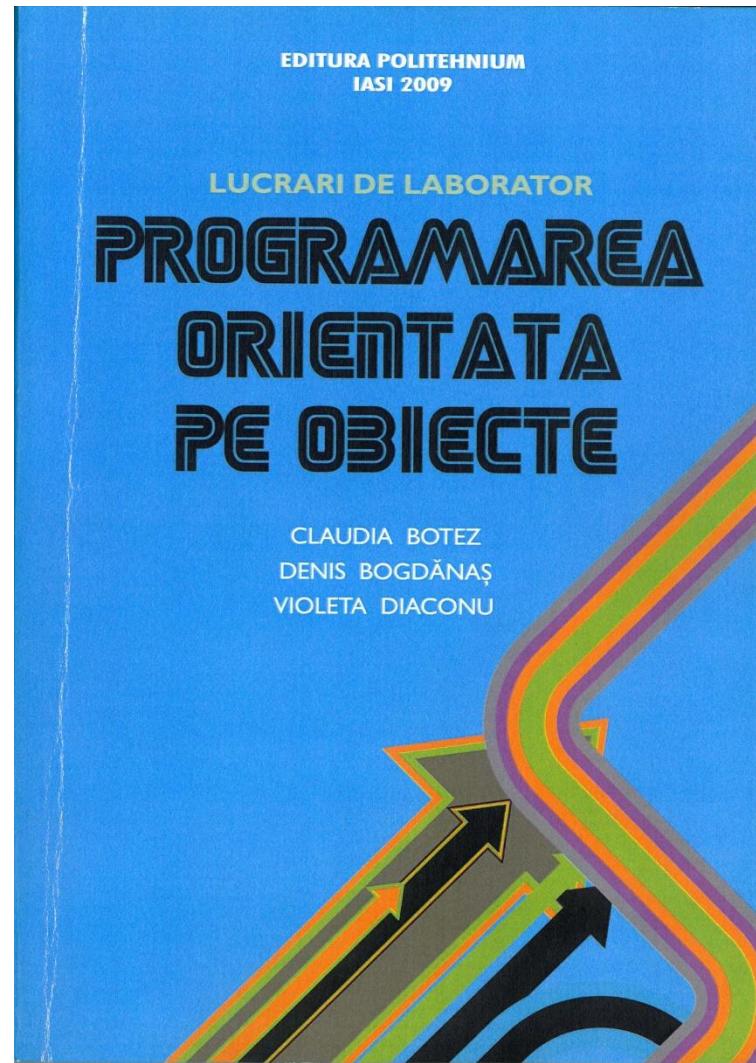
---

- ▶ Programarea orientată pe obiecte, *Claudia Botez*



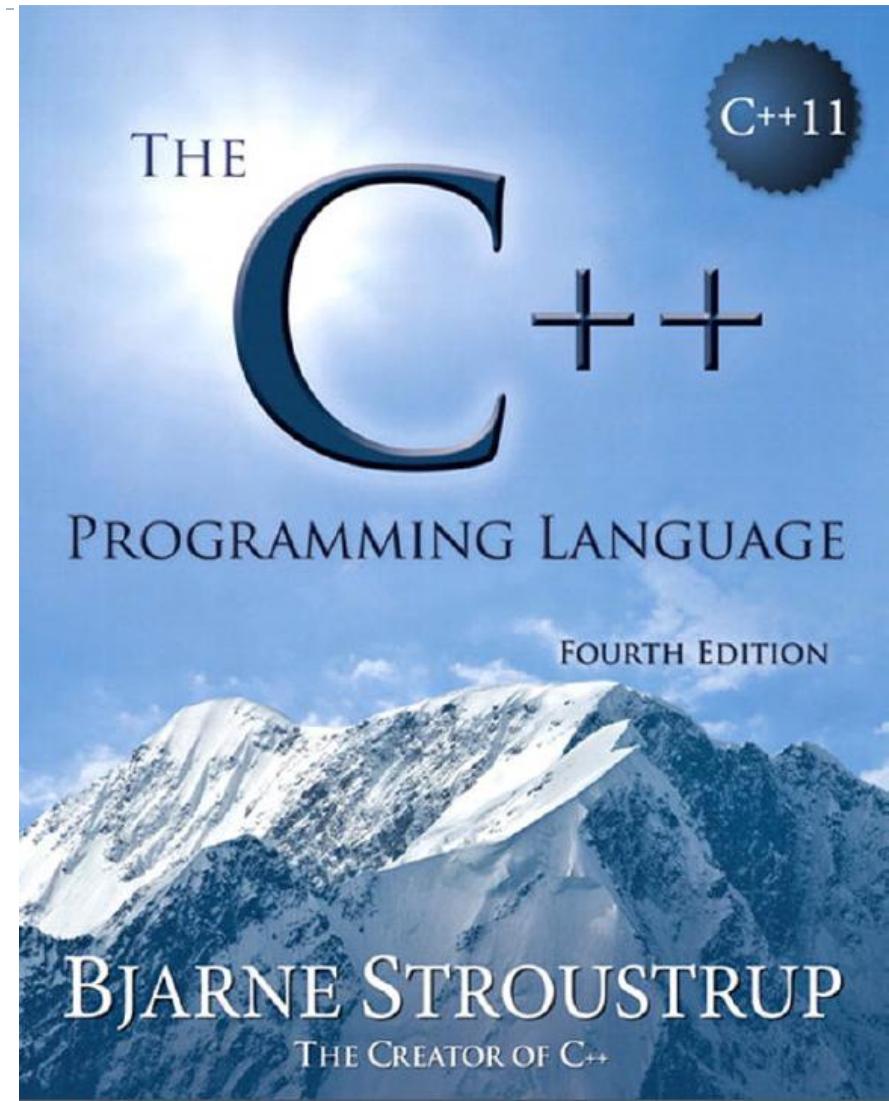
# Bibliografie

- ▶ Programarea orientată pe obiecte, *Claudia Botez*
- ▶ Lucrări de laborator, Programarea orientată pe obiecte, *Claudia Botez*



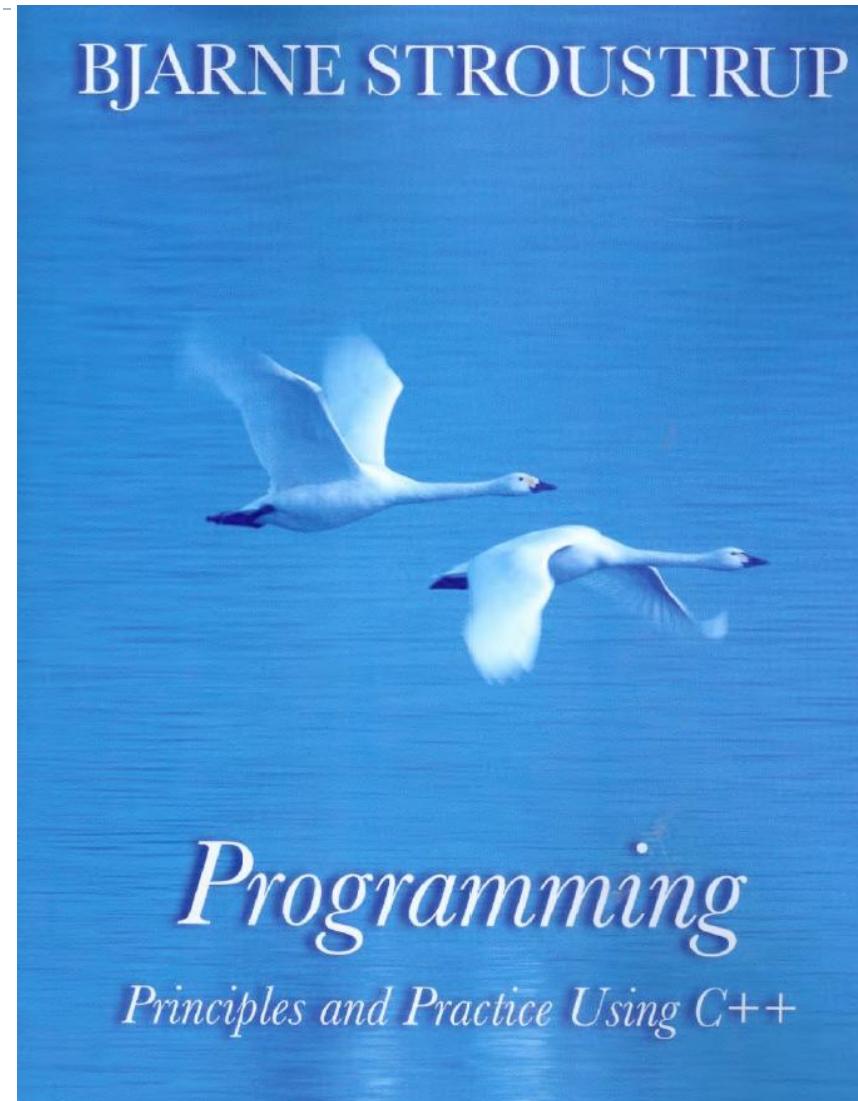
# Bibliografie

- ▶ Programarea orientată pe obiecte, *Claudia Botez*
- ▶ Lucrări de laborator, Programarea orientată pe obiecte, *Claudia Botez*
- ▶ **The C++ programming language fourth edition, Bjarne Stroustrup**



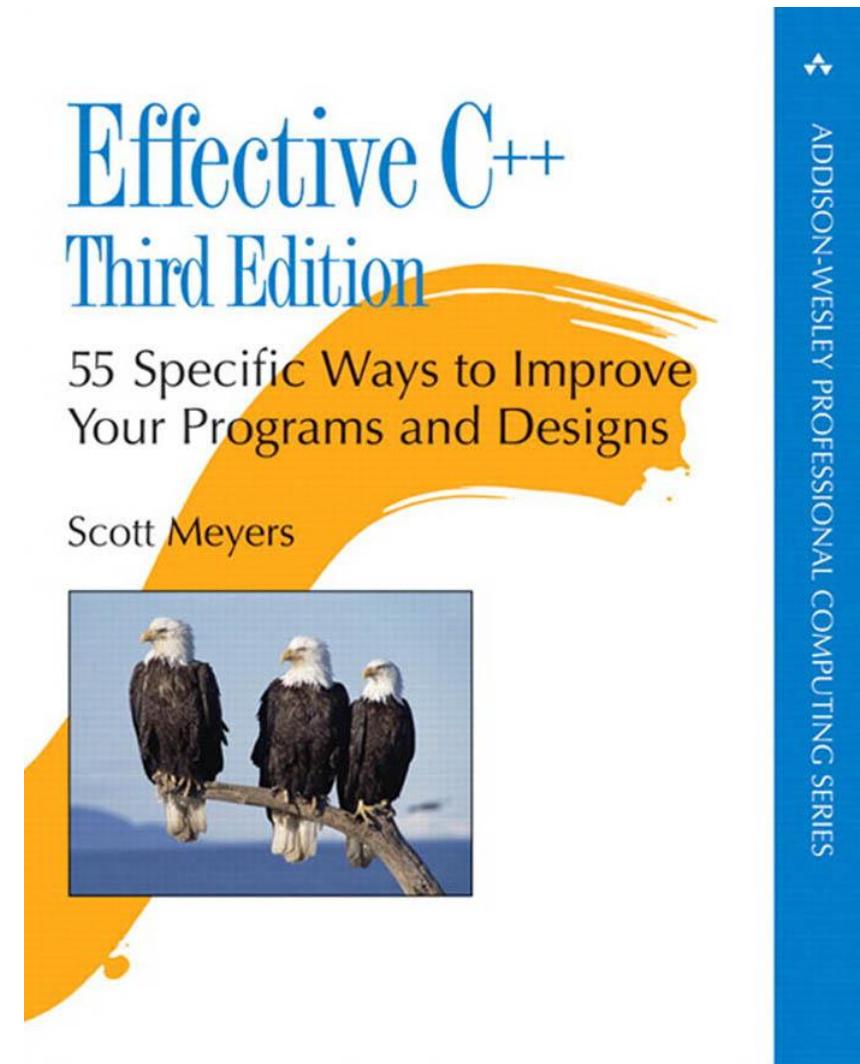
# Bibliografie

- ▶ Programarea orientată pe obiecte, *Claudia Botez*
- ▶ Lucrări de laborator, Programarea orientată pe obiecte, *Claudia Botez*
- ▶ The C++ programming language fourth edition, *Bjarne Stroustrup*
- ▶ **Programming principles and practice using C++,  
*Bjarne Stroustrup***



# Bibliografie

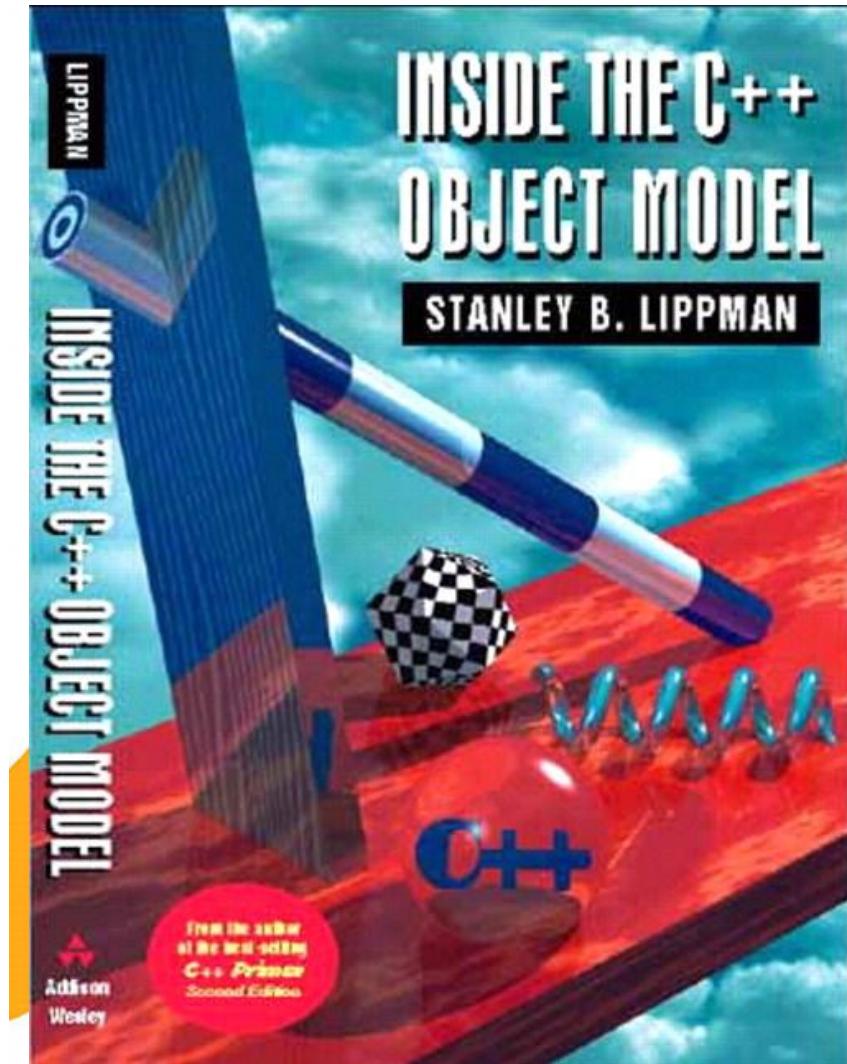
- ▶ Programarea orientată pe obiecte, *Claudia Botez*
- ▶ Lucrări de laborator, Programarea orientată pe obiecte, *Claudia Botez*
- ▶ The C++ programming language fourth edition, *Bjarne Stroustrup*
- ▶ Programming principles and practice using C++, *Bjarne Stroustrup*
- ▶ **Effective C++, third edition, Scott Meyers**



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

# Bibliografie

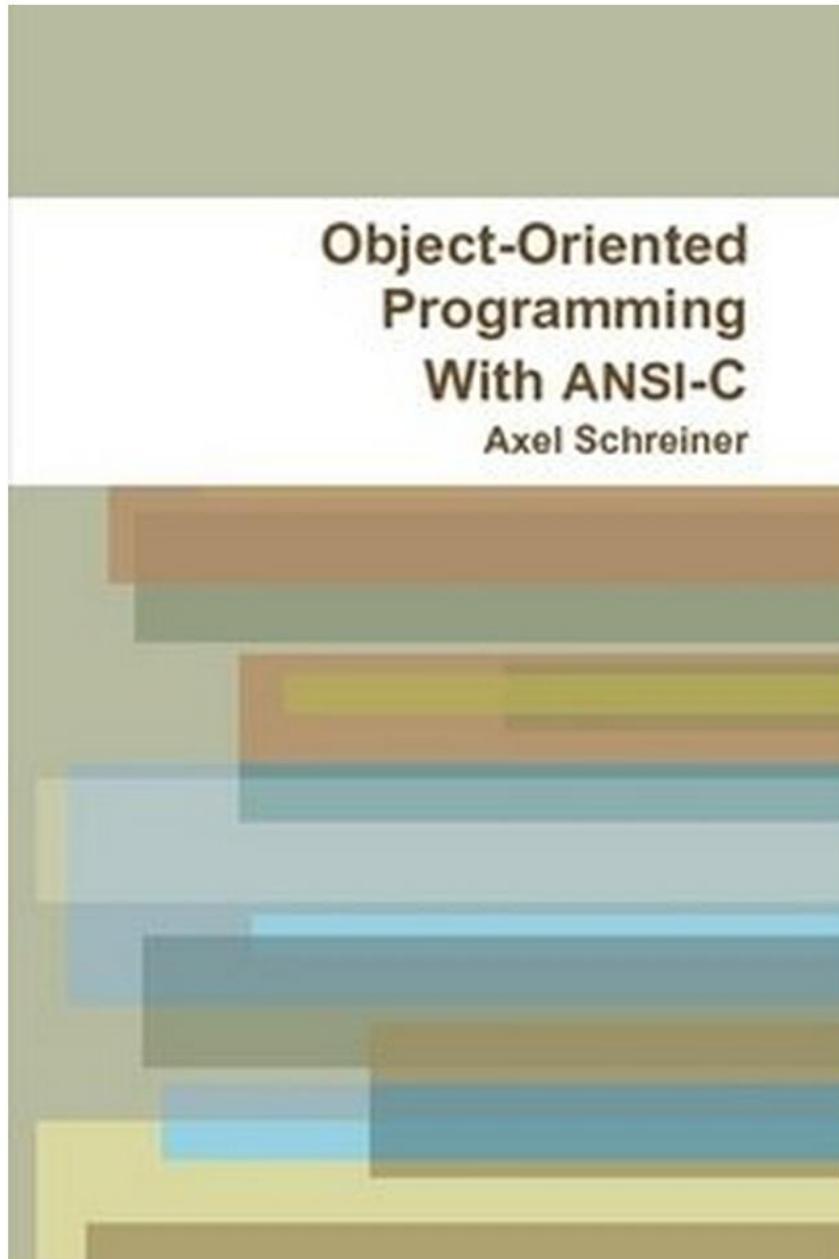
- ▶ The C++ programming language fourth edition, *Bjarne Stroustrup*
- ▶ Programming principles and practice using C++, *Bjarne Stroustrup*
- ▶ Effective C++, third edition, *Scott Meyers*
- ▶ Inside the C++ object model, *Stanley B. Lippman*



# Bibliografie

---

- ▶ Object-Oriented  
Programming with ANSI-C,  
*Axel Schreiner*



# Principalele tehnici de programare

---

- ▶ Activitatea de proiectare, codificare, testare și documentare a programelor se numește activitate de programare.
- ▶ ***definirea generală a problemei***  
În această fază se realizează schema logică conceptuală de rezolvare a problemei, se structurează datele de intrare/ieșire și se stabilesc algoritmii de calcul;
- ▶ ***stabilirea logicii programelor***  
Această fază constă în descrierea algoritmilor în pseudocod sau utilizând scheme logice
- ▶ ***codificarea și testarea programelor***  
Acum, se scrie textul sursă într-un limbaj, de obicei de nivel înalt și se testează programul cu date de test;



# Principalele tehnici de programare

---

- ▶ ***implementarea și exploatarea programelor***
  - ▶ Programele sunt date în exploatare și se fac eventualele adaptări. Programele pot fi eventual împachetate în biblioteci utilizator;
- ▶ ***documentarea programelor***
  - ▶ *partea I-a:* documentația tehnică, care cuprinde: definirea problemei, schema conceptuală, logica problemei, textul sursă, și datele de test.
  - ▶ *partea – II-a:* documentația de operare



# Tehnici de programare

---

- ▶ Monolică
- ▶ Procedurală
- ▶ Modulară
- ▶ Programării pe obiecte



# Tehnica programării monolitice

---

- ▶ Toată problema este rezolvată de la cap la coadă într-un singur program (funcție)
  - ▶ Dezavantaje
    - ▶ Dificultate în depistarea erorilor (în programe mari)
    - ▶ Dificultate în întreținere
    - ▶ Codul nu poate fi reutilizat



# Tehnica programarii procedurale

---

- ▶ Nevoia de a reutiliza codul deja scris
- ▶ Secvența de instrucțiuni organizate în sensul rezolvării unei probleme se numește **procedură** sau **subprogram** sau **subrutină**
  - ▶ Procedura nu întoarce nimic
  - ▶ Funcția poate întoarce o valoare sau nu (void)
- ▶ Utilizare procedură/funcție → proces de abstractizare realizat pe baza parametrilor
  - ▶ la definirea unei proceduri/funcții se lucrează cu parametri formali
  - ▶ la apelul unei proceduri/funcții se lucrează cu parametri reali, efectivi sau actuali



# Tehnica programării procedurale

---

- ▶ Procesul de abstractizare prin intermediul parametrilor se numește abstractizare procedurală



- ▶ Utilizatorul nu trebuie să știe decât tipul parametrilor  $a$  și  $b$  și tipul valorii returnate



# Tehnica programarii modulare

---

- ▶ Definiția lui Bjarne Stroustrup, spune că un **modul** este un set de proceduri înrudite împreună cu datele pe care le manevrează.
- ▶ De multe ori datele unui modul sunt “ascunse”, adică accesul la ele este limitat, fiind protejate. De fapt accesul la aceste date este indirect, prin procedurile modulului (mai ales în programarea orientată pe obiecte).



# Tehnica programării modulare

---

- ▶ Fiecare modul se găsește într-un fișier sursă, deci este **o unitate de compilare**. Prin urmare, modulele se compilează separat, rezultând din fiecare câte un modul obiect, care apoi se leagă într-un program executabil, de către editorul de legături.



# Tehnica programarii modulare

## Fișiere header (antet)

---

- ▶ Împărțirea unui program mare în mai multe module se face respectând condiția ca orice modul să fie compilat independent. Acest lucru se întâmplă dacă *orice nume este utilizat numai după ce a fost declarat.*
- ▶ Legăturile între module se realizează prin intermediul funcțiilor și al variabilelor globale, pentru care definiția apare într-un fișier, iar în celelalte fișiere în care sunt folosite, ele sunt doar declarate.



# Tehnica programarii modulară

## Ce pot conține fișierele header

1) declarații de funcții:

```
void ff(int);  
extern void eroare (char *);
```

2) declarații de variabile:

```
extern int i;  
extern double f;
```

3) definiții de tipuri, enumerări:

```
typedef int lungime;  
struct complex {  
    double re, im;  
};  
enum stare {OFF, ON};
```



# Tehnica programarii modulare

## Ce pot conține fișierele header

4) definiții de constante, dar *nu* masive

```
const char ESC = '\x1B';
```

5) directive preprocesor

```
#include <stdio.h>
#define MAX 100
```

6) definiții de funcții inline, specifice limbajului C++

```
inline int increment (int i)
{
    return ++i;
}
```



# Tehnica programarii modulare

## Ce **nu** pot conține fișierele header

1) definiții de variabile globale

```
int i = 100;  
double f;
```

2) definiții de funcții, care nu sunt inline:

```
void eroare (char *mesaj)  
{  
    printf ("\n %s\n", mesaj);  
}
```

3) definiții de masive (tablouri):

```
const int cifre [] = {0,1,2,3,4,5,6,7,8,9};
```

### *Utilizare*

*Un singur fișier antet  
Mai multe fișiere antet*



# Tehnica programării modulare

## Compilarea condiționată

### Fis.h

```
#ifndef __CONST__H
#define __CONST__H

extern int i;
void ff(int);
//.....continutul fisierului
#endif
```

### Fis.h

```
#pragma once

extern int i;
void ff(int);
//.....continutul fisierului
```



# Tipuri de date C++

---

- ▶ char, unsigned char, signed char (**1 octet**)
- ▶ short [int], unsigned short [int] (**2 octeți**)
- ▶ int, unsigned [int] (**4 octeți**)
- ▶ long [int], unsigned long [int] (**4 octeți**)
- ▶ long long [int] (**8 octeți**)
- ▶ float (**4 octeți**)
- ▶ double, long double (**8-10 octeți**)
- ▶ bool (**1 octet**)



# PROGRAMARE ORIENTATĂ PE OBIECTE

Curs 2

*Facilități ale limbajului C++ utilizate în POO*

*Domeniu*

*Namespace*

*Tipul referință*

*Argumente cu valori predefinite*

*Supraîncărcarea numelor de funcții*

*Constante, pointeri și constante*

# Domeniu (scope)

---

- ▶ O declarație introduce un nume (*identifier*) într-un domeniu astfel încât acel identifier poate fi folosit doar într-o parte specifică a unui program.
  - ▶ **Domeniu local** – un bloc este o secvență de cod delimitată de **{}**. Orice nume declarat în interiorul unui bloc este numit **nume local**;
  - ▶ **Domeniu clasă, structură** – un nume dacă este declarat în interiorul unei clase/structuri este numit nume membru. Domeniul de valabilitate începe de la deschiderea cu **{** a declarației clasei și până la încheierea declarației clasei;
  - ▶ **Domeniu namespace** – unu nume este numit nume membru al *namespace* dacă este definit într-un namespace în afara oricărei funcții, clase etc membre a spațiului;
  - ▶ **Domeniu global** – unu nume este global dacă este definit în afara oricărei funcții, clase, namespace etc. Domeniul de valabilitate din momentul declarației și până la sfârșitul fișierului;



# Domeniu (scope)

---

- ▶ **Domeniu instrucțiune** (statement scope) – un nume este în domeniu instrucțiune dacă este definit în interiorul () unei instrucțiuni **for**, **while**, **if** sau **switch**. Domeniul de valabilitate este limitat doar la instrucțiune. Este considerat nume local;
- ▶ **Domeniu funcție** – un nume declarat este valabil tot corpul funcției.
- ▶ O declarație a unui nume într-un bloc poate ascunde o declarație din blocul părinte sau una globală. Astfel un nume poate fi redefinit (*poate referi un alt tip*) în interiorul oricărui bloc existent. După ieșirea din blocul respectiv se poate reveni la definiția anterioară.



# Domeniu (scope)

---

```
int x; // variabilă globală x

void f(void)
{
    int x; // variabilă locală x ascunde variabila globală x
    x = 1; // atribuire variabilei locale x
    {
        int x; // ascunde prima declarație locală a variabile x
        x = 2; // atribuire variabilei locale secundare x
    }
    x = 3; // atribuire primei variabile locale x
}

int *p = &x; //initializare cu adresa variabilei globale x
```

- ▶ Ascunderea (*umbrirea*) numelor este inevitabilă în programele mari



# Domeniu (scope)

---

- ▶ Un nume global ascuns poate fi accesat cu ajutorul operatorului de rezoluție de domeniu `::`.

```
int x;
void f2(void)
{
    int x = 1; // ascunde variabila globală x
    ::x = 2;   // atribuire variabile globale x
    x = 2;     // atribuire variabilei locale x
}
```

- ▶ O variabilă locală ascunsă nu poate fi accesată.



# Domeniu (scope)

---

- ▶ Domeniul unui nume care nu este membru al unei clase începe din momentul declarației complete înainte de inițializare:

```
int x = 97;  
void f3()  
{  
    int x = x; // fără sens: inițializarea variabilei x cu  
               // valoarea sa neinițializată  
}
```



# Domeniu (scope)

---

- ▶ Este posibilă utilizarea unui singur nume pentru a referi două obiecte diferite:

```
int x = 11;
void f4(void) // fără sens: utilizarea a două nume într-un
               // singur domeniu, ambele numite x
{
    int y = x; // utilizarea variabilei globale x: y = 11
    int x = 22;
    y = x;    // utilizarea variabile locale x: y = 22
}
```



# Namespace (continuare)

---

- ▶ Orice program constă din mai multe părți separate.
- ▶ Ex.
  - ▶ Programul **”Hello, world!”** implică cel puțin două părți: cererea utilizatorului de a tipări **”Hello, world!”** și sistemul I/O care va realiza tipărirea
  - ▶ Namespace permite gruparea unor entități cum ar fi clase, obiecte și funcții sub același nume. Astfel, domeniul global poate fi divizat în subdomenii, fiecare cu numele său.
  - ▶ Formatul este: *namespace identifier*  
    {  
        *entitati*  
    }  
Unde *identifier* este un nume de identificare valid iar *entitatile* sunt clase, obiecte, variabile și funcții care sunt incluse în acel namespace



# Namespace (continuare)

---

- ▶ Exemplu: *namespace myFirstNamespace*

```
{  
    int a = 15;  
    int b = 17;  
}  
//accesarea variabilelor  
myFirstNamespace::a;  
myFirstNamespace::b;
```

- ▶ În exemplul de mai sus a și b sunt variabile declarate în namespace-ul *myFirstNamespace*. Pentru a accesa variabilele respective din afara *myFirstNamespace* se va folosi operatorul de domeniu.



# Namespace (continuare)

---

- ▶ Funcționalitatea namespace-urilor este utilă atunci când există posibilitatea ca un obiect global sau funcție să aibă aceeași denumire cauzând erori de redefinire.

```
namespace mySecondNamespace
{
    int a = 25;
    int b = 27;
}

int main (void)
{
    cout << myFirstNamespace::a << endl;
    cout << mySecondNamespace::a << endl;

    return 0;
}
```



# Namespace (continuare)

---

- ▶ Cuvântul cheie ***using*** este folosit pentru a introduce un nume/entitate dintr-un namespace în regiunea curentă de declarații

```
int main (void)
{
    using myFirstNamespace::a;
    using mySecondNamespace::b;

    cout << a << endl;
    cout << b << endl;
    cout << myFirstNamespace::b << endl;
    cout << mySecondNamespace::a << endl;

    return 0;
}
```



# Namespace (continuare)

---

- ▶ Cuvântul cheie **using** poate fi folosit pentru a introduce un întreg namespace

```
int main (void)
{
    using namespace myFirstNamespace;

    cout << a << endl;
    cout << b << endl;
    cout << mySecondNamespace::a << endl;
    cout << mySecondNamespace::b << endl;

    return 0;
}
```



# Namespace (continuare)

---

- ▶ Se pot utiliza mai multe namespace-uri dacă se ține cont de domeniu

```
int main (void)
{
    {
        using namespace first;
        cout << x << endl;
    }
    {
        using namespace second;
        cout << x << endl;
    }
    return 0;
}
```

- ▶ Se pot declara nume alternative pentru namespace-uri existente astfel:

```
namespace new_name = current_name;
```



# Tipul referință în C++

---

- ▶ Referința este un alias pentru o anume variabilă. Dacă în C, transmiterea parametrilor unei funcții se face prin valoare (inclusiv și pentru pointeri), în C++ se adaugă și transmiterea parametrilor prin referință.
- ▶ Dacă tipul pointer se introduce prin construcția: *tip \**, tipul referință se introduce prin *tip &*.
- ▶ O variabilă referință trebuie să fie inițializată la definirea sau declararea ei cu numele unei alte variabile.

```
void f(void)
{
    int i = 1;
    int &r = i;      // r și i refere același i
    int x = r;       // x=1
    r = 2;          // i=2
}
```



# Tipul referință în C++

---

- ▶ Fiecare referință trebuie inițializată

```
int i = 1;
int &r1 = i;    //ok
int &r2;          //error
extern int &r3;    //ok, r3 initializat in alta parte
```

- ▶ Inițializarea unei referințe este ceva diferit față de asignare (atribuire). În ciuda aparențelor nici un operator nu acționează asupra referințelor

```
void g(void)
{
    int ii = 0;
    int &rr = ii;
    rr++;           //ii este incrementat cu 1
    int *pp = &rr;   //pp pointează către ii
}
```



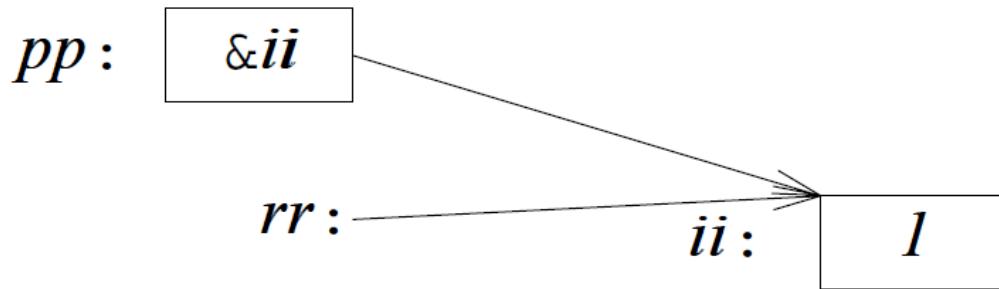
# Tipul referință în C++

---

- ▶  $rr++$  nu incrementează referința  $rr$ . Operatorul  $++$  este aplicat unui ***int*** care este ***ii***.
- ▶ În consecință referința nu poate fi schimbată după inițializare. Întotdeauna va referi obiectul cu care a fost inițializată.
- ▶ Pentru a obține un pointer referit de  $rr$  se poate scrie ***&rr***
- ▶ Implementarea evidentă a unei referințe este un pointer constant care este dereferențiat de fiecare dată când este folosit dacă se are în vedere că nu poate fi manipulat precum un pointer.



# Tipul referință în C++



- ▶ O referință poate fi folosită ca argument într-o funcție astfel încât funcția să schimbe valoarea obiectului respectiv.

```
void increment(int &aa){ aa++; }
void f(void)
{
    int x = 1;
    increment(x); //x=2
}
```

- ▶ La apelul funcției increment argumentul **aa** devine un alt nume al variabilei **x**.

