

Programarea Orientată pe Obiecte (POO)

Prelegere



Introducere în cursul de
Programarea Orientată pe Obiecte (POO)

Silviu GÎNCU, doctor în pedagogie
Email: silviu.gincu@isa.utm.md

- Prelegeri (30 ore)
 - sunt prezentate noțiuni teoretice și exemple de aplicare practică a acestora;
 - nu ezitați să adresați întrebări.
- Seminare (15 ore)
 - sunt analizate/elaborate programe cu aplicarea conceptelor studiate în cadrul orelor de curs;
- Laboratoare (30 ore)
 - se elaborează și se depanează programe prin aplicarea conceptelor POO în practică;
 - se aplică ceea ce s-a predat la curs în cadrul orei de curs
- Evaluări
 - Examen final 40% + Atestarea_1 30% + Atestarea_2 30%
 - Atestare_1, _2 : rezultatul activităților desfășurate în cadrul orelor de prelegere, seminare și laboratoare.

Conținutul curricular al cursului POO

- § Paradigme de programare. Concepte de bază ale programării orientate obiect
- § Elemente ale limbajului C++ din perspectiva implementării principiilor POO
- § Clase. Constructori și destructori. Apelul constructorilor
- § Funcții și clase friend. Clase friend
- § Supraîncărcarea operatorilor unari și binari
- § Derivarea simplă și multiplă a claselor. Funcții virtuale pure. Clase abstracte. Polimorfism
- § Relații între clase. Agregare, Compoziția
- § Programarea generică. Funcții și clase template. Containeri. Iteratori. Algoritmii
- § Excepții. Tipuri de excepții predefinite

dr. Silviu GÎNCU

Referințe bibliografice

1. David Vandevoorde, Nicolai M. Josuttis "C++ Templates: The Complete Guide". Addison Wesley, 2018
2. Silviu Gîncu, "Metodologia rezolvării problemelor de informatică în stilul orientat pe obiecte", 2012
3. Diana ȘTEFĂNESCU, "Cristina SEGAL, Inițiere în Limbajul C++"
4. Kris Jamsa, Lars Kland, "TOTUL DESPRE C și C++ (MANUALUL FUNDAMENTAL DE PROGRAMARE IN C SI C++)", 2001
5. Grady Booch, "Object-oriented analysis and design with applications", 1998

Noțiuni generale despre cursul POO

§ Scop

- Acumularea de cunoștințe și aptitudini necesare realizării unor aplicații orientate obiect

§ Obiective

- Prezentarea și învățarea conceptelor programării orientate obiect
- Abilitatea de a elabora aplicații orientate obiect

§ *Limbaj de programare folosit pentru ilustrarea conceptelor de programare*

orientată obiect: C++

dr. Silviu GÎNCU

Prelegerea 1

Paradigme de programare. Concepte de bază ale POO

dr. Silviu GÎNCU

SUMAR

§ Paradigme de programare:

- *Programarea nestructurată*
- *Programarea procedurală*
- *Programarea modulară*
- *Programarea orientată obiect*

§ Principii ale programării orientate obiect:

- Abstractizare
- Incapsulare
- Modularizare
- Ierarhizare

§ Caracteristici ale programării orientate obiect

dr. Silviu GÎNCU

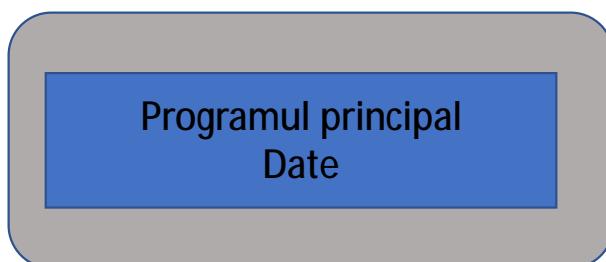
Paradigmă de programare

- § Noțiunea de paradigmă de programare se definește ca fiind o metodă de conceptualizare a modului de execuție al calculelor într-un calculator, precum și a modului de structurare și organizare a taskurilor responsabile cu execuția calculelor.
- § O noțiune frecvent utilizată în locul celei de paradigmă de programare este cea de stil de programare, deși semnificația sa nu este foarte clară definită.
- § Paradigma este o idee, un set de reguli care precizează modul în care se construiește un program într-un anume limbaj de programare. Paradigma de programare are o mare influență asupra modului în care se gândește rezolvarea unei probleme.
- § Exemplu. Să zicem că la reședința unei familii se strică încuietoarea de la ușă.
 - Soția nu are cunoștințe tehnice, așa că soluția ei să cheme un lăcătuș să schimbe încuietoarea.
 - Soțul, pe lângă cunoștințe, are și puțin orgoliu de bărbat și vrea să o repare personal.Ambele abordări conduc spre aceeași soluție.
Soția și soțul acționează sub paragime diferite.

dr. Silviu GÎNCU

Programarea nestructurată

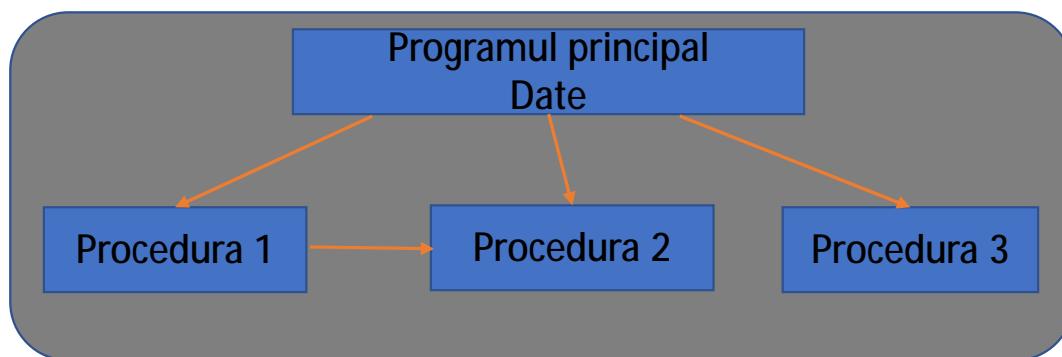
- § Programe simple / mici ca dimensiune care conțin doar o singură metodă
- § Program = succesiune de comenzi care modifică date globale
- § Dezavantaje
 - Greu de întreținut cu când codul devine mai lung
 - Mult cod duplicat (copy / paste)
- § Exemple: programe scrise în: asamblare, limbajul C, limbajul Pascal



```
test.c
//declarații date
int main(){
    //declarații date locale
    //instructiuni
}
```

Programarea procedurală

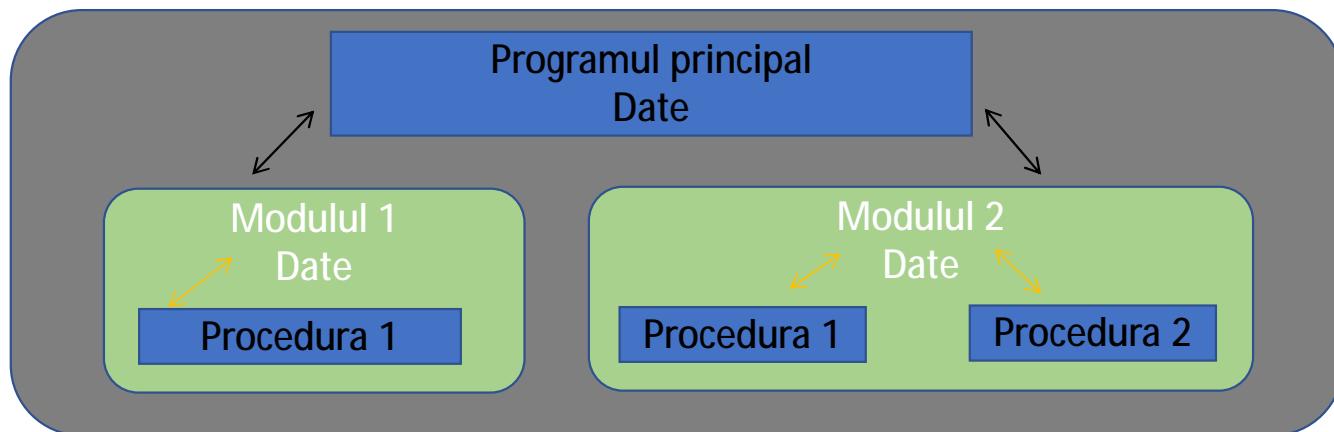
- § Se bazează pe noțiunea de procedură (funcție)
- § Procedura stochează algoritmul pe care dorin să îl (re)folosim
- § Dezavantaje
 - Menținerea, gestiunea diferitor structuri de date și algoritmi care prelucrează, gestionează datele
- § Exemple: programe scrise în: limbajul C, Pascal, Fortran, Algol



```
test.c
double sqrt() {...}
void f(double x){
... sqrt(x); ...
}
int main(){
...
double a=_sqrt(9);
f(sqrt(121.49));
...
}
```

Programarea modulară

- § Dimensiunea programului crește → Organizarea datelor
- § Partiționarea subprogramelor astfel încât datele să fie ascunse în module
- § Dezavantaje
 - Doar un singur modul există o dată într-un program
- § Exemple: programe scrise în: limbajul C, Modula-2



Programarea modulară

```
stiva.h
// declarea interfeței
char pop();
void push(char);
const dim_stiva= 100;
```

```
main.c
#include "stiva.h"
void functie(){
    push('c');
    char c = pop();
    if (c != 'c')
        error("imposibil"); }
```

```
stiva.c
#include "stiva.h"
char v[dim_stiva];
char* p = v;
// stiva este inițial goală
char pop() {
    // extrage element
}
void push(char c) {
    // adaugă element
}
```

Unele aspecte ce țin de organizarea codului de program în module

Codul este înpărțit în mai multe fișiere:

- § header (.h) - conțin declarații (interfața);
- § de implementare (.c | .cpp) - conțin definiția(implementarea) funcțiilor.