# Tipul referință în C++

---

- ▶ Totuși pentru a menține lizibilitatea unui program se recomandă a se evita utilizarea funcțiilor care își modifică argumentele. În schimb se pot folosi ponteri.

```
int next(int p) { return p + 1; }
void incr(int*p) { (*p)++; }
void g(void)
{
    int x = 1;
    increment(x); //x=2
    x = next(x); //x=3
    incr(&x);    //x=4
}
```

- ▶ **increment(x)** nu furnizează nici un indiciu că variabila x este modificată



# Argumente cu valori predefinite

---

- ▶ O îmbunătățire adusă limbajului C constă în posibilitatea specificării valorilor argumentelor unei funcții atunci când se declară prototipul ei. Aceste argumente sunt considerate a fi **predefined**.
- ▶ Fie redefinirea funcției de copiere a două siruri:

```
char *MyStrcpy( char *dest, char *src, int sense = 0 );
```

- ▶ Următoarele apeluri sunt corecte:

```
MyStrcpy( sirDest, sirSrc );
MyStrcpy( sirDest, sirSrc, 0 );
MyStrcpy( sirDest, sirSrc, 1 );
```



# Argumente cu valori predefinite

---

```
char *MyCopy( char *dest, char *src, int sense )
{
    size_t i; /*!< Index pentru parcurgere a sirului sursa*/
    size_t srcLength = strlen(src);

    for( i = 0; *(src + i); i++ )
    {
        /*(dest + i) = *(src + i); /*!< Copiere normala*/
        /*(dest + i) = *(src + srcLength - 1 - i); /*!< Copiere
                                                   inversa*/

        *(dest + i) = *(src + i + sense*(srcLength - 1 - 2 * i));
    }

    return dest;
}
```



# Argumente cu valori predefinite

---

## ► Apel:

```
char src[]="Ana are mere";
char dest[20]={0};
```

*//copiere normala (stanga-dreapta)*

```
MyCopy(dest, src);
    cout << "Sirul sursa este: " << src << "\n";
    cout << "Sirul destinatie este: " << dest << "\n";
```

*//copiere inversa (dreapta-stanga)*

```
MyCopy(dest, src,1);
    cout << "Sirul sursa este: " << src << "\n";
    cout << "Sirul destinatie este: " << dest << "\n";
```



# Argumente cu valori predefinite

---

- ▶ Compilatorul recunoaște argumentele prin poziția lor.
- ▶ La apelul funcției pot fi lăsate nespecificate doar ultimele argumente, pentru care se iau valorile predefinite.
- ▶ Declararea unui argument ca având valoare implicită se face o singura dată.
- ▶ Concluzie:
  - ▶ La proiectarea unei funcții cu argumente predefinite, este bine să ordonăm argumentele în cadrul listei de argumente astfel: mai întâi lista parametrilor fischi (cei care se modifică cel mai des la apelul funcției), urmată de lista parametrilor cu valori implicate (cei care se modifică cel mai rar la apelul funcției)



# Supraîncărcarea funcțiilor

---

- ▶ În C++ pot exista mai multe funcții cu același nume, dar cu liste diferite de argumente. Aceste funcții sunt **supraîncărcate**.
- ▶ Supraîncărcarea se utilizează pentru a da același nume tuturor funcțiilor care fac același tip de operație.
- ▶ Numele unei funcții și ansamblul argumentelor sale, ca număr și tipuri, se numește **semnătura** acelei funcții. Se observă că funcțiile supraîncărcate au semnături diferite.



# Supraîncărcarea funcțiilor cu un singur argument

---

## ► Ex.1:

```
short int MyAbs(short int x)
{
    return (x<0)? -x:x;
}
```

```
Long MyAbs(Long x)
{
    return (x<0)? -x:x;
}

double MyAbs(double x)
{
    return (x<0)? -x:x;
}
```

//Functii pentru valoare  
absoluta in C\_ANSI  
*int abs(int);  
Long Labs(Long);  
double fabs(double);*

//Apel  
*int a = -3;  
Long c = -5;  
double f = -23.2;*  
  
*MyAbs(a) ;  
MyAbs(c) ;  
MyAbs(f) ;*



# Supraîncărcarea funcțiilor cu două argumente

## ► Ex. 2: Ridicarea la putere

//Declaratii

```
int MyPow( int n, int p );
float MyPow( float n, int p );
double MyPow( double n, int p );
```

//Definitii

```
int MyPow( int n, int p ){
    return (p < 0)? 0:((p == 0)? 1: n * MyPow(n, p-1));
}
```

```
float MyPow( float n, int p ){
    return (p < 0)? 0:((p == 0)? 1: n * MyPow(n, p-1));
}
```

```
double MyPow( double n, int p ){
    return (p < 0)? 0:((p == 0)? 1: n * MyPow(n, p-1));
}
```

//Apel

```
int x = 3, p = 3;
float y = 2.0;
double z = 3.3;
```

```
MyPow( x, p );
```

```
MyPow( y, p );
```

```
MyPow( z, p );
```

# Supraîncărcarea funcțiilor

---

- ▶ Declararea functiilor supraincarcate respecta regulile de domeniu cunoscute. Astfel, daca se declara un prototip intr-un bloc, domeniul lui va fi acel bloc, ascunzand celelalte variante de prototipuri

```
//Declaratii  
int Fct(char);  
int Fct(char*);  
void Functie(void);
```



# Supraîncărcarea funcțiilor

```
int Fct(char a) {
    cout << "int Fct(char) returneaza " << a << endl;
    return a;
}
int Fct(char *s){
    printf( "int Fct(char*s) returneaza adresa sir %s: -> %X\n", s, s);
    return 1;
}
void Fct(Long nr){
    cout << "void Fct(Long nr) afiseaza nr.Long=" << nr << endl;
}
void Functie(void) {
    void Fct(Long);
    cout << "Apel al functiei void Functie(void)" << endl;
    printf("\t"); Fct('a');
    printf("\t"); Fct(678);                                //Apel
}
                                            Fct('A');
                                            Fct("Ana are mere");
                                            Functie();
```

# Supraîncărcarea funcțiilor cu argumente implicate

```
//Declaratii                                //Apel
void Function( void );
void Function( int, int = 0 );
void Function( char, char = 'a', int = 0 );
void Function( void )
{
    cout << "Apel void Function(void)" << endl;
}

void Function( int a, int b )
{
    cout << "Apel void Function( int, int = 0 )" << endl;
}

void Function( char a, char b, int c )
{
    cout << "Apel void Function( char, char = 'a', int = 0 )" << endl;
}
```



# Supraîncărcarea funcțiilor

---

- ▶ Compilatorul C++ selectează funcția corectă prin compararea tipurilor argumentelor din apel cu cele din declarație.
- ▶ Când facem supraîncărcarea funcțiilor, trebuie să avem grijă ca numărul și/sau tipul argumentelor versiunilor supraîncărcate să fie diferite. Nu se pot face supraîncărcări dacă listele de argumente sunt identice:

```
int Compute( int x );  
double Compute( int x );
```

- ▶ O astfel de supraîncărcare (numai cu valoarea returnată diferită) este ambiguă. Compilatorul nu are posibilitatea să discearnă care variantă este corectă și semnalează eroare.



# Constanțe

---

- ▶ C++ oferă conceptul de constantă definită de utilizator, **const**, pentru a exprima noțiunea că o valoare nu se schimbă direct.
- ▶ Contextele în care sunt utile:
  - ▶ Variabile care nu își modifică valoarea după inițializare
  - ▶ Constantele simbolice permit o întreținere mai ușoară a programului
  - ▶ Majoritatea pointerilor sunt folosiți pentru a se citi o valoare(adresă) și nu pentru a fi scriși
  - ▶ Majoritatea parametrilor unei funcții sunt citiți nu scriși
- ▶ Cuvântul **const** poate fi adăugat unei declarații pentru a face obiectul respectiv constant și trebuie inițializat.

```
const int model = 90;           //model este const
const int v[] = { 1, 2, 3, 4 }; //v[i] este const
const int x;                  //error: nu există inițializare
```

---



# Constante

---

- ▶ Declararea unui obiect **const** va asigura ca valoarea acestuia nu se modifică în cadrul blocului.

```
void f( void )
{
    model = 200;    //eroare
    v[2]++;        //eroare
}
```

- ▶ **Obs.** Cuvântul cheie **const** modifică un tip în sensul că restricționează modurile în care un obiect poate fi folosit.

```
void g(const X* p)
{
    //(*p) nu poate fi modificat
}
void h(void)
{
    X val; //val poate fi modificat
    g(&val);
}
```

# Constante

---

- ▶ Funcție de compilator, se poate aloca spațiu pentru variabila respectivă sau nu.

```
const int c1 = 1;  
const int c2 = 2;  
const int c3 = my_f(3); //valoarea lui c3 nu este cunoscuta la compilare  
extern const int c4; //valoarea lui c4 nu este cunoscuta la compilare  
const int *p = &c2; //este necesar sa se aloca spatiu pentru c2
```

- ▶ Pentru vectori se alocă memorie deoarece compilatorul nu poate să știe ce element al vectorului este folosit într-o expresie.
- ▶ Constantele sunt utilizate în mod frecvent ca limite pentru vectori sau *case labels*:



# Constante

---

```
const int a = 42;
const int b = 99;
const int max = 128;
int v[max];
void f(int i)
{
    switch(i)
    {
        case a:
            //...
        case b:
            //...
    }
}
```

- ▶ Enumeratori sunt o alternativă pentru astfel de cazuri



# Pointeri și constante

- ▶ Prin folosirea unui pointer sunt implicate două obiecte: pointerul însuși și obiectul pointat
- ▶ Precedarea declarației unui pointer cu un *const* face obiectul (nu pointerul) constant.
- ▶ Declararea unui pointer constant presupune utilizarea *\*const* și nu doar *\**.

```
void f1(char* p){  
    char s[] = "Gica";  
    const char* pc = s;          //pointer catre constant  
    pc[3] = 'g';                //error: pc pointeaza catre constant  
    pc = p;                     //ok  
    char *const cp = s;          //pointer constant  
    cp[3] = 'a';                //ok  
    cp = p;                     //error: cp este constant  
    const char *const cpc = s; //pointer constant catre constant  
    cpc[3] = 'a';              //error: cpc pointeaza catre constant  
    cpc = p;                   //error: cpc este constant  
}
```

# Pointeri și constante

---

- ▶ Operatorul care face un pointer constant este `*const`. Nu există declaratorul `const*`. Dacă este întâlnit este considerat că face parte din tipul de bază.

```
char *const cp; //pointer constant catre char  
char const* pc; //pointer catre char constant  
const char* pc2; //pointer catre char constant
```

- ▶ Modificatorul `const` se folosește frecvent la funcții, la declararea parametrilor formali de tip pointer sau referințe, pentru a interzice funcțiilor respective modificarea datelor spre care pointează parametrii respectivi.

```
char* strcpy(char* p, const char* q); //nu poate modifica (*q)
```

- ▶ Protecția datelor cu ajutorul `const` nu este totală pentru toate compilatoarele



# PROGRAMARE ORIENTATĂ OBIECTE

Curs 3  
*Facilități ale limbajului C++  
Principii POO*

*Pointeri către un obiect de tip necunoscut*

*Pointer nul*

*Struct/class layout*

*Forward declaration*

*Câmpuri de biți*

*Functii inline*

*Class*

*Constructori de initializare*

*Destructor*

# *Pointeri către un obiect de tip necunoscut*

---

- ▶ Într-o secvență de cod *low level* este necesar să transmită adresa unei locații de memorie fără să cunoască tipul datei/datelor stocate în acea locație.
- ▶ Pentru asta se folosește pointerul către tipul de date necunoscut **void\***
- ▶ Un pointer către orice tip de dată poate fi atribuit unei variabile de tip **void\***.
- ▶ Un pointer pe funcție sau la un membru al unei clase nu poate fi atribuit unei variabile de tip **void\***.
- ▶ Un pointer de tip **void\*** poate fi atribuit/comparat unui/cu alt pointer de tip **void\***.
- ▶ Un pointer de tip **void\*** poate fi convertit explicit la orice tip de date.



# *Pointeri către un obiect de tip necunoscut*

```
int f(int *p)
{
    void *pv = p; //conversie implicita
    *pv; //eroare, un pointer pe void* nu poate fi dereferentiat
    ++pv; //eroare, un pointer pe void* nu poate fi incrementat

    int *pi = (int*)pv; //conversie explicita la int*
}
```

## ► Sfaturi:

- ▶ A nu se utiliza pointeri convertiți explicit către un alt tip de date decât cel inițial
- ▶ A se utiliza pointeri la void\* doar pentru a fi:
  - ▶ transmiși ca parametru funcțiilor iar tip acestora nu este necesar a fi cunoscut;
  - ▶ returnat de funcții.
- ▶ A se converti explicit pointierii la void\* atunci când sunt folosiți;



# Pointeri către un obiect de tip necunoscut

---

- ▶ Având în vedere că pointeri la **void\*** sunt folosiți în programarea *low level*, acolo unde resursele hardware sunt manipulate, apariția unor astfel de pointeri la nivel *high level* trebuie privită cu suspiciune deoarece cu siguranță se datorează unor erori de proiectare.

## Pointer nul

- ▶ **nullptr** reprezintă un pointer nul, un pointer ce nu pointează către nici un obiect.
- ▶ Poate fi atribuit oricărui tip de pointer

```
int *pi = nullptr;
double *pd = nullptr;
int i = nullptr; // eroare : i nu este un pointer
```



# Pointer nul

---

- ▶ Înainte de a fi introdus **nullptr**, era folosit zero (**0**) ca notație pentru pointer nul. `int *x = 0;`
- ▶ Nici un obiect nu este alocat la adresa **0**, iar zero (**0**) este cea mai comună reprezentare a **nullptr**.
- ▶ Zero (**0**) este un int. Totuși în conversiile standard **0** este folosit ca o constantă a unui pointer sau pointer la membru.
- ▶ O altă reprezentare a unui pointer nul a fost macrodefiniția **NULL**. `int *p = NULL;`
- ▶ Totuși există diferențe în definiția **NULL** în diferite implementări:

```
#define NULL 0  
#define NULL 0L  
#define NULL (void*)0 /*C style*/
```

- ▶ `int *p = NULL; //eroare: nu poate fi atribuit un void* la un int*`

# Struct/class layout

- ▶ O structură/clasă își păstrează membri în ordinea în care au fost declarați:

```
struct DataCalendaristica
{
    char zi; // [1:31]
    int an;
    char Luna; // [1:12]
};
```

- ▶ Aranjarea în memorie a membrilor ar putea fi aceasta:



- ▶ Totuși, dimensiunea unui obiect nu este suma dimensiunilor membrilor



# Struct/class layout

- ▶ Ex: pe 32 de biți dimensiunea unui obiect de tip ***DataCalendaristica*** este de 12 octeți și nu de 6.



- ▶ Optimizare:

```
struct Data
{
    int an;
    char zi; // [1:31]
    char Luna; // [1:12]
};
```



# Definiția și utilizarea unei structuri

---

- ▶ Un nume (identifier) devine accesibil imediat ce a fost scris și nu după declarare completă:

```
struct Link
{
    Link* previous;
    Link* successor;
};
```

- ▶ Totuși nu se poate declara un obiect de un anumit tip dacă declarația tipului respectiv nu este finalizată

```
struct Link
{
    Link data;
};
```

- ▶ Eroare, deoarece compilatorul nu poate determina dimensiunea tipului **Link**.



# Forward declaration

---

- ▶ Referiri simultane:

```
struct List; // declararea numelui unei structuri: Va fi  
             //definita mai tarziu
```

```
struct Link  
{  
    Link* prev;  
    Link* next;  
    List* member_of;  
    int data;  
};
```

```
struct List  
{  
    Link* head;  
};
```

- ▶ Numele unei structuri poate fi folosit înainte de a fi definită structura, dar nu poate instanția un obiect.

# Câmpuri de biți

---

- ▶ Tipul de date **char** este cel mai mic obiect ce poate fi alocat independent
- ▶ Se poate considera risipă de spațiu utilizarea unui **char** în declararea unei variabile binare (singurile valori pe care le poate avea sunt 0 și 1)
- ▶ Există posibilitatea de împacheta astfel de variabile ca fiind câmpuri de biți într-o structură/clasă.
- ▶ Un membru este definit ca fiind un câmp de biți prin specificarea numărului de biți pe care îl ocupă
- ▶ Un câmp de biți trebuie să fie de tip **int** sau **enum**
- ▶ Nu se poate obține adresa unui câmp de biți
- ▶ Sunt permise câmpurile fără nume



# Câmpuri de biți

---

```
struct student
{
    unsigned int nrMatricol;
    int : 8;
    bool promovat : 1; // [0..1]
    bool camin : 1; // [0..1]
    unsigned int grupa : 11; // [1101..1410]
    unsigned int nota : 4; // [1..10]
    unsigned int vârsta : 7; // [1..127]
};
```

- ▶ Utilizarea câmpurilor de biți pentru împachetarea variabilelor salvează un căți va octeți în detrimentul dimensiunii și vitezei de execuție a unui program.
- ▶ Pot fi o alternativă convenabilă pentru lucru pe biți.



# Functii inline

```
inline int factorial(int n)
{
    return (n<2) ? 1 : n*fac(n-1);
}
```

- ▶ Specificatorul `inline` sugerează compilatorului să nu apeleze funcția ci să:
  - ▶ înlocuiască apelul cu secvența de cod conținută de funcție
  - ▶ ori să genereze cod
- ▶ Pentru un apel de genul ***factorial(6)*** un compilator „inteligent” va genera constanta ***720***
- ▶ Gradul de „inteligentă” al unui compilator nu poate fi legiferat, motiv pentru care un compilator poate genera constanta ***720***, un altul ***6\*factorial(5)***, un simplu apel de funcție ***factorial(6)***
- ▶ Pentru a avea garanția că o valoare este calculată la compilare se va declara funcția ca fiind ***constexpr***
- ▶ Pentru a avea garanția că o funcție este ***inline*** se va defini (nu doar declara) în domeniul în care este utilizată.

# Tehnica Programării Orientate pe Obiecte

## Prezentare generală

---

- ▶ Orice structură poate fi definită printr-o clasă în care toate elementele sunt publice.

```
class Durata
{
    private: //declara'ie implicita
        int ora, min, sec;
    public:
        void Seteaza(int, int, int);
        void Scrie(void);
        void Citeste(void);
        void Aduna(void);
        int EsteEgalăCu(Durata);
};
```

- ▶ Tipurile abstracte de date se definesc folosind noțiunea (conceptul) de clasă (c++ **class**).



# Tehnica Programării Orientate pe Obiecte

## Prezentare generală

---

- ▶ **O clasă** definește atât reprezentarea datelor cât și funcțiile care au acces la date și le pot prelucra.
- ▶ Utilizatorii clasei au acces numai la componentele **publice**
- ▶ Datorită acestei încapsulări, componentele unei clase sunt de 3 tipuri, funcție de **nivele de protecție** specificate prin intermediul etichetelor **private**, **protected** și **public**.



# Tehnica Programării Orientate pe Obiecte

## Prezentare generală

- ▶ Într-o definiție de clasă pot exista mai multe secțiuni cu același nivel de protecție

```
class X
{
public:
    //-----membri publici
private:
    //-----membri privati
public:
    //-----membri publici
};
```

- ▶ Ideea de protejare deosebește de fapt structura de clasă.

```
struct X
{
    //-----
};
```

```
class X
{
public:
    //-----
};
```



# Tehnica Programării Orientate pe Obiecte

## Prezentare generală

---

- ▶ Asociind unei clase valori concrete pentru membrii săi se obține un **obiect**.
- ▶ Un obiect este o **instanțiere** (sau instanță) a clasei care definește tipul său.
- ▶ Variabilele membre definesc **proprietățile** obiectului, în timp ce funcțiile definesc **comportamentul**.
- ▶ Extinderea valabilității operatorilor existenți în limbaj se numește **supraîncărcarea operatorilor**
- ▶ Variabilele membre se numesc scurt membri iar funcțiile membre metode



# Tehnica Programării Orientate pe Obiecte

## Prezentare generală

---

- ▶ Clasele pot conține:
  - ▶ variabile de orice tip
  - ▶ funcții de orice tip
  - ▶ metode de tip constructori – initializează membri unei clase
  - ▶ metodă de tip constructor de copiere – implementează operația de copiere a unui obiect
  - ▶ metodă de tip destructor – realizează distrugerea obiectului (în special eliberarea memoriei alocate dinamic)
  - ▶ definiția unor noi tipuri de date
  - ▶ funcții operator
  - ▶ etc.



# Tehnica Programării Orientate pe Obiecte

## Prezentare generală

---

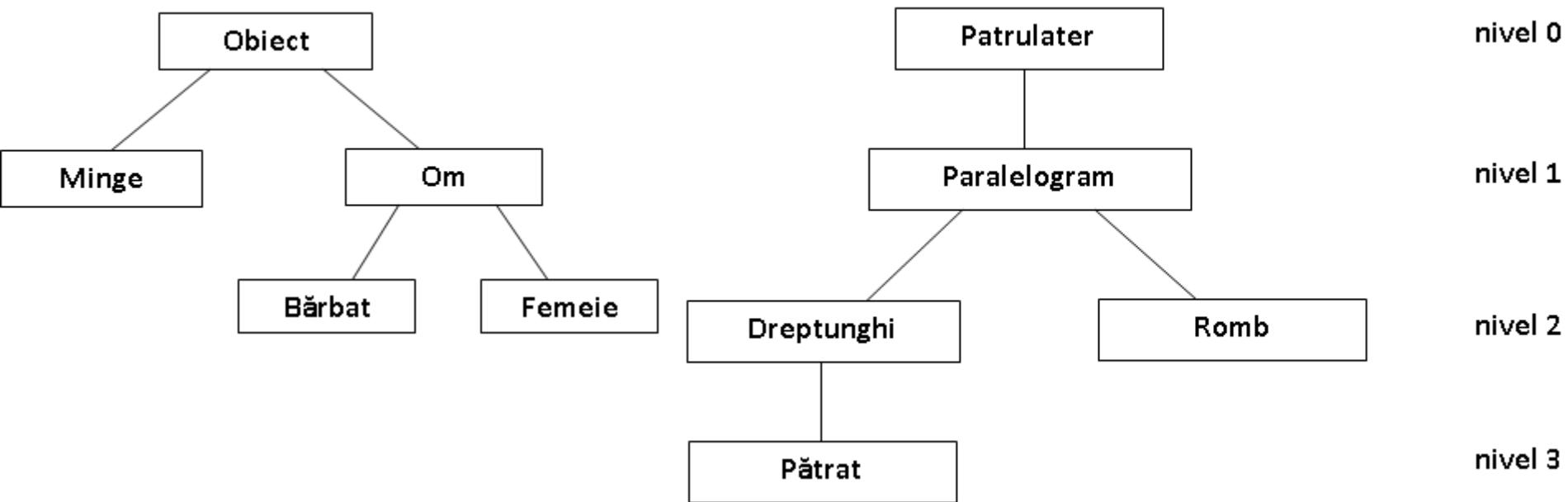
- ▶ Când se lucrează cu un tip abstract de date, el va fi descris prin prezentarea interfeței și a implementării sale.
- ▶ **Interfața** este reprezentată de **membrii publici** ai clasei.
- ▶ De obicei, diferite tipuri de date abstracte înrudite au elemente comune.
- ▶ Precizarea acestor elemente comune duce la o “ierarhizare” arborescentă a tipurilor abstracte de date.
- ▶ Prin ierarhizarea claselor, componentele unei clase rămân valabile și pentru clasele de nivel ierarhic inferior. Această mecanism se numește **moștenire**.
- ▶ Folosind moștenirea, se pot obține clase noi, prin adăugarea de componente noi la o clasă deja existentă, numite clase derivate



# Tehnica Programării Orientate pe Obiecte

## Prezentare generală

- ▶ Clasa din care se obțin clasele derivate se numește **clăsă de bază**.



# Tehnica Programării Orientate pe Obiecte

## Prezentare generală

---

- ▶ Dintr-un tip abstract de date se poate crea un obiect. Cum un tip abstract de date este o colecție de variabile rezultă:
  - ▶ Variabilele din tipul abstract de dată reprezintă informația care dă proprietățile tipului
  - ▶ Funcțiile din tipul abstract de dată reprezintă operațiile ce pot fi făcute cu datele aceluui tip și reprezintă comportamentul tipului
- ▶ Clasele abstrakte nu se pot instanția în mod direct deoarece nu există suficiente informații pentru a construi acele instanțieri
- ▶ Un obiect poate fi descris folosind proprietățile altui obiect la care se adaugă proprietăți suplimentare. Astfel se pot defini clase grupate ierarhic, procedeul fiind numit moștenire
- ▶ Polimorfism – o anumită operatie (funcție) poate funcționa diferit cu obiecte diferite, dar înrudite



# Tehnica Programării Orientate pe Obiecte

## Moștenire, polimorfism

---

- ▶ Un obiect poate fi descris folosind proprietățile altui obiect la care se adaugă proprietăți suplimentare. Astfel se pot defini clase grupate ierarhic, procedeul fiind numit moștenire
- ▶ Polimorfism – o anumită operație (funcție) poate funcționa diferit cu obiecte diferite, dar înrudite



# Programarea orientată pe obiecte

---

- ▶ Scopul limbajului C++ este acela de a furniza programatorului o unealtă pentru crearea de noi tipuri de date.
- ▶ În plus, clasele derivate și şabloanele (templates), furnizează căi, metode de organizare a claselor „înrudite”, astfel încât programatorul să disponă de avantajele oferite.
- ▶ Un tip de date este o reprezentare concretă a unui concept.
- ▶ Ex: tipul de date predefinit ***float*** cu operatorii ***+, -, \****, etc. asigură o aproximare concretă a conceptului matematic a unui număr real.
- ▶ O clasă este un tip definit de utilizator. Se proiectează un nou tip pentru a furniza un concept care nu se regăsește în lista tipurilor predefinite. Ex: tipul de date complex.



# Programarea orientată pe obiecte

---

- ▶ Un program care furnizează tipuri de date cât mai apropiate de conceptele aplicației tinde să fie mai ușor de înțeles și ușor de modificat comparativ cu un program care nu face acest lucru.
- ▶ Un set foarte bine ales de tipuri definite de utilizator face programul mult mai concis.



# Clasa – tip definit de utilizator

---

## ▶ Funcții membre

- ▶ Fie implementarea conceptului de dată calendaristică utilizând *struct* pentru a defini reprezentarea *Date* (dată calendaristică) și un set de funcții pentru manipularea variabilelor de acest tip.

```
struct Date
{
    int d, m, y;
};

void InitData(Date &d, int, int, int); //initializare
void AddYear(Date &d, int n);
void AddMonth(Date &d, int n);
void AddDay(Date &d, int n);
```

- ▶ Nu există o corespondență explicită între tipul Date și aceste funcții.



# Clasa – tip definit de utilizator

---

- ▶ Această corespondență poate fi stabilită declarând funcțiile ca membre ale structurii:

```
struct Date
{
    int d, m, y;
    void Init(int dd,int mm,int yy);
    void AddYear(int n);
    void AddMonth(int n);
    void AddDay(int n);
};
```

- ▶ Funcțiile declarate în definiția unei structuri/clase sunt numite funcții membre și pot fi invocate doar de tipul respectiv sau apropiat utilizând sintaxa standard de acces la membrii unei structuri



# Clasa – tip definit de utilizator

```
Date myBirthday;  
void f(void)  
{  
    Date today;  
    today.Init(16, 10, 1996);  
    myBirthday.Init(30, 12, 1950);  
    Date tomorrow = today;  
    tomorrow.AddDay(1);  
}
```

- ▶ Deoarece diverse structuri pot avea funcții membre cu același nume, trebuie specificat numele unei structuri când este definită o funcție

```
void Date::Init(int dd, int mm, int yy)  
{  
    d = dd;  
    m = mm;  
    y = yy;  
}
```

# Clasa – tip definit de utilizator

---

- ▶ Într-o funcție membră, numele membrilor pot fi folosiți fără o referință explicită la un obiect.
- ▶ Construcția:

```
class X { ... };
```

este numită *definirea clasei* deoarece definește un nou tip de date. Din motive istorice definirea unei clase mai este numită și *declararea clasei*. Din acest motiv definiția clasei poate fi replicată în diverse surse utilizând `#include` fără să fie încălcată regula definiției unice.



# Clasa – controlul accesului

- ▶ Declarația structurii *Date* de mai înainte pune la dispoziție un set de funcții pentru manipularea membrilor. Totuși nu specifică faptul că sunt singurele funcții care au acces direct la membri. Această restricție poate fi exprimată folosind **class** în loc de **struct**:

```
class Date
{
    int d, m, y;
public:
    void Init(int dd, int mm, int yy);
    void AddYear(int n);
    void AddMonth(int n);
    void AddDay(int n);
};
```

- ▶ Cuvântul cheie **public** separă corpul clasei în două părți. Membri, *privăți*, din prima parte pot fi folosiți doar de funcțiile membre. Membri publici constituie interfața publică a obiectelor clasei.

# Clasa – controlul accesului

---

- ▶ Structura este simplu o clasă a cărei membri sunt implicit publici.
- ▶ Funcțiile membre unei clase se definesc în același mod ca la o structură:

```
inline void Date::AddYear(int n)
{
    y += n;
}
```

- ▶ Totuși funcții nemembre nu pot accesa membri privați:

```
void Timewarp(Date &d)
{
    d.y= 200; //eroare: Date::y este privat
}
```

- ▶ Avantaje: acces controlat, actualizare prin intermediul interfaței



# Clasa – specificatori de acces

---

- ▶ Modifică drepturile de acces către membrii clasei
  - ▶ **private**: membrii privați pot fi accesați numai de membrii aceleiași clase sau *de membrii claselor prietene*
  - ▶ **protected**: membrii protejați sunt accesibili în cadrul aceleiași clase sau *dintr-o clasă prietenă, dar și din clasele derivate din acestea*
  - ▶ **public**: membrii publici sunt accesibili de oriunde din domeniul de vizibilitate al obiectului de tip clasă
- ▶ În cadrul unei clase, specificatorul de acces implicit este *private*
- ▶ În cadrul unei structuri, specificatorul de acces implicit este *public*



# Clasa – constructori

---

- ▶ Utilizarea funcției *Init()* pentru a initializa obiectele clasei nu este elegantă și generatoare de erori.
- ▶ O abordare potrivită este de a permite programatorului să declare o funcție al cărei scop explicit este acela de inițializare. Deoarece astfel de funcții se construiesc obiecte au fost numite *constructori*.
- ▶ Un constructor are același nume ca și clasa:

```
class Date
{
    //...
    Date(int, int, int); //constructor
};
```



# Clasa – constructori

---

```
Date today = Date(8, 11, 2012);
Date Xmas(25, 12, 2012);      //formă abreviată
Date MyBirthday;            //eroare: lipsa parametri de initializare
Date Release1_0(10, 12); //eroare: lipsește al treilea argument
```

- ▶ Este de dorit a se furniza diferite moduri de inițializare a unui obiect

```
class Date
{
    int d, m, y;
public:
    Date(int, int, int); //day, month, year
    Date(int ,int);     //day, month, anul curent
    Date(int);          //day, Luna si anul curent
    Date(void);         //valori implicite
    Date(const char*);  //data sir de caractere
};
```



# Clasa – constructori

---

- ▶ Regulile de supraîncărcare a funcțiilor se aplică și în cazul constructorilor

```
Date today(4);  
Date july4(„1 Dec, 2012”);  
Date guy(„8Nov”);  
Date now;
```

- ▶ Constructori pot avea argumente predefinite

```
class Date  
{  
    int d, m, y;  
public:  
    Date(int dd = 0, int mm = 0, int yy = 0);  
    Date(void):d(0), m(0), y(0){};  
};
```



# Constructori și destructori

---

## ▶ Constructorul:

- ▶ metodă specială a unei clase
- ▶ are același nume ca și clasa
- ▶ nu are un tip de return
- ▶ este apelat automat la declararea obiectului

## ▶ Cu ajutorul constructorului:

- ▶ se creează un obiect pentru care se face alocarea spațiului de memorie necesar
- ▶ se pot inițializa variabilele membre ale clasei

## ▶ Constructor implicit

- ▶ Se apelează atunci când nu există nicio declarație a unui constructor în clasă



# Constructori

---

- ▶ Alte tipuri de constructori:
  - ▶ Constructor fără listă de argumente
  - ▶ Constructor de inițializare (cu listă de argumente)
  - ▶ Constructor de copiere
- ▶ Dacă se definește cel puțin un constructor, nu se va mai genera constructorul implicit
- ▶ Pot exista mai mulți constructori ai aceleiași clasei
- ▶ Toate declarațiile obiectelor trebuie să respecte un model din mulțimea constructorilor clasei



# Clasa - Destructor

---

- ▶ Ajută la eliberarea zonelor de memorie
- ▶ Are rol opus constructorului
- ▶ Nu are tip de return
- ▶ Are numele clasei precedat de caracterul ~
- ▶ Nu primește nici un parametru
  - ▶ O clasă are un singur destructor!
- ▶ Se apelează automat când domeniul de vizibilitate a obiectului s-a terminat
- ▶ Se recomandă să se realizeze atunci când clasa conține pointeri care sunt alocați dinamic în constructor sau în alte metode prezente în clasă

*~Date(**void**);*



---

Vă mulțumesc !