Pentru a avea access la funcțiile declarate într-un modul (bibliotecă de funcții) se folosește directiva #include

Preprocessorul include fișierul referit în fișierul sursă în locul unde apare directiva. Există două variante pentru a referi un modul: #include "fisierul_meu.h" - caută fișierul în directorul curent
#include <iostream.h> - caută fișierul în bibliotecile system

Întru evitarea dublei incluziuni a fișierelor header, în cadrul acestora se includ următoarele instrucțiuni:
#ifndef titlu generic – numele clasei
#define titlu generic
// definiții / conținutul clasei
#endif

Sarcină practică

Fișierul date.in conține informații despre notele ale studenților la unitățile de curs studiate în cadrul unui semestru. Să se elaboreze un program prin intermediul căruia să se realizeze următoarele operații:

- Citirea datelor din fișier și afișarea acestora la consolă;
- Determinarea studentului cu cea mai mare reușită;
- Determinarea disciplinei pentru care studenții au cele mai înalte note;
- Afișarea datelor despre studenți în ordine crescătoare a numelui;
- Modificarea notei la disciplina selectată;
- Afișarea datelor despre studenți în ordine descrescătoare a reușitei semestriale.

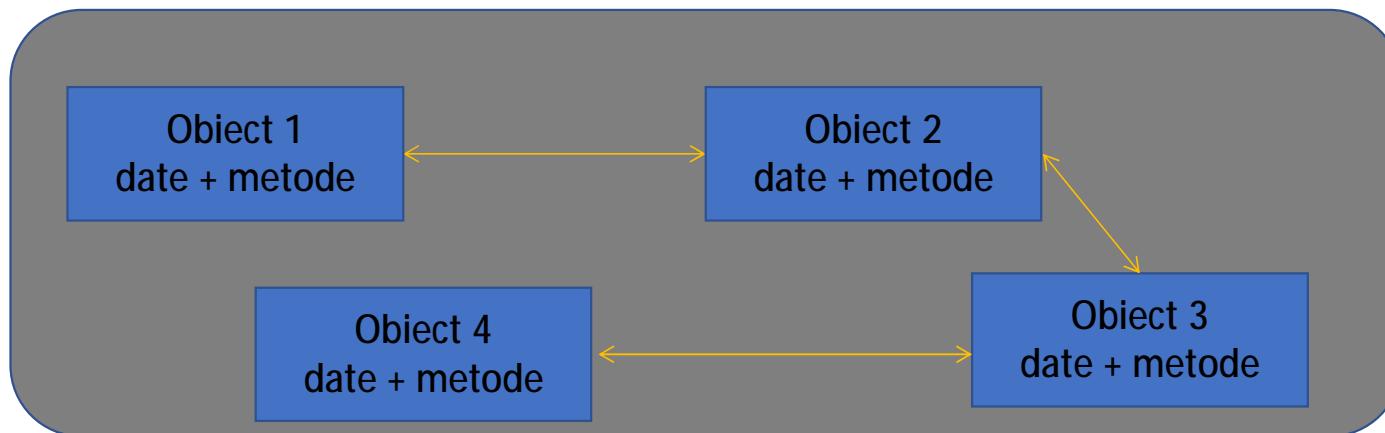
Întrebări de control:

1. Ce paradigmă de programare ve-ți utiliza la rezolvarea acestei probleme ? Justificați alegerea.
2. Rezolvați problema cu utilizarea paradigmelor de programare prezentate. Ce dificultăți întâmpinați la aplicare acestora ?

dr. Silviu GÎNCU

Programarea orientată pe obiecte

- § Obiecte care interacționează, fiecare gestionând starea proprie
- § Incapsularea datelor și a procedurilor într-un tot întreg
- § Exemple: programe scrise în: Simula, C++, Java, Eiffel, etc.



Paradigma POO

§ În procesul de elaborare a aplicațiilor în stilul orientat pe obiecte, distingem următoarele etape:

1. Identificarea obiectelor și a structurii acestora în acord cu specificațiile problemei:

- identificare date;
- identificare metode.

2. Stabilirea relațiilor dintre obiecte.

§ Conform problemei, se cunoaște:

§ Datele despre notele a n studenți la unitățile de curs studiate;

§ Se solicită următoarele operații:

- Citirea/afișarea datelor;
- Studentul cu cea mai mare reușită;
- Disciplinei pentru care studenții au cele mai înalte note;
- Afișarea datelor despre studenți în ordine ...;
- Modificarea notei la disciplina selectată.

dr. Silviu GÎNCU

Nume obiect	student
date	Informații despre student, informații despre unitățile de curs
metode	Citire, afișare, modifică nota, determinare reușită

Nume obiect	Lista studenti
date	Nr de studenți, Student []
metode	Citire, afișare, afișare cond., ordonare (conform criteriului)

SUMAR

§ Paradigme de programare:

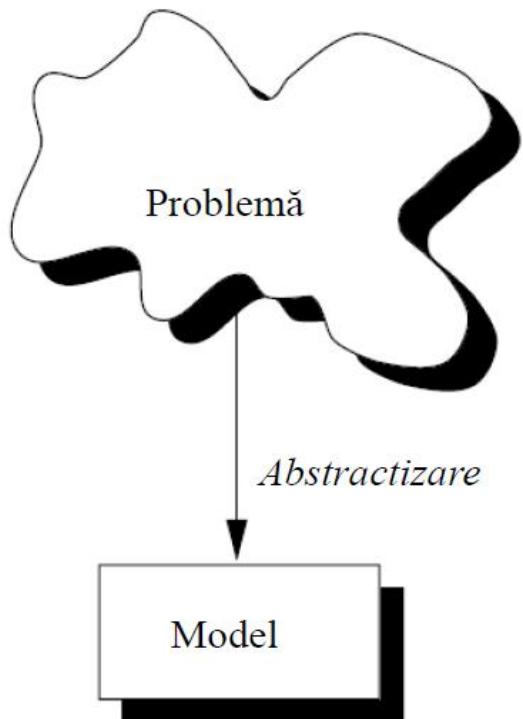
- *Programarea nestructurată*
- *Programarea procedurală*
- *Programarea modulară*
- *Programarea orientată obiect*

§ Principii ale programării orientate obiect:

- Abstractizare
- Incapsulare
- Modularizare
- Ierarhizare

§ Caracteristici ale programării orientate obiect

dr. Silviu GÎNCU



Abstractizare

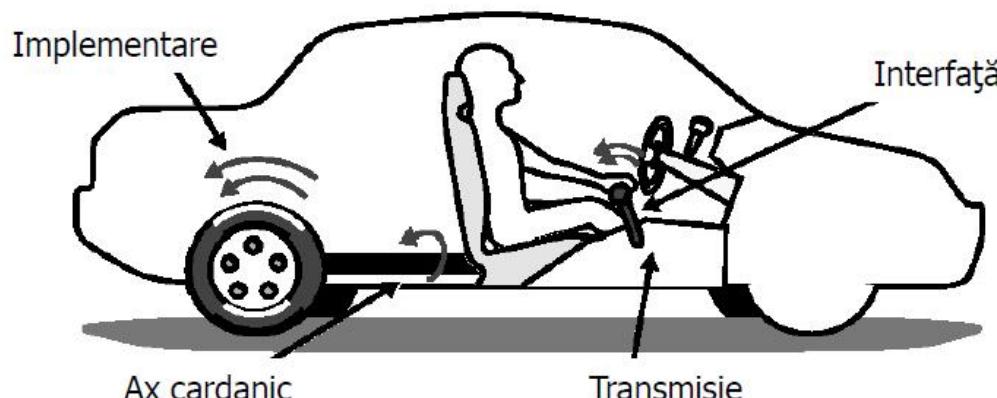
Abstractizare - este procesul de grupare a datelor și metodelor de prelucrare specifice rezolvării unei probleme

Abstracțiunea:

- 1) exprimă toate caracteristicile esențiale ale unui obiect care fac ca acesta să se distingă de alte obiecte;
- 2) oferă o definire precisă a granițelor conceptuale ale obiectelor din perspectiva unui privitor extern.

Incapsulare

- § Combinarea datelor și metodelor într-o singură structură de date, definind totodată modul în care obiectul și restul programului pot referi datele din obiect.
- § Concept care definește apartenența unor proprietăți și metode față de un obiect.
- § Constă în separarea aspectelor externe ale unui obiect care sunt accesibile altor obiecte de aspectele interne ale obiectului care sunt ascunse altor obiecte.



Modularizare

- § Modalitatea prin care un program este divizat în subunități (module) ce pot fi compilate separat.
- § Un modul grupează abstracții (clase) legate logic între ele.



Ierarhizare

§ Reprezintă o ordonare a abstracțiunilor. Principalele tipuri sunt:

- Moștenirea (ierarhia de clase) relație între clase în care o clasă împărțăște structura și comportarea definită în una sau mai multe clase (semantic implică o relație de tip "is a").
- Agregarea (ierarhia de obiecte) relație între două obiecte în care unul dintre obiecte aparține celuilalt obiect. (semantic implică o relație de tip "part of").

Alte 3 principii

- § Tipizare – model de protejare a obiectelor, prin intermediul căruia se face distincție între clase, astfel obiectele nu pot fi modificate, sau după caz, modificarea acestora este limitată.
- § Concurență (parallelism) – permite ca mai multe obiecte diferite să fie în execuție în același timp. Sistemele care implică acest principiu funcționează cu procesare multiple, în care mai multe procese (fire de execuție) sunt executate simultan pe procesoare diferite.
- § Persistență – presupune că în anumite situații obiectele există și după închiderea procesului (firul de execuție) care l-a creat.

Esențiale pentru limbajele de POO sunt următoarele principii: Abstractizare, Incapsulare, Modularizare, Ierarhizare. Se consideră că un limbaj de programare este orientat pe obiecte dacă susține primele patru principii.

SUMAR

§ Paradigme de programare:

- *Programarea nestructurată*
- *Programarea procedurală*
- *Programarea modulară*
- *Programarea orientată obiect*

§ Principii ale programării orientate obiect:

- Abstractizare
- Incapsulare
- Modularizare
- Ierarhizare

§ Caracteristici ale programării orientate obiect

dr. Silviu GÎNCU

Obiecte

- § Un obiect este o reprezentare a unei entități din lumea reală asupra căruia se poate întreprinde o acțiune sau care poate întreprinde o acțiune. Un obiect este o instanță a unei clase. El este unic determinat de numele său și are o stare reprezentată de valorile atributelor sale la un anumit moment particular
- § Un obiect este caracterizat de:
 - nume
 - attribute (date)
 - valorile atributelor la un moment dat definesc o starea obiectului
 - Metode (funcții/ operații)
 - subprograme, implementează comportamentul
 - apel metodă = transmitere mesaj, acțiune asupra obiectului
 - execuție metodă = răspuns la mesaj, reacție a obiectului, poate duce la modificarea stării

Exemple de obiecte

- Automobil

- Imobil

- Animal

dr. Silviu GÎNCU

Identificarea atributelor și metodelor

§ În cadrul unei bănci un cont bancar are:

- un titular, sold, o rata a dobânzii, număr de cont etc.
- și se pot efectua operații de:
- depunere, extragere, interogare sold.

§ Extragerea atributelor:

- titular, sold, rata a dobânzii, numar de cont

§ Extragerea metodelor:

- depunere, extragere, interogare sold

Clase

§ O clasă este o colecție de obiecte cu aceeași structură (caracteristici) și același comportament (metode sau operații). O clasa este o implementare a unui tip de date abstract. Ea definește atributele și metodele care implementează structura de date respectiv operațiile tipului de date abstract.

§ Clasa Bicicletă:

- atrbute
 - tip cadru
 - dimensiunea roții
 - număr de viteze
- metode
 - accelerează
 - frânează

Obiecte - Bicilete

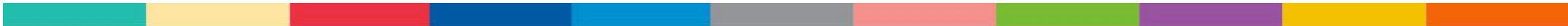


Teme pentru acasă

- § A învăță și repeta conceptele, principiile prezentate în cadrul lecției
- § A realiza sarcina practică expusă în slide-ul 14
- § A identifica 3 obiecte din 3 domenii diferite și pentru acestea a descrie:
 - Atributele caracteristice fiecărui obiect;
 - Metodele specifice fiecărui obiect.

Prelegerea următoare

1. Operații de intrare/ieșire în limbajul C++
2. Fișiere în limbajul C++
3. Pointeri. Alocarea dinamică a memoriei prin intermediul operatorilor.
4. Prelucrarea tablourilor prin intermediul pointerilor
5. Pointeri și structuri



Programarea Orientată pe Obiecte (POO)

Prelegere

Facilități C++

Silviu GÎNCU, doctor în pedagogie
Email: silviu.gincu@isa.utm.md

SUMAR

1. Operații de intrare/ieșire în limbajul C++
2. Fișiere în limbajul C++
3. Pointeri. Alocarea dinamică a memoriei prin intermediul operatorilor.
4. Prelucrarea tablourilor prin intermediul pointerilor
5. Pointeri și structuri

Facilități C++ vs C (1)

Limbajul C++ apare la începutul anilor '80 și îl are ca autor pe Bjarne Stroustrup. El este o variantă de limbaj C îmbunătățit, mai riguroasă și mai puternică, completată cu construcțiile necesare aplicării principiilor programării orientate pe obiecte (POO).

Câteva elemente de noutate:

1. I/O - C++ furnizează obiectele cin și cout, în plus față de funcțiile scanf și printf din C. Pe lângă alte avantaje, obiectele cin și cout nu necesită specificarea formatelor.

Exemplu:

```
cin >> variabila;  
cout<<"sir de caractere"<<variabila<<endl;
```

Facilități C++ vs C (2)

2. Limbajul C++ permite utilizarea mai multor funcții care au același nume, caracteristică numită supraîncărcarea funcțiilor. Identificarea lor se face prin numărul de parametri și tipul lor.

Exemplu:

```
int suma (int a, int b) {return a + b;}  
float suma (float a, float b) {return a + b;}  
int main (){ suma (3,5); (2.3, 9);}
```

3. C++ - doi noi operatori pentru alocarea dinamică de memorie, care înlocuiesc familiile de funcții "free" și "malloc" și derivatele acestora.

Astfel, pentru alocarea dinamică de memorie se folosește operatorul new, iar pentru eliberarea memoriei se folosește operatorul delete.

Operații de intrare/ieșire în limbajul C++

În C++, pentru realizarea operațiilor de I/O are loc prin intermediul "fluxurilor de intrare/ieșire" - este un obiect care conține datele și metodele necesare operațiilor cu acel flux. Pentru operații de I/O la consola sunt definite trei variabile de tip flux, numite:

- § cin (console input) pentru citirea fluxului de date;
- § cout (console output) pentru afișarea fluxului de date;
- § cerr (console errors) pentru indicarea erorilor.

Pentru afișarea datelor la consolă vom scrie: `cout<<flux1<<flux2<<...<<fluxn;`

unde în calitate de flux pot fi variabile și constante.

Pentru citirea datelor de la consolă vom scrie: `cin>>var1>>var2>>...>>varn;`

unde var1, var2, ..., varn sunt variabile.

Problema 1

De la tastatură se citesc două numere întregi. Să se elaboreze un program prin intermediul căruia se va afișa la ecran suma, diferența, produsul câtul și restul dintre cele două numere.

```
#include<iostream>
using namespace std;
int a,b,sum,dif,prod,cit,rest;
main(){
    cout<<"Dati doua numere intregi"<<endl;
    cout<<"a=";cin>>a;
    cout<<"b=";cin>>b;
    sum=a+b; dif=a-b; prod=a*b; cit=a/b; rest=a%b;
    cout<<a<<"+ "<<b<<"=" <<sum<<endl;
    cout<<a<<" - "<<b<<"=" <<dif<<endl;
    cout<<a<<" * "<<b<<"=" <<prod<<endl;
    cout<<a<<"/ "<<b<<"=" <<cit<<endl;
    cout<<a<<"%" <<b<<"=" <<rest<<endl;
}
```

dr. Silviu GÎNCU

```
Dati doua numere intregi
a=14
b=3
14+3=17
14-3=11
14*3=42
14/3=4
14%3=2
```

Formatarea prin manipulatori

Manipulatorii sunt funcții speciale, care pot fi folosite împreună cu operatorii de inserție într-un flux de ieșire sau de extractie dintr-un flux de intrare, în scopul modificării caracteristicilor formatului informațiilor de intrare/ieșire.

Manipulatorii furnizează, ca rezultat, fluxul obținut în urma acțiunii manipulatorilor. Pentru a avea acces la manipulatori se va include biblioteca iomanip.

Manipulatorii fără parametri (dex, oct, hex, endl) se folosesc astfel:

- § cout<<manipulator;
- § cin>>manipulator;

Prototipul manipulatorilor cu parametri este:

- § cout<<manipulator (argument)<<afisare_date;
- § cin>>manipulator (argument)>>citire_date;

dr. Silviu GÎNCU

Manipulatori C++ cu parametrii

Manipulator	Intrare/ieșire	Acțiune
setbase(int baza)	I/O	Stabilește baza de conversie 8 10 sau 16.
setfill(char c)	I/O	Definește caracterul de umplere (cel implicit este spațiul liber, blank-ul)
setprecision (int p)	I/O	Definește precizia pentru numerele reale
setw(int w)	I/O	Definește lățimea câmpului (numărul de octeți care vor fi citiți sau afisați)

Exemplu:

```
int a=24; double c=123.1234567;
cout<<a<<endl<<"a="<<setw(5)<<a<<endl;
cout<<"c="<<setw(7)<<setprecision(2)<<c;
cout << setfill ('x') << setw (10);
cout << 77 << endl;
```

Fișiere (1)

Definiție

Un fișier este o structură dinamică, situată în memoria secundară (pe floppy disk-uri, harddisk-uri, flash). Fișierele sunt de mai multe tipuri :

de tip text - un astfel de fișier conține o succesiune de linii, separate prin '`\n`';

de tip binar - un astfel de fișier conține o succesiune de octeți, fară nici o structură.

Declarare

!

`fstream nume_variabila_fisier;`

`fstream` – reprezintă tipul de date fișier în limbajul C++

`nume_variabila_fisier` – identificator, precizează numele variabilei de tip fișier

Se va include biblioteca `fstream.h`

Fișiere (2)

Algoritmul de utilizarea a fișierelor în cadrul unui program

1. Declararea variabilei de tip fișier **fstream**

2. Deschiderea fișierului – creare unei conexiuni între variabila declarată și fișierul extern. La crearea conexiunii se va indica modul de deschidere al fișierului, care poate fi:

- § Doar pentru citire;
- § Doar pentru scriere;
- § Atât pentru scriere cât și pentru citire

3. Realizarea operațiilor de scriere/citire a informației

4. Închiderea fișierului (distrugerea conexiunii dintre fișierul extern și variabila utilizată în cadrul programului).