# PROGRAMARE ORIENTATĂ OBIECT

Curs 4

*Programarea orientată pe obiecte în C++*

Constructor de copiere

Constructor explicit

Funcții membre inline

Membri statici

This

Operatori new și delete

Funcții membre constante

Prieteni

# Constructorul de copiere

---

- ▶ Implicit obiectele pot fi copiate. În particular, un obiect poate fi inițializat cu copia unui alt obiect din aceeași clasă

*Date d = today;*

- ▶ Copierea implicită a obiectelor este o copiere membru cu membru.
- ▶ Constructorul de copiere este un constructor special al clasei
- ▶ Este utilizat pentru a copia un obiect existent de tipul clasei
- ▶ Are un singur argument, o referință către obiectul ce va fi copiat
- ▶ Forma generală este:

*Date(**const** Date& obj);*  
*Date(Date& obj);*

- ▶ În cazul în care o clasă conține variabile de tip pointer,
- ▶ trebuie făcută alocare de memorie

# Constructorul de copiere

```
class Data
{
    unsigned char zi, Luna;
    unsigned int an;
public:
    void Afisare(void){ cout <<“Data:
                            “<<an<<“.”<<Luna<<“.”<<zi<<endl;};
    Data(unsigned char z, unsigned char l, unsigned int a);
    Data(void);
    Data(Data &);
    ~Data();
};

Data::Data(Data & d)
{
    zi=d.zi;
    Luna=d.Luna;
    an=d.an;
}
```

```
int main(void)
{
    Data d3(d2);
    d3.afisare();
    return 0;
}
```

# Constructorul de copiere

---

- ▶ Cazuri în care se apelează constructorul de copiere:

- ▶ La inițializarea unui obiect nou creat cu un alt obiect

*Data d2 = d1;*

- ▶ La transferul parametrilor unei funcții prin valoare
  - ▶ Funcție ce returnează un obiect
- ▶ În clasă se adaugă:

*public:*

*Data Increment();*

- ▶ Definiția funcției:

```
Data Data::Increment (void)
{
    Data temp;
    temp.zi = zi + 1;
    temp.Luna = Luna + 1;
    temp.an = an;
    cout << "inainte de return" << endl;
    return temp;
}
```

# Constructorul de copiere

---

```
Data::Data(void)
{
    cout << "S-a apelat constr. fara argumente" << endl;
}

Data::Data(unsigned char z, unsigned char l, int a)
{
    cout << "S-a apelat constr. cu Lista de argumente" << endl;
    zi = z;
    Luna = l;
    an = a;
}

Data::Data(Data &d)
{
    cout << "S-a apelat constr. de copiere" << endl;
    zi=d.zi;
    Luna=d.Luna;
    an=d.an;
}
```



# Constructorul de copiere

```
int main(void)
{
    cout<<"d1: ";
    Data d1 (9,1,2012);

    cout<<"d2: ";
    Data d2;
    d2 = d1.Increment();

    cout << "dupa apelul functiei increment" << endl;

    d2=d1;
    return 0;
}
```

*d1: S-a apelat constr. cu lista de argumente*

*d2: S-a apelat constr. fara argumente*

*S-a apelat constr. fara argumente*

*inainte de return*

***S-a apelat constr. de copiere***

►*dupa apelul functiei increment*

# Constructorul de copiere

---

```
class Persoana
{
    char * nume;
    int varsta;
public:
    Persoana(void);
    Persoana (char *sName, int v);

    ~Persoana (void);
    void Afiseaza(void);
    Persoana SchimbaVarsta(int);
};
```



# Constructorul de copiere

---

```
#include <iostream>
#include "header.h"

using namespace std;

Persoana::Persoana(void)
{
    nume = new char [1];
    varsta = 0;
}
```

```
Persoana::Persoana(char * n, int v)
{
    nume = new char [strlen (n) + 1];
    strcpy_s (nume, strlen (n) + 1, n);
    varsta = v;
}
```



# Constructorul de copiere

---

```
Persoana::~Persoana (void)
{
    if (nume)
        delete [] nume;
}

void Persoana::Afiseaza(void)
{
    cout << "Nume=" << nume << " varsta=" << varsta<< endl;
}

Persoana Persoana::SchimbaVarsta(int n)
{
    Persoana temp (nume, varsta);
    varsta = n;
    return temp;
}
```



# Constructorul de copiere

---

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

int main (void)
{
    Persoana p1 ("Pavel",22);
    p1.Afiseaza();

    Persoana p2 ("Ana",23);

    Persoana p3 = p2.SchimbaVarsta(25);

    p2.Afiseaza();

    p3.Afiseaza();
    return 0;
}
```

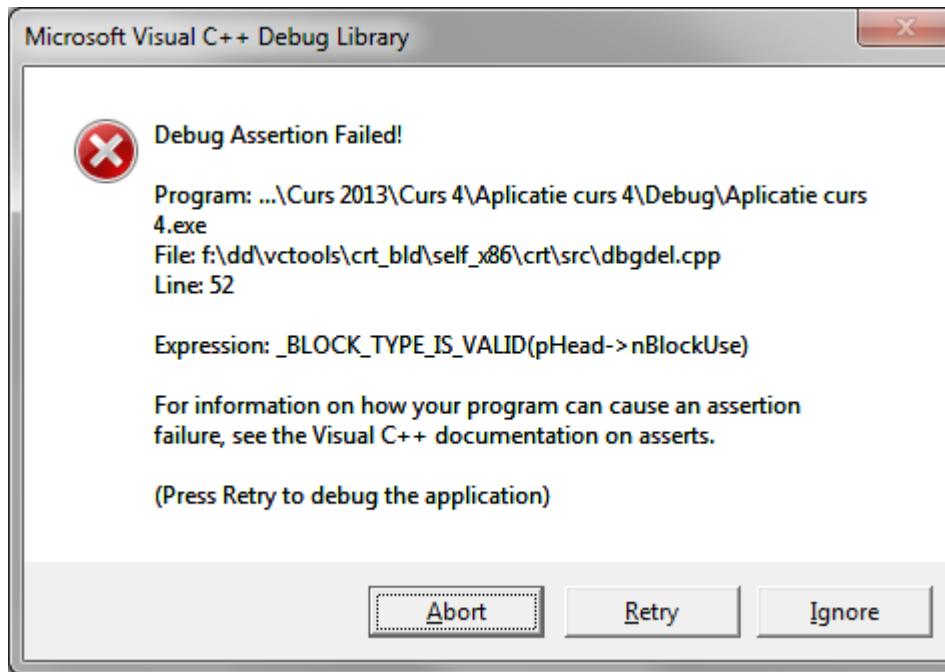


# Constructorul de copiere

Nume=Pavel varsta=22

Nume=Ana varsta=25

Nume=TTTTTTTETl2TT varsta=23



# Constructorul de copiere

---

- ▶ De ce ? Pentru ca lipsește constructorul de copiere!
- ▶ Se adauga în clasa un constructor de copiere:  
*Persoana (Persoana & p);*
- ▶ Cu definiția

```
Persoana::Persoana (Persoana & p)
{
    nume = new char [strlen (p.nume) + 1];
    strcpy_s (nume, strlen (p.nume) + 1, p.nume);
    varsta = p.varsta;
}
```



# Constructor explicit

---

- ▶ Este constructorul ce nu poate lua parte la nici o conversie implicită
- ▶ Se declară folosind cuvântul cheie **explicit**
- ▶ Numai constructorii cu un singur argument pot lua parte într-o conversie implicită

```
class A
{
public:
    explicit A(int);
};
```

```
A::A(int)
{}
```

```
A a1 = 37;
```

*error C2440: 'initializing' : cannot convert from 'int' to 'A'*



# Funcții inline

---

- ▶ Pentru funcții membru, dacă corpul funcției este definit în cadrul clasei, atunci acele funcții sunt implicit inline, cu condiția să respecte regulile de scriere a unor funcții inline

```
class Persoana
{
    char * nume;
    int varsta;
public:
    //...
    char*GetName(void)
    {
        return nume;
    }
    int GetVarsta(void);
    //...
};

inline int Persoana::GetVarsta(void){ return varsta;}
```

# Membri statici

---

- ▶ O variabilă care este membră a unei clase dar nu membră a unui obiect al clasei respective se numește membru ***static***.
- ▶ Similar, o funcție care necesită accesul la membri unei clase, fără să fie apelată pentru un obiect oarecare, se numește funcție membră *statică*.



# Membri statici

---

```
class Data
{
    unsigned char zi, Luna;
    unsigned int an;
    static Data dataImplicita;
public:
    Data(unsigned char zz = 0, unsigned char ll = 0,
          unsigned int aa = 0);
    static void SetImplicitData(unsigned char,
                               unsigned char, unsigned int);
};

Data::Data(unsigned char zz, unsigned char ll, unsigned int aa)
{
    zi = zz ? zz : dataImplicita.zi;
    Luna = ll ? ll : dataImplicita.Luna;
    an = aa ? aa : dataImplicita.an;
}
```

# Clasa – membri statici

---

- ▶ Membri statici, atât functiile cât și variabilele, trebuie să fie definite (initializați) cumva:

```
Data Data::dataImplicita(16, 12, 1770);
```

//sau

```
void Data::SetImplicitData(unsigned char zz, unsigned char ll,
                           unsigned int aa)
{
    Data::dataImplicita = Data(zz, ll, aa);
}

int main(void)
{
    Data::SetImplicitData(16, 12, 1770);
}
```



# Pointerul **this**

---

- ▶ Pointerul **this** este o variabilă predefinită în C++ accesibilă în corpul oricărei metode *non-static*e din cadrul unei clase
- ▶ Valoarea pointerului este dată de adresa obiectului pentru care s-a apelat o anume metodă non-statică din clasă
- ▶ Este folosit:
  - ▶ Pentru a înlătura ambiguitățile dintre un parametru al unei funcție și o variabilă membră
  - ▶ În cazurile când este necesar un pointer către obiectul pentru care s-a apelat o anumită metodă
- ▶ Compilatorul C++ convertește apelul funcției non-staticice apelate și pune ca prim parametru pointerul **this** .



# Operatorii new, delete, new[] și delete[]

---

- ▶ Operatorii ***new, delete, new[]*** și ***delete[]*** sunt implementați folosind funcțiile:

```
void* operator new(size_t);  
void operator delete(void*);  
void *operator new[](size_t);  
void operator delete[](void*);
```

- ▶ La folosirea operatorului new pentru alocarea unui singur obiect sau variabile, este apelat automat constructorul implicit sau, dacă există constructori declarati, cel ce corespunde listei de argumente
- ▶ La folosirea operatorului delete, este apelat automat destructorul clasei



# Operatorii new, delete

---

```
► int main()
{
    cout<<"d1: ";
    Data d1;

    cout<<"d2: ";
    Data * d2 = new Data;

    cout<<"d3: ";
    Data * d3 = new Data (1,2,2003);
    cout<<"d4: ";
    Data * d4 = new Data (*d3);
    cout<<"d5: ";
    Data * d5 = (Data*)malloc(sizeof(Data));
    cout<<endl;
    cout<<"final:";

    delete d2;
    delete d3;
    delete d4;
    free(d5);
    //delete d5;
    return 0;
} ▶ }
```

# Operatorii new, delete

---

*d1: S-a apelat constr. fara argumente*

*d2: S-a apelat constr. fara argumente*

*d3: S-a apelat constr. cu lista de argumente*

*d4: S-a apelat constr. de copiere*

*d5:*

*final:s-a apelat destructorul*

*s-a apelat destructorul*

*s-a apelat destructorul*



# Alocarea si dealocarea tablourilor

---

- ▶ Pentru alocare de memorie unui tablou se folosește operatorul ***new[]***
  - ▶ Se apeleaza constructorul corespunzător pentru fiecare element (cel implicit sau cel fără argumente dacă este declarat)
- ▶ Pentru dealocare se foloseste operatorul ***delete []***
  - ▶ Se apelează destructorul pentru fiecare element

```
Date *dv;  
dv = new Data [10];
```

```
delete [] dv;
```



# Functii membre constante

---

- ▶ Fie clasa:

```
class Data {  
    unsigned char zi, Luna;  
    unsigned int an;  
  
public:  
    unsigned char Zi(void) const { return zi; }  
    unsigned char Luna(void) const { return Luna; }  
    unsigned int An(void) const;  
    unsigned int AddAn(unsigned int aa) {an+=aa;}  
};
```

- ▶ Prin prezența cuvântului cheie **const** se specifică compilatorului că funcțiile respective nu pot modifica conținutul obiectului *Data*.

```
inline unsigned int Date::An(void) const  
{  
    return an++; //eroare: incercare de modificare a unei variabile  
           //membre intr-o functie constanta  
}
```



# Funcții membre constante

- ▶ O funcție membră constantă definită în afara clasei necesită sufixul **const**.

```
inline unsigned int Date::An(void) const //corect
{
    return y;
}
inline unsigned int Date::year(void) //eroare
{
    return y;
}
```

- ▶ O funcție membră constantă poate fi apelată atât de obiecte **const** cât și de obiecte *non-const*.

```
void f(Data &d, const Data &cd){
    unsigned int i = d.An(); //ok
    d.AddAn(1);             //ok
    unsigned int j = cd.An(); //ok
    cd.AddAn(1);            //eroare: obiect constant
}
}
```

# Functii membre constante

---

- ▶ Pentru un obiect declarat cu ***const*** dacă se apelează o metodă care nu este ***const***, compilatorul dă o eroare de compilare.



# Prietenii

---

- ▶ Ascunderea membrilor unei clase este anulată pentru prietenii acelei clase, care pot fi funcții sau alte clase.
- ▶ O funcție prietenă are acces la toți membrii clasei, inclusiv la cei privați ori protejați.
- ▶ Regula este valabilă pentru toți membrii unei clase prietene
- ▶ Pentru fiecare prieten al unei clase, există o declarație în interiorul clasei, ***friend***.
  - ▶ *Funcții prietene care aparțin domeniului global*
  - ▶ *Funcții prietene care sunt membre ale unei clase*
  - ▶ *Clase prietene*



# Functii prietene globale

```
class Time
{
    int min, sec, hour;
public:
    Time(void) : min(0), sec(0), hour(0){};
    friend int min(Time& t);
    friend int sec(Time& t);
    friend int hour(Time& t);
}

inline int min(Time& t) { return t.min; }
inline int sec(Time& t) { return t.sec; }
inline int hour(Time& t) { return t.hour; }

Time t;
cout << "min: "<<min(t)<<, sec: "<<sec(t)<<, hour:
"<<hour(t)<<endl;
```

- ▶ Funcțiile prietene sunt utile pentru supraîncărcarea operatorilor

# Funcții membru prietene

---

- ▶ O funcție membru a unei clase poate fi funcție prietenă pentru altă clasă

```
class XXX {  
public:  
    void f(void);  
};  
  
class YYY {  
    int nr;  
public:  
    YYY(int n): nr(n){}  
    friend void XXX::f(void);  
};  
  
void XXX::f(void) {  
    YYY ob(10);  
    cout <<ob.nr<<endl;  
}
```



# Clase prietene

---

- ▶ Declarând o clasă prietenă în cadrul unei alte clase, se extinde calitatea de prieten la toate funcțiile membru

```
class YYY
{
    friend class XXX;
}
```



---

Vă mulțumesc !



# PROGRAMARE ORIENTATĂ OBIECTE

Curs 5

*Programarea orientată pe obiecte în C++*

Prietenii  
Pointeri la membri  
Supraîncarcarea operatorilor

# Prietenii

---

- ▶ Ascunderea membrilor unei clase este anulată pentru prietenii acelei clase, care pot fi funcții sau alte clase.
- ▶ O funcție prietenă are acces la toți membrii clasei, inclusiv la cei privați ori protejați.
- ▶ Regula este valabilă pentru toți membrii unei clase prietene
- ▶ Pentru fiecare prieten al unei clase, există o declarație în interiorul clasei, ***friend***.
  - ▶ *Funcții prietene care aparțin domeniului global*
  - ▶ *Funcții prietene care sunt membre ale unei clase*
  - ▶ *Clase prietene*



# Functii prietene globale

```
class Time
{
    int min, sec, hour;
public:
    Time(void) : min(0), sec(0), hour(0){};
    friend int min(Time& t);
    friend int sec(Time& t);
    friend int hour(Time& t);
}

inline int min(Time& t) { return t.min; }
inline int sec(Time& t) { return t.sec; }
inline int hour(Time& t) { return t.hour; }

Time t;
cout << "min: "<<min(t)<<, sec: "<<sec(t)<<, hour:
"<<hour(t)<<endl;
```

- ▶ Funcțiile prietene sunt utile pentru supraîncărcarea operatorilor

# Funcții membru prietene

---

- ▶ O funcție membru a unei clase poate fi funcție prietenă pentru altă clasă

```
class XXX {  
public:  
    void f(void);  
};  
  
class YYY {  
    int nr;  
public:  
    YYY(int n): nr(n){}  
    friend void XXX::f(void);  
};  
  
void XXX::f(void) {  
    YYY ob(10);  
    cout <<ob.nr<<endl;  
}
```



# Clase prietene

---

- ▶ Declarând o clasă prietenă în cadrul unei alte clase, se extinde calitatea de prieten la toate funcțiile membru

```
class YYY  
{  
    friend class XXX;  
}
```



# Pointeri la membri

---

- ▶ Se presupune tipul de date:
  - ▶ `typedef void (*PtrFuncție)(int);`
- ▶ și declarațiile pentru funcțiile *fct()* și *push()*
  - ▶ `void fct(int);`
  - ▶ `void Stiva::Push(int);`
- ▶ Deși aparent cele două funcții au aceeași structură ele nu sunt de același tip, prin urmare:
  - ▶ `PtrFunctie pf1 = fct; //corect`
  - ▶ `PtrFunctie pf2 = &Stiva::Push; //eroare`
- ▶ A doua funcție, pe lângă argumentul *int* mai primește ca argument implicit pointerul *this*
- ▶ Pentru a defini un pointer la funcția *Stiva::Push()* se va utiliza un concept nou: ***pointer la membru***



# Pointeri la membri

---

- ▶ Un pointer la membru al unei clase X se definește folosind construcția **X::\***
  - ▶ `typedef void (Stiva::*PFmembru)(int);`
  - ▶ `PFmembru pf3 = &Stiva::Push;`
- ▶ Utilizarea pointerilor la membri (dereferențierea lor) se face cu ajutorul unor operatori speciali:
  - ▶ `.*` sau `->*`
- ▶ Cu operatori de mai sus se selectează un membru pe baza unui pointer la membru
- ▶ Se folosește operatorul `.*` când obiectul este specificat prin nume
- ▶ Se folosește operatorul `->*` când obiectul este specificat prin pointer.
- ▶ Obiectul apare ca operand stâng iar membrul ca operand drept



# Pointeri la membri

---

- ▶ Conceptul de pointer la membru este valabil și pentru membri de tip dată (adrese relative față de începutul zonei de memorie alocate obiectului)

```
class X
{
public:
    double d;
    X(void):d(0){}
    void fct(void)
    {
        cout << "S-a apelat fct\n";
    }
};
```



```
int main(void)
{
    X obj, obj4; obj4.d = 1;
    double *pd1;
    double X::*pd2;
    void (X::*pf3)(void);

    pd1 = &obj.d;
    pd2 = &X::d;
    pf3 = &X::fct;

    cout <<*pd1<<obj.*pd2<<obj4.*pd2;
    (obj.*pf3)();

    X *px = new X();
    pd1 = &px->d;

    cout <<*pd1<<px->*pd2;
    (px->*pf3)();

    return 0;
}
```



# Supraîncărcarea operatorilor

---

- ▶ În fiecare domeniu au fost dezvoltate notații, convenții pentru a discuta, prezenta diverse concepte într-un mod cat mai convenabil.
- ▶ Ex.

$$x + y * z;$$

este mult mai clar decât

*multiplică y cu z și adună rezultatul la x*

- ▶ C++ are un set de operatori pentru tipurile de date predefinite
- ▶ Pentru tipurile de date definite de utilizatori operatorii trebuie redefiniți
- ▶ Ex. Dacă este nevoie de numere complexe, matrici algebrice sau siruri de caractere se vor defini clase pentru reprezentare

# Supraîncărcarea operatorilor

- ▶ Definirea operatorilor pentru astfel de clase permite programatorului utilizarea unor notații convenționale și convenabile pentru manipularea obiectelor comparativ cu utilizarea funcțiilor clasice.

```
class complex
{
    double re, im;
public:
    complex(double r, double i) : re(r), im(i) {}
    complex operator+(complex);
    complex operator*(complex);
};
```

- ▶ Au fost definiti operatorii `complex::operator+()` și `complex::operator*()` pentru a furniza semnificatie +, \*.
- ▶ Ex. Fie `complex b, c;`

$b+c$  inseamna de fapt `b.operator+(c)`

# Introducere

---

- ▶ Regulile de prioritate a operatorilor se respectă

```
void f(void)
{
    complex a = complex(1, 3.1);
    complex b = complex(1.2, 2);
    complex c = b;
    a = b + c;
    b = b + c * a;
    c = a * b + complex(1, 2);
}
```

- ▶ Astfel  $b=b+c*a$  înseamnă  $b=b+(c*a)$  nu  $b=(b+c)*a$
- ▶ Utilizarea operatorilor definiti de utilizatori nu este restrictionată doar la tipuri concrete. Astfel proiectarea unor interfețe generale si abstracte poate conduce la supraîncărcarea operatorilor ->, [ ], ( )



# Functii operator

- ▶ Pot fi definite urmatoarele funcții operator

+	-	*	/	%	^	&
/	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	/=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	//	++	--	->*	,
->	[]	()	<i>new</i>	<i>new</i> []	<i>delete</i>	<i>delete</i> []

- ▶ Următorii operatori nu pot fi definiți de utilizator

- :: (apartenenta),
  - . (selectare membru)
  - .\* (selectare membru prin intermediul unui pointer la functie).

- ▶ Operatorii =, ->, ->\*, (), [] pot fi supraîncărcăți numai ca funcții membru.
- ▶ Nu este posibil definirea de noi operatori, în schimb se recomandă folosirea funcțiilor. Ex. *pow()* în loc de \*\*

# Functii operator

---

- ▶ Denumirea funcției operator este cuvântul cheie *operator* urmat de operatorul insusi, ex. *operator<<*.
- ▶ O functie operator este declarata și apelată ca orice altă funcție. Folosirea operatorului este o prescurtare a unui apel explicit a functiei operator

```
void f(complex a, complex b)
{
    complex c = a + b;           //prescurtare
    complex d = a.operator+(b);  //apel explicit
}
```

- ▶ Pentru orice tip, fundamental sau abstract, compilatorul are definite variante implicate ale lui *operator=* (atribuire) si *operator&* (adresa lui ...)



# Operatori binari si unari

- ▶ Operatorii binari pot fi definiti atât de funcții membre nestatice având un singur argument sau functii nemembre cu doua argumente.
- ▶ Pentru fiecare operator @,  $aa@bb$  poate fi interpretat ca  $aa.\text{operator}@(bb)$  sau  $\text{operator}@(aa, bb)$
- ▶ Dacă sunt definite amândouă algoritmul de potrivire va determina care sa fie folosită

```
class X {  
public:  
    void operator+(int);  
    X(int);  
};  
void operator+(X, X);  
void operator+(X, double);  
void f(X a) {  
    a+1;    //a.operator+(1)  
    1+a;    //::operator+(X(1),a)  
    a+1.0;  //::operator+(a,1.0)  
}
```

# Operatori binari si unari

---

- ▶ Operatorii unari atât cei prefixati cât și cei postfixati, pot fi definiți atât de funcții membre nestatice fără argument cât și de funcții nemembre având doar un argument.
- ▶ Pentru orice operator prefixat @,  $@aa$  poate fi interpretat ca  $aa.operator@()$  sau ca  $operator@(aa)$ .
- ▶ Daca ambele sunt definite, regulile de potrivire determina care va fi folosita.
- ▶ Pentru orice operator postfixat @,  $aa@$  poate fi interpretat ca  $aa.operator@(int)$  sau ca  $operator@(aa, int)$ .
- ▶ Daca ambele sunt definite regulile de potrivire determina care va fi folosita.
- ▶ Un operator poate fi declarat doar pentru sintaxa definită pentru el.



# Operatori binari si unari

---

```
class X
{
    //functii membre (pointerul this este implicit):
    X*operator&(void);           //unar prefixat &(adresa ...)
    X operator&(X);              //binar&(si)
    X operator++(int);            //postfixat de incrementare
    X operator&(X, X);            //eroare: ternar
    X operator/(void);            //eroare: unar/
};

//functii nemembre:
X operator-(X);                  //prefixat unar minus
X operator-(X, X);                //binar minus
X operator--(X&, int);           //postfixat decrementare
X operator-(void);                //eroare: nici un operand
X operator-(X, X, X);             //eroare:ternar
X operator%(X);                  //eroare:unar %
```



# Operatori unari - detaliere

---

- ▶ Operatorii unari au un singur argument: obiectul asupra căruia se aplică. Acest argument poate fi dat explicit, dacă se supraîncarcă o variantă nemembru, sau implicit prin pointerul **this**, dacă se supraîncarcă o variantă membru
- ▶ Fie clasa *Time*:

```
class Time {  
    int min, sec, hour;  
public:  
    Time& operator++(void);  
}  
Time& Time::operator++(void) {  
    hour += (min += ++sec/60) /60;  
    sec %= 60; min %= 60; hour %=24;  
    return *this;  
}  
Time time;  
++time;  
time++;  
time.operator++();
```

# Operatori unari - detaliere

---

- ▶ *Operatorul* `++` implementat ca mai sus, are două forme de apel acceptate, ca operator prefix, și ca operator sufix, dar, pentru tipul abstract implementat, ele sunt echivalente
- ▶ Dacă se dorește ca operator`++` forma sufix să fie diferit de forma prefix se recurge la un compromis între supraîncărcarea unui operator unar și a unuia binar:
  - ▶ Se cunoaște că un operator unar operează asupra unui singur operand, forma operatorului fiind

*tip* `operator++();`

*tip* `operator--();`

- ▶ unde operandul este dat prin argumentul implicit *this*
- ▶ Un operator binar are doi operanzi: cel implicit (current) și cel explicit dat ca argument în metodă:
  - ▶ *tip* `operator++(argument);`
- ▶ Se implementează o operație binară în care al doilea operand nu este utilizat



# Operatori unari - detaliere

---

- ▶ Forma sufix: 

```
Time& Time::operator++(int i)
{
    hour++;
    hour %= 24;
    return *this;
}
int i = 1;
Time time;
time++;
time.operator++(i);
++time;
```
- ▶ Concluzie: `time.operator++();`

- ▶ Dacă ar fi lipsit definiția variantei sufix atunci atât `time++` cât și `++time` ar fi fost rezolvate prin varianta prefix
- ▶ Dacă ar fi lipsit definitia variantei prefix, ar fi apărut eroare la `++time` căci varianta sufix cere un argument în lista de argumente



# Operatori unari - detaliere

- ▶ Dacă `operator++` este definit ca funcție globală atunci are acces la partea privată a obiectului `Time` numai dacă este funcție prietenă.

```
class Time
{
    int min, sec, hour;
public:
    friend Time& operator++(Time&);
}
Time& operator++(Time& t)
{
    t.hour += (t.min += ++t.sec/60 ) /60;
    t.sec %= 60;
    t.min %= 60;
    t.hour %=24;
    return t;
}
```

- ▶ Apel: `Time time; time++; ++time; operator++(t);`  
*//varianta prefix*

# Operatori unari - detaliere

---

- ▶ Varianta pentru forma sufix

```
class Time
{
    int min, sec, hour;
public:
    friend Time& operator++(Time&);
    friend Time& operator++(time&, int);
}

Time& operator++(Time& t, int i)
{
    t.hour++;
    t.hour %= 24;
    return t;
}
```

- ▶ Pentru supraîncărcarea unui operator, în formă nemembru și neprieten trebuie folosite funcțiile de acces la membri privați din clasa *Time*

---

Vă mulțumesc !



# PROGRAMARE ORIENTATĂ OBIECT

Curs 6

*Moștenire*



# Introducere

---

- ▶ Limbajul C++ a împrumutat ideea claselor și ierarhiei de clase din limbajul Simula. De asemenea a împrumutat ideea de modelare a noilor concepte/tipuri de date prin intermediul claselor.
- ▶ Un concept (noțiune, idee) nu există de sine stătător ci coexistă cu alte concepte și ceea ce este mai important, în relație cu acestea.
- ▶ Ex. conceptul de mașină introduce alte noțiuni: roată, motor, șofer, pieton, camion, ambulanță, drum, ulei etc.
- ▶ Având în vedere utilizarea claselor pentru reprezentarea conceptelor, problema care apare este cum să fie reprezentată relația dintre concepte.
- ▶ Noțiunea de clasă derivată este utilizată pentru a explica similitudinile dintre clase, relația erarhică.



# Introducere

---

- ▶ Conceptele de cerc și triunghi au ceva în comun și anume conceptul de figură. Astfel se definește clasa *Cerc* și clasa *Triunghi* ca având clasa *Figura* în comun.
- ▶ Clasa comună *Figura* este referită clasa de bază sau superclasă și clasa derivată (*Cerc* sau *Triunghi*) din clasa *Figura* ca fiind clasa derivată sau subclasă.
- ▶ Limbajul de programare oferă facilități pentru construirea de noi clase pe baza celor existente:
  - ▶ *Implementation inheritance*: utilizarea facilităților furnizate de clasa de bază în clasa derivată
  - ▶ *Interface inheritance*: utilizarea diferitelor clase derivate prin intermediul interfeței furnizate de clasa de bază comună (run-time polimorfism sau dynamic polimorfism).



# Se considera clasa Date

---

```
const int monthDays[]={31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
  
int CheckBisect(int year)  
{  
    return (((year%4)==0)&&(!(((year%100)==0)&&((year%400) != 0))));  
}  
  
class Date  
{  
    int d, m, y;  
public:  
    Date(int dd=0, int mm=0, int yy=0):d(dd), m(mm), y(yy) {};  
    //metode Get Set  
    Date operator-(const Date&dc){ /*...*/};  
};
```



# Clase derivate

- ▶ Se consideră un program ce gestionează persoanele angajate dintr-o firmă:

```
class Angajat
{
    char *firstName;
    char *familyName;
    char middleInitial;
    Date hiringDate;
    short int departament;
};
```

```
class Manager
{
    Angajat ang; //datele de angajat
                  //ale managerului
    Angajat *group; //oameni in
                     //subordine
    short int Level;
};
```

- ▶ Un manager este un *Angajat*. Datele de *Angajat* sunt stocate în membrul *ang* al obiectului *Manager*.
- ▶ Nu este nimic care să „spună” compilatorului că *Manager* este un *Angajat*.
- ▶ De asemenea un *Manager\** nu este un *Angajat\**. Nu se pot pune *Manager\** într-un vector cu *Angajat\**

# Clase derivate

---

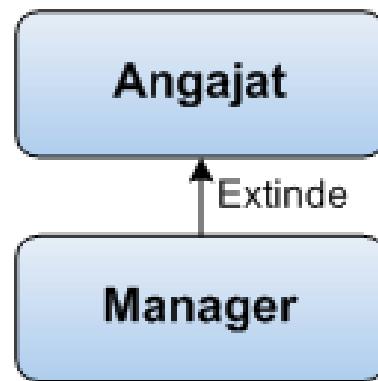
- ▶ Se poate converti explicit *Manager*\* în *Angajat*\* sau pur și simplu să fie pusă adresa membrului *ang* într-o listă de angajați, dar nu se recomandă
- ▶ Abordarea corectă este de a afirma în mod explicit că *Manager* este un *Angajat* dar cu noi informații adăugate.