Deschiderea fișierelor

Deschiderea fișierului se efectuează prin intermediul metodei `open`.

Mod de utilizare: `nume_file.open(adresa_file, mod_de_deschidere)`;

`adresa_file` – specifică adresa fizică a fișierului în calculator

argumentul `mod_de_deschidere` trebuie să aibă una din valorile:

- § `ios::in` este permisă doar citirea datelor dintr-un fișier existent;
- § `ios::out` crează un nou fișier, sau dacă există deja, distrugе vechiul conținut;
- § `ios::app` deschide un fișier pentru adăugare, pointerul de acces se plasează la sfîrșit;
- § `ios::ate` deschide un fișier existent pentru citire sau scriere, pointerul de acces se plasează la sfîrșit;
- § `ios::trun` deschide un fișier și ștergere vechiul conținut;
- § `ios::binary` specifică fișier de tip binar.

În argumentul `mod_de_deschidere` se pot combina prin intermediul operatorul OR (SAU)

Deschiderea fișierelor. Exemple

`f.open("date.txt",ios::in);` - se creează legătura dintre variabila f și fișierul **date.txt**, aflat în directorul curent. Prin intermediul programului în fișierul date.txt se vor realiza operații de citire.

!!! Observație

fișierul date.txt trebuie să fie în dosarul curent la momentul apelului funcției open.

`f.open("C:\\temp\\date.txt",ios::out);` - se creează legătura dintre variabila f și fișierul **date.txt**, aflat în mapa temp din directorul C. Prin intermediul programului în fișierul **date.txt** se vor realiza operații de scriere în fișier.

!!! Observație

dr. Silviu GÎNCU

Informația din fișierul date.txt va fi ștearsă, deoarece cursorul este poziționat la început de fișier pentru scriere. Dacă fișierul nu există în locul specificat, atunci el va fi creat de către program

Citirea datelor din fișier

```
fstream fisier;
```

```
fisier.open("date.txt",ios::in);
```

```
int a; float b; char c, d[100];
```

```
fisier>>a;//a=12
```

```
fisier>>b;//b=4.56
```

```
fisier>>c;//c='w'
```

```
fisier>>d;//d="Imi"
```

Pentru același conținut al fișierului și aceleași variabile

```
fisier>>b;//b=12
```

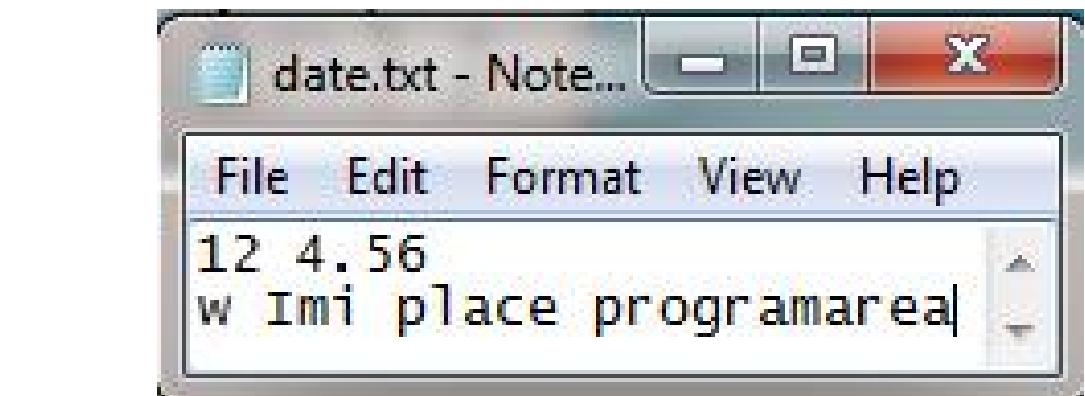
```
fisier>>d;//d="4.56"
```

Funcția get() citește caracter cu caracter

```
fisier.get(c); cout<<c; fisier.get(c); cout<<c;
```

```
fisier.get(c); cout<<c; fisier.get(c); cout<<c;
```

va afișa la consolă textul = "12 4"



```
fisier.close() - închide fișierul
```

Funcții de prelucrare a fișierelor

- § `eof ()`; - returnează nenul dacă este sfârșit de fișier;
- § `get(ch)`; - extrage un caracter în `ch`;
- § `read(str, MAX)` ; - extrage pînă la `MAX` caractere în `str` sau pînă la EOF;
- § `write(str, SIZE)` ; - inserează `SIZE` caractere din vectorul `str` în stream.
- § `seekg(positie)`; - setează distanța (în bytes) a pointerului de fișier față de începutul fișierului;
- § `seekg(positie, seek_dir)`; - setează distanța (în bytes) a pointerului de fișier față de poziția specificată `seek_dir`, care poate lua valori precum:
 - `ios::beg` - început de fișier
 - `ios::cur` - poziție curentă
 - `ios::end` - sfîrșit de fișier
- § `tellp()`; - returnează poziția pointerului de fișier în bytes.
- § `getline(str, MAX, [DELIM])`; - extrage pînă la `MAX` caractere din `str` sau pînă la caracterul `DELIM` ('\0' sau '\n').

Generalități

Un fișier este văzut în C++ ca un obiect, deci ca o variabilă de un tip clasă. Se pot folosi 3 clase predefinite în biblioteca `fstream.h`:

- § `fstream` pentru fișiere ce pot fi folosite în citire sau în scriere;
- § `ifstream` pentru fișiere din care este permisă doar citirea;
- § `ofstream` pentru fișiere în care este permisă doar scrierea.

Exemple de apeluri ale funcției `open()` pentru deschidere de fișiere:

- § `ifstream input; input.open("intrare.txt");`
- § `ofstream output; output.open("iesire.txt");`
- § `fstream inout; inout.open("fisier.txt", ios::in | ios::out);`

Pointeri

Un pointer este o variabilă care conține adresa unui Obiect (altă variabilă sau funcție).

Orice variabilă are două elemente caracteristice:

- § valoarea conținută în variabilă;
- § valoarea adresei locației de memorie.

Adresa locației de memorie în care este stocată o variabilă se poate obține aplicând operatorul de adresă (operatorul &) înaintea numelui variabilei. Operatorul de adresă poate fi utilizat pentru obținerea valorii adresei oricărei variabile.

```
int a=18;  
  
cout<<"valoarea lui a este "<<a<<endl;  
  
cout<<"adresa lui a este "<<&a;
```

```
valoarea lui a este 18  
adresa lui a este 0x69feec
```

Operatorii de adresare și derefențiere

Ca și în cazul variabilelor de orice tip, variabilele pointer declarate și neinitializate conțin valori aleatoare.

Pentru a atribui variabilei p valoarea adresei variabilei x, se va utiliza operatorul de adresă într-o expresie de atribuire de forma: p=&x;

Spunem că pointerul p indică spre variabila x. Pentru a obține valoarea obiectului indicat de un pointer se utilizează operatorul de derefențiere (operatorul *).

Pentru a indica adresă inexistentă, se utilizează ca valoare a unui pointer, constanta NULL (se acceptă și 0).

Exemplu de utilizare:

```
int x=3,*p;  
p=&x;  
cout<<*p; //afișează valoarea 3  
*p=5;    //valoarea 5 se scrie pe adresa indicată de p  
cout<<x; //afișează valoarea 5
```

Pointerii și adresele

Operatorii unari * și & au o precedență mai mare decât operatorii aritmetici, în consecință:

- § $y = *ip + 1$; preia obiectul spre care indică ip, îl adună cu 1 și atribuie rezultatul lui y
- § $*ip += 1$; incrementează obiectul spre care indică ip, ca și $++ *ip$ și $(*ip)++$

Operatorii unari precum * și ++ se asociază de la dreapta la stânga.

Pointerii fiind variabile, pot fi folosiți fără a fi diferențiați. Dacă iq este un alt pointer spre tipul int, atunci:

$iq = ip$; copiază conținutul lui ip în iq, făcând astfel ca iq să indice spre ceea ce indică ip.

Operatorul de adresă (&) se aplică numai obiectelor din memorie:

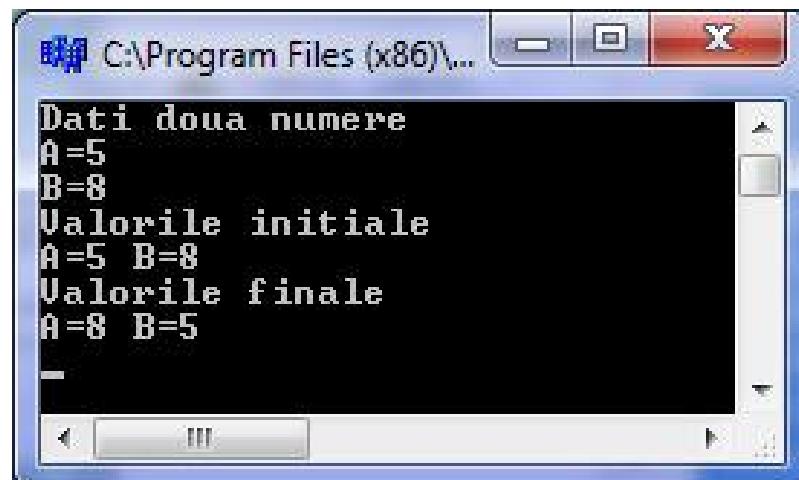
- § variabile
- § elemente de tablou.

Problema 2

De la tastatură se citesc două numere A și B. Elaborați un program prin intermediul căruia să se schimbe între ele valorile variabilelor A și B.

```
#include<iostream>
using namespace std;
float A,B,*a,*b,c;
main(){
    cout<<"Dati doua numere"=>endl;
    cout<<"A=";cin>>A; cout<<"B=";cin>>B;
    cout<<"Valorile initiale"=>endl;
    cout<<"A="<<A<<" B="<<B<<endl;
    a=&A; b=&B; c=A; *a=*b; *b=c;
    cout<<"Valorile finale"=>endl;
    cout<<"A="<<A<<" B="<<B<<endl;
}
```

dr. Silviu GÎNCU



Pointerii și tablouri (1)

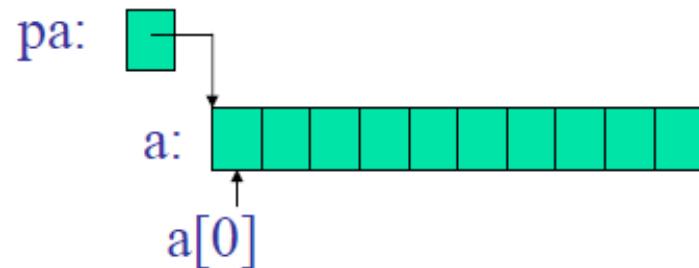
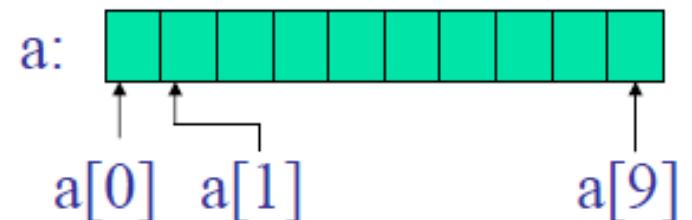
Orice operație care poate fi realizată cu ajutorul tablourilor cu indici poate fi de asemenea efectuată cu ajutorul pointerelor.

Fie dată declarația `int a[10];` - definește un tablou de dimensiune 10, adică un bloc de 10 elemente consecutive notate $a[0]$, $a[1]$, ..., $a[9]$.

Și declarația: `int *pa;`

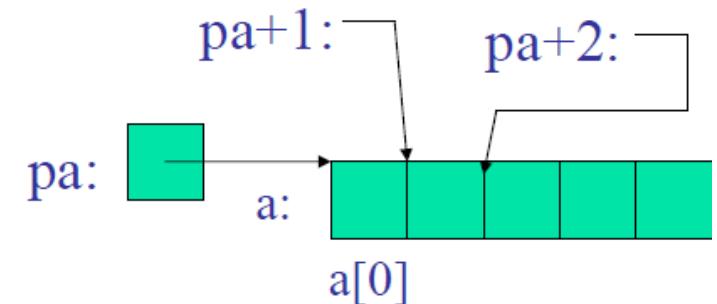
atribuirea `pa = &a[0];` îl setează pe `pa` să indice spre elementul zero al lui `a`; astfel, `pa` conține adresa lui $a[0]$.

Instrucțiunea `x = *pa;` va copia conținutul lui $a[0]$ în `x`.



Pointerii și tablouri (2)

Dacă pa indică spre un anumit element din tablou, atunci, $pa + 1$ indică spre următorul element, $pa + i$ indică spre o locație aflată la i elemente după pa , iar $pa - i$ indică spre o locație aflată cu i elemente înainte.
instrucțiunea $pa = \&a[0]$; poate fi scrisă $pa = a$;



Fie următoarele declarații: $T tab[N], * p;$
tab un tablou de elemente de tip T și p un pointer la tipul T :

În acest sens următoare atribuirile sunt echivalente, și au ca efect faptul că p va indica adresa primului element al tabloului tab :

$p=tab; p=&tab; p=&tab[0];$

Atenție !!! Atribuirea $tab=p$; nu este permisă.

dr. Silviu Gîncu

Prelucrarea tablourilor

Se consideră următoarele declarații: int *ptr,i,N=9,tab[]={1,2,3,4,5,6,7,8,9};

Următoarele instrucțiuni afișează la ecran elementele tabloului

```
for(i=0;i<N;i++) cout<<" "<<tab[i];
ptr=&tab[0];
for(i=0;i<N;i++) cout<<" "<<*(ptr+i);
for(ptr=tab;ptr<tab+N;ptr++) cout<<" "<< *ptr;
```

Instrucțiunea

```
for(ptr=tab;ptr<tab+N;ptr++)
cout<<"adresa: "<<ptr<<" Valoarea: "<<*ptr<<endl;
```

Va afișa adresa de memorie a fiecărui element
din tablou urmată de valoare de pe adresa

dr. Silviu GÎNCU

Problema 3

Pentru un tablou unidimensional să se determine elementul maximal și poziția lui în tablou. În cazul în care sunt mai multe elemente maximele pozițiile acestora se vor afișa în ordine descrescătoare.

```
#include<iostream>
using namespace std;
float t[100], *p, Max; int i, n;
main(){
    cout << "Dati numarul de elementele ale tabloului" << endl; cout << "n=" << cin >> n; p = t; Max = 0;
    for(i = 0; i < n; i++) { *p = rand() % 5; if(Max < *p) Max = *p; p++; }
    cout << "Elementele tabloului" << endl; for(i = 0; i < n; i++) cout << t[i] << " ";
    cout << endl << "Maxim=" << Max << "pozitia elementelor maxime" << endl;
    p = &t[n - 1];
    for(i = n - 1; i >= 0; i--) { if(*p == Max) cout << i << " "; p--; }
}
```

dr. Silviu GÎNCU

Pointerii spre caractere

Se consideră declarația:

```
char *pmesaj;
```

atunci instrucțiunea `pmesaj = "acum este timpul";`

îi atribuie lui `pmesaj` un pointer către tabloul de caractere.

Există o deosebire importantă între următoarele declarații:

```
char tmesaj[ ] = "acum este timpul"; /* un tablou */
```

```
char *pmesaj = "acum este timpul"; /* un pointer */
```



`tmesaj:` 

`pmesaj:` 

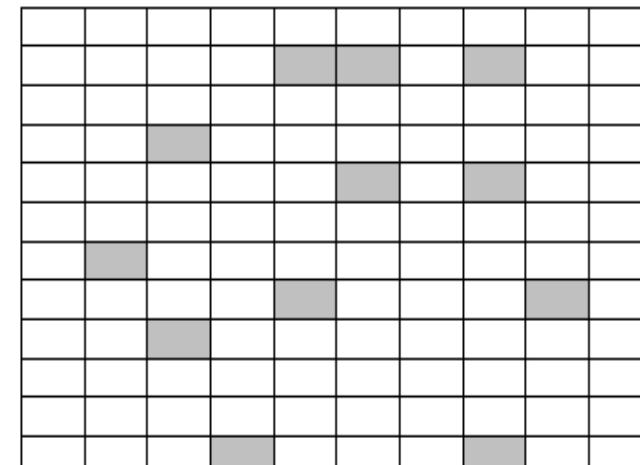
Alocarea dinamică a memoriei (1)

Memoria internă a calculatorului se reprezintă, sub formă de unei succesiuni de octeți. Fiecare octet, care intră în componența memoriei calculator-ului, are o adresă.

Adresa unui octet este un număr scris în baza 16, care reprezintă numărul de odine al octetului. Octeții sunt împărțiți în trei zone de memorie:

- q Zona Blocul de date BD
- q Zona Stiva
- q Zona Heap

Volumul fiecărui bloc de memorie este împărțit astfel:



Căsuțele colorate reprezintă locații de memorie ocupate

Blocul de Date
64 KO

Stiva
16 KO

Heap
256 KO

Alocarea dinamică a memoriei (2)

Spre deosebire de limbajul C, în limbajul C++ s-au introdus doi operatori noi, pentru alocare dinamică a memoriei new și pentru eliberarea memoriei delete, destinați să înlocuiască funcțiile de alocare și eliberare.

Operatorul new are ca operand un nume de tip, urmat în general de o valoare inițială pentru variabila creată (între paranteze rotunde). Rezultatul lui new este o adresa (un pointer de tipul specificat) sau NULL dacă nu există suficientă memorie liberă.

Sintaxa:

```
tipdata_pointer = new tipdata;  
tipdata_pointer = new tipdata(val_initializare);  
tipdata_pointer = new tipdata[nr_elem];  
delete tipdata_pointer;  
delete [] tipdata_pointer;
```

Tipdata reprezintă tipul datei (predefinit sau obiect) pentru care se alocă dinamic memorie, iar tipdata_pointer este o variabilă pointer către tipul tipdata.

dr. Silviu GÎNCU

Exemple de utilizare

```
int * p = new int(3); // alocare cu initializare
```

Operatorul new are o formă puțin modificată la alocarea de memorie pentru vectori, pentru a specifica numărul de componente:

```
int * v = new int [n];
```

Operatorul delete are ca operand o variabilă pointer și are ca efect eliberarea blocului de memorie adresat de pointer, a cărui mărime rezultă din tipul variabilei pointer sau este indicată explicit.

`delete p;` eliberează zona de memorie ocupată de variabila p.

`delete [] v;` eliberează zona de memorie ocupată de variabila v,

care este un tablou unidimensional.