```
class Manager : public Angajat
{
    Angajat *group;           //oameni in subordine
    short int level;
};
```



# Clase derivate

- ▶ Clasa *Manager* are membri *firstName*, *departament* etc. și în *plus group*, *Level* etc.
- ▶ Derivarea este reprezentată grafic printr-o săgeată de la clasa derivată către clasa de bază



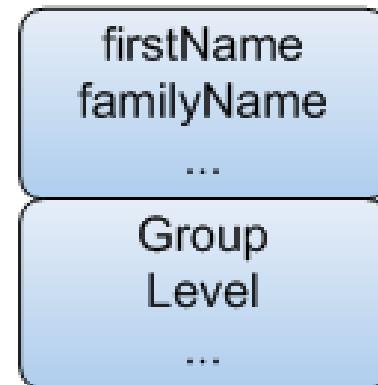
# Clase derivate

- ▶ În mod obișnuit se spune că o clasă derivată moștenește proprietățile din clasa de bază, motiv pentru care relația se numește moștenire.
- ▶ O implementare populară și eficientă a noțiunii de clasă derivată este un obiect al clasei derivate reprezentat ca un obiect al clasei de bază cu informații în plus ce aparțin doar clasei derivate, adăugate la final.

Angajat



Manager



# Clase derivate

---

- ▶ Derivând *Manager* din *Angajat* face ca *Manager* să fie un subtip al tipului *Angajat*. Astfel se poate crea un vector de angajați în care unele dintre ei sunt manageri

```
Angajat a1;  
Manager m1;  
Angajat v[2] = {a1, m1};
```



# Clase derivate

---

- ▶ Având în vedere că *Manager* este un *Angajat* atunci și *Manager\** poate fi folosit ca un *Angajat\**. Similar și *Manager&* poate fi folosit ca un *Angajat&*.
- ▶ Totuși un *Angajat* nu este necesar un *Manager* astfel un *Angajat\** nu poate fi folosit ca un *Manager\**.
- ▶ În general dacă o clasă *Derivată* are o clasă publică *Bază*, atunci un pointer *Derivată\** poate fi atribuit unui pointer de tip *Bază\** fără conversie explicită de tip. Învers, din *Bază\** în *Derivată\** conversia trebuie realizată în mod explicit.
- ▶ O clasa trebuie definită înainte de a fi folosită ca o clasă de bază



# Moștenirea C++

---

- ▶ Pentru a deriva o clasă dintr-alta, se folosește operatorul “:” la declararea clasei deriveate după următorul format:

```
class ClasaDerivata : public ClasaBaza  
{ ... }
```

- ▶ Specifierul de acces (*public*) poate fi înlocuit de unul dintre ceilalți doi specifatori *private* sau *protected*

```
class ClasaBaza  
{  
    //membri clasa de baza  
};
```

```
class ClasaDerivata: [public|protected|private] ClasaBaza  
{  
    //membrii clasa derivata  
};
```



# Moștenirea – specificatori de acces

Modificatorii de protecție utilizați în lista claselor de bază, definesc protecția în clasa derivată a elementelor moștenite. O clasă derivată va moșteni membrii clasei de bază, dar accesul la aceștia va fi restricționat de modificatorii de protecție.

Accesul în clasa de bază	Specificatorul de acces din lista claselor de bază	Accesul în clasa derivată a membrului moștenit
Private	Private	inaccesibil
Protected		private
Public		private
Private	Protected	inaccesibil
Protected		protected
Public		protected
Private	Public	inaccesibil
Protected		protected
Public		public



# Moștenirea – specificatori de acces

---

- ▶ La moștenirea private, membrii moșteniti cu protecția *protected* sau *public*, vor avea protecția *private* în clasa derivată. Această înseamnă că nu vor mai putea fi accesati de o clasă derivată din clasă ce derivă din clasa de bază de unde sunt moșteniți. Practic această tehnică închide ierarhia, și posibilitatea de extindere a acestieia.
- ▶ **De obicei, o ierarhie de clase nu este o ierarhie finală, ea putând fi dezvoltată adăugând clase noi, care derivă din clasele terminale.**
- ▶ Pentru a facilita extinderea ierarhiei fără probleme se folosește moștenirea publică.



# Functii membre

```
class Angajat
{
private:
    char *firstName;
    char *familyName;
    char middleInitial;
    Date hiringDate;
    short int departament;
public:
    void Print(void) const;
    char* GetFullName(void);
};

char* Angajat::GetFullName(void)
{
    char *strTmp = new char[strlen(firstName)+strlen(familyName) + 4];
    sprintf(strTmp, "%s %c. %s", firstName, middleInitial, familyName);
    return strTmp;
}

class Manager : public Angajat
{
private:
    Angajat *group;
    short int level;
public:
    void Print(void) const;
};
```

# Functii membre

---

- ▶ O metodă a clasei derivate poate folosi membri publici și `protected` din clasa de bază ca și cum ar apartine clasei derivate

```
void Manager::Print(void)
{
    char *fullName = GetFullName();
    cout << "Numele: " << fullName << "\n";
    delete []fullName;
}
```

- ▶ Totuși o clasă derivată nu poate accesa membri privați ai clasei de bază (eroare de compilare)

```
void Manager::Print(void)
{
    cout << "Numele: " << familyName << "\n";
}
```

- ▶ Acolo unde este necesar, membri privați din clasa de bază pot fi declarați ca fiind `protected` astfel încât să fie accesibili și în clasele derivate.



# Funcții membre

---

- ▶ Cea mai bună soluție de implementare ar fi:

```
void Manager::Print(void)
{
    Angajat::Print(); //afiseaza informatii Angajat
    cout << Level << "\n"; //afiseaza informatii specifice Manager
    //...
}
```

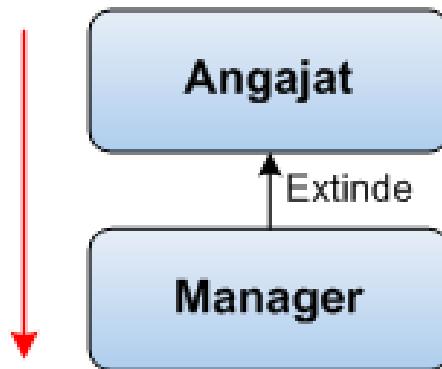
- ▶ Având în vedere că *Print* a fost redefinită în clasa Manager trebuie specificat în mod clar care funcție membră să fie apelată

*Angajat::Print();*



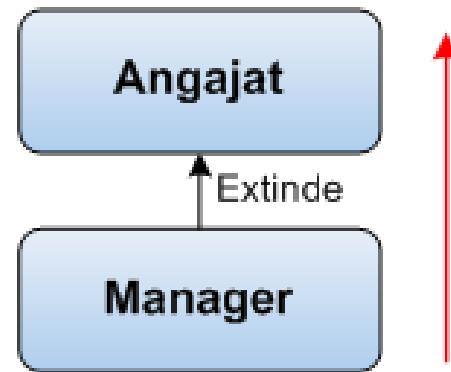
# Constructori și destructori

- ▶ Ordinea de apelare a constructorilor pentru un obiect dintr-o clasă derivată: de la clasa de bază către clasa derivată
  - ▶ Constructorul clasei de bază
  - ▶ Constructorul clasei deriveate



# Constructori și destructori

- ▶ Ordinea de apelare a destructorilor pentru un obiect dintr-o clasă derivată: de la clasa derivată către clasa de bază
  - ▶ Destructorul clasei deriveate
  - ▶ Destructorul clasei de bază



# Constructorii apelați implicit

---

- ▶ Pentru o instanță a unei clase de bază, constructorul apelat implicit este constructorul fără listă de argumente (dacă există).
- ▶ Prinapeluri de forma:

*Manager::Manager( ... ) : Angajat( ... ) {}*

se poate controla ce constructor din clasa de bază va fi folosit  
(se forțează compilatorul pentru a apela un anumit constructor)



# Moștenirea “în lant” (în cascadă)

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

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

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

```
int main()
{
    C objC;
}
```

*La rulare, se va afișa:*  
A  
B  
C

# Suprascrierea funcțiilor din clasa de bază

---

```
void Angajat::Print(void) const
{
    std::cout << "Nume: " << firstName << " " << middleInitial << ". „
                           << familyName << " \n";
}

void Manager::Print(void) const
{
    std::cout << "Nume: " << firstName << " " << middleInitial << ". „
                           << familyName << " \n";
    std::cout << level << "\n";
    //...
}

sau

void Manager::Print(void) const
{
    Angajat::Print();
    std::cout << level << "\n";
    //...
```

---

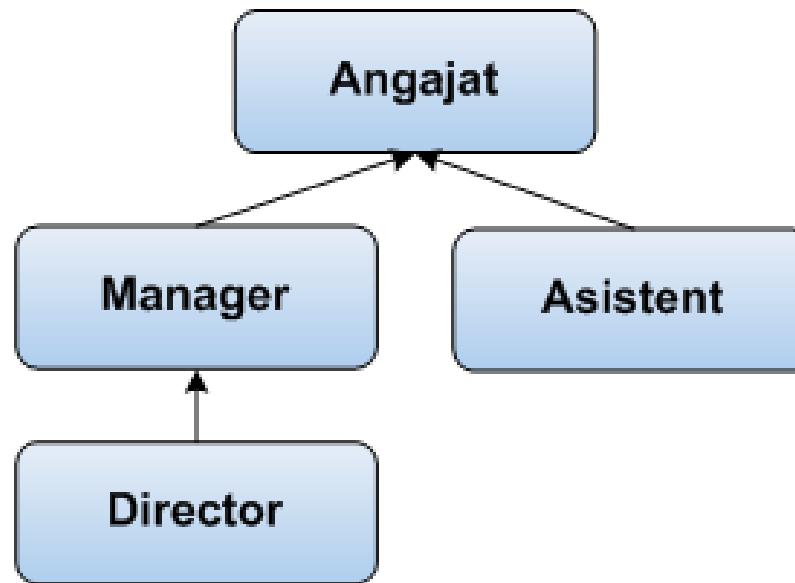


# Ierarhie de clase

- ▶ O clasă derivată poate fi o clasă de bază:

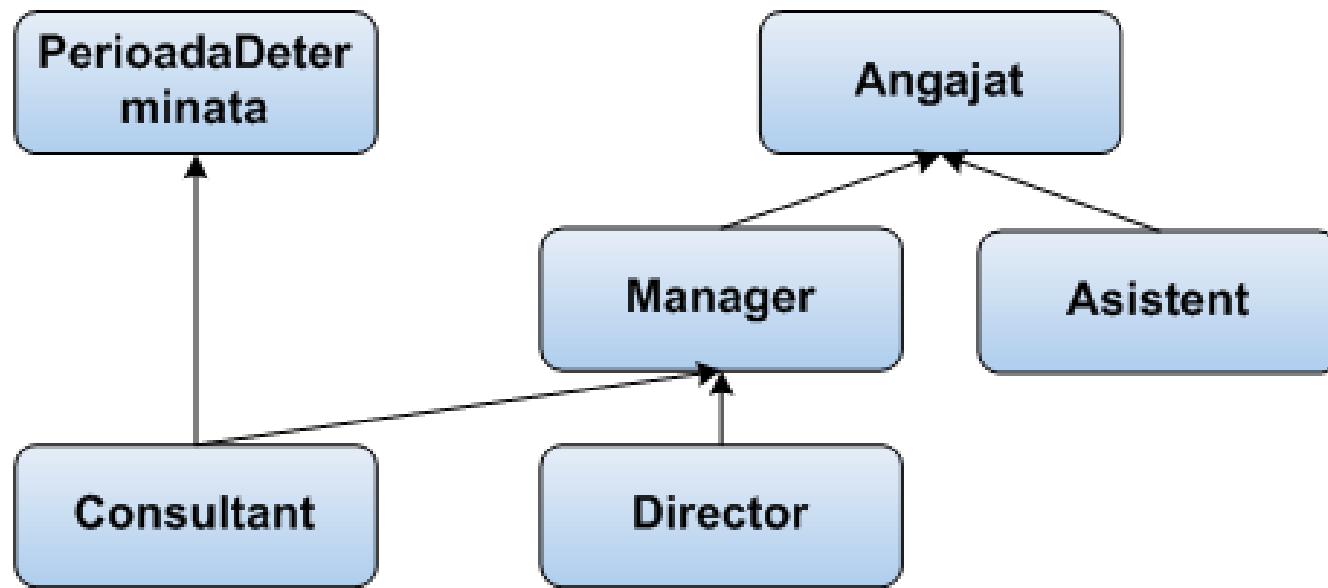
```
class Angajat { /* ... */;
class Manager : public Angajat { /* ... */ };
class Director : public Manager { /* ... */ };
class Asistent : public Angajat { /* ... */ };
```

- ▶ Un set de clase relateionate în acest fel se numește ierarhie de clase



# Ierarhie de clase

- ▶ În general o ierarhie de clase are forma unui arbore dar poate fi și sub forma unui graf



- ▶ Moștenirea multiplă apare atunci când o clasă derivată are mai multe clase de bază.

```
class PerioadaDeterminata /* ... */;
class Consultant : public PerioadaDeterminata, public Manager
{ /* ... */};
```

# Sintaxă

---

```
class Bi
{
    //membri clasa de baza
};

class D: [public|protected|private]B1,...,[public|protected|private]Bn
{
    //membrii clasa derivata
};
```



# Moștenire multiplă

---

```
class B1
{
protected:
    int m_b1;
public:
    B1(int b1) : m_b1(b1) { cout << "B1(int)\n"; }
    ~B1(void) {cout << "~B1()\n"; }
    void Print(void) {cout << m_b1 << endl;}
};

class B2
{
protected:
    float m_b2;
public:
    B2(float b2): m_b2(b2) { cout << "B2(float)\n"; }
    ~B2(void) {cout << "~B2()\n"; }
    void Print(void) {cout << m_b2 << endl;}
};
```



# Moștenire multiplă

---

```
class B3{  
protected:  
    int m_ib3;  
    double m_db3;  
public:  
    B3(int ib3, double db3): m_ib3(ib3), m_db3(db3){cout<<  
                                            "B3(int, double)\n";}  
    ~B3(void) {cout << "~B3()\n";}  
    void Print(void) {cout<<m_ib3<<endl; cout<<m_db3<<endl;}  
};
```



# Moștenire multiplă

---

```
class D: public B1, private B2, protected B3 {  
protected:  
    char c;  
public:  
    D(int i1, float f1, int i2, double d1, char cc):B3(i2,  
            d1), B2(f1), B1(i1),  
            c(cc){cout<<"D(int,float,int,double)\n";}  
    ~D(void) { cout << "~D()\n";}  
    void afisare() {  
        B1::Print();  
        B2::Print();  
        B3::Print();  
        cout << c << endl;  
    }  
};
```



# Moștenire multiplă

```
int main(void)
{
    D d(2, 3.5f, 4, 4.55, 'a');
    d.Print();
    return 0;
}
```

Name	Value	Type
d	{c=97 'a'}	D
B1	{m_b1=2}	B1
m_b1	2	int
B2	{m_b2=3.5000000}	B2
m_b2	3.5000000	float
B3	{m_ib3=4 m_db3=4.5499999999999998}	B3
m_ib3	4	int
m_db3	4.5499999999999998	double
c	97 'a'	char

Rulare:

**Apel constructori:**

B1(int)

B2(float)

B3(int, double)

D(int, float, int, double)

2

3.5

4

4.55

a

**Apel destructori:**

~D()

~B3()

~B2()

~B1()



---

Vă mulțumesc !



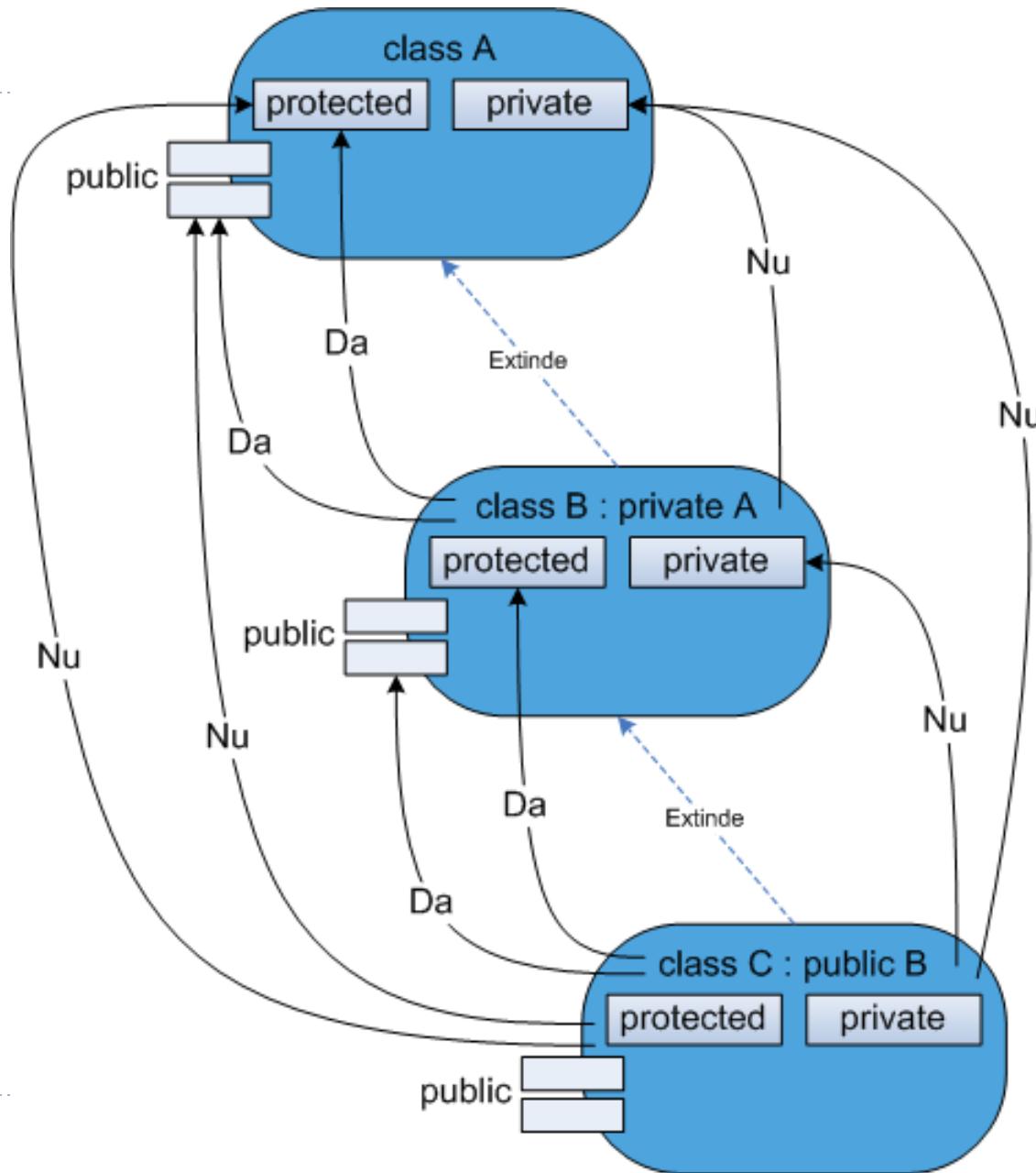
# PROGRAMARE ORIENTATĂ PE OBIECTE

Curs 7

*Moștenire - continuare*

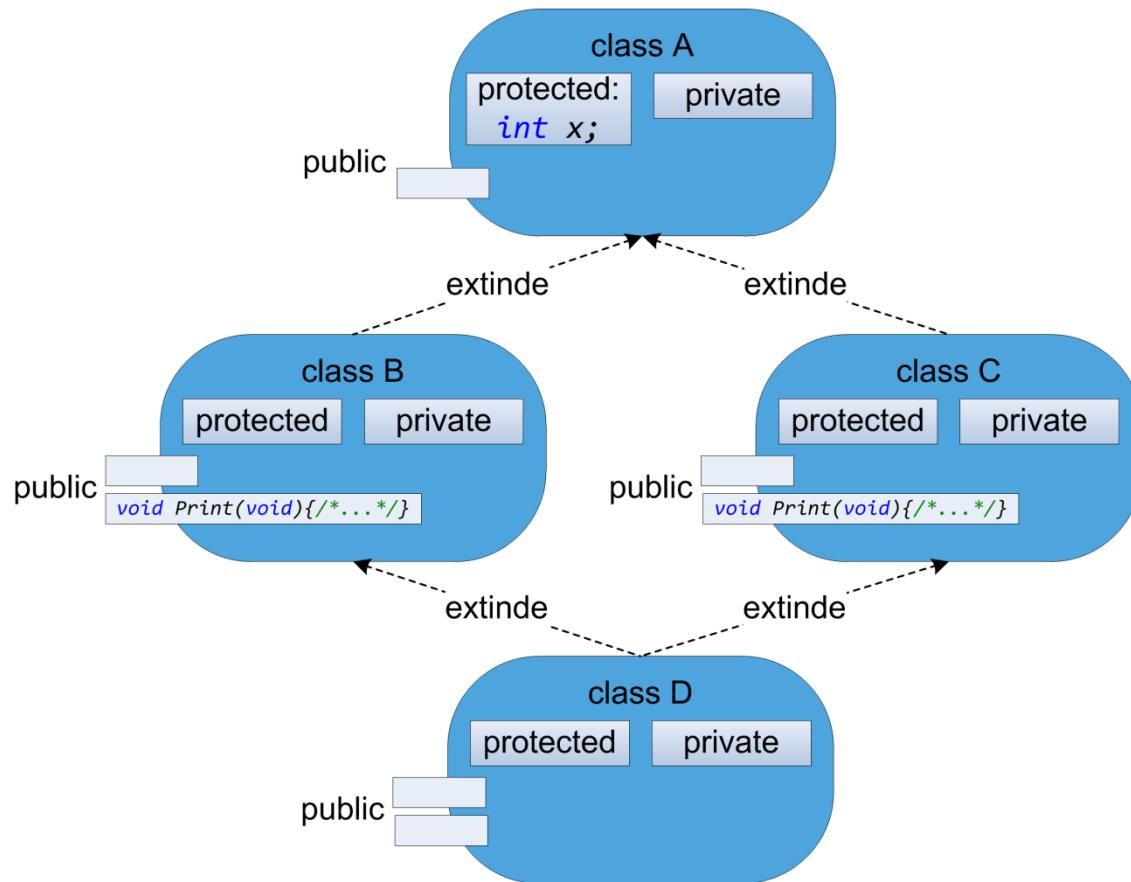
Moștenire multiplă – probleme  
Polimorfism

# Controlul accesului la membri clasei de bază



# Moștenirea multiplă - probleme

- ▶ Moștenire în formă de diamant
- ▶ O clasă moștenește 2 clase care au o funcție membră cu aceeași semnătură



# Moștenirea multiplă – studiu de caz

## Problema diamantului

---

- ▶ Fie următoarea ierarhie de clase

```
class A
{
protected:
    int x;
public:
    A(void) {x = 0; cout << "A()\n";}
    A(int xx) {x = xx; cout << "A(int)\n";}
    ~A(void) {cout << "~A()\n";}
};

class B: public A
{
public:
    B(void){cout << "B()\n";}
    B(int xx): A(x) {cout << "B(int)\n";}
    ~B(void) {cout << "~B()" << endl;}
};
```



# Moștenirea multiplă – studiu de caz

## Problema diamantului

---

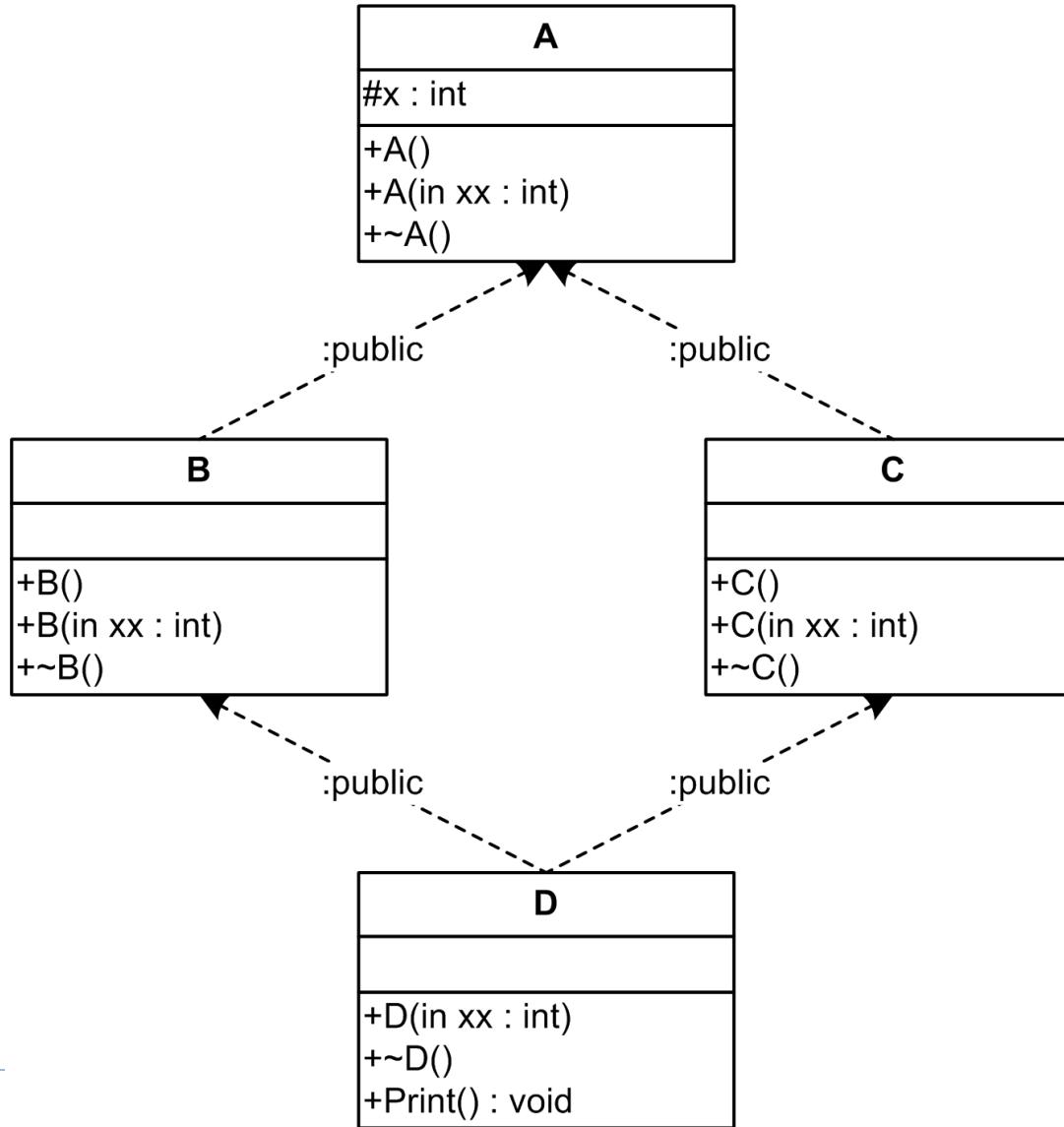
```
class C: public A
{
public:
    C(void){ cout << "C()\n"; }
    C(int xx) : A(xx) {cout << "C(int)\n"; }
    ~C(void) {cout << "~C()\n"; }
};

class D: public B, public C
{
public:
    D(int xx): B(xx), C() {cout << "D(int)\n"; }
    ~D(void) {cout << "~D()\n"; }
    void Print(void) {cout << x << endl;};
};
```



# Moștenirea multiplă – studiu de caz

## Problema diamantului



# Moștenirea multiplă – studiu de caz

## Problema diamantului

```
int main (void)
{
    A a(6); B b(7); C c(8); D d(5);
    d.Print();
    cout << "sizeof(a) = " << sizeof(a) << endl;
    cout << "sizeof(b) = " << sizeof(b) << endl;
    cout << "sizeof(c) = " << sizeof(c) << endl;
    cout << "sizeof(d) = " << sizeof(d) << endl;
    return 0;
}
```

### ▶ Compilare:

*error C2385: ambiguous access of 'x' in 'D'*

### ▶ Posibile „solutii” de implementare ale functiei *D::Print()*:

```
void D::Print(void) {cout << A::x << endl;} // se afiseaza 5
void D::Print(void) {cout << B::x << endl;} // se afiseaza 5
void D::Print(void) {cout << C::x << endl;} // se afiseaza 0
```

# Moștenirea multiplă – studiu de caz

## Problema diamantului

---

### ► Dimensiuni ale obiectelor:

*sizeof(a) = 4*

*sizeof(b) = 4*

*sizeof(c) = 4*

*sizeof(d) = 8*

### ► Apelarea constructorilor:

*A(int)*

*B(int)*

*A()*

*C()*

*D(int)*

*θ*

*sizeof(d)=8*

*~D()*

*~C()*

*~A()*

*~B()*

*~A()*



# Moștenirea multiplă – studiu de caz

## Problema diamantului

- ▶ Pentru rezolvarea cazului de ambiguitate se declara clasa A virtuala astfel încât constructorul clasei virtuale va fi apelat prima dată și apoi, celelalte apeluri explicite ale constructorului clasei virtuale vor fi ignorate

```
class B: public virtual A
{
public:
    B(void){cout << "B()\n";}
    B(int x): A(x) {cout << "B(int)\n";}
    ~B(void) {cout << "~B()";}
};

class C: virtual public A
{
public:
    C(void){cout << "C()\n";}
    C(int x): A(x) {cout << "C(int)\n";}
    ~C(void) {cout << "~C()\n";}
};
```

# Moștenirea multiplă – studiu de caz

## Problema diamantului

- ▶ Apel main:

```
A()  
B(int)  
C()  
D(int)  
D::x=0  
sizeof(d)=16  
~D()  
~C()  
~B()  
~A()
```

- ▶ Pentru a se forța apelarea constructorului de initializare, se reimplementează constructorul de initializare al clasei D.

```
class D: public B, public C  
{  
public:  
    D(int x): A(x) {cout << "D(int)\n";}  
    ~D(void) {cout << "~D()\n"; }  
    void Print(void){cout << x << endl;};  
};
```

# Moștenirea multiplă – studiu de caz

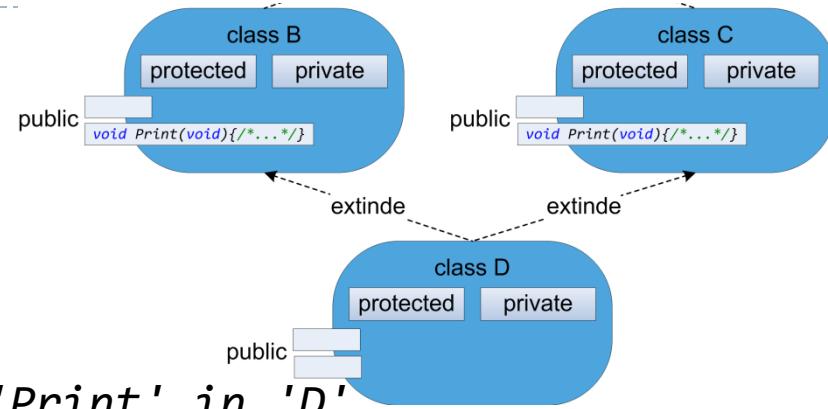
## Problema diamantului

---

- ▶ Apel main:
  - $A(int)$
  - $B()$
  - $C()$
  - $D(int)$
  - $D::x=5$
  - $sizeof(d)=12$
  - $\sim D()$
  - $\sim C()$
  - $\sim B()$
  - $\sim A()$



# O clasă moștenește 2 clase care au o funcție membră cu aceași semnătură



```
d.Print();
//...
error C2385: ambiguous access of 'Print' in 'D'
```

## ▶ Soluții:

- ▶ redefinirea funcției respective în clasa derivată

```
void D::Print(void)
{
    cout << "D::x=" <<x<< endl;
}
```

- ▶ utilizarea operatorului de rezoluție apartenență la domeniu pentru a specifica în mod clar domeniul de care aparține funcția

```
d.B::Print();
```



# Ierarhii de clase

---

- ▶ Odată cu moștenirea trebuie rezolvată o problemă: Având un pointer la clasa de bază (*Base* \*), cărei clase derivate aparține obiectul pointat. Sunt 4 soluții:
  - ▶ Asigura-te că pointerul pointează către obiecte de același tip (nu se recomandă dar poate fi o soluție pentru containere omogene)
  - ▶ Plasarea unui membru în clasa de bază pentru a caracteriza tipul de date.
  - ▶ A se utiliza *dynamic\_cast* pentru convertirea pointerului
  - ▶ A se utiliza funcții virtuale

```
class Angajat
{
protected:
    enum TipAngajat{man, ang};
    TipAngajat angType;
public:
    Angajat(void): angType(ang){/*...*/};
    //...
};
```

# Ierarhii de clase

```
class Manager : public Angajat
{
    //...
public:
    Manager(void){angType=man; /*...*/};
    //...
};

void Angajat::Print(void) const{
    switch(angType)
    {
        case ang:
            cout << "Angajat -> Nume: " << firstName << " " <<
                middleInitial << ". " << familyName << "\n";
            break;
        case man:
            cout << "Manager -> Nume: " << firstName << " " <<
                middleInitial << ". " << familyName;
            const Manager*p = static_cast<const Manager*>(this);
            cout << ", Level" << (((Manager&)(*p)).GetLevel()) << "\n";
            break;
    }
}
```

# Ierarhii de clase

---

```
int main(void)
{
    Date da(5,10,1977);
    Date dm(10,5,1968);
    cout<<"*****\n";

    Manager m("Gica", "Popescu", 'G', dm, 3, 10, 8);
    Angajat a("Mitica", "Gheorghe", 'I', da, 1);

    Angajat *pa;
    pa = &a;
    pa->Print();

    pa = &m;
    pa->Print();

    return 0;
}
```



# Ierarhii de clase

---

Apel main:

*Angajat -> Nume: Mitica I. Gheorghe*

*Manager -> Nume: Gica G. Popescu, Level 8*

- ▶ Această soluție nu se recomandă deoarece prezintă probleme de întreținere, fiind o soluție generatoare de erori



# Funcții virtuale

---

- ▶ Utilizarea funcțiilor virtuale elimină problemele generate de soluția cu membru de caracterizare a tipului obiectului, permitând programatorului să declare funcții în clasa de bază ce pot fi redefinite în fiecare clasă
- ▶ Compilatorul și linkeditorul va garanta corespondența corectă între obiecte și funcțiile aferente.

```
class Angajat
{
private:
    //...
public:
    //...
    virtual void Print(void) const;
};
```



# Funcții virtuale

```
int main(void)
{
    Date da(5,10,1977);
    Date dm(10,5,1968);

    cout<<"*****\n";
    Manager m("Gica", "Popescu", 'G', dm, 3, 10, 8);
    Angajat a("Mitica", "Gheorghe", 'I', da, 1 );

    Angajat *pa;
    pa = &a;
    cout << "Angajat -> ";
    pa->Print();

    pa = &m;
    std::cout << "Manager -> ";
    pa->Print();

    return 0;
}
```

# Functii virtuale

---

- ▶ Cuvântul cheie *virtual* indică faptul că *Print()* se comportă ca o interfață atât pentru funcția *Print()* definită în clasa *Angajat* cât și pentru funcția *Print()* definită în clasa *Manager* derivată din *Angajat*.
- ▶ Compilatorul va asigura că pentru un obiect *Angajat* va fi apelată funcția *Print()* corespunzătoare, dacă în clasele deriveate aceasta a fost definită.
- ▶ Toate funcțiile *Print()* redefinite trebuie să aibă aceeași semnătură
- ▶ O funcție virtuală trebuie definită în clasa unde a fost declarată (cu excepția cazurilor când sunt declarate ca fiind virtuale pure)
- ▶ Determinarea funcției *Print()* corespunzătoare, independent de tipul *Angajat*-ului se numește **polimorfism**.



# Polimorfism

---

- ▶ Un tip de date cu funcții virtuale se numește *tip de date polimorfic* sau mai exact ***run-time polymorphic type***.
- ▶ Pentru a obține un comportament polimorfic obiectele trebuie manipulate prin intermediul pointerilor sau referințelor
- ▶ O funcție ce suprascră o funcție virtuală devine la rândul ei virtuală.
- ▶ Pentru a implementa polimorfismul, compilatorul trebuie să rețină unele informații în fiecare obiect de tip *Angajat* și să le folosească pentru a apela funcția *Print()* corespunzătoare.
- ▶ Conectarea unui apel al unei funcții de corpul acesteia se numește legătura



# Polimorfism

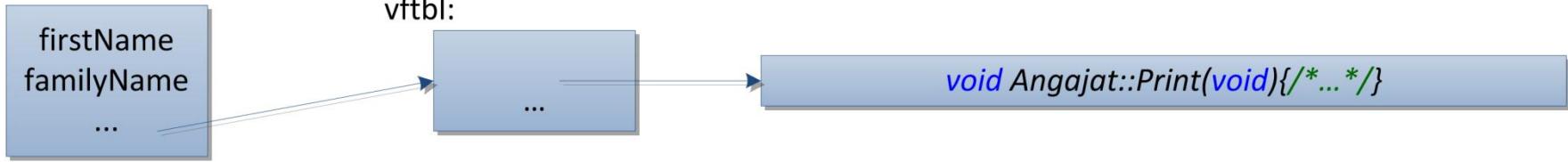
---

- ▶ Când legătura se realizează înainte de rularea unui program (de către compilator și linkeditor), se numește **legare timpurie sau legare statică** (*early binding*)
- ▶ Legarea implicită în C++ este cea statică
- ▶ Polimorfismul este un mecanism prin care legarea dintre apelul unei metode și corpul acesteia se face la momentul rulării (**legarea dinamică**) (*late binding*)
- ▶ În mod uzual compilatorul convertește numele unei funcții virtuale într-un index din cadrul unui tabel de pointeri la funcții. Acel tabel este numit *tabelul cu funcții virtuale* sau mai simplu **vftbl**.
- ▶ Fiecare clasă cu funcții virtuale are propriul său **vftbl**

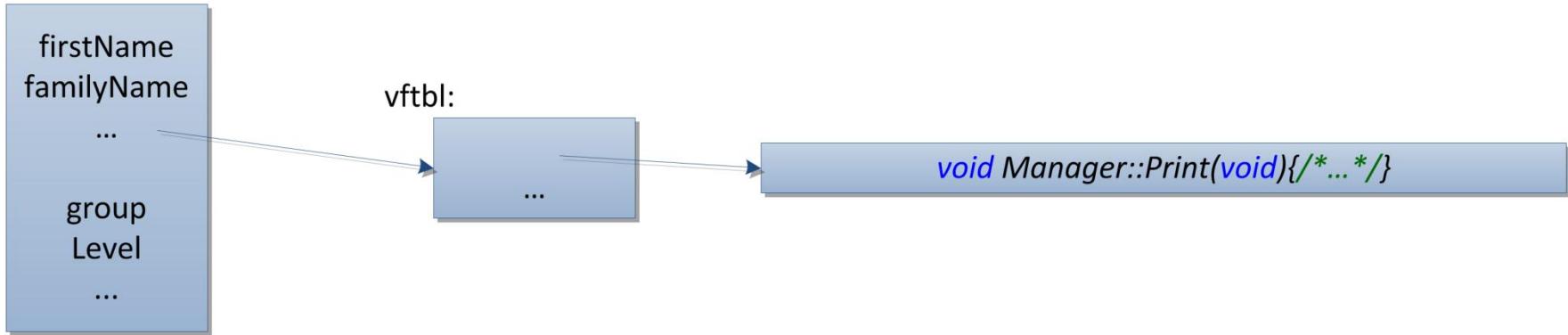


# Tabela funcțiilor virtuale

Angajat



Manager



# Tabla funcțiilor virtuale

---

- ▶ Conține pointeri către funcțiile virtuale
- ▶ Este creată pentru fiecare clasă ce conține funcții membre virtuale sau suprascrisă funcții virtuale
- ▶ Există numai una pentru o anumită clasă
- ▶ Există clase pentru care nu este creată
- ▶ Când un obiect este creat, se adaugă un membru ascuns, un pointer către această tabelă virtuală
- ▶ Compilatorul generează automat codul în constructori pentru inițializarea pointerului către această tabelă
- ▶ Se accesează în momentul când se execută o funcție virtuală



---

Vă mulțumesc !



# PROGRAMARE ORIENTATĂ PE OBIECTE

Curs 8

*Polimorfism - continuare*

Polimorfism  
Type casting

# Tabla funcțiilor virtuale

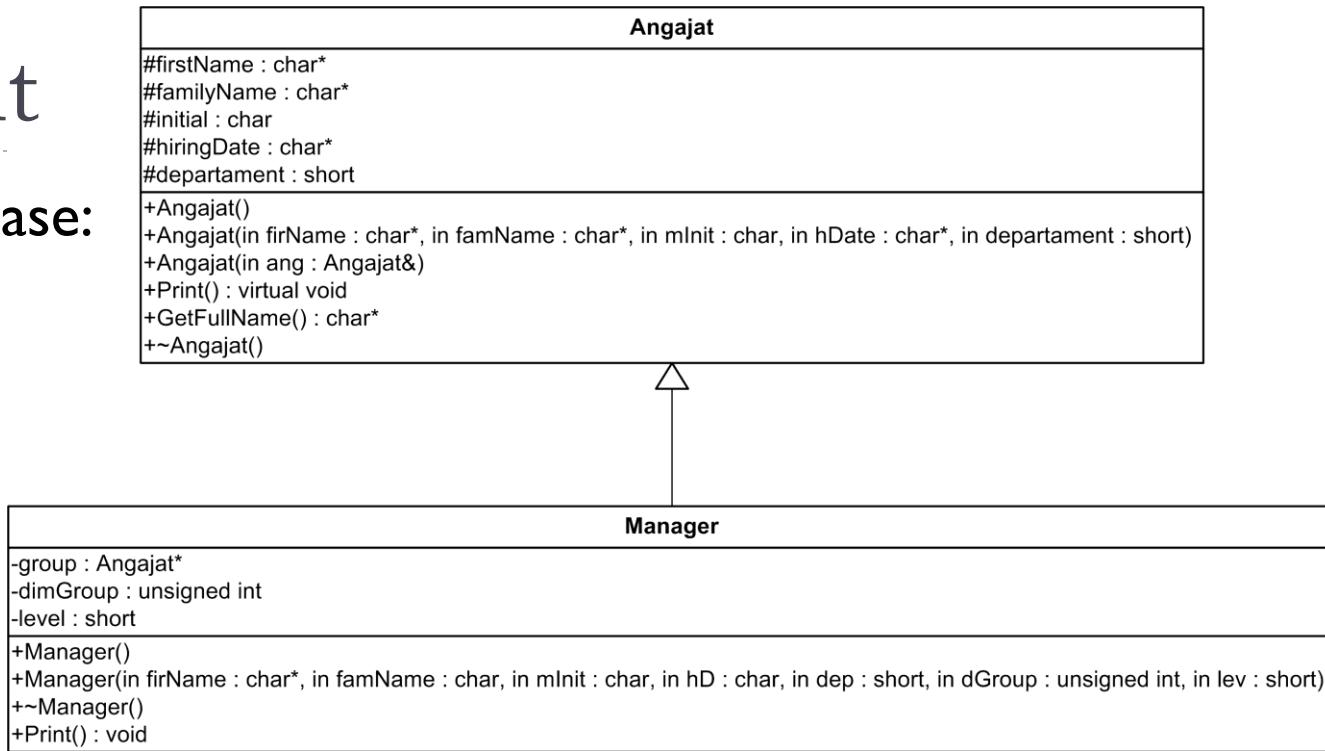
---

- ▶ Conține pointeri către funcțiile virtuale
- ▶ Este creată pentru fiecare clasă ce conține funcții membre virtuale sau suprascrisă funcții virtuale
- ▶ Există numai una pentru o anumită clasă
- ▶ Există clase pentru care nu este creată
- ▶ Când un obiect este creat, se adaugă un membru ascuns, un pointer către această tabelă virtuală
- ▶ Compilatorul generează automat codul în constructori pentru inițializarea pointerului către această tabelă
- ▶ Se accesează în momentul când se execută o funcție virtuală



# Apel explicit

- ### ▶ Fie ierarhia de clase:



- ▶ Apelând o funcție cu ajutorul operatorului apartenență la domeniu asigură că mecanismul virtual nu este folosit:

```
void Manager::Print(void) const
```

{

```
Angajat::Print(); //nu este un apel virtual  
std::cout << "Level = "<<level <<"\n"; //afiseaza informatii  
specifice Manager
```



# Clase abstracte

---

- ▶ Clasa Angajat poate fi utilizată de sine stătător, ca interfață pentru clasele derivate și ca parte a claselor derivate.
- ▶ Totuși sunt clase precum Figura care reprezintă concepte abstracte pentru care obiectele nu există.
- ▶ O Figura are sens doar dacă o clasă este derivată din ea.

```
class Figura
{
public:
    virtual void rotate(int) { cout << "Figura::rotate\n"; }
    virtual void draw() const { cout<<"Figura::draw\n"; }
};
```

- ▶ O alternativă este ca funcțiile din clasa Figura să fie declarate ca fiind funcții virtuale pure



# Clase abstracte

```
class Figura
{
public:
    virtual void Rotate(int) = 0; // pure virtual function
    virtual void Draw() const = 0; // pure virtual function
    virtual ~Figura(){}; //virtual
};
```

- ▶ O clasă cu cel puțin o funcție virtuală este o clasă abstractă și nu poate instanția nici un obiect.

```
Figura fig;
error C2259: 'Figura' : cannot instantiate abstract class
```

- ▶ O clasă abstractă se dorește a fi o interfață pentru obiectele accesate prin intermediul pointerilor și referințelor.
- ▶ O clasă abstractă trebuie să aibă un destructor virtual.
- ▶ Deoarece o clasă abstractă nu instanțiază obiecte, în mod uzual nu are constructori.



# Clase abstracte

---

- ▶ O clasă abstractă poate fi folosită doar ca interfață pentru alte clase:

```
class Punct
{
public:
    int x;
    int y;
    Punct():x(0),y(0){};
};

class Cerc : public Figura
{
private:
    Punct centru;
    int raza;
public:
    Cerc(): raza(0){};
    Cerc(Punct p, int r);
    void rotate(int) {};
    void draw() const {};
};
```

# Clase abstracte

---

- ▶ O funcție virtuală pură ce nu a fost definită în clasa derivată rămâne o funcție virtuală pură. Astfel clasa derivată este de asemenea abstractă.

```
class Poligon : public Figura //clasă abstractă
{
public:
    void draw()const {};
};
```

- ▶ O clasă abstractă furnizează o interfață fără a expune detaliile de implementare



# Constructori și deconstructori

---

- ▶ Constructori virtuali ? – **NU**

```
class Punct
{
public:
    int x;
    int y;
    virtual Punct():x(0),y(0){};
};

...
error C2633: 'Punct' : 'inline' is the only
legal storage class for constructors
```

- ▶ În cazul constructorilor, pentru crearea unui obiect este necesară cunoașterea exactă a tipului acestuia
- ▶ În plus, tabela virtuală nu a fost inițializată și deci nu se poate crea dacă nu se cunoaște tipul obiectului



# Constructori și destructori

---

- ▶ Destructorii claselor de bază trebuie declarați virtuali pentru a asigura apelarea destructorilor din clasele derivate
- ▶ În caz contrar există pericolul de a nu dealoca toată memoria utilizată
- ▶ În cazul destructorilor, se apelează toți destructorii dinspre clasa derivată către clasa de bază (în ordinea inversă a apelării constructorilor)



# Type casting

---

- ▶ Conversia unei expresii dintr-un anumit tip în alt tip este cunoscută sub numele de Type casting
- ▶ Conversia dintr-o clasă de bază către o clasă derivată este în mod uzuale denumită *downcast* datorită modului în care este reprezentată grafic ierarhia de clase.
- ▶ Conversia dintr-o clasă derivată către o clasă de bază este în mod uzuale denumită *upcast*.
- ▶ Similar, conversia de la o clasă derivată către o altă clasă derivată se numește *crosscast*.
- ▶ Conversia se realizează cu ajutorul operatorilor:
  - ▶ *dynamic\_cast <>*
  - ▶ *static\_cast <>*
  - ▶ *reinterpret\_cast <>*
  - ▶ *const\_cast <>*

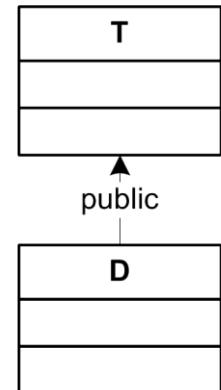


# Operatorul *dynamic\_cast*

- ▶ Operatorul *dynamic\_cast* are nevoie de doi operanzi: un tip de date încadrat între două paranteze unghiulare  $< T^* >$  și un pointer încadrat de două paranteze  $(p)$

*dynamic\_cast<T\*>(p)*

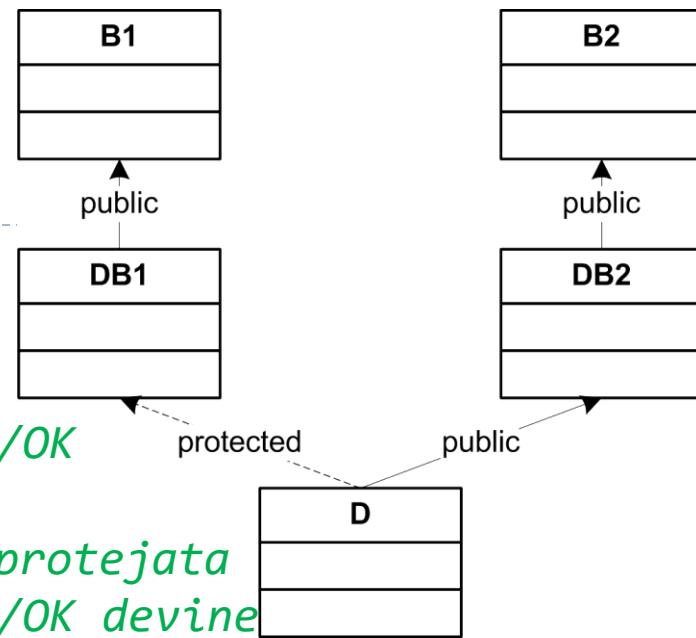
- ▶ Dacă  $p$  este de tipul  $T^*$  sau a unui tip  $D^*$  derivat din  $T$  atunci rezultatul este exact ca și cum s-ar atribui  $p$  unui  $T^*$ .
- ▶ Se observă că prezența operatorului *dynamic\_cast* nu este necesară. Totuși este bine de știut că operatorul *dynamic\_cast* nu permite accidental încălcarea regulei de protecție private sau protected a clasei de bază.



# Operatorul *dynamic\_cast*

```
void f(D* p)
{
    DB2* pdb2_1 = p;           //OK
    DB2* pdb2_2 = dynamic_cast<DB2*>(p); //OK

    DB1* pdb1_1 = p;   //eroare DB1 clasa protejata
    DB1* pdb1_2 = dynamic_cast<DB2*>(p); //OK devine
nullptr
```



- ▶ Odată ce *dynamic\_cast* folosit ca upcast este exact o simplă atribuire nu implică nici un overhead și este sensitiv contextului lexical.
- ▶ Scopul operatorului este de a fi utilizat în situațiile în care corectitudinea conversiei nu poate fi determinată de compilator.

# Operatorul *dynamic\_cast*

- ▶ Astfel operatorul *dynamic\_cast<T\*>(p)* analizează tipul obiectului pointat de *p*. Dacă acel obiect este de clasa *T* ori are o clasă unică *T*, *dynamic\_cast* returnează un pointer de clasă *T\**, altfel *nullptr*.
- ▶ Dacă *p* este *nullptr* atunci *dynamic\_cast<T\*>(p)* returnează *nullptr*.
- ▶ Operatorul *dynamic\_cast* necesită un pointer sau o referință către un tip polimorfic pentru downcast ori crosscast.

```
class B1
{
public:
    virtual void f(void){};
}

void g(B1* pb1)
{
    DB1* pdb1 = dynamic_cast<DB1*>(pb1); //OK
}
```

```
class DB1:public B1
{
public:
    void f(void){};
}
```

# Operatorul *dynamic\_cast*

```
class B2
{
public:
    void f(void){};
}

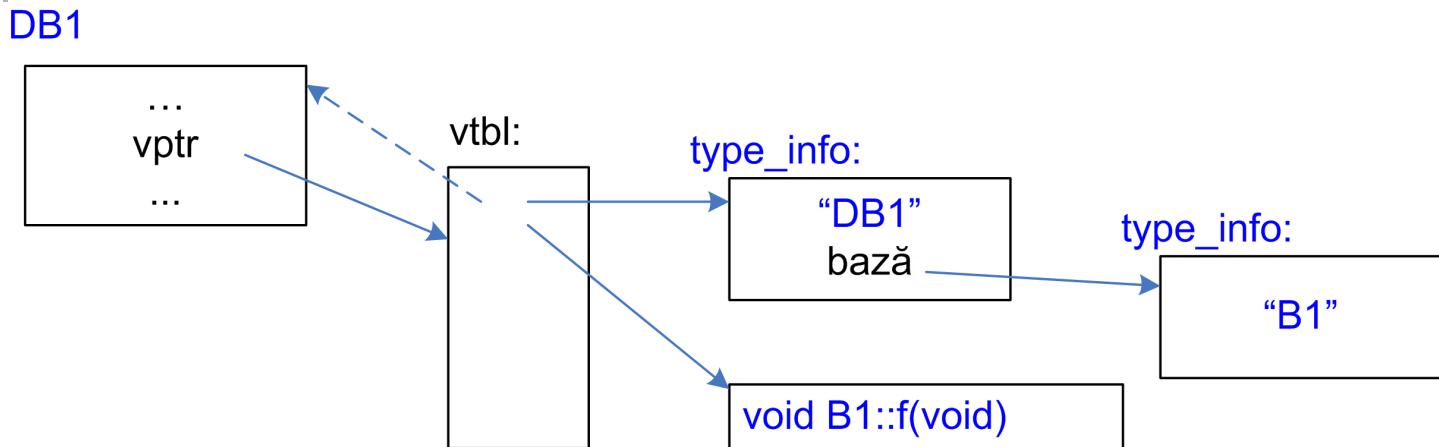
void h(B2* pb2)
{
    DB2* pdb2 = dynamic_cast<DB2*>(pb2); //eroare B2 nu este
                                                //polimorfic
}
```

```
class DB2:public B1
{
    //...
public:
    void f(void){};
}
```

- ▶ Necesitatea ca tipul pointerului să fie polimorfic simplifică implementarea operatorului *dynamic\_cast* deoarece există posibilitatea de atașare a „*type information object*” unui obiect plasând un pointer către *type information* în *vtbl*.



# Operatorul *dynamic\_cast*



- ▶ Săgeata punctată reprezintă un offset (informație) de regăsire a obiectului având un pointer către un subobiect polimorfic.
- ▶ Restricționarea data de utilizarea operatorului *dynamic\_cast* are sens din punct de vedere logic. Dacă un obiect nu are funcții virtuale nu poate fi manipulat în siguranță fără a se cunoaște exact tipul acestuia. Totodată trebuie avut grija ca un astfel de obiect să nu fie adus într-un context în care să nu i se cunoască tipul.

# Operatorul *dynamic\_cast*

---

- ▶ Tipul pointerului țintă a operatorului *dynamic\_cast* nu trebuie să fie polimorfic. Acest lucru permite ascunderea unui tip concret într-un tip polimorfic, trasmisia lui prin intermediul unui obiect al sistemului I/O și apoi despachetarea acestuia într-un tip concret.
- ▶ Se poate face un *dynamic\_cast* către *void\** cu scopul de a determina adresa de început a unui obiect polimorfic.

```
void h(B2* pb2, B1* pb1)
{
    void* pdb1 = dynamic_cast<DB1*>(pb1); //OK
    void* pdb2 = dynamic_cast<DB2*>(pb2); //eroare B2 nu este
                                                //polimorfic
```

- ▶ Aceste conversii sunt utile atunci când se interacționează cu funcții low level.



# dynamic\_cast la referință

---

- ▶ **Dynamic\_cast la o referință** este o aserțiune: „obiectul referit este de tipul respectiv”.

*dynamic\_cast<T&>(r)*

- ▶ Rezultatul operatorului ***dynamic\_cast*** este implicit testat de implementarea operatorului. Dacă operandul referit nu este de tip așteptat se aruncă o excepție ***bad\_cast***



# Operatorul *static\_cast*

---

- ▶ Utilizarea operatorului *dynamic\_cast* implică existența unui operator polimorfic deoarece nu există informație stocată într-un operator nonpolimorfic.
- ▶ Utilizarea operatorului *dynamic\_cast* implică un cost de timp, cu alte cuvinte sunt mii de linii de cod scrise ce trebuie executate până când *dynamic\_cast* este disponibil.
- ▶ Totodată *dynamic\_cast* nu poate converti dintr-un *void\**.
- ▶ Pentru acest lucru se utilizează *static\_cast*:

```
DB1* f1(void* p)
{
    B1* pb1 = static_cast<B1*>(p);
    return dynamic_cast<DB1*>(pb1);
}
```



# Operatorul *static\_cast*

---

- ▶ Realizează conversii între pointeri către clase înrudite (bază-derivată, derivată - bază). Acest lucru asigură faptul că cel puțin clasele sunt înrudite dacă un pointer corespunzător este convertit.

```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
```

- ▶ Nu efectuează nici o verificare la rulare asupra obiectului, deci nu se recomandă a fi folosit decât dacă știi ce faci.
- ▶ Necesară verificări suplimentare din partea programatorului pentru a fi sigur ca s-a efectuat cu succes conversia
- ▶ Poate fi folosit și la o conversie ce nu implică pointeri dar implică conversii implicite

```
double d=3.14159265;
int i = static_cast<int>(d);
```

# Operatorul *reinterpret\_cast*

---

- ▶ Convertește un pointer către orice tip de pointer, chiar dacă pointează către clase complet diferite (nerelaționate).
- ▶ Nu este verificat nici conținutul pointat nici tipul către care se pointează
- ▶ Se utilizează în cod low-level

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```



# Operatorul *const\_cast*

---

- ▶ Se utilizează în manipularea proprietății *const* a unui obiect

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

void Print (char * str)
{
    cout << str << endl;
}

int main (void)
{
    const char * c = "exemplu";
    Print ( const_cast<char *> (c) );
    return 0;
}
```



---

Vă mulțumesc !



# PROGRAMARE ORIENTATĂ PE OBIECTE

Curs 9

*Template*

Template

# Necesitatea parametrizării tipului de date

---

- ▶ Fie funcția ce realizează suma elementelor unui vector de elemente de tip double

```
double Sum0(double* a, int n)
{
    double res = 0;

    for(int i = 0; i < n; ++i)
    {
        res += a[i];
    }
    return res;
}
```



# Necesitatea parametrizării tipului de date

---

- ▶ Fie un vector cu elemente de tip float. Poate fi utilizată funcția *Sumθ* pentru suma elementelor?

```
double a[10];
float b[10];
// ...
double sa = sumθ(a, 10); // OK
float sb = sumθ(b, 10); // eroare de compilare
```

- ▶ Soluție clasică:

```
float Sumθ(float* a, int n)
{
    float res = 0;

    for(int i = 0; i < n; ++i)
    {
        res += a[i];
    }
    return res;
}
```

- ▶ Cum se poate generaliza această funcție ?

# Necesitatea parametrizării tipului de date

---

- ▶ Pentru majoritatea tipurilor de date utilizate ar trebui definită câte o funcție
  - ▶ Spațiul ocupat de program crește
- 
- ▶ Posibilitatea de a trimite tipul de date ca parametru ar permite existența unei singure funcții pentru suma elementelor unui vector

```
template<typename T>
T sum1(T* a, int n)
{
    T res = 0;
    for(int i = 0; i < n; ++i)
    {
        res += a[i];
    }
    return res;
}
```



# Necesitatea parametrizării tipului de date

---

```
double a[10];
float b[10];
int c[10];
// ...
double sa = sum1(a, 10); // OK
float sb = sum1(b, 10); // OK
int sc = sum1(c, 10); // OK
```



# Introducere

---

- ▶ Template-urile (șabloane) furnizează suport pentru programare generică utilizând tipurile de date ca parametri
- ▶ Mecanismul template al limbajului C++ permite ca un tip de date sau o valoare să fie un parametru în definirea unei clase, funcții sau alias.
- ▶ Template-urile furnizează o cale directă de a reprezenta o gamă largă de concepte generale și căi simple de a le combina.
- ▶ Rezultatul din punct de vedere al claselor și funcțiilor poate fi un cod mai puțin general la run-time și eficient din punct de vedere al spațiului.
- ▶ Template-urile au fost introduse având ca scop proiectarea, implementarea și utilizarea librăriei standard.
- ▶ Librăria standard cere un mare grad de generalitate, flexibilitate și eficiență.



# Introducere

---

- ▶ În consecință, tehniciile care pot fi utilizate în proiectarea și implementarea libăriei standard sunt eficace și eficiente în proiectarea soluțiilor pentru o mare varietate de probleme.
- ▶ Aceste tehnici permit unui programator să ascundă implementări sofisticate în spatele unei simple interfețe și să expună complexitatea utilizatorului atunci când acesta are nevoie și cere acest lucru.



# Un simplu template string

---

- ▶ Se consideră un sir de caractere definit ca o clasă ce memorează caractere și furnizează operații precum căutare, comparare, concatenare.
- ▶ Se dorește ca acest comportament să poate fi aplicat mai multor tipuri de caractere: char, unsigned char, caractere chinezești, caractere grecești etc.
- ▶ Cu alte cuvinte se dorește reprezentarea noiiunii de „sir de caractere” cu o dependență cât mai mică de un specific tip de caracter



# String template

---

- ▶ Fie clasa String:

```
class String
{
    char *ptr;
    int sz;
public:
    String() : sz(0), ptr(0) {};
    explicit String(const char*p);
    String(const String&);

    char& operator[](int n) { return ptr[n]; }

    String& operator+=(char c);
    String& operator=(const String&);

    ~String() { if (ptr) delete[] ptr; }
};
```

- ▶ Clasa gestionează siruri de caractere de tip char.
- ▶ Pentru a deveni o clasă generală se parametrizează tipul de date



# String template

```
template<typename T>
class String {
    T *ptr;
    int sz;
public:
    String() : sz(0), ptr(0) {};
    explicit String(const T*p);
    String(const String&);
    T& operator[](int n) { return ptr[n]; }
    String& operator+=(T c);
    String& operator=(const String&);
    String(String&& x);
    ~String() { if (ptr) delete[] ptr; }
};
```

- ▶ Prefixul `template<typename T>` specifică faptul că un template este declarat și că tipul argument `T` va fi folosit în declarație
- ▶ După introducere, tipul `T` este utilizat ca oricare alt tip de date

# String template

---

- ▶ Domeniul tipului de date  $T$  se extinde până la finalul declarației prefixate de `template<typename T>`.
- ▶ Poate fi utilizat și prefixul echivalent `template<class T>`
- ▶ În ambele cazuri  $T$  este un nume de tip de date și nu numele unei clase.
- ▶ Utilizare:  
`String<char> cs;`  
`String<unsigned char> us;`  
`String<wchar_t> ws;`  
  
`struct jchar { /* ... */ }; // caractere nipone`  
  
`String<jchar> js;`
- ▶ Cu excepția sintaxei speciale a numelui, obiectul `String <char> cs` are aceeași funcționalitate ca cea definită anterior.
- ▶ Făcând `String` un template permite furnizarea facilitărilor implementate pentru orice tip de caracter

# String template

---

- ▶ Libraria standard oferă clasă template pentru manipularea sirurilor de caractere sub numele de *basic\_string* similar clasei templetizate prezentate în curs.
- ▶ În librăria standard, *string* este un sinonim pentru *basic\_string<char>*.

```
using string = std::basic_string<char>;
```

- ▶ Astfel pentru secvența *basic\_string<char> sir*; poate fi utilizat *string sir*;
- ▶ În general aliasurile sunt utile pentru evita denumirile lungi ale claselor generate. De asemenea se preferă a nu se cunoaște detaliile despre cum un tip este definit. Un alias permite ascunderea faptului că un tip este generat dintr-un template



# Definirea unui template

---

- ▶ O clasă generată dintr-un template este o clasă normală. Din acest motiv utilizarea unei clase template nu implică nici un mecanism run-time comparativ cu o clasă normală.
- ▶ Utilizarea template-urilor conduce la o descreștere a codului generat deoarece codul pentru o funcție membră a unei clase template este generată dacă acel membru este utilizat.
- ▶ Înainte de a proiecta o clasă template este preferabil de a proiecta, testa, depana o clasă concretă (String). Astfel o serie de erori de implementare pot fi eliminate în contextul exemplului concret.
- ▶ Pentru a putea înțelege generalitatea unui template se va imagina comportarea acestuia pentru un timp concret de date.
- ▶ Pe scurt: o componentă generică este dezvoltată ca o generalizare a unui sau mai multe exemple concrete și nu pornind de la principii



# Definirea unui template

---

- ▶ Membri unei clase template sunt declarați și definiți exact ca la clasele non-template
- ▶ Totuși când un membru este declarat în afara clasei trebuie în mod explicit declarat template-ul

```
template<typename T>
String<T>::String() : sz{0}, ptr{ch}
{
    ch[0] = {};
}
```

```
template<typename T>
String& String<T>::operator+=(T c)
{
    // ... concatenare
    return *this;
}
```



# Definirea unui template

---

- ▶ Parametrul template  $T$  este un parametru și nu un nume pentru un tip specific. În domeniul  $\text{String}\langle T \rangle$ , identificatorul  $\langle T \rangle$  este numele template-ului. Astfel numele constructorului este  $\text{String}\langle T \rangle::\text{String}$ .
- ▶ Nu este posibilă supraîncărcarea numelui unei clase template.

```
template<typename T>
class String { /* ... */ };
```

- ▶ Un tip utilizat ca template trebuie să furnizeze interfața așteptată de template  $\text{String}$ ; // eroare



# Instanțierea unui template

---

- ▶ Procesul de generare a unei clase ori funcții dintr-o listă de argumente template este denumit adesea *template instantiation*. O versiune a template-ului pentru o listă specifică de argumente este numită **specializare**.

```
String<char> cs;  
  
void f(void)  
{  
    String<jchar> js;  
    cs = "Abcd efg hgi";  
}
```

- ▶ Pentru secvența de mai sus compilatorul generează declarația claselor *String<char>* și *String<jchar>*, pentru fiecare în parte destructori și constructori cu listă vidă de parametri și funcția operator *String<char>::operator=(char\*)*
- ▶ Alte funcții membre nefiind folosite nu sunt generate.



- 
- ▶ Evident, template-urile furnizează o modalitate de a genera foarte mult cod din relativ mici definiții. În consecință o atenție sporită trebuie avută pentru a nu umple memoria cu definiții de funcții aproape identice.

# Derivare

```
template<typename T>
class B
{
    /* ... */
};
```

```
template<typename T>
class D : public B<T>
{
    /* ... */
};
```

```
template<typename T> void f(B<T>*);  
  
void g(B<int>* pb, D<int>* pd)  
{  
    f(pb); // f<int>(pb)  
    f(pd); // f<int>(static_cast<B<int>*>(pd));  
        // conversie standard D<int>* to B<int>*  
}
```

# Sfaturi

---

- ▶ A se utiliza template-uri pentru
  - ▶ a implementa algoritmi ce se adresează mai multor tipuri de argumente
  - ▶ a implementa containere
- ▶ Cele două declarații `template<typename T>` și `template<class T>` sunt sinonime
- ▶ Înainte de a proiecta un template se va proiecta și implementa o versiune nontemplate. Ulterior se va generaliza parametrizând tipul/tipurile de date
- ▶ O funcție virtuală nu poate fi o funcție template membră
- ▶ A se supraîncărca funcțiile template pentru a se obține aceeași semantică pentru mai multe tipuri de argumente ()
- ▶ A se utiliza aliasuri de template-uri pentru a simplifica notația și a ascunde detaliile de implementare
- ▶ A se include definițiile template în orice fisier sursă unde este necesar

---

Vă mulțumesc !