Alocarea dinamică a memoriei pentru un tablou bidimensional

1. Declarare

Tip_data **nume;

2. Alocarea unui tablou de adrese **nume=new tip_data*[nr_lini];**

3. Alocarea unui tablou unidimensional pentru fiecare adresă

for(int i=0;i<nr_linii;i++) nume[i]=new tip_data[nr_coloane]

4. Procesare ... prelucrarea elementelor tabloului **nume[i][j]**

5. Eliberare memorie pentru coloane

for(int i=0;i<nr_linii;i++) delete [] nume[i];

6. Eliberare memorie pentru linii

delete [] nume;

Exemplu de Alocarea dinamică a memoriei

```
*****Pasul 1*****
int **t,n;
main(){
    cout<<"n="; cin>>n;
*****Pasul 2-3*****
t=new int*[n];
for(int i=0;i<n;i++) t[i]=new int[n];
*****Pasul 4*****
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++) t[i][j]=i+j;
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++) cout<<t[i][j]<<" ";
        cout<<endl;
    }
```

```
*****Pasul 5-6*****
for(int i=0;i<n;i++) delete [] t[i];
delete [] t;
}
```

Problema 4

În mod aleatoriu, se generează un tablou bidimensional cu numere întregi mai mari decât 10 și mai mici decât 32000, de n linii și m coloane, (n, m – se citesc de la tastatură). Să se elaboreze un program prin intermediul căruia se va crea un vector care va conține doar numerele suma cifrelor cărora este număr par. Elementele vectorului creat se vor afișa în ordine crescătoare.

Indicati nr. de coloane=7

Elementele tabloului bidimensional

11	17	14	10	19	14	18
18	12	14	15	15	11	17
11	11	15	12	17	16	11
14	12	13	12	12	11	16

Elementele tabloului creat

11	11	11	11	11	11	11	13	15
15	15	17	17	17	17	19		

```
#include<iostream>
#include<iomanip>
#include<stdlib.h>
using namespace std;
int i,j,n,m,k=0,r,s;
int **t,*v;
main(){
cout<<"Indicati nr. de linii="; cin>>n;
cout<<"Indicati nr. de coloane=";cin>>m;
t=new int*[n];
for(i=0;i<n;i++) t[i]=new int[m];
```

```
cout<<"Elementele tabloului
bidimensional"<<endl;
for(i=0;i<n;i++){
    for(j=0;j<m;j++){
        r=t[i][j]=rand()%32000+10;
        cout<<setw(8)<<t[i][j]; s=0;
        while(r>0){s+=r%10;r=r/10;}
        if(s%2==0) k++;
    }
    cout<<endl;
}
```

```
v=new int[k]; k=-1;
for(i=0;i<n;i++)
for(j=0;j<m;j++){
s=0; r=t[i][j];
while(r>0){s+=r%10;r=r/10;}
if(s%2==0)v[++k]=t[i][j]; }
for(j=1;j<=k;j++){
r=v[j]; i=j-1;
while(r < v[i] && i>=0){
v[i+1]=v[i]; i=i-1; }
v[i+1]=r; }
```

```
cout<<"Elementele tabloului creat"<<endl;
for(i=0;i<=k;i++) cout<<setw(8)<<v[i];
for(i=0;i<n;i++) delete t[i];
delete [] t;
delete [] v;
}
```

Structuri și pointeri

Structurile sunt frecvent referite prin intermediul unei variabile pointer, de exemplu:

```
struct student {  
    char nume[20], prenume[20]; int an_studiu;  
} *p ;  
  
student st={"Anatol", "Manole", 2}; p=&st;
```

Referirea unui element al unei structuri indicate de un pointer ca în exemplul anterior este posibilă, prin intermediul operatorului de selecție indirectă: "->" (săgeată):

```
cout<<"Prenume= " <<p->prenume;
```

Adresare se poate realiza și prin intermediul operatorului punct:

```
cout<<"Nume= " <<(*p).nume;
```

Alocare dinamică în cazul pointerilor de tip structură

Ca și în cazul altor tipuri de date, pointerilor de tip structură le poate fi alocată memorie în mod dinamic.

Se consideră tipul de date student, declarat anterior: struct student *p,t, a[15], *b;

Pentru a aloca dinamic memorie pointerului t vom scrie: p=new student;

iar pentru eliberare ei: delete p;

În cazul tablourilor: b=new student[10];

iar pentru eliberare delete [] b;

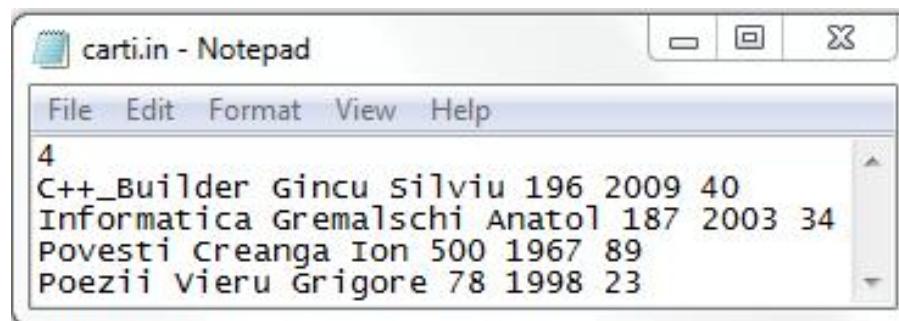
!!! Atenție. După alocarea memoriei accesul către câmpurile variabilelor p și t respectiv a și b se realizează la fel,
prin intermediul operatorului . (punct).

Problema 5

Ionel a cumpărat mai multe cărți. Pentru o mai bună evidență a lor el a hotărât să elaboreze un program prin intermediul căruia să gestioneze mai eficient datele despre cărți. El a hotărât ca despre o carte să păstreze informații referitoare la: *titlul cărții; numele și prenumele autorului; numărul de pagini; anul ediției; prețul*. Să se elaboreze un program prin intermediul căruia se va crea un meniu cu următoarele opțiuni:

- Afișarea tuturor cărților;
- Afișarea cărților tipărite recent (ultimii trei ani);
- Afișarea cărților unui anumit autor;
- Afișarea tuturor cărților în ordine descrescătoare a prețului.

Pentru ca programul să fie mai eficient, datele despre cărți se vor citi din fișierul *carti.in*. Structura fișierului:



```
carti.in - Notepad
File Edit Format View Help
4
C++_Builder Gincu Silviu 196 2009 40
Informatica Gremalschi Anatol 187 2003 34
Povesti Creanga Ion 500 1967 89
Poezii Vieru Grigore 78 1998 23
```

```
#define anul 2021
using namespace std;
struct carte{
    char titlu[40],n_a[15],p_a[15];
    int nr_p,an_ed;
    float pret;
}*t;
void citire_f(){
ifstream f("carti.in"); f>>n;
t=new carte[n];
for(i=0;i<n;i++){
    f>>t[i].titlu>>t[i].n_a>>t[i].p_a;
    f>>t[i].nr_p>>t[i].an_ed>>t[i].pret;
}
f.close();
}
```

```
void sortare(){
    carte aux;
    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
            if(t[j].pret>t[i].pret) {
                aux=t[i];
                t[i]=t[j];
                t[j]=aux;
            }
}
```

```
void meniu(){
    cout<<"Afisati "<<endl<<"1 - Toate cartile "<<endl;
    cout<<"2 - Cartile tiparite recent"<<endl;
    cout<<"3 - Cartile unui autor"<<endl;
    cout<<"4 - In ordine descreșcătoare a prețului"<<endl;
    cout<<"0 - Iesire";
}

void antet(){
    cout<<setw(20)<<"Titlu"<<setw(15)<<"Nume";
    cout<<setw(15)<<"Prenume"<<setw(8)<<"Pagini";
    cout<<setw(8)<<"Anul ed"<<setw(8)<<"Preț"<<endl;
}

void afisare_c(carte t){
    cout<<setw(20)<<t.titlu<<setw(15)<<t.n_a;
    cout<<setw(15)<<t.p_a<<setw(8)<<t.nr_p;
    cout<<setw(8)<<t.an_ed<<setw(8)<<t.pret<<endl;
}
```

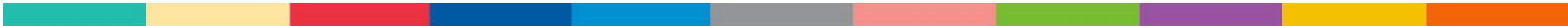
```
int main(){
    int i,j,n; char c, nume[15];
    citire_f(); do{    system("cls");
    meniu();    c=getch();    system("cls");
    switch(c){ case '1':cout<<"Lista tuturor cartilor"<<endl; antet();
        for(i=0;i<n;i++) afisare_c(t[i]); getch();break;
    case '2': cout<<"Cartile tiparite recent"<<endl; antet();
        for(i=0;i<n;i++) if(anul-t[i].an_ed<=3) afisare_c(t[i]); getch();break;
    case '3': cout<<"Indicati numele autorului"; cin>>nume;
        cout<<"Cartile autorului "<<nume<<endl; antet();
        for(i=0;i<n;i++) if(strcmpi(nume,t[i].n_a)==0)afisare_c(t[i]);getch();break;
    case '4': sortare();    cout<<"In ordine descrescatoare a pretului"<<endl;
        antet(); for(i=0;i<n;i++) afisare_c(t[i]); getch();break;
    }
}while(c!='0');
delete []t;
}
```

Teme pentru acasă

- § A învăță și repeta particularităile limbajului C++, în caz particular cele prezentate în cadrul lecției
- § A depana cele 5 programe prezentate în cadrul lecției

Prelegerea următoare

1. Generalități POO
2. Definirea claselor
3. Constructori. Tipuri de constructori
4. Destructor
5. Manevrarea dinamica a obiectelor
6. Implementarea tipului de date Array după modelul orientat pe obiect



Programarea Orientată pe Obiecte (POO)

Prelegere

*Definirea claselor.
Funcții speciale Constructor și destructor*



SUMAR

1. Generalități POO
2. Definirea claselor
3. Constructori. Tipuri de constructori
4. Destructor
5. Manevrarea dinamica a obiectelor
6. Implementarea tipului de date Array după modelul orientat pe obiect

Program

§ Programare structurată

\emptyset Structuri de date + Algoritmi = **Program**

§ Programare orientată obiect

\emptyset Date + Metode = **Obiect**

Clasă - Obiect

- § O **clasa** este o implementare a unui tip de date abstract. Ea definește atributele și metodele care implementează structura de date respectiv operațiile tipului de date abstract.
- § **Obiectul** reprezintă o instanță a unei **clase**.