# PROGRAMARE ORIENTATĂ PE OBIECTE

Curs 10  
*UML*



# Software academic și industrial

---

- ▶ Exemplu: O problemă de 10000 LOC
  - ▶ Student: 2 luni (5000 LOC / lună)
  - ▶ Firmă: 1000 LOC/lună, 10 luni-om
  - ▶ Firma embedded systems: 100 LOC/lună
- ▶ Software academic
  - ▶ Versiune demo funcțională
  - ▶ Hobby: nu necesită documentație și interfață complexă cu utilizatorul, defectele sunt corectate când apar
- ▶ Software industrial
  - ▶ Plătit de client
  - ▶ Influențează mediul de afaceri
  - ▶ Necesită o abordare inginerescă
- ▶ Toată lumea consideră importantă disciplina de lucru, dar nuexistă un acord general asupra modalităților de aplicare



# Ingineria programării

---

- ▶ Ingineria programării reprezintă aplicarea unei abordări sistematice, disciplinare și cuantificabile pentru dezvoltarea, operarea și întreținerea produselor software
- ▶ (*Glosarul terminologiei ingineriei programării, IEEE, Institute of Electrical and Electronics Engineers, 1990*)



# Software industrial

---

- ▶ Este construit pentru a rezolva unele probleme din organizația clientului
  - ▶ Funcționarea incorectă poate provoca pierderi financiare și chiar pierderea de vieți omenești
- ▶ Trebuie să aibă calitate foarte bună
  - ▶ Testare riguroasă înainte de livrare (30%-50% din efortul total)
  - ▶ Dezvoltarea este împărțită pe faze pentru a corecta defectele din vreme (necesită documentație)
- ▶ Are cerințe de recuperare a datelor, toleranță la defecte, portabilitate
  - ▶ Acestea conduc la creșterea dimensiunilor



# Regula lui Brooks

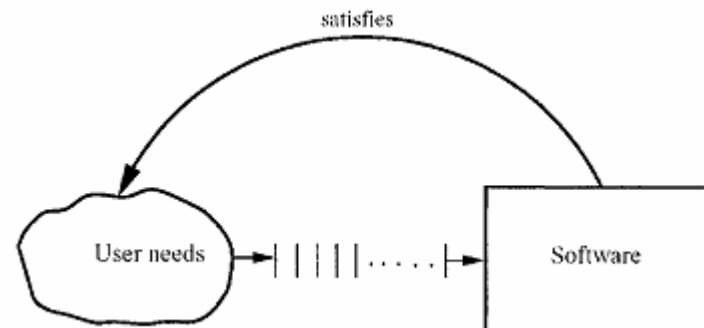
---

- ▶ Software-ul industrial în comparație cu software-ul academic
  - ▶ Productivitate: 1 / 5
  - ▶ Dimensiune dublă
- ▶ Software-ul industrial necesită de 10 ori mai mult efort decât software-ul academic



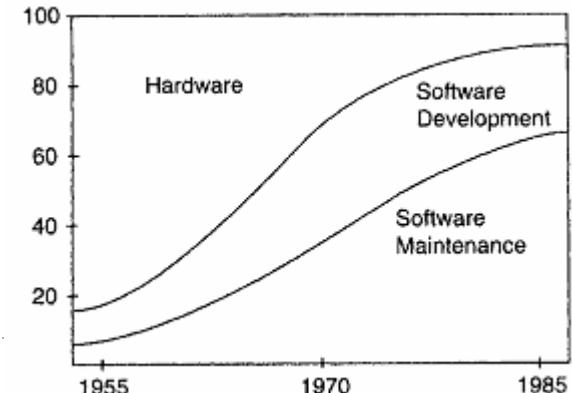
# Scopul ingineriei programării

- ▶ Utilizarea unor metodologii pentru dezvoltarea de software
  - ▶ Rezultate repetabile
  - ▶ Apropiere de știință
  - ▶ Îndepărțarea de metodele ad-hoc cu rezultate imprevizibile
- ▶ Scopul dezvoltării de software este satisfacerea nevoilor clienților sau utilizatorilor



# Costul produselor software

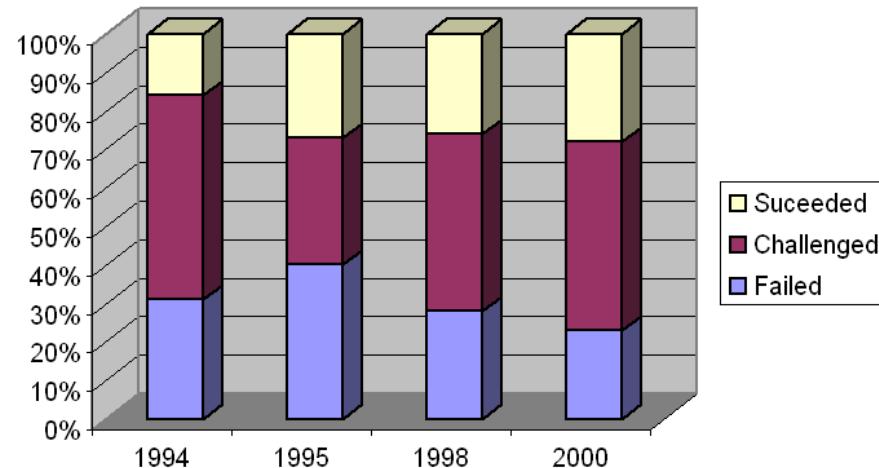
- ▶ Costul software-ului depinde în primul rând de efortul oamenilor
- ▶ Productivitatea este frecvent măsurată în linii de cod (LOC) / lună-om
- ▶ Productivitatea medie pentru o aplicație nouă este de 300-1000 LOC / lună-om
  - ▶ 8000 \$ / lună - 8-25 \$ / LOC
  - ▶ Un program mediu de 50.000 LOC poate costa aproximativ 1.000.000 \$



# Întârzieri și instabilitate

- ▶ Din 600 de firme, 35% aveau proiecte informaticе scăpate de sub control din punct de vedere al bugetului și timpului de execuție
- ▶ Raportul Standish Group privind finalizarea proiectelor IT în SUA

	<b>Failed</b>	<b>Challenged</b>	<b>Succeeded</b>
<b>1994</b>	31%	53%	16%
<b>1995</b>	40%	33%	27%
<b>1998</b>	28%	46%	26%
<b>2000</b>	23%	49%	28%



# Lipsa de incredere

---

- ▶ Software care:
  - ▶ Nu face ce trebuie
  - ▶ Face ce nu trebuie
- ▶ În sisteme complexe (incluzând componente electrice, mecanice, hidraulice), de cele mai multe ori software-ul este problema cea mai mare
- ▶ Defectele software-ului nu se datorează uzurii, ci erorilor de proiectare și implementare



## Defecte „celebre”

---

- ▶ 28 iulie 1962 – sonda spațială Mariner I
- ▶ 1982 – conducta sovietică de gaz trans-siberiană
- ▶ 1983 – sistemul sovietic de avertizare nucleară
- ▶ 1985-1987 – acceleratorul medical Therac-25
- ▶ 1988-1996 – generatorul de numere aleatorii al protocolului Kerberos



# Reprogramarea

---

- ▶ Cerințele nu sunt specificate complet
  - ▶ Schimbarea lor conduce la refacerea tuturor fazelor ulterioare
- ▶ Pentru proiecte cu durată lungă, cerințele clientului se modifică
- ▶ Reprogramarea consumă 30%-40% din efortul total de dezvoltare



# Intretinerea

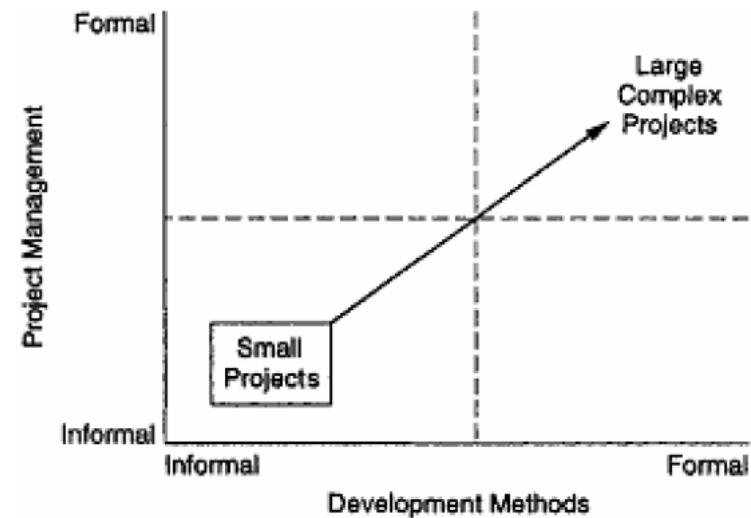
---

- ▶ Întreținere corectivă
  - ▶ Eliminarea erorilor
- ▶ Întreținere adaptivă
  - ▶ Includerea de funcționalități suplimentare
  - ▶ Legea evoluției software-ului: software-ul schimbă mediul, apoi trebuie să se adapteze la noul mediu
- ▶ Întreținerea costă de obicei mai mult decât dezvoltarea unei aplicații
  - ▶ Presupune înțelegerea codului, modificarea, testarea de regresiune
- ▶ În timpul dezvoltării, întreținerea este deseori neglijată
- ▶ Raport de cost 60:40 – 80:20



# Scala proiectelor

- ▶ Proiectele complexe necesită metode diferite de dezvoltare față de proiectele de mici dimensiuni
  - ▶ Presupun formalizarea procedurilor ingineresci și a managementului de proiect
- ▶ Proiecte:
  - ▶ Mici: < 10 KLOC
  - ▶ Medii: 10-100 KLOC
  - ▶ Mari: 100-1000 KLOC
  - ▶ Foarte mari: peste 1 milion LOC



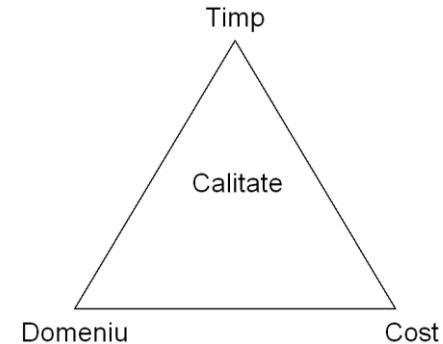
# Scala proiectelor

- ▶ Previziuni:
  - ▶ 1946: Goldstein, von Neumann – max. 1000 instrucțiuni
  - ▶ 1981: Bill Gates – max. 640 KB RAM
- ▶ Realitatea:
  - ▶ 1966: sistemul de operare IBMOS 360: 5000 de ani-om
  - ▶ 1977: naveta spațială NASA: cc. 40 milioane LOC
  - ▶ 1983: System V versiunea 4.0 Unix: 3,7 milioane LOC
  - ▶ 1992: sistemul de rezervare abiletelor KLM: 2 milioane LOC în limbaj de asamblare

Size (KLOC)	Software	Languages
980	gcc	ansic, cpp, yacc
320	perl	perl, ansic, sh
305	teTeX	ansic, perl
200	openssl	ansic, cpp, perl
200	Python	python, ansic
100	apache	ansic, sh
90	cvs	ansic, sh
65	sendmail	ansic
60	xfig	ansic
45	gnuplot	ansic, lisp
38	openssh	ansic
30,000	Red Hat Linux	ansic, cpp
40,000	Windows XP	ansic, cpp

# Triunghiul managementului de proiect

- ▶ Ingineria programării este condusă de 3 factori majori: costul, timpul și domeniul de aplicare (întinderea, anvergura)



- ▶ **Costul este o măsură a resurselor utilizate pentru sistem**
  - ▶ În cazul software-ului este dominat de costul de personal
  - ▶ Costul poate fi estimat ca efort (luni-om) \* cost mediu (lunar)
  - ▶ Include costul suplimentar pentru hardware și instrumentele de dezvoltare
- ▶ **Timpul**
  - ▶ Mediul de afaceri dorește reducerea timpului de livrare
  - ▶ Creșterea productivității (KLOC / lună-om) determină scăderea timpului și a costului
- ▶ **Domeniul (engl. “scope”) este dat de specificații**

# Calitatea

---

- ▶ **Calitatea presupune, conform standardului internațional al calității produselor software:**
  - ▶ **Funcționalitate (functionality)**
    - ▶ Asigurarea funcțiilor care satisfac nevoile exprimate explicit sau implicate
    - ▶ Include *securitatea: persoanele neautorizate să nu aibă acces iar celor autorizate să nu le fie refuzat accesul*
  - ▶ **Încredere (reliability)**
    - ▶ Menținerea unui nivel specificat de performanță
  - ▶ **Utilizabilitate (usability)**
    - ▶ Capacitatea de a fi înțeles, învățat și utilizat
  - ▶ **Eficiență (efficiency)**
    - ▶ Asigurarea unor performanțe adecvate relativ la volumul de resurse utilizate
  - ▶ **Mantenabilitate (maintainability)**
    - ▶ Capacitatea de a fi modificat pentru corecții, îmbunătățiri sau adaptări
  - ▶ **Portabilitate (portability)**
    - ▶ Capacitatea de a fi adaptat pentru medii diferite exclusiv pe baza mijloacelor existente înprodus



# Calitatea

---

- ▶ Importanța fiecărei dimensiuni depinde de natura proiectului
  - ▶ Sistem critic: încredere
  - ▶ Joc: utilizabilitate
- ▶ Înainte de dezvoltare, trebuie specificat obiectivul principal de calitate
- ▶ Încrederea este considerată în general cea mai importantă
  - ▶ Se măsoară în defecte / KLOC
  - ▶ Bunele practici curente: mai puțin de 1 defect / KLOC
  - ▶ Definirea unui defect depinde de proiect sau de standardele organizației dezvoltatoare



# Consecvență și repetabilitate

---

- ▶ Succesele trebuie să fie repetabile
  - ▶ Calitatea și productivitatea trebuie să fie consecvente
- ▶ Acest lucru permite unei organizații:
  - ▶ Să prevadă cu acuratețe rezultatele proiectelor
  - ▶ Să își îmbunătățească procesele de dezvoltare
- ▶ Se impune standardizarea unor proceduri și folosirea unor metodologii



# Concluzii

---

- ▶ Ingineria programării este o colecție de metode și recomandări pentru dezvoltarea eficientă de programe de mari dimensiuni
- ▶ Software-ul nu este doar o mulțime de programe, ci include documentația și datele asociate
- ▶ Fazele fundamentale ale dezvoltării programelor sunt: analiza, proiectarea, implementarea și testarea
- ▶ Istoria sistemelor de calcul și a programării se întinde pe aproape 70 de ani iar dezvoltarea prezentă a domeniului este fără precedent



# Limbajul unificat de modelare UML

---

- ▶ 1. Modelarea
- ▶ 2. Limbajul unificat de modelare
- ▶ 3. Clasificarea diagramelor UML 2.0
- ▶ 4. Diagramele UML 2.0
- ▶ 5. Concluzii



# Modelarea

---

- ▶ Un model este o simplificare a unui anumit sistem, care permite analizarea unei din proprietățile acestuia
  - ▶ Reține caracteristicile necesare
- ▶ Folosirea de modele poate încorda abordarea problemelor complexe, facilitând comunicarea și înțelegerea
  - ▶ Divide et impera
- ▶ Exemple:
  - ▶ Formalismul matematic
  - ▶ Reprezentările din fizică
- ▶ Orice limbaj „intern” poate fi folosit pentru modelare, însă într-un context formal este nevoie de standardizare



# Limbajul unificat de modelare, UML

---

- ▶ Limbaj pentru specificarea, vizualizarea, construirea și documentarea elementelor sistemelor software
  - ▶ Un limbaj grafic care ne permite să reproducem „pe hârtie” ceea ce este produs în procesul dezvoltării a unui sistem software
  - ▶ Poate fi folosit și pentru alte sisteme, cum ar fi procesele de afaceri (business processes)



# Versiuni și standardizare

---

- ▶ Ianuarie 1997: UML 1.0 a fost propus spre standardizare în cadrul OMG (Object Management Group)
- ▶ Noiembrie 1997: versiunea UML 1.1 a fost adoptată ca standard de către OMG
- ▶ Martie 2003: a fost lansată versiunea 1.5
- ▶ Octombrie 2004: versiunea 2.0
- ▶ August 2011: versiunea 2.4.1
- ▶ UML este standardul ISO/IEC 19501:2005



# UML

---

- ▶ Ca orice limbaj, UML are:
  - ▶ Notații (alfabetul de simboluri)
  - ▶ Sintaxă și gramatică (reguli pentru combinarea simbolurilor)
- ▶ UML este un instrument de comunicare
- ▶ UML nu este o metodologie de dezvoltare
  - ▶ Dar este determinat de cele mai bune practici îndomeniu



# Clase de diagrame

---

## ▶ **Diagrame de structură**

- ▶ Prezintă elementele unei specificații independent de timp
- ▶ Includ: diagramele de clase, structuri compuse, componente, desfășurare (deployment), obiecte și pachete

## ▶ **Diagrame de comportament**

- ▶ Prezintă trăsăturile comportamentale ale sistemului
- ▶ Includ: diagramele de activități, mașini de stare și cazuri de utilizare, precum și cele 4 diagrame de interacțiune

## ▶ **Diagrame de interacțiune**

- ▶ Scot în evidență interacțiunile dintre obiecte
- ▶ Includ: diagramele de secvențe, comunicare, interacțiuni generale (interaction overview) și cronometrare (timing)



# Diagrame de structura

---

## ▶ Ce contine sistemul:

- ▶ Clase
- ▶ *Structuri compuse*
- ▶ Componente
- ▶ Desfășurare
- ▶ Obiecte
- ▶ *Pachete*



# Diagrame de comportament

---

- ▶ Ce se intampla in sistem

- ▶ Activități
- ▶ Mașini de stare
- ▶ Cazuri de utilizare



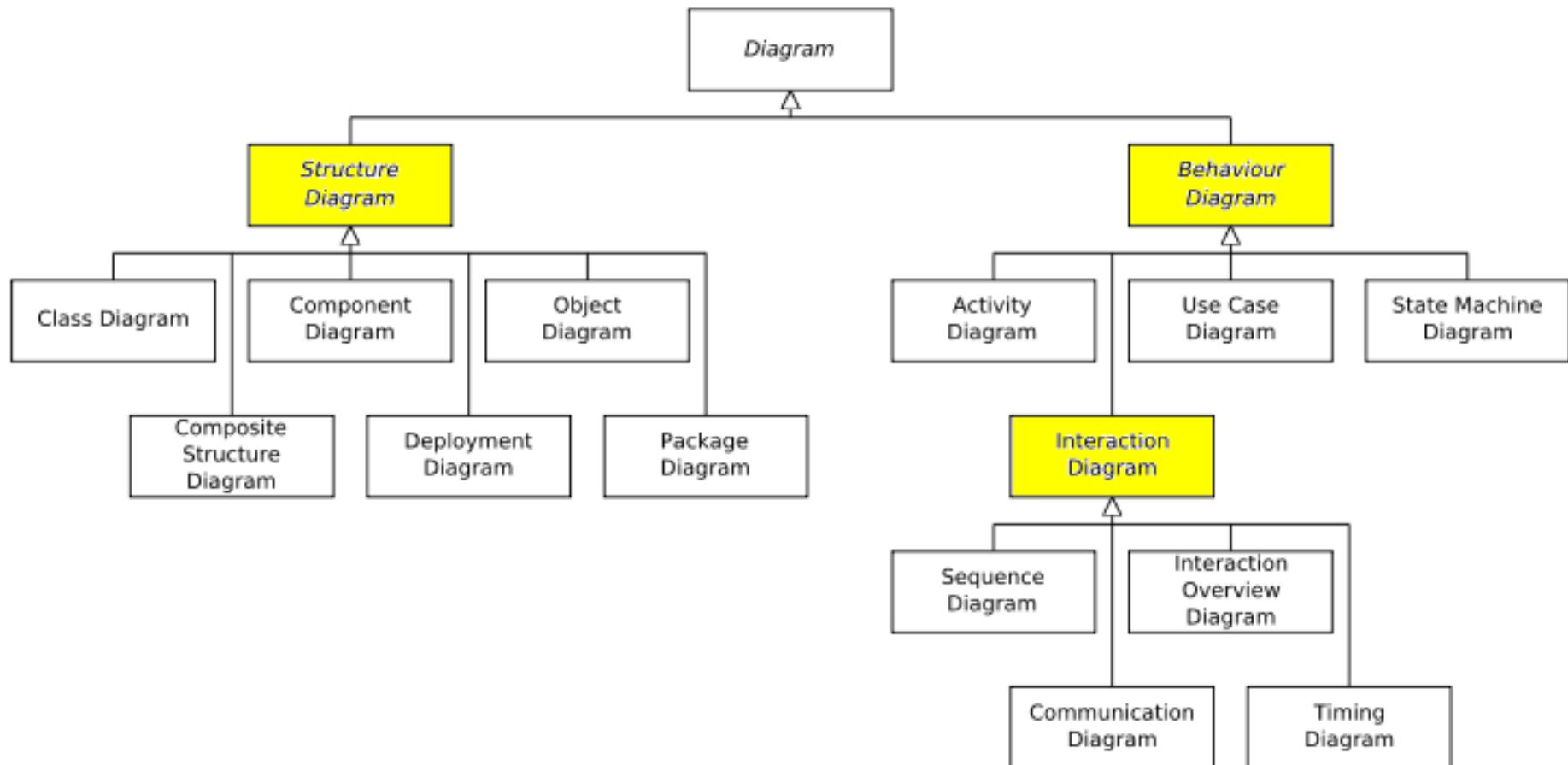
# Diagrame de interacțiune

---

- ▶ Fluxurile de control și date dintre componentele sistemului
  - ▶ Secvențe
  - ▶ Comunicare
  - ▶ *Interacțiuni generale*
  - ▶ *Cronometrare*

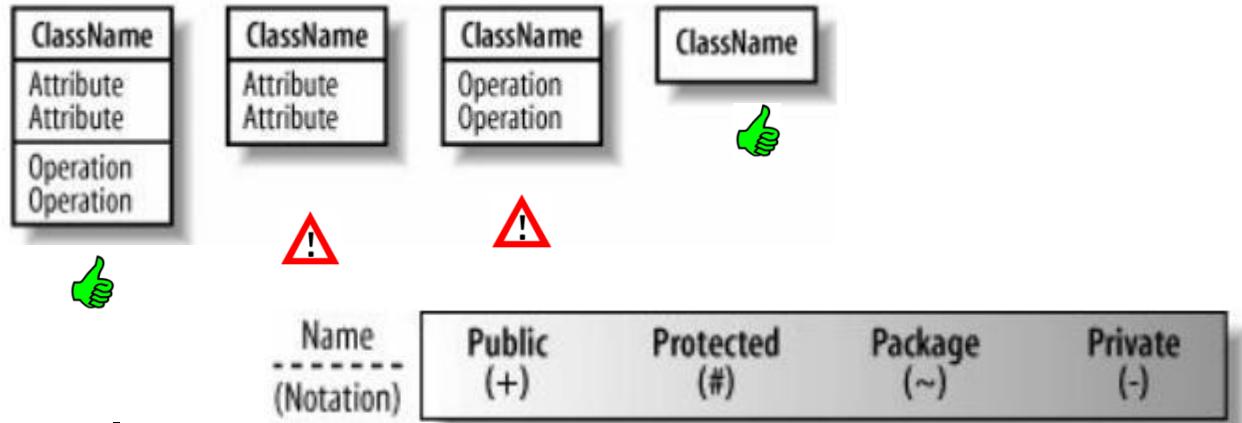


# Diagramme UML 2.0



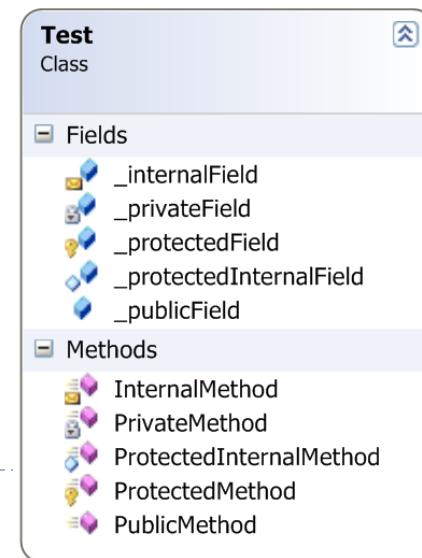
# Diagrama de clase

## ▶ Clasa



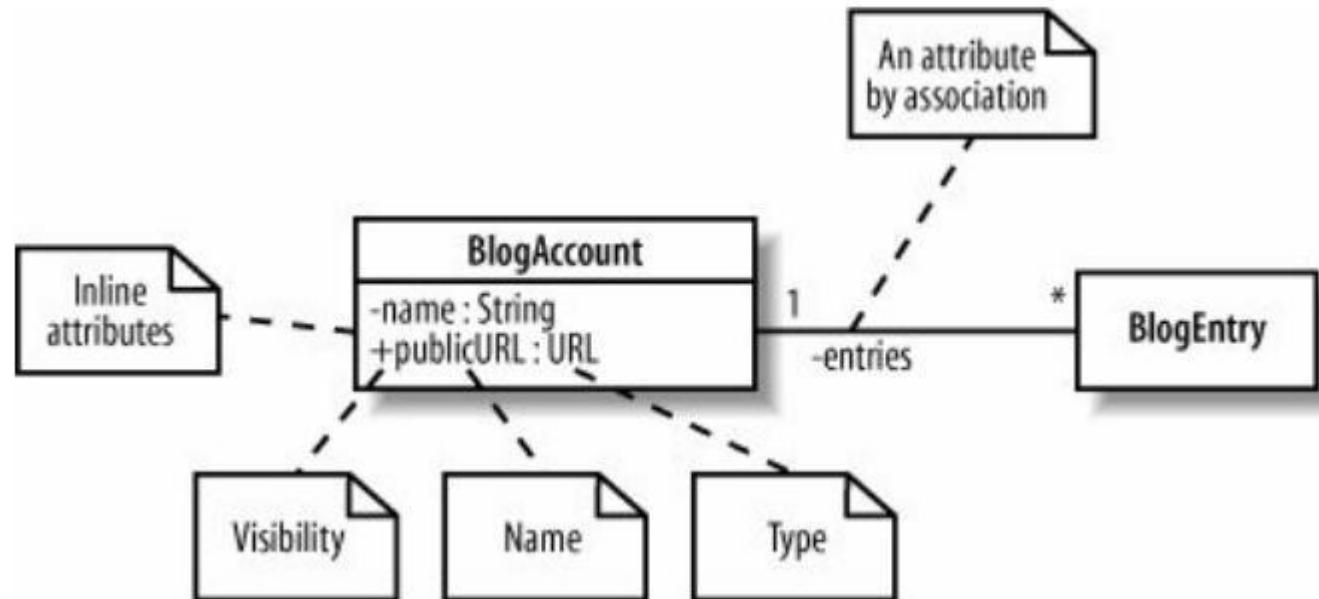
## ▶ Vizibilitatea trasaturilor

- ▶ Public
- ▶ Protejat
- ▶ Pachet
- ▶ Privat

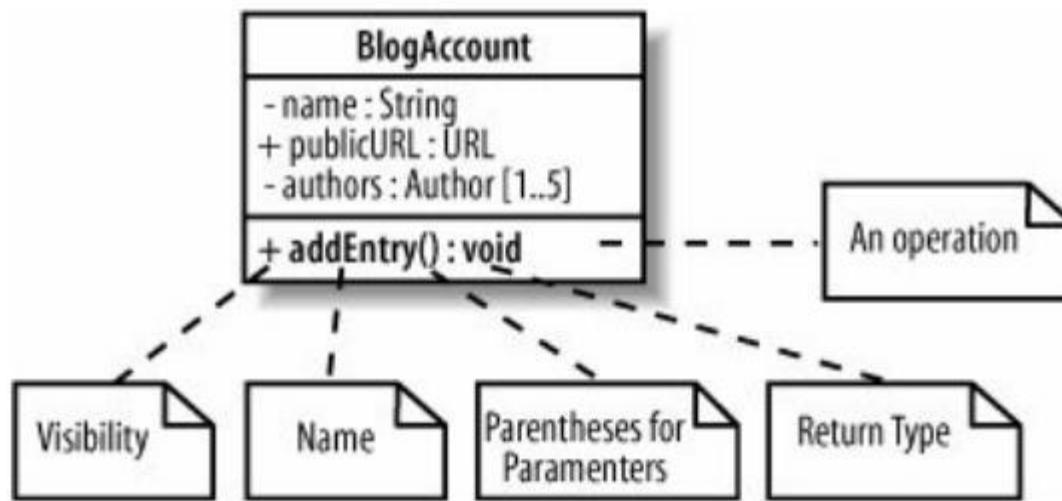


# Reprezentarea atributelor

- ▶ În interorul clasei (inline)
- ▶ Prin asociere

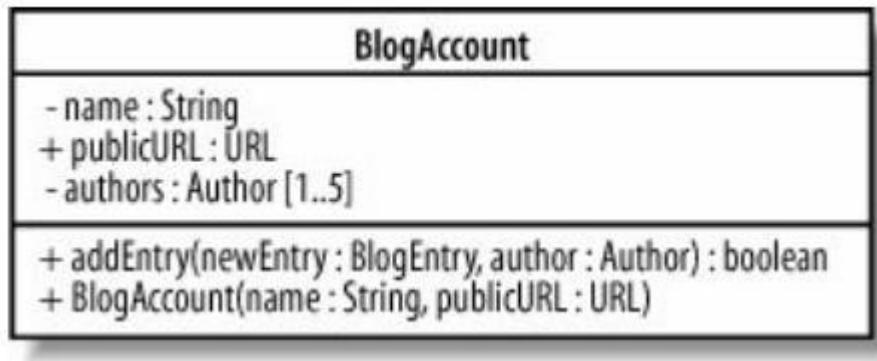
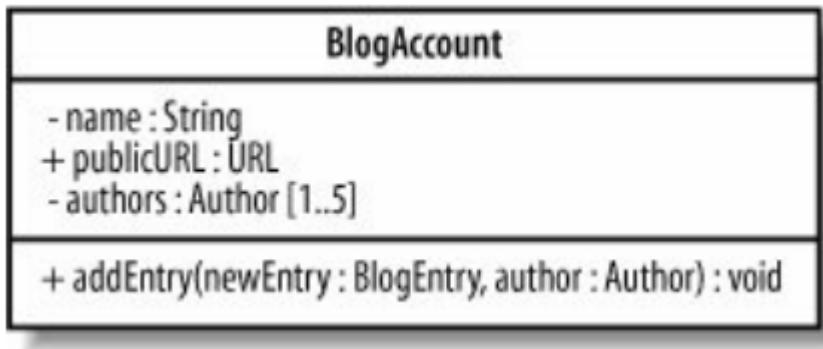


# Operatii



# Parametri sau tipurile de return

---



# Trasaturi statice

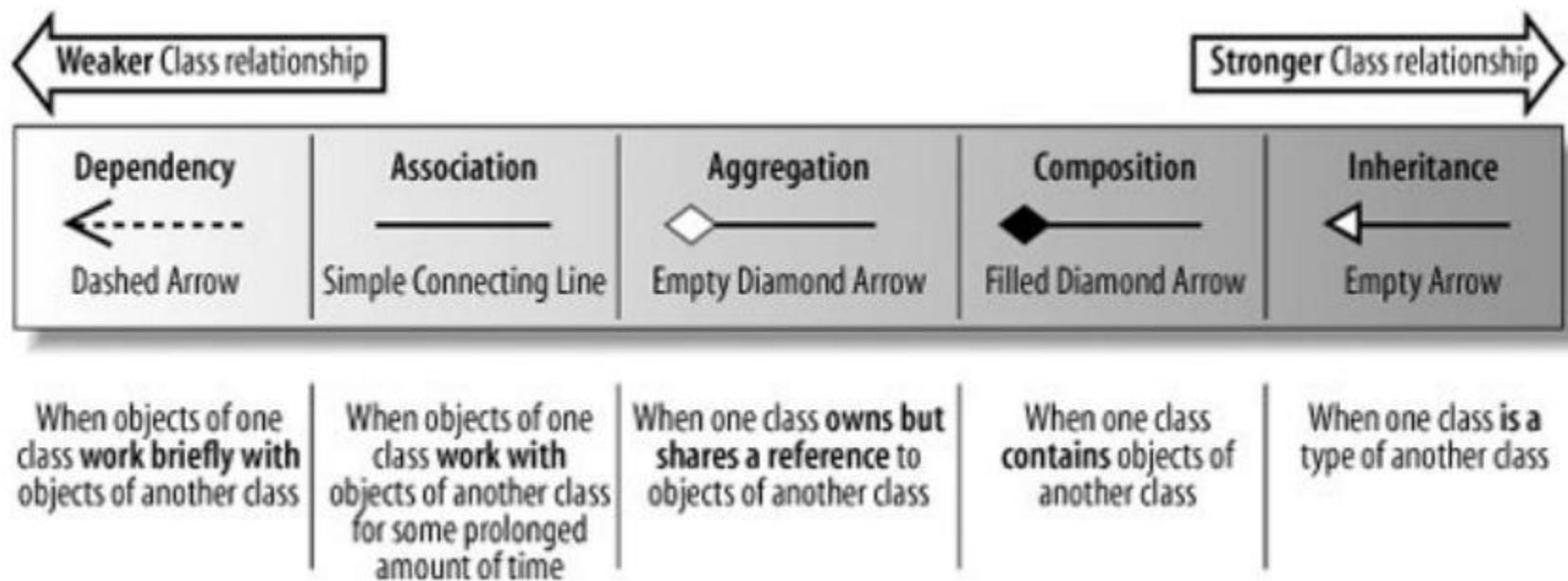
---

- ▶ Trăsături (*features*) = *atribute și operații*
- ▶ Trăsăturile statice se subliniază

Math
+ Abs(val : double) : double
+ Sin(angle : double) : double
+ Exp(val : double) : double

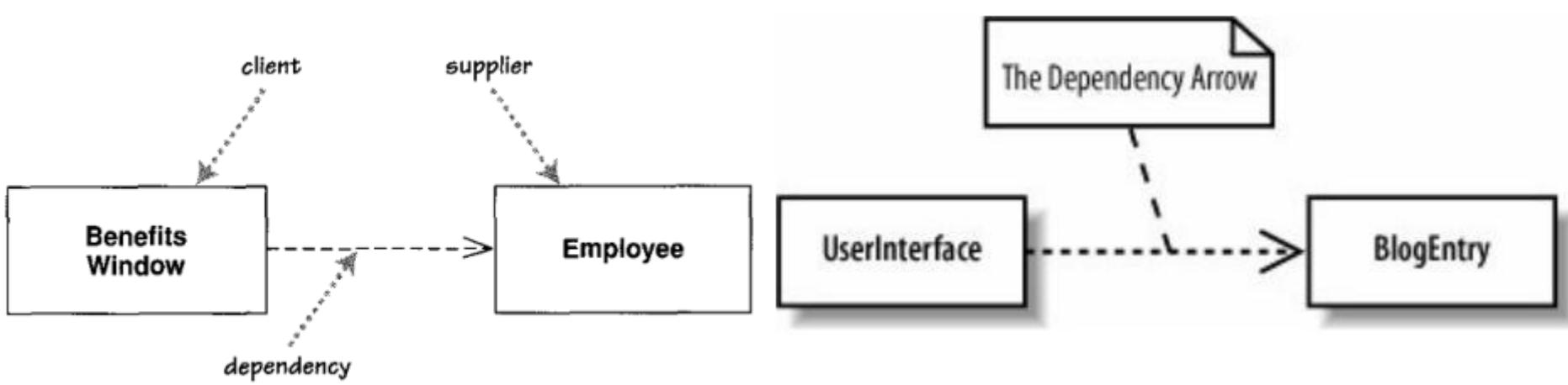


# Relatii intre clase



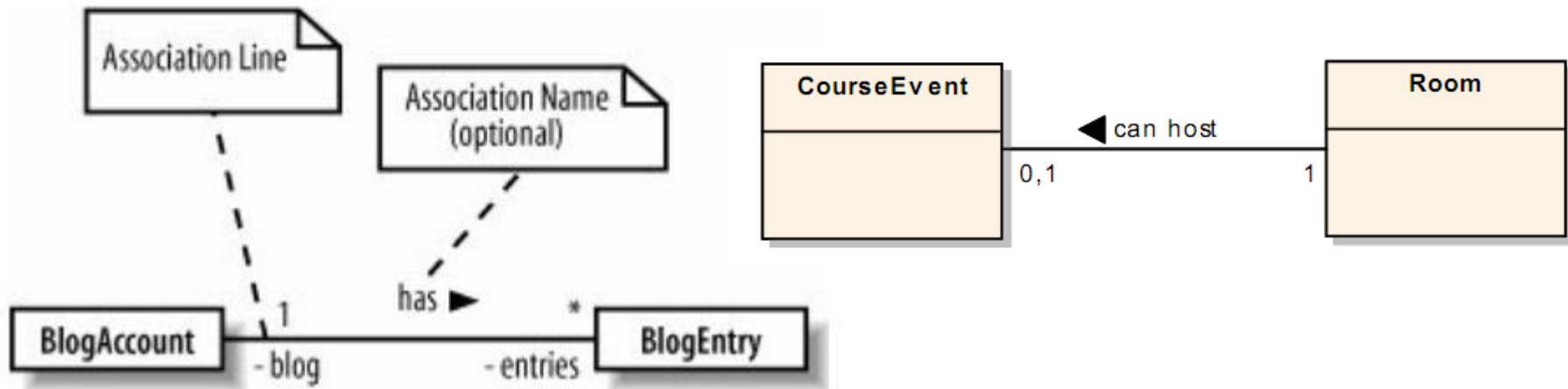
# Dependenta

- ▶ O clasă folosește pentru scurt timp o altă clasă
  - ▶ Exemplu: trimiterea unui mesaj - metodele clasei Math
- ▶ Din punct de vedere al implementării:
  - ▶ Instanțierea unei clase într-o metodă
  - ▶ Primirea unui obiect ca parametru într-o metodă
  - ▶ Crearea și returnarea unui obiect dintr-o metodă



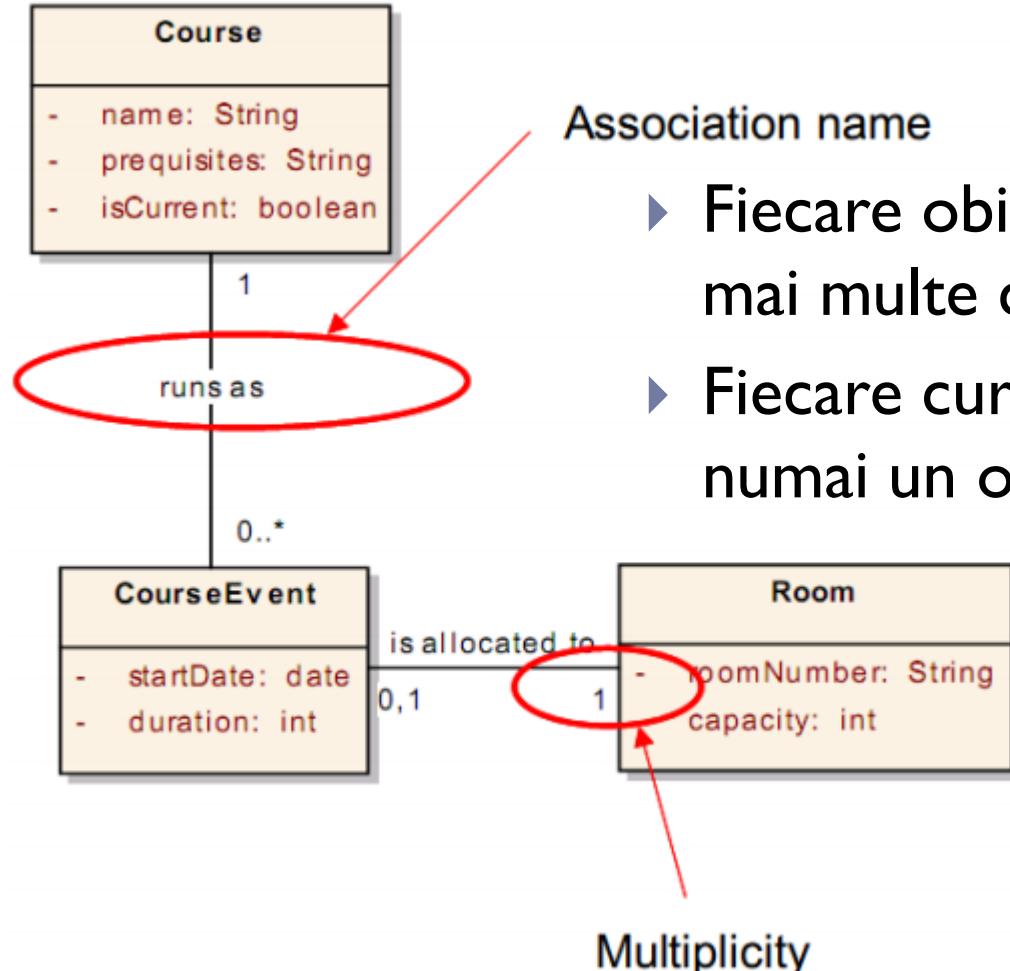
# Asocierea

- ▶ O clasă folosește un timp îndelungat o altă clasă
- ▶ De obicei, o clasă are un câmp instantiat din cealaltă clasă



- ▶ Directia de citire este de obicei de la stanga la dreapta și de sus în jos
- ▶ Directia de citire se poate indica explicit
  - ▶ Săgeata care indică directia de citire nu trebuie pusă pe linia de asociere!