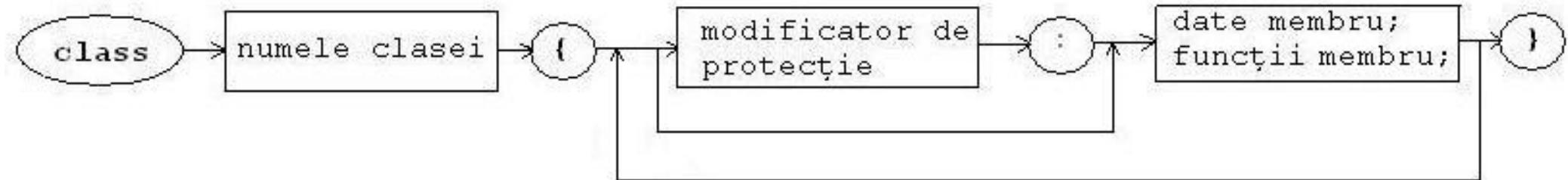
§ Clasa Bicicletă:

- *attribute*
 - tip cadru
 - dimensiunea roții
 - număr de viteze
- *metode*
 - accelerează
 - frânează

Obiecte - Bicilete



Sintaxa definirii unei clase



class <nume clasa>{

[<modificator de acces>:]

<tip data> <nume_data_membru>;

[<modificator de acces>:]

<tip metoda> <nume_metoda>(<lista parametri formali>);

};

§ date

§ metode

Protecția datelor și funcțiilor membre

- § **private** – datele și funcțiile aflate sub influența acestui modificador NU pot fi accesate din afara clasei.
- § **protected** - datele și funcțiile aflate sub influența acestui modificador nu pot fi accesate din afara clasei, cu excepția claselor derivate
- § **public** – datele și funcțiile aflate sub influența modificadorului public, pot fi accesate în afara clasei

<pre>class complex{ private: double Im, Re; public: double Modulul(); };</pre>	<pre>complex z; z.re = 2.5; // Incorrect deoarece re este private double d=z.Modulul(); // corect</pre>
--	---

Protectia datelor și funcțiilor membre - exemplu

```
class Curs{  
    char *cursId;      //vizibilitate privată  
public:             // vizibilitate - publică  
    string *nume;  
protected:          // vizibilitate - protejată  
    int nrCredite;  
public:             // vizibilitate - publică  
    void modificaNumarCredite (int);  
    void afisareIstoricCurs();  
    void informatiiEvaluariStudenti();  
private:            // vizibilitate - privată  
    void afisareInformatii();  
};
```

dr. Silviu GîNCU

```
Curs c;  
c.cursID="";           //inaccesibil  
c.nume="POO";         //acesibil  
c.nrCredite=8;        //inaccesibil  
c.modificaNumarCredite (8); //acesibil  
c.afisareIstoricCurs(); //acesibil  
c.informatiiEvaluariStudenti(); //acesibil  
c. afisareInformatii(); //inaccesibil  
};
```

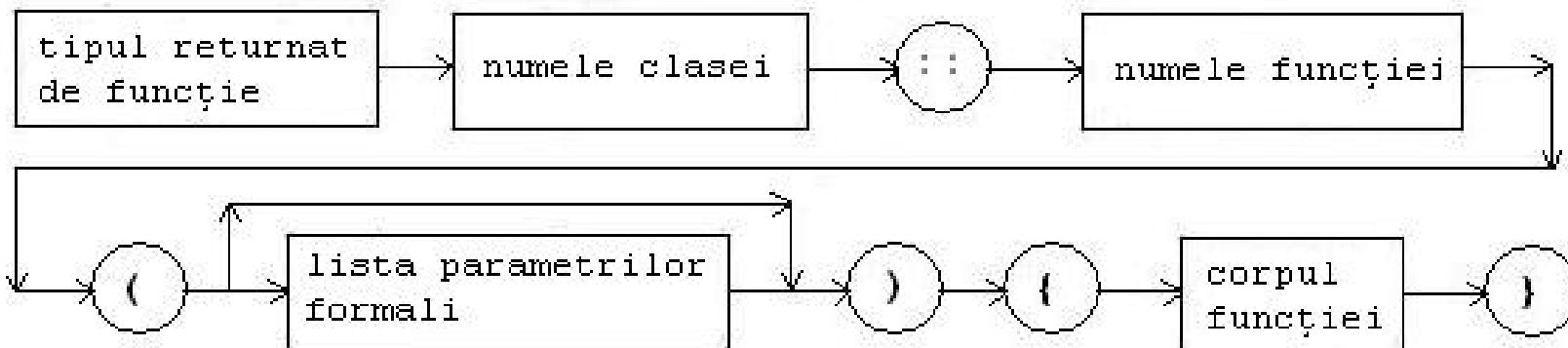
Clase - Definirea metodelor

§ **imediat**, în interiorul clasei -> metode **inline**

- definiție completă (antet + corp), Se realizează pentru metodele simple (3- 5 instrucțiuni)

§ În afara clasei -> prin utilizarea operatorului de rezoluție ::

- antetul metodei în interiorul clasei, iar corpul metodei în afara clasei



Problema 1

Să se implementeze tipul de date timp, care va avea în calitate de date: ora, min și sec, iar în calitate de metode: inițializare, afișare timp format scurt/lung.

```
#include<iostream>
using namespace std;
class Time{
public:
    void SetTime(int, int, int);
    void PrintShort();
    void PrintLong ();
private:
    int hour; //0-23
    int minute; //0-59
    int second; //0-59
};
```

```
void Time::SetTime(int h, int m, int s){
    hour = (h >= 0 && h < 24) ? h : 0;
    minute = (m >= 0 && m < 60) ? m : 0;
    second = (s >= 0 && s < 60) ? s : 0;
}
void Time::PrintShort(){
    cout << (hour < 10 ? "0" : "") << hour << ":";
    cout << (minute < 10 ? "0" : "") << minute;
}
```

Problema 1

```
void Time::PrintLong(){
    cout << ((hour == 0 || hour == 12) ? 12 : hour % 12);
    cout << ":" << (minute < 10 ? "0" : "") << minute;
    cout << ":" << (second < 10 ? "0" : "") << second;
    cout << (hour < 12 ? " AM" : " PM");
}

int main(){
    Time t;//instantiaza obiectul t de tip Time
    t.SetTime(13, 27, 6);
    cout << "\n\nOra in format scurt dupa SetTime este ";
    t.PrintShort();
    cout << "\nOra in format lung dupa SetTime este ";
    t.PrintLong();
    t.SetTime(99, 99, 99);
    cout << "\n\nDupa asignarea valorilor invalide:";
    cout << "\nOra in format scurt: ";
    t.PrintShort();
    cout << "\nOra in format lung: ";
    t.PrintLong();
}
```

Obiecte

§ **Obiectul** reprezintă o instanță a unei **clase**

§ Declararea obiectelor:

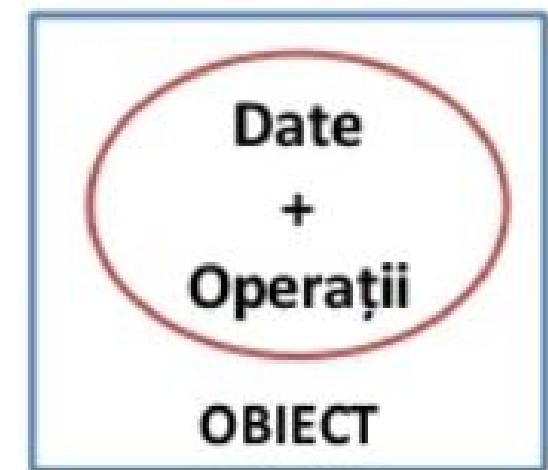
<nume_clasă> <nume_object>;

§ Alocarea și inițializarea unui obiect

- la declarare
- prin apelul automat al constructor-ului

§ De-alocarea

- la sfârșitul programului/funcției
- prin apelul automat al destructor-ului



Accesarea componentelor unei clase

§ Componentele clasei sunt accesate:

- direct – în interiorul metodelor membre ale clasei
- prin intermediul operatorului “.” – operatorul este precedat de un obiect
- prin utilizarea operatorului “->” – operatorul este precedat de o referință a unui obiect
- prin utilizarea operatorului “::” – operatorul este precedat de numele unei clase

```
class Carte{  
    char *titlu;    int pret;  
    static int nr;  
public:  
    char *getTitlu(){return this->titlu;}  
    int getPret();  
};  
int Carte::nr=10;  
int Carte::getpret(){return pret;}  
int main(){Carte c;  
cout<<c.getpret();  
cout<< Carte::nr;  
}
```

Obiectul carte

De la obiecte spre clasă

nume	cărți
atribute	titlu, autor, editura, an_apariție, ISBN, preț
metode	obține_titlu, obține_autor, modifică_preț, afișează_informații

atribute	titlu	Poezii	Geniu Pustiu	Marile speranțe
	autor	Mihai Eminescu	Mihai Eminescu	Charles Dickens
	editura	Polirom	Polirom	Univers
	an apariție	2007	2005	2003
	ISBN	973-567-545-1	973-565-545-2	971-267-441 -1
	preț	25	20	35

dr. Silviu GÎNCU

Problema 2

Să se creeze o clasă care să implementeze tipul de date carte, descris pe slide-ul anterior.

```
class carte{
    char titlu[40], autor[25], ISBN[20], editura[20];
    int an_aparitie;
    float pret;
public:
    void set();
    void set(char*, char*, char*, char*, int, float);
    char * obtine_titlu();
    char * obtine_autor();
    void modifica_pret();
    void afiseaza_informatii();
};
```

```
int main(){
    carte a;
    a.set();
    char *titlu;
    titlu=a.obtine_titlu();
    char autor[40];
    strcpy(autor, a.autor); // eroare
    // nu avem acces la câmpul autor
}
```

Inițializarea Obiectelor

§ În cazul datelor de tipuri predefinite de date, este posibilă inițializarea acestora la momentul declarării.

Exemplu:

```
int x=10;  
float a[2]={3,8};
```

§ În cazul obiectelor, inițializarea la declarare se face prin intermediul unor funcții speciale numite constructori.

§ Sintaxa declarării unui constructor

```
class IdNumeClasa {  
    IdNumeClasa (<listaParametri>);  
};  
IdNumeClasa::IdNumeClasa (<listaParametri>){  
    //instructiuni  
}
```

Funcția specială constructor

§ Constructor – o metodă publică specială:

Ø numele metodei = nume_clasă

Ø fără tip de return (nici măcar void)

§ Scop:

- rezervă spațiu pentru datele membre
- initializează datele membre
- apelat automat la declararea unui obiect

§ Constructorii pot fi supraîncărcați pentru a oferi diverse metode de inițializare a obiectelor clasei

§ O clasă poate avea mai mulți constructori

§ O clasă poate avea un singur constructor implicit

§ Fiecare clasă conține cel puțin un constructor

§ Tipuri de Constructori:

Ø Impliciți

Ø Cu parametri

Ø De copiere

Ø De conversie

dr. Silviu GÎNCU

Constructori implicați

- § definit de utilizator – constructor ce nu are niciun parametru
- § generat de compilator – daca o clasă nu are niciun constructor definit atunci compilatorul generează unul automat, fară parametri al cărui corp nu conține nicio instrucțiune
- § constructor cu toți parametri implicați

```
class Time {  
public:  
    Time(){hour=minute=second=0;  
        cout<<“Apel constructor\n”;}  
    ...  
};  
...  
Time z;      z.PrintLong();
```

Constructori cu parametri

- § cu parametri ce nu iau valori implicite;
- § cu parametri ce iau valori implicite.

```
class Time {  
    ...  
public:  
    Time(int h=0, int m = 0, int s= 0){  
        hour=h; minute=m; second=s;  
    } //parametri implicați
```

```
class Time {  
    ...  
public:  
    Time(int h, int m, int s){  
        hour=h; minute=m; second=s;  
    } //parametri ce nu iau valori implicite
```

Apelul constructorilor

```
int main(){  
    Time t1;           //toate argumentele implice  
    Time t2(2);        //minute și second implice  
    Time t3(21, 34);   //second implicit  
    Time t4(12, 25, 42); //toate valorile specificate  
    Time t5(27, 74, 99); //toate valorile eronate  
}
```

t1 – reprezintă timpul **0:0:0**
t2 – reprezintă timpul **2:0:0**
t3 – reprezintă timpul **21:34:0**
t4 – reprezintă timpul **12:25:42**

Constructori de copiere

- § definiți de utilizator;
- § generați de compilator
- § primește ca argument o referință la un obiect din clasă și initializează obiectul nou creat folosind datele conținute în obiectul referință. Pentru crearea unui obiect printr-un constructor de copiere, argumentul transmis trebuie să fie o referință la un obiect din aceeași clasă.
- § este definit de programator; în caz contrar, compilatorul generează un constructor de copiere care copiază datele membru cu membru din obiectul referință în obiectul nou creat.

dr. Silviu GÎNCU

```
class IdNumeClasa {  
    IdNumeClasa (IdNumeClasa &ob);  
};  
  
class Time {  
    ...  
    Time(Time &t1){  
        hour = t1.hour;  
        minute = t1.minute;  
        second = t1.second;  
    }  
};  
  
Time t1(2,25,30), t2(t1);  
t1.PrintLong(); t2.PrintLong();
```

Problema 3

Pentru tipul de dată Data calendaristică se definească o clasă care să conțină toate cele 3 categorii de constructori descriși.

```
class Date{
public:
    Date(){day=month=year=0;}
    Date (int a, int b, int c){day=a; month=b; year=c;}
    Date (Date const &c){
        day=c.day; month=c.month; year=c.year;}
    void print();
private:
    int day, month, year;
};
void Date::print() {
    cout<<day<<"/ " <<month " / " <<year<<endl;
}
```

```
int main(){
    Date date1, date2(15,9,2018), date3(date2);
    cout << "date1 = ";           date1.print();
    cout << "date2 = ";           date2.print();
    cout<<"Dupa copierea membru cu membru, date1 = ";
    date1 = date2;               date1.print();
    cout<< "Constructorul de copiere, date3 = ";
    date3.print();
}
```

Funcția specială destructor

§ Destructor – o metodă publică specială:

Ø numele metodei = ~nume_clasă

Ø fără tip de return (nici măcar void)

§ Scop:

- eliberează resursele de memorie alocate de constructor

§ apelat automat la sfârșitul domeniului de vizibilitate:

- la sfârșitul programului -> pentru obiectele globale
- la sfârșitul funcției -> pentru obiectele locale

§ Nu are parametri

§ O clasă poate avea un singur destructor

```
class CreateAndDestroy{
public:
    CreateAndDestroy(); //constructor
    ~CreateAndDestroy(); //destructor
};
CreateAndDestroy::CreateAndDestroy(){
    cout << "Constructorul obiectului " << endl;
}
CreateAndDestroy::~CreateAndDestroy(){
    cout << "Destructorul obiectului " << endl;
}
int main(){
    CreateAndDestroy t;
}
```

Manevrarea dinamica a obiectelor

- § Obiectele, ca și alte tipuri de date ale limbajului C++, pot fi gestionate dinamic prin intermediul operatorilor **new** și **delete**.
- § De exemplu, prin intermediul instrucțiunii **Date *p=new Date(20,10,2021);** se realizează rezervarea de memorie pentru obiectul specificat (Data calendaristică), după care se va apela constructorul obiectului.
- § Asemănător se poate declara un tablou de obiecte:
Date *d=new Date[10]; se realizează rezervarea de memorie pentru 10 obiecte de tip (Data calendaristică), după care se va apela constructorul pentru fiecare din cele 10 obiecte.
- § Pentru eliberarea memoriei se va utiliza operatorul **delete**
 - **delete p;** - pentru un obiect
 - **delete [] d;** - pentru tablou

dr. Silviu GÎNCU

Problema 4

Să se creeze o clasă care să implementeze tipul de date Array (tablou unidimensional) cu metode corespunzătoare tipului de dată creat

```
class Array{
public:
    Array(){c=0;t=NULL;}
    Array(int const);
    Array(Array const &);
    ~Array();
    void print();
    void put_n(int);
private:
    int *t,c;
};
Array::Array(int const k){
    c=k; t=new int[c]; put_n(100);
}
```

```
Array::Array(Array const &k){
    if(k.c==0){c=0; t=NULL;} else{
        c=k.c; t=new int[c];
        for(int i=0;i<c;i++) t[i]=k.t[i];
    }
}
Array::~Array(){if(c!=0) delete [] t;}

void Array::put_n(int n){
    if(c==0) {c=10; t=new int[c];}
    for(int i=0;i<c;i++) t[i]=rand()%n;
}
```

```
void Array::print(){
    if(c==0) cout<<"Vectorul nu contine elemente"<<endl;
    else{
        cout<<"Elementele vectorului: ";
        for(int i=0;i<c;i++) cout<<setw(5)<<t[i]; cout<<endl;
    }
}
```

```
int main(){
    cout<<"Obiectul a1"<<endl;
    Array a1; a1.print(); a1.put_n(50); a1.print();
    cout<<"Obiectul a2"<<endl;
    Array a2(5); a2.put_n(100); a2.print();
    cout<<"Obiectul a3"<<endl;
    Array a3(a1); a3.print();
    cout<<"Obiectul a4"<<endl;
    Array a4(a2); a4.print();
}
```

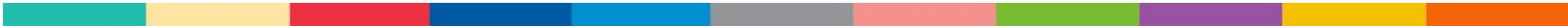
```
Obiectul a1
Vectorul nu contine elemente
Elementele vectorului: 41 17 34 0 19 24 28 8 12 14
Obiectul a2
Elementele vectorului: 91 95 42 27 36
Obiectul a3
Elementele vectorului: 41 17 34 0 19 24 28 8 12 14
Obiectul a4
Elementele vectorului: 91 95 42 27 36
```

Teme pentru acasă

- § A învăța și repeta conceptele prezentate în cadrul lecției
- § A depana programele prezentate în cadrul lecției
- § A dezvolta programul de pe slide-ul 14 (clasa carte), cu implementarea metodelor descrise și după caz completarea cu noi metode
- § A completa clasa Array cu metode de calcul (suma elementelor), metode de căutare (min, max, etc), ordonare a elementelor, precum și de modificare a elementelor tabloului

Prelegerea următoare

1. Cuvântul cheie „**this**”
2. Date statice
3. Pointeri la date și funcții membre
4. Tablouri de obiecte
5. Funcții **friend**
6. Clase **friend