# Validarea asocierilor

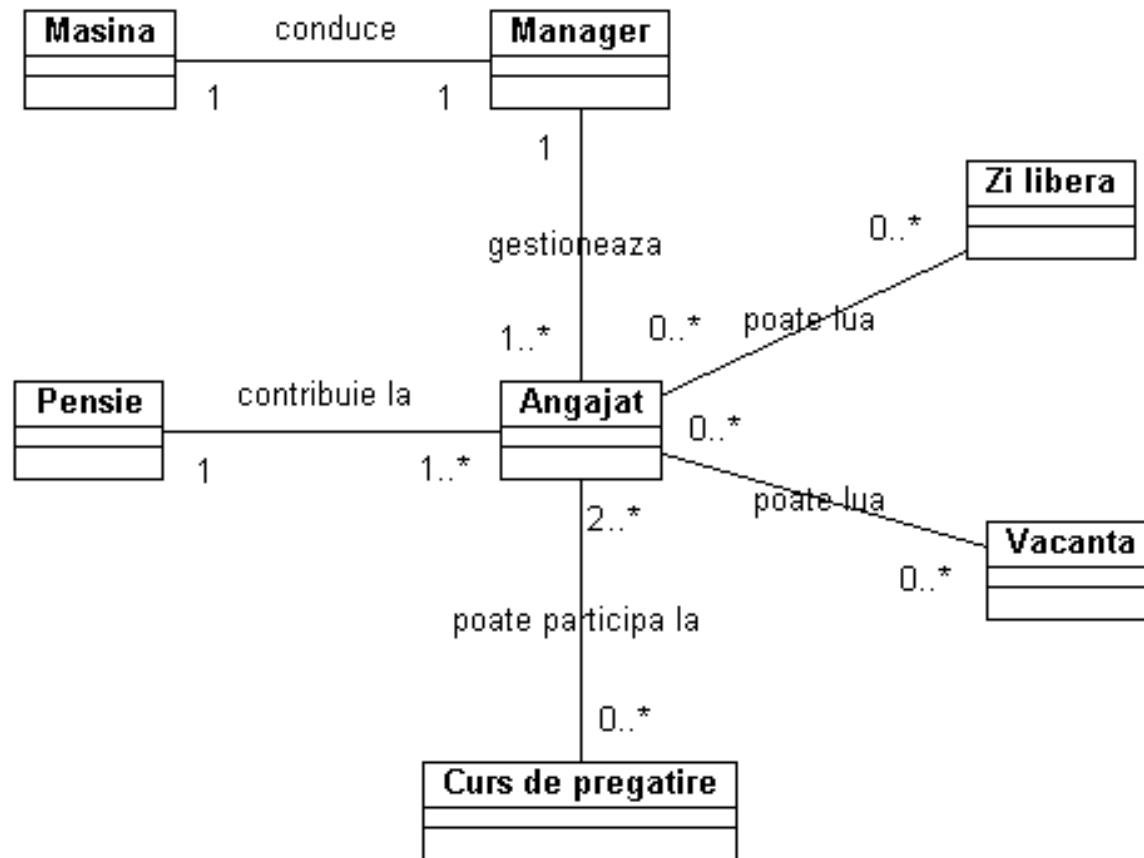


Association name

- ▶ Fiecare obiect este predat ca 0 sau mai multe cursuri.
- ▶ Fiecare curs este pentru unul și numai un obiect.

Multiplicity

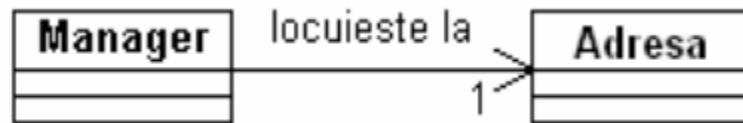
# Asociere complexă



# Asociere unidirectională

---

- ▶ Numai o clasă „știe” de cealaltă



# Multiplicitatea asocierii



Oricât de multe



Una sau mai multe



Între 1 și 8



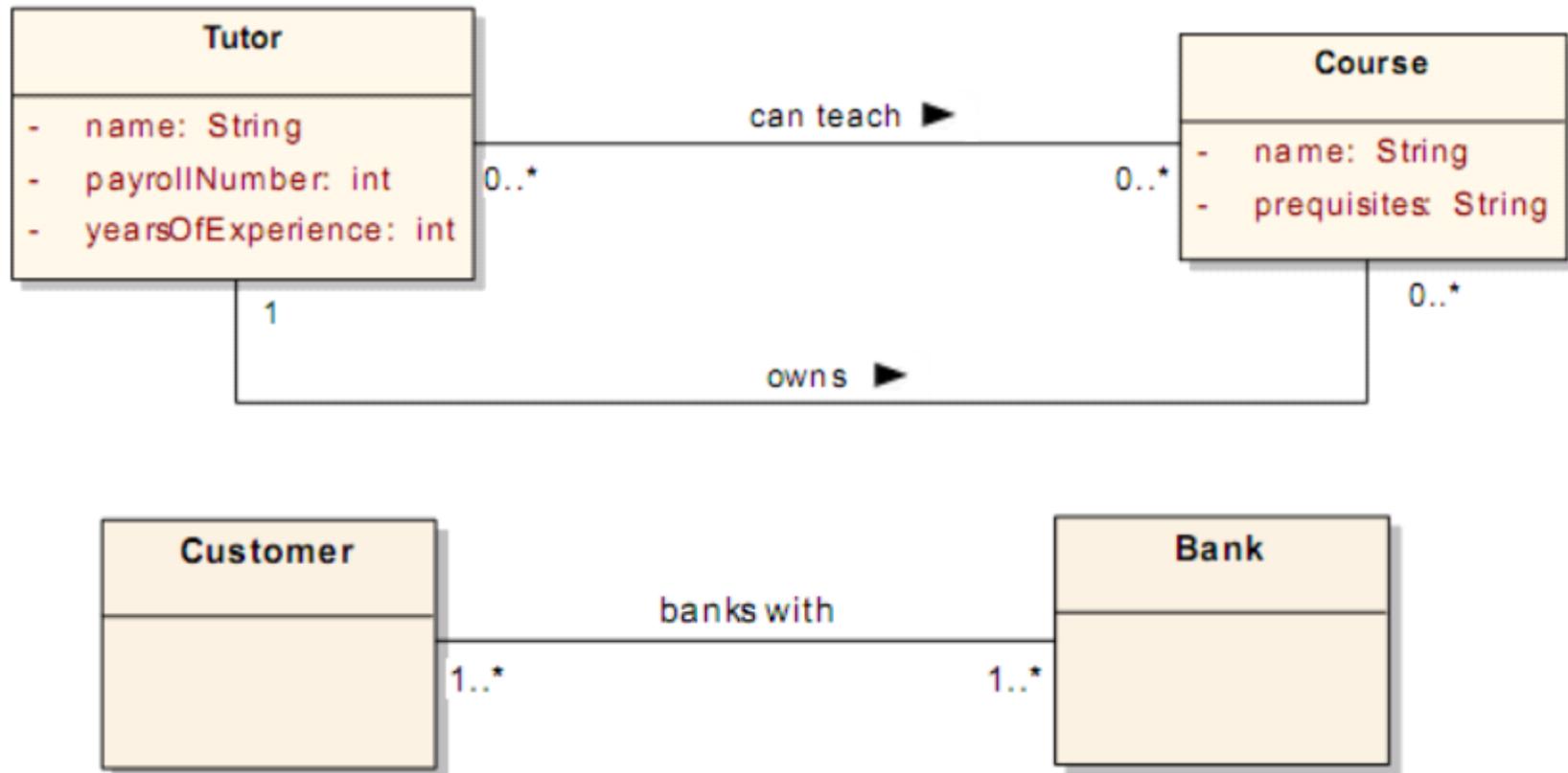
Exact 18



Multime specificată

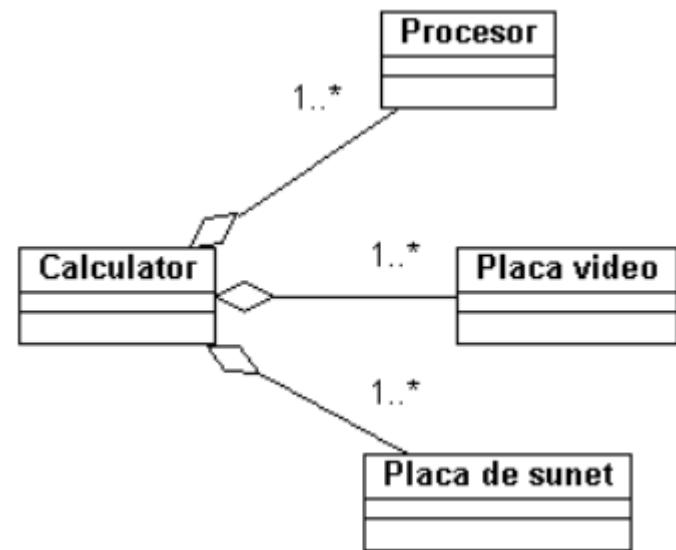
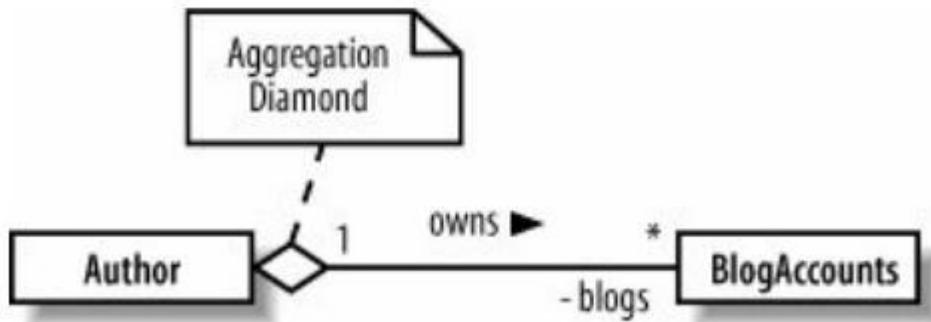


# Asocieri multiple



# Agregarea

- ▶ O clasă are dar *partajează obiecte dincealaltă clasă*



# Agregarea

---

```
class Program
{
    static void Main(string[] args)
    {
        Procesor procesor = new Procesor();
        PlacaVideo placaVideo = new PlacaVideo();
        PlacaDeSunet placaDeSunet = new PlacaDeSunet();
        Calculator calculator1 = new Calculator(procesor, placaVideo, placaDeSunet);
        Calculator calculator2 = new Calculator(procesor, placaVideo, placaDeSunet);
    }
}

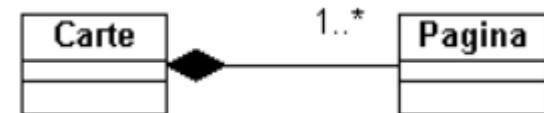
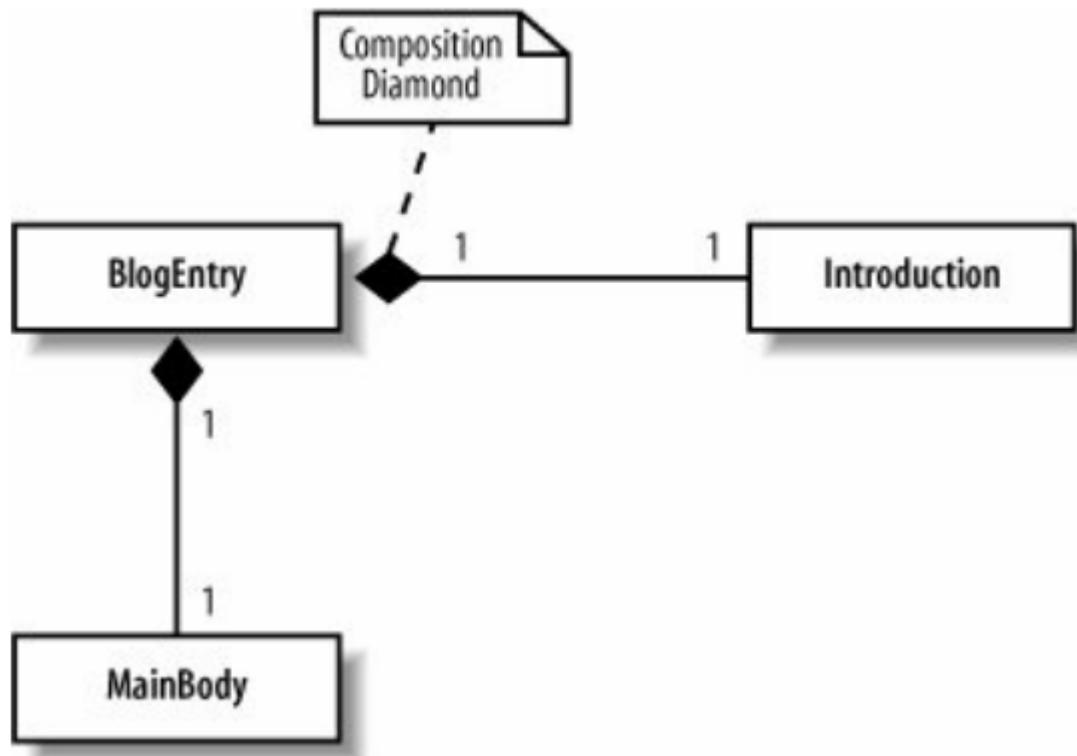
public class Calculator
{
    private Procesor _procesor;
    private PlacaVideo _placaVideo;
    private PlacaDeSunet _placaDeSunet;

    public Calculator(Procesor procesor, PlacaVideo placaVideo, PlacaDeSunet placaDeSunet)
    {
        _procesor = procesor;
        _placaVideo = placaVideo;
        _placaDeSunet = placaDeSunet;
    }
}
```



# Compunerea

## ► Atributele compun clasa



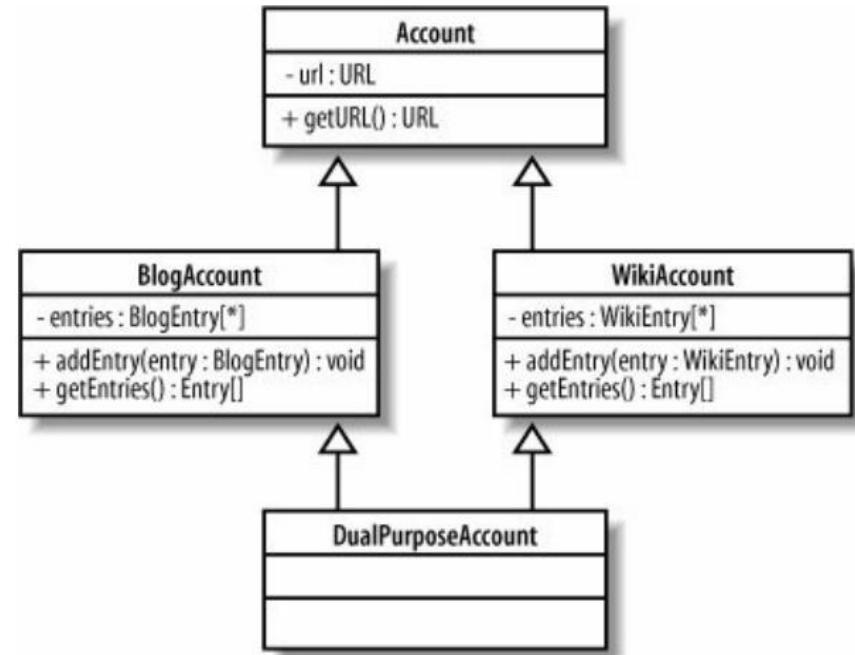
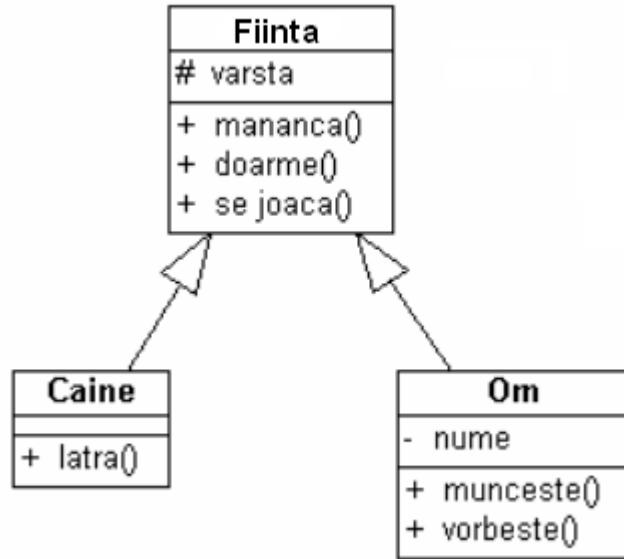
```
public class Carte
{
    private List<Pagina> _pagini;

    public Carte()
    {
        _pagini = new List<Pagina>();
    }
}

public class Pagina
{}
```

# Moștenirea

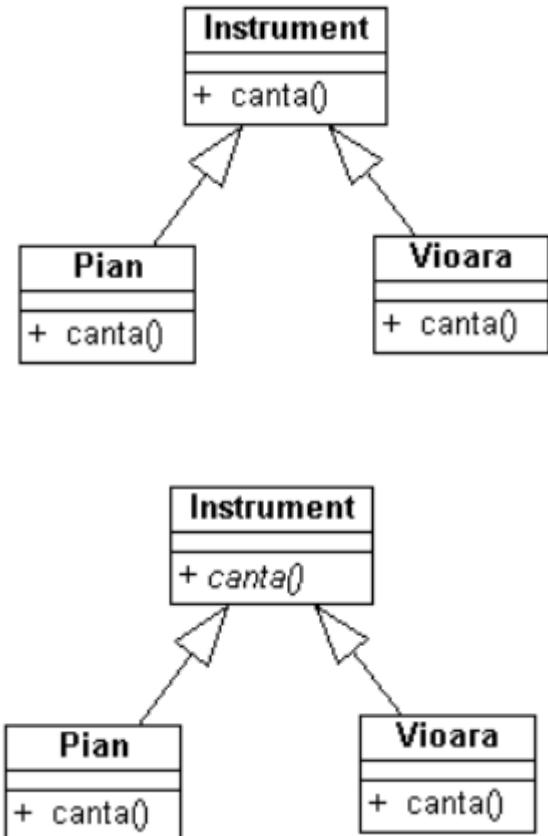
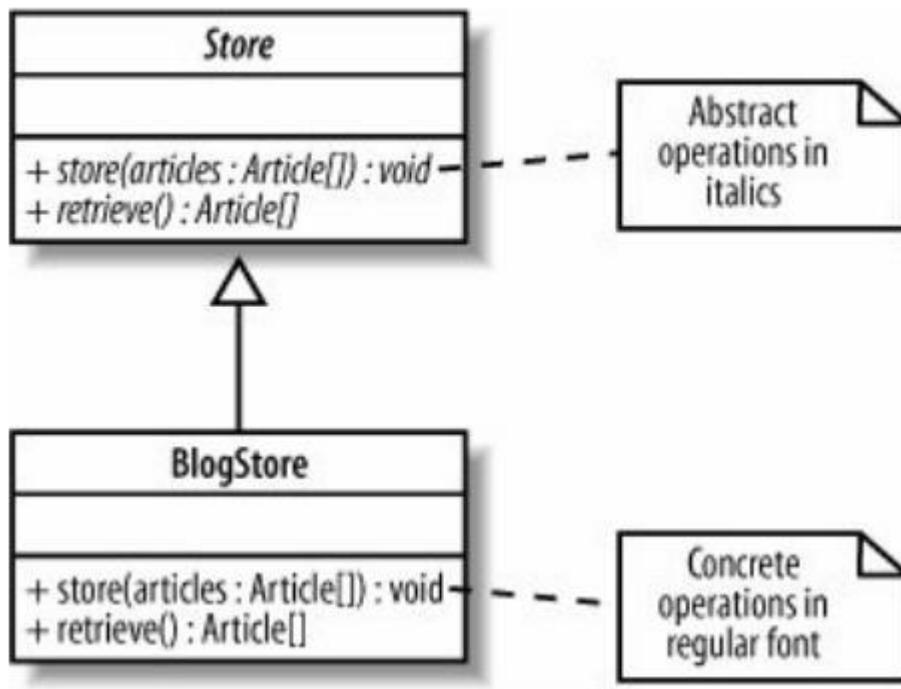
- ▶ Este o relație de tip ESTE-UN / ESTE-O



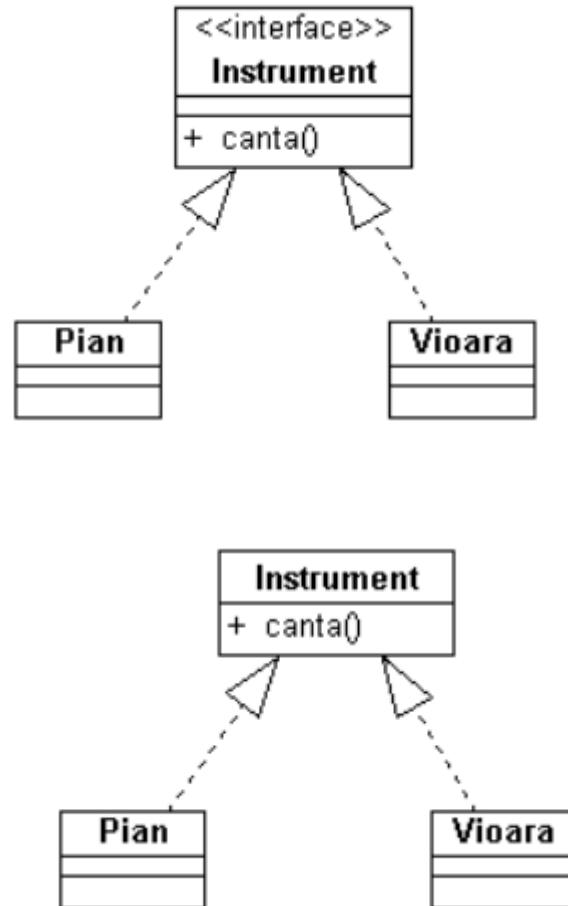
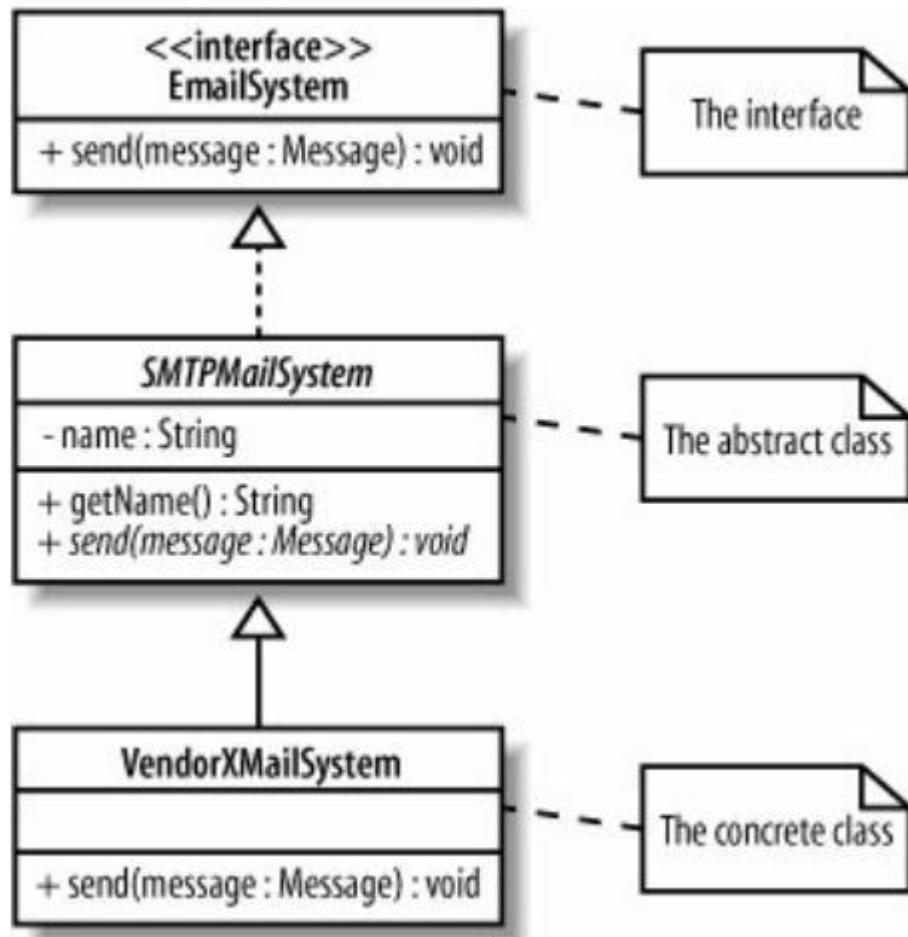
- ▶ Componerea ar trebui preferată moștenirii
  - ▶ Moștenirea este cea mai puternică formă decuplare
  - ▶ În general, componerea este mai ușor degestionat



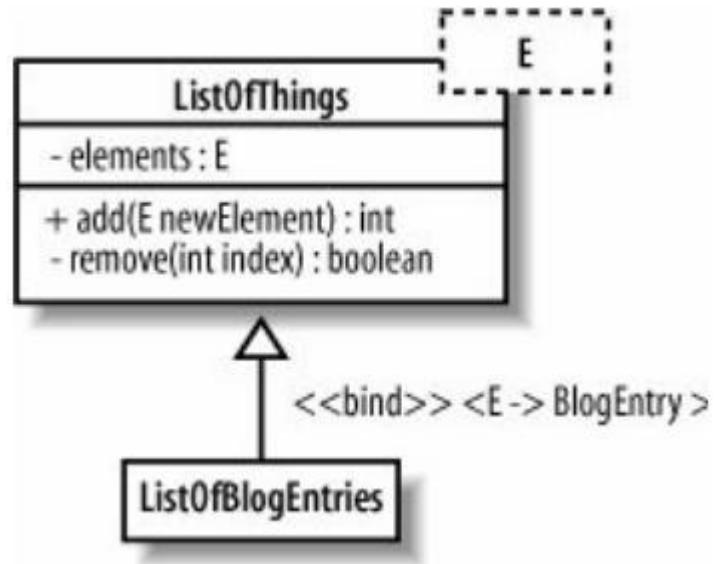
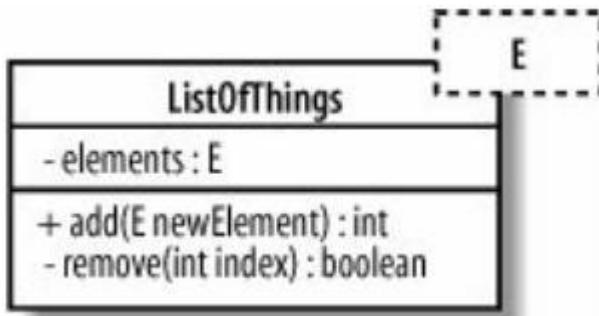
# Clase și operații abstracte



# Interfete



# Template-uri



# PROGRAMARE ORIENTATĂ PE OBIECTE

Curs 11  
*STL*

Containere

# Introducere – simplă clasă vector

---

- ▶ Fie clasa *MyVector*:

```
class MyVector
{
    size_t sz;
    int *v;
public:
    MyVector(void): sz(0), v(0) {};
    size_t GetSize(void){return sz; };
    int GetItem(size_t idx){ return v[idx]; }
    void PushBack(int);
    int PopBack(void);
    ~MyVector(void){ delete []v; };
};
```



# Introducere – simplă clasă vector

---

- ▶ Definițiile funcțiilor *PushBack* și *PopBack*:

```
void MyVector::PushBack(int x)
{
    int* aux = new int[sz+1];
    for(size_t i=0;i<sz;i++)
    {
        aux[i] = v[i];
    }
    aux[sz] = x;
    sz++;
    delete []v;
    v = aux;
}
```

```
int MyVector::PopBack(void)
{
    int tmp = v[sz-1];
    int* aux = new int[sz-1];
    for(size_t i=0;i<sz-1;i++)
    {
        aux[i] = v[i];
    }
    sz--;
    delete []v;
    v = aux;
    return tmp;
}
```

# Introducere – simplă clasă vector

---

- ▶ Utilizarea tipului de date *MyVector*:

```
int main(void)
{
    MyVector v;

    v.PushBack(1);
    v.PushBack(2);
    v.PushBack(3);
    v.PushBack(4);
    v.PushBack(5);

    for(size_t i = 0; i < v.GetSize(); i++)
    {
        cout << v.GetItem(i) << endl;
    }
    return 0;
}
```



# Introducere – class MyVector – lift up

---

- ▶ Fie clasa template *MyVector*:

```
template<class T>
class MyVector
{
    size_t sz;
    T*v;
public:
    MyVector(void): sz(0), v(0) {};
    size_t GetSize(void){return sz; };
    T GetItem(size_t idx){ return v[idx]; }
    void PushBack(T);
    T PopBack(void);
    ~MyVector(void){ delete []v; };
};
```



# Introducere – class MyVector – lift up

- ▶ Definițiile funcțiilor template *PushBack* și *PopBack*:

```
template<typename T>
void MyVector<T>::PushBack(T x)
{
    T* aux = new T[sz+1];
    for(size_t i=0; i<sz; i++)
    {
        aux[i] = v[i];
    }
    aux[sz] = x;
    sz++;
    delete []v;
    v = aux;
}
```

```
template <typename T>
T MyVector<T>::PopBack(void)
{
    T tmp = v[sz-1];
    T* aux = new T[sz-1];
    for(size_t i=0; i<sz-1; i++)
    {
        aux[i] = v[i];
    }
    sz--;
    delete []v;
    v = aux;
    return tmp;
}
```

# Introducere – class MyVector – lift up

---

```
int main(void)
{
    MyVector<double> v;

    v.PushBack(1.1);
    v.PushBack(2.2);
    v.PushBack(3.3);
    v.PushBack(4.4);
    v.PushBack(5.5);

    for(size_t i = 0; i < v.GetSize(); i++)
    {
        cout << v.GetItem(i) << endl;
    }
    return 0;
}
```



# Introducere – adăugarea unei clase iterator

```
▶ template<class T>
  class MyVector
  {
      size_t sz;
      T*v;
  public:
      class MyIterator
      {
          //...
      };
      MyIterator Begin(void){return v;};
      MyIterator End(void){return v+sz;};

      MyVector(void): sz(0), v(0) {};
      size_t GetSize(void){return sz; };
      T GetItem(size_t idx){ return v[idx]; }
      void PushBack(T);
      T PopBack(void);
      ~MyVector(void){ delete []v; };
  };
```

# Introducere – class MyIterator

---

```
▶ class MyIterator
{
    int *p;
public:
    MyIterator(void):p(0){};
    MyIterator(int *x):p(x){};
    MyIterator(const MyIterator& mit) : p(mit.p) {}
    MyIterator& operator++() {++p;return *this;}
    MyIterator operator++(int)
    {
        MyIterator tmp(*this);
        operator++();
        return tmp;
    }
    bool operator==(const MyIterator& rhs) {return p==rhs.p;}
    bool operator!=(const MyIterator& rhs) {return p!=rhs.p;}
    int& operator*() {return *p;}
};
```



# Introducere – utilizare iterator

---

```
int main(void)
{
    MyVector<int> v;

    v.PushBack(1);
    v.PushBack(2);
    v.PushBack(3);
    v.PushBack(4);
    v.PushBack(5);

    for(MyVector::MyIterator it = v.Begin(); it!=v.End(); it++)
    {
        cout << *it << endl;
    }
    return 0;
}
```



# Introducere - STL

---

- ▶ Librăria standard este un set de componente specificate în standardul ISO C++, furnizate cu un comportament identic de către orice implementare C++.
- ▶ Componentele sunt reutilizabile
- ▶ În anii '70 componentele folosite de programatori erau structurile de control și funcțiile
- ▶ În anii '80 programatorii foloseau clase dintr-o gamă largă de biblioteci dependente de platformă
- ▶ Odată cu standardul STL din anul '97 se introduc componente definite prin clase independente de platformă
- ▶ Structurile de date sunt colecții de date (containeri) organizate după diverse reguli



# Introducere - STL

---

- ▶ În C++ structurile de date sunt obiecte ce conțin colecții de obiecte:
  - ▶ Clasa vector reprezintă un vector de obiecte de tip int
  - ▶ Prin utilizarea template-urilor se redefinește clasa vector la vector<T> astfel încât se extinde acest tip de date la vector<char>, vector<double>, vector<Angajat> sau orice tip de dată
  - ▶ Similar se poate proceda cu implementarea structurilor de tip stivă, lista, arbori, grafuri etc.
- ▶ STL este o bibliotecă de clase template dar conține și implementări ale structurilor de date.
- ▶ În C și C++ elementele unui tablou sunt accesate prin intermediul pointerilor. În C++ STL elementele unui container sunt accesate prin intermediul iteratorilor care sunt tot pointeri dar care se comportă intelligent.



# Introducere - STL

---

- ▶ Containerii implementează operații primitive.
- ▶ Algoritmii ce utilizează containeri sunt independenți de tipurile de datele conținute.
- ▶ În STL s-a evitat folosirea moștenirii și a funcțiilor virtuale din considerente de performanță.
- ▶ S-a evitat utilizarea operatorilor new și delete în favoarea alocatorilor (permite metode de control pentru alocare și dealocare de memorie)
- ▶ Managementul erorilor



# STL - structură

---

- ▶ **STL cuprinde trei elemente principale:**
  - ▶ Containeri – obiecte ce conțin obiecte
  - ▶ Iteratori – pointeri „inteligenti” pentru acces la elementele unui container
  - ▶ Algoritmi – funcționalități de acces și prelucrare asupra elementelor containerilor.



# Containere

<b>Containere</b>	<b>Secvențiale</b>	<b>vector</b>	Implementează vectori alocați dinamic.	#include <vector>
		<b>list</b>	Lista liniară dublu înlănțuită	#include < list >
		<b>deque</b>	Asemănător containerului vector, operațiile putându-se realiza la ambele capete.	#include < deque >
	<b>Asociative</b>	<b>set</b>	Mulțime sortată de elemente unice.	#include < set >
		<b>multiset</b>	Mulțime sortată de elemente.	#include < set >
		<b>map</b>	Stochează perechi sortate de tip <cheie, valoare> în care o cheie identifică în mod unic o valoare.	#include < map >
		<b>multimap</b>	Stochează perechi sortate de tip <cheie, valoare> în care o cheie identifică una sau mai multe valori.	#include < map >
	<b>Adaptive</b>	<b>stack</b>	Structură de tip stivă.	#include < stack >
		<b>queue</b>	Structură de tip coadă	#include < queue >
		<b>priority_queue</b>	Structură de tip coadă în care elementelor le sunt asociate priorități.	#include < queue >



# Iteratori

---

- ▶ Iteratorii sunt obiecte care se comportă asemănător pointerilor și care sunt utilizati pentru a accesa elementele unui container.
- ▶ Iteratorii se aseamănă cu pointerii, dar sunt de fapt obiecte ce adresează alte obiecte. Cu ajutorul lor pot fi adresate elemente ale containerelor care aparțin anumitor intervale. Iteratorii reprezintă interfața de comunicație între algoritmi și containere, fiind preluati ca parametrii de către algoritmi. Containerele le furnizează algoritmilor o cale de acces către elementele lor prin intermediul iteratorilor.
- ▶ Algoritmii furnizează funcționalități de acces și prelucrare asupra elementelor containerelor



# Iteratori

<b>Iteratori</b>	<b>Acces aleatoriu</b>	Stochează și regăsește valori. Elemente pot fi accesate aleatoriu.
	<b>Bidirecționali</b>	Stochează și regăsește valori. Iteratorul poate înainta și reveni.
	<b>Înainte</b>	Stochează și regăsește valori. Iteratorul poate doar înainta.
	<b>De intrare</b>	Regăsește dar nu stochează valori. Iteratorul poate doar înainta.
	<b>De ieșire</b>	Stochează dar nu regăsește valori. Iteratorul poate doar înainta.
<b>Algoritmi</b>	Functii globale care oferă servicii generale cum ar fi sortări, reordonări, modificări, copieri, căutări etc.	#include <algorithm>



# Algoritmi

---

- ▶ Algoritmii STL se împart în patru mari categorii:
  - ▶ Algoritmi care modifică ordinea elementelor în container - modifying sequence operations: *copy()*, *replace()*, *transform()* și *remove()*.
  - ▶ Algoritmi care nu modifică ordinea elementelor în container – non-modifying sequence operations: *for\_each()*, *find()*, *count()* și *equal()*.
  - ▶ Algoritmi de sortare și operații similare: *sort()*, *equal\_range()*, *merge()* și *includes()*.
  - ▶ Algoritmi generali pentru operații numerice: *min()*, *max()*.



# Containere

---

- ▶ Containerele *secvențiale* sunt colecții liniare și ordonate de date în care accesul se face pe baza poziției elementului în cadrul containerului.
  - ▶ vector (adaugare pe la un singur capăt)
  - ▶ list (adaugare pe la ambele capete)
  - ▶ deque (adăugare pe la ambele capete)
- ▶ Ordinea elementelor este dată de ordinea în care au fost adăugate
- ▶ Containere *asociative* se diferențiază prin faptul că stocarea elementelor se face pe baza unor chei. Ordinea elementelor este dată de valorile cheilor și relația dintre ele. Accesul este direct prin intermediul cheii.
  - ▶ set
  - ▶ multiset
  - ▶ map
  - ▶ multimap

# Containere

---

- ▶ Containere adaptive – adaugă funcționalități containerelor sevențiale.
- ▶ Nu pot fi parcuse cu ajutorul iteratorilor întrucât nu sunt folosite în mod independent
- ▶ Programatorul trebuie să aleagă containerul de bază căruia să îi aplice un container adaptiv.
- ▶ Un container *stack* poate adapta containere *vector*, *List*, *deque*, un container *queue* poate adapta *List* și *deque*, *priroty\_queue* poate adapta un *vector* sau *deque*.



# Containere secvențiale - exemplu

---

```
class CStudent
{
private:
    int nrMat;
    char nume[20];
public:
    CStudent(int nr = 0, char* n = "Student"):nrMat(nr);
    CStudent(const Student& s);
    int GetNrMat(void);
    char* GetNume(void);
    void SetNume(char* n);
    bool operator<(Student& s);
};
```



# Containere secvențiale - exemple

```
void main(void){  
    vector<int> vectInt;  
    vectInt.push_back(5);  
    vectInt.push_back(0);  
    vectInt.push_back(15);  
    vectInt.push_back(13);  
    for(int i=0; i<vectInt.size(); i++)  
        cout<<vectInt[i].GetNrMat()<<" "<<  
    vectInt[i].GetNume()<    cout<<endl;  
  
    CStudent stud1(1, „Popescu”);  
    CStudent stud2(2, „Marian”);  
    CStudent stud3(3, „Gica”);  
    vector<Student> vectStud;  
    vectStud.push_back(stud1);  
    vectStud.push_back(stud2);  
    vectStud.push_back(stud3);  
    for(int i=0; i<vectStud.size(); i++)  
        cout<<vectStud[i]<<"\n";  
    cout<<endl;
```

```
List<int> listInt;
listInt.push_front(-4);
listInt.push_back(15);
listInt.insert(listInt.begin(),19);
listInt.insert(listInt.end(),3);

List<int>::iterator it;
for(it=listInt.begin(); it!=listInt.end(); it++)
    cout<<*it<<" ";
cout<<endl;
listInt.sort();
cout<<"Lista sortata"<<endl;
for(it=listInt.begin(); it!=listInt.end(); it++)
    cout<<*it<<" ";
cout<<endl;
```



```
list<Student> listStud;
listStud.push_back(stud2);
listStud.push_front(stud3);
listStud.insert(listStud.end(), s1);

list<Student>::iterator it;
for(it=listStud.begin(); it!=listStud.end(); it++)
    cout<<(*it).GetNrMat()<<" "<< vectInt[i].GetNume()<<endl;
cout<<endl;

listStud.sort();

for(it=listStud.begin(); it!=listStud.end(); it++)
    cout<<*it<<" ";
cout<<endl;
```



---

Vă mulțumesc !



# PROGRAMARE ORIENTATĂ OBIECT

Curs 12

Tratarea erorilor  
Operatii I/O

# Introducere

---

- ▶ Fie un program compus din mai multe părți/module dezvoltate separat.
- ▶ O parte a programului este apelată pentru a executa un anumit task. Respectiva parte poate fi văzută ca o colecție de funcții, o „librărie”.
- ▶ O librărie este un cod obișnuit dar în contextul gestiunii erorilor proiectantul/dezvoltatorul (autorul) acelei librării nu cunoaște din ce fel de programe va face parte librăria respectivă.
- ▶ Astfel:
  - ▶ Autorul acelei librării poate detecta erorile ce apar la rularea programului (*run time*) dar, în general, nu știe ce decizie să ia.
  - ▶ Utilizatorul unei librării poate gestiona o eroare de run time dar nu o poate detecta ușor

# Modul traditional de gestiune al erorilor

---

- ▶ **I.Terminarea programului (abordare drastică).**

- ▶ Fie funcția:

```
void *xmalloc(size_t size)
{
    void *pv = malloc(size);
    if(!pv)
    {
        exit(-1);
    }
    else
    {
        return pv;
    }
}
```

- ▶ Totuși pentru majoritatea erorilor se poate face mai mult:
  - ▶ Afisarea/logarea unui mesaj de eroare înainte de a închide programul.

```
cout << "Eroare alocare memorie" << endl;
```

---

# Modul traditional de gestiune al erorilor

---

- ▶ În particular, o librărie nu „știe” care este scopul și strategia programului în care este încapsulată. Astfel, în cazul unei erori, nu trebuie doar să apeleze `exit(...)` sau `abort()`.
- ▶ O librărie care înlătură programul într-un mod necondiționat nu poate fi utilizată într-un program care nu trebuie să „crape”.

# Modul traditional de gestiune al erorilor

---

- ▶ **2. Returnarea unui cod de eroare.**
- ▶ Nu este întotdeauna convenabil deoarece unele funcții sunt apelate în expresii a căror valoare urmează a fi determinată.  
*double x = sqrt(15);*
- ▶ Funcția poate fi modificată pentru a returna o pereche de valori (o structură cu rezultatul funcției și un cod de eroare ce caracterizează rezultatul). Dar acest lucru nu convine deoarece pentru fiecare apel trebuie verificat codul de eroare.
- ▶ De asemenea, programatorii ignoră sau uită să testeze codul de eroare returnat. Exemplu, funcția printf() returnează o valoare negativă dacă apare o eroare la scrierea în stream. Acel cod de eroare niciodată nu este testat.
- ▶ Constructorii nu returnează nimic.

# Modul traditional de gestiune al erorilor

---

- ▶ **3. Returnează o valoare însă lasă programul într-o stare de eroare**
- ▶ Funcția apelată nu notifică în nici un fel programul despre starea de eroare.
- ▶ Foarte multe funcții din librăria standard C setează variabila globală *errno* pentru a indica o eroare.

```
double x = sqrt(-1.0);
```

- ▶ Valoarea variabilei x nu are sens având în vedere că variabila *errno* a fost setată pentru a indica un argument inacceptabil pentru funcția *sqrt*. Programele scrise nu testează variabile *errno*.
- ▶ Având în vedere existența variabilei globale *errno* pot să apară probleme în prezența concurenței.

# Modul traditional de gestiune al erorilor

---

## ► 4. Apelul unei funcții error-handler:

```
if(error)
{
    ErrorHandlerFunction();
}
```

- Intrebarea care apare: „Ce face funcția *ErrorHandlerFunction()*”.
- Dacă funcția nu rezolvă complet problema atunci aceasta trebuie să:
  - Termine programul
  - Returneze un cod de eroare
  - Seteaze o stare de eroare
  - Arunce o excepție
- Dacă funcția gestionează eroarea cu succes atunci de ce este considerată eroare?
- În mod tradițional astfel de abordări co-există în programe

# Excepții

---

- ▶ Noțiunea de excepție este furnizată pentru a obține informații din punctul unde este detectată o eroare în punctul în care poate fi gestionată.
- ▶ O funcție ce nu poate gestiona o problemă, aruncă o excepție (*throws*) sperând ca apelantul (direct ori indirect) să gestioneze problema.
- ▶ O funcție care „vrea” să gestioneze o anumită problema indică acest lucru prințând excepția respectivă:
  - ▶ O componentă de apel indică tipurile de erori pe care le poate gestiona specificând excepțiile respective în clauza *catch* a unui bloc *try*.
  - ▶ Componenta apelată care nu își poate duce taskul la bun sfârșit raportează eroarea aruncând o excepție

# Excepții

---

```
void Taskmaster()
{
    try {
        auto result = DoTask();
        // ok
    }
    catch (someError)
    {
        // eroare în indeplinirea taskului. Gestionarea.
    }
}
int DoTask()
{
    // ...
    if /*daca nu au aparut erori*/
        return result;
    else
        throw someError{};
}
```

- ▶ Funcția *TaskMaster* „întreabă *DoTask()*” în legătură cu o sarcină de indeplinit. Dacă funcția *DoTask()* și-a îndeplinit sarcina și întoarce un rezultat corect, totul este bine. Altfel, trebuie să raporteze erori aruncând excepții.
- ▶ *TaskMaster()* este pregătită să gestioneze eroarea apărută dar totodată este posibil ca funcția *DoTask()* să apeleze la rândul ei alte funcții care la randul lor pot arunca excepții.
- ▶ O altfel de eroare decât cea pentru care *TaskMaster()* este pregătită indică un eșec al funcției în indeplinirea sarcinei și trebuie gestionat de secvența de cod ce a apelat *TaskMaster()*.
- ▶ O funcție apelată nu trebuie doar să returneze o eroare dacă a apărut.
- ▶ Pentru ca programul să fie funcțional, o funcție apelată trebuie să lase programul într-o stare bună fără „scurgeri” de resurse.

# Excepții

---

- ▶ **Mecanismul de gestiune a excepțiilor:**
  - ▶ Este o alternativă la tehniciile tradiționale atunci când acestea sunt insuficiente, neelegante ori generatoare de erori.
  - ▶ Este complet; poate fi folosit la gestiunea tuturor erorilor detectate
  - ▶ Permite programatorului să separe codul de gestiune al erorilor de restul codului.
- ▶ **O excepție este un obiect aruncat pentru a reprezenta apariția unei erori.** Poate fi de orice tip ce poate fi copiat dar se recomandă a se utiliza doar tipuri definite de utilizator special pentru acel scop. Astfel se reduce șansa ca două librării diferite să folosească același cod de eroare pentru a reprezenta erori diferite.
- ▶ **Cel mai simplu mod de a defini o excepție este de a defini o clasă special pentru acel tip de eroare**

# Excepții

---

```
class RangeError {};  
  
void f(int n)  
{  
    if (n<0 || max<n) throw RangeError {};  
    // ...  
}
```

- ▶ O excepție poate purta informații despre eroarea pe care o reprezintă

# Aruncarea și prinderea exceptiilor

- ▶ Pot fi aruncate exceptii de orice tip cu conditia sa poate fi copiate

```
class NoCopy {  
    NoCopy(const NoCopy& x) = delete; // fara constructor de copiere  
};  
class MyError{  
    // ...  
};  
void f(int n){  
    MyError me;  
    switch (n){  
        case 0: throw me; // OK  
                  break;  
        case 3: throw MyError{}; // valabil doar in C++11  
                  break;  
        case 1: throw NoCopy{}; // eroare: nu se poate copia NoCopy  
                  break;  
        case 2: throw MyError; // eroare: MyError este un tip  
                  break;  
    }  
}
```

# Aruncarea și prinderea exceptiilor

---

- ▶ Obiectul exceptie este o copie a celui aruncat. Astfel este initializată o variabilă temporară de tipul variabilei aruncate cu variabila aruncată. Această variabilă poate fi copiată de câteva ori înainte de a fi prinsă: exceptia este trimisă înapoi de la funcția apelata la funcția apelantă până când un handler potrivit este găsit.
- ▶ Tipul exceptiei este folosit pentru a selecta un handler al unui bloc *try*.
- ▶ Informația dintr-un obiect exceptie este folosită în mod obișnuit pentru mesaje de eroare sau pentru „help recovery”

# Aruncarea și prinderea exceptiilor

---

- ▶ Având în vedere că o excepție este copiată de câteva ori înainte de a fi prinsă, nu se pun informații mari în ea.
- ▶ Exceptii ce conțin câteva cuvinte sunt foarte comune.
- ▶ Unele exceptii nu poartă nici o informație; numele tipului este suficient pentru a raporta o eroare.

# noexception Function

---

- ▶ Unele funcții nu aruncă exceptii iar unele chiar nu ar trebui. Pentru a indica acest lucru se poate declara funcția respectivă ca funcție *noexcept*

```
double compute(double) noexcept;
```
- ▶ Este util pentru un programator deoarece nu trebuie să mai scrie blocul *try/catch* pentru apelul funcției respective și de asemenea modulul de optimizare al compilatorului nu mai are în vedere toată calea petru găsirea unui handler.
- ▶ Specificatorul *noexcept* spune doar că funcția nu gestionează exceptii.

# Prinderea exceptiilor

---

- ▶ Fie:

```
void f()
{
    try
    {
        throw E{};
    }
    catch(H)
    {
        // ...
    }
}
```

- ▶ Handlerul este invocat atunci când:

- ▶ H este de același tip cu E
- ▶ H este clasa de bază pentru E
- ▶ H și E sunt tipuri de date pointeri către H sau E

- 
- ▶ Dacă un handler prinde o excepție și decide că nu o poate gestiona complet o poate rearunca.

```
void f()
{
    try
    {
        throw E{};
    }
    catch(H)
    {
        if(...)
        {

        }
        else
        {
            throw;
        }
    }
}
```

---

▶ Prinderea oricărei exceptii:

```
void f()
{
    try
    {
        throw E{};
    }
    catch(...)
    {
        //...
    }
}
```

## ► Handlere multiple

```
void f()
{
    try
    {
        //...
    }
    catch(H)
    {
        //...
    }
    catch(F)
    {
        //...
    }
    catch(G)
    {
        //...
    }
}
```

# Exemple

---

```
double radical (double nr)
{
    if (nr < 0.0)
        throw "Nu pot extrage radicalul unui nr negativ!";
    return sqrt (nr);
}
```

# Exemple

---

```
int main ()
{
    double nr;
    cin >> nr;
    try
    {
        cout << "Radical din " << nr << " este " <<
            radical (nr);
    }
    catch (char *e)
    {
        cout << "Eroare: " << e << endl;
    }
    return 0;
}
```

# Excepții ne tratare

---

- ▶ Considerăm următoarea funcție main pentru exemplul de extragere a radicalului:

```
int main ()
{
    double nr;
    cin >> nr;

    cout << "Radical din " << nr << " este " << radical (nr);
    return 0;
}
```

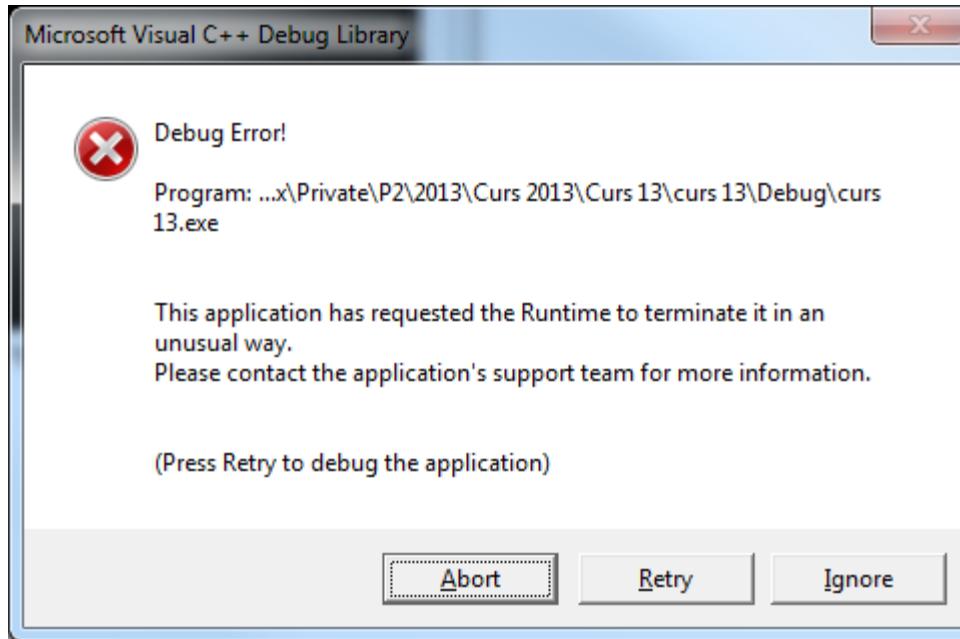
- ▶ Funcția de extragere a radicalului ramâne aceeași.

```
double radical (double nr)
{
    if (nr < 0.0)
        throw "Nu pot extrage radicalul unui nr negativ!";
    return sqrt (nr);
}
```

---

# Excepții ne tratare

- ▶ La introducerea unei valori negative, rularea programului se va opri brusc și va fi semnalată o eroare:



- 
- ▶ Funcțiile pot lansa exceptii care pot să nu aibă nici o modalitate de tratare a lor
  - ▶ Pentru a preveni aceste situații, se poate introduce un handler catch-all:

```
try
{
    cout << "Radical din " << nr << " este " << radical (nr);
}
catch (char *e)
{
    cout << "Eroare: " << e << endl;
}
catch (...)
{
    cout << "Eroare necunoscuta" << endl;
}
```

# Excepții în funcții membre

---

- ▶ Se folosesc
  - ▶ În special în constructori pentru a semnaliza unele erori la crearea obiectelor
  - ▶ Când se dorește semnalarea unei erori fără a duce la terminarea programului

# Excepții standard

---

- ▶ Librăria c++ standard furnizează o clasă de bază pentru tratarea obiectelor furnizate ca excepții
- ▶ Această clasa este *std::exception*
- ▶ Contine o funcție virtuală *what* ce returnează un *char\**

```
int main()
{
    try {
        while (true) {
            new int[1000000000uL];
        }
    } catch (const std::bad_alloc& e) {
        cout << "Allocation failed: " << e.what() << '\n';
    }
}
```



-----

# I/O

---

- ▶ Limbajul C++ nu are instructiuni specifice operațiilor de intrare/ieșire
- ▶ La baza operațiilor I/O se află conceptul de *stream* (flux). Prin *stream* se intlege un *flux* de date de la o *sursă* la o *destinație* sau *consumator*
  - ▶ sursa – tastatura, fisier pe disk, zona de memorie
  - ▶ destinație – monitor, fisier pe disk, zona de memorie
- ▶ C++ conține două ierarhii de clase: una are ca radacina clasa *streambuf*, iar cealaltă, clasa *ios*
- ▶ *streambuf* – furnizează funcții generale pentru lucrul cu zonele tampon (buffers) și permite tratarea operațiilor de intrare/ieșire fără formatări complexe. Clase derivate – *strstreambuf* și *filebuf*

- 
- ▶ Clasa *ios* are un pointer spre *streambuf*. Contine membri pentru a controla interfața *streambuf* și pentru tratarea erorilor. Clase derivate: *istream* pentru gestiunea intrărilor și *ostream* pentru gestiunea iesirilor

```
class istream : virtual public ios  
class ostream : virtual public ios
```

- ▶ O a treia clasa, numita *iostream* deriva din ambele clase:
- ▶ Clasele *istream*, *ostream*, și *iostream* sunt, fiecare, clase de bază pentru alte trei derivari:

```
class istream_withassign : public istream  
class ostream_withassign : public ostream  
class iostream_withassign : public iostream
```

---

- 
- ▶ Clasa *istream* – permite realizarea de conversii formatare sau neformatate ale caracterelor încărcate din obiecte de tip *streambuf*
  - ▶ Clasa *ostream* permite conversii formatare sau neformatate în caractere care se pastreaza în obiecte de tip *streambuf*
  - ▶ Clasa *iostream* mosteneste facilitatile ambelor clase de baza permitand operatii în ambele directii
  - ▶ Clasele cu sufixul *withassign* furnizeaza streamuri standard: *cin, cout, cerr, clog*
  - ▶ Obiectul *cin* (console input) este o instantiere a clasei *istream\_withassign* si el corespunde fisierului standard de intrare definit de pointerul *stdin*
-

- ▶ Analog, obiectul *cout* (console output) este o instantiere a clasei *ostream\_withassign* și el corespunde fisierului standard de iesire definit de *stdout*
- ▶ Obiectele *cerr* și *clog* sunt si ele instantieri ale clasei *ostream\_withassign* și corespund fisierului standard de iesire definit de pointerul *stderr*.
- ▶ Clasele cu sufixul *withassign* se deosebesc de claselor lor de baza prin faptul că supraincarca operatorul de atribuire.

```
class istream_withassign : public istream
{
    istream_withassign();
    istream& operator = (istream&);
    istream& operator = (streambuf&);
}
```

- ▶ Obiectele *cin*, *cout*, *cerr*, *clog*, se instantieaza o singura data, chiar daca fisierul respectiv se include de mai multe ori.

# Iesire standard

---

- ▶ Iesirile standard se pot realiza folosind operatorul “`<<`” supraincarcat pentru operatii de iesire, numit si *operator de inserare*
- ▶ Operandul stang este un obiect al clasei *ostream*
- ▶ Operandul din dreapta are un tip pentru care a fost supraincarcat operatorul “`<<`”. Acesta poate fi un obiect predefinit sau un tip abstract.  
`ostream& operator << (tip);`
- ▶ Implicit operatorul de inserare este supraincarcat pentru tipurile predefinite: *char, short, int, Long, unsigned, unsigned Long, char \*, float, double, Long double, void \**
- ▶ Regulile de prioritate se pastreaza
- ▶ Deoarece la supraincarcarea operatorului de inserare se returneaza o referinta la obiectul curent, operatorii de inserare se pot aplica intuitiv

# Exemple

---

- ▶ 1. `cout << i;`
- ▶ 2. `cout << "abc";`
- ▶ 3. `cout << "\n";`
- ▶ 4. `char c[] = "abc";  
cout << "c";`
- ▶ 5. `cout <<"i="<`
- ▶ 1. `printf("%d", i);`
- ▶ 2. `printf("%s", "abc");`
- ▶ 3. `printf("\n");`
- ▶ 4. `char c[] = "abc";  
printf("%s", c);`
- ▶ 5. `printf("i=%d\n", i+10)`

# Functii membre

---

- ▶ `int width(); //returneaza dimensiunea curenta a campului`
- ▶ `int width(int w); //seteaza dimensiunea campului`
- ▶ `char fill(); //returneaza caracterul de umplere`
- ▶ `char fill(char c); //seteaza caracterul de umplere`

```
cout.width(5);                                printf("%5d", i);
cout << i;
```

```
cout.width(5);                                printf("%05d", i);
cout.fill('0');
cout << i;
```

# Indicatori de format

---

- ▶ Sunt definiti in clasa `ios` și reprezinta biti din dublul cuvant `x_flag`, membru al clasei

```
public enum
{
    skipws=0x0001, //Skip white space on input.
    left=0x0002, //Left-align values; pad on the right with the fill character.
    right=0x0004, //Right-align values; pad on the left with the fill character (default alignment).
    internal=0x0008, //Add fill characters after any leading sign or base indication, but before the
                     //value.
    dec=0x0010, //Format numeric values as base 10 (decimal) (default radix).
    oct=0x0020, //Format numeric values as base 8 (octal).
    hex=0x0040, //Format numeric values as base 16 (hexadecimal).
    showbase=0x0080, //Display numeric constants in a format that can be read by the C++ compiler.
    showpoint=0x0100, //Show decimal point and trailing zeros for floating-point values.
    uppercase=0x0200, //Display uppercase A through F for hexadecimal values and E for scientific values.
    showpos=0x0400, //Show plus signs (+) for positive values.
    scientific=0x0800, //Display floating-point numbers in scientific format.
    fixed=0x1000, //Display floating-point numbers in fixed format.
    unitbuf=0x2000, //Cause ostream::osfx to flush the stream after each insertion. By default, cerr is
                    //unit buffered.
    stdio=0x4000, //Cause ostream::osfx to flush stdout and stderr after each insertion.
};
```

- 
- ▶ Indicatorii mai sus enumerati sunt grupati in trei grupe:
    - ▶ *adjustfield* – *right*, *left* si *internal* – defineste pozitiile caracterelor de umplere
    - ▶ *basefield* – *dec*, *oct* si *hex* – defineste baza de numeratie
    - ▶ *floatfield* – *scientific*, *fixed* – definesc formatul de afisare
  - ▶ In cadrul unei grupe numai un bit poate fi setat
  - ▶ Biti indicatori de format pot fi setati prin intermediul functiei *setf* membra a clasei *ios*  
*Long setf( Long f);*  
*Long setf( Long bit, Long grupa);*
  - ▶ Ambele returneaza valoarea precedenta a *x\_flag*.
  - ▶ Prima functie modifica formatul curent setand bitii corespunzatori bitilor de 1 din *f*. Ceilalți corespunzatori bitilor de 0 din *f* raman nemodificati
-

- 
- ▶ Biti indicatori de format pot fi resetati prin intermediul functiei *unsetf* membra a clasei *ios*.

*void unsetf(*Long* mask);*

```
int i = 123;
```

```
cout.width(5);
cout.fill(' ');
coutsetf(ios::left, ios::adjustfield);
coutsetf(ios::showpos);
cout << i;
```

# Manipulatori

---

- ▶ Permit definirea formatelor pentru operatiile de intrare/iesire.
- ▶ Sunt functii membru speciale ce returneaza o referinta la un stream. In acest fel apelurile manipulatorilor se pot inlantui.
- ▶ Practic, facilitatile oferite de functiile *width*, *setf*, *fill* etc. pot fi realizate cu ajutorul manipulatorilor

Manipulator	Actiune
<i>dec</i>	Seteaza indicatorul de conversie in zecimal
<i>oct</i>	Seteaza indicatorul de conversie in octal
<i>hex</i>	Seteaza indicatorul de conversie in hexazecimal
<i>ws</i>	Seteaza indicatorul skipws de salt peste caracterele albe
<i>endl</i>	Insereaza newline si videaza zona tampon a streamului
<i>ends</i>	Insereaza caracterul nul
<i>flush</i>	Videaza zona tampon a unui obiect al clasei ostream

Manipulator	Actiune
<code>setbase(int n)</code>	Seteaza baza dfe conversie egala cu n;//unde n poate avea una din valorile 0, 8, 10 sau 16. Valoarea 0 inseamna conversie implicita, adica in zecimal
<code>resetiosflags(Long f)</code>	Sterge biti de format specificati de parametrul f
<code>setiosflags(Long f)</code>	Seteaza bitii de format specificati de parametrul f
<code>setfill(int n)</code>	Seteaza caracterul de umplere la valoarea lui n
<code>setprecision(int n)</code>	Seteaza precizia la valoarea lui n
<code>setw(int n)</code>	Seteaza dimensiunea campului la valoarea lui n

```
int i = 123;
```

```
cout << setw(5) << resetiosflags(ios::internal|ios::right) <<
setiosflags(ios::left) << setfill('0') << i;
```

# Iesire neformatata

---

```
ostream& ostream::put(char c);
```

```
ostream& ostream::write(const signed char c, int n);
```

```
ostream& ostream::write(const unsigned char c, int n);
```

- ▶ Asupra acestor functii bitii de format nu au nici un efect

# Supraincarcarea operatorului <<

---

```
ostream& operator << (stream& iesire, complex z)
{
    return iesire << z.real << "+i" << z.imag;
}

class complex
{
    double real, imag;
public:
    ...
    friend ostream& operator <<(ostream&, complex);
    ...
};
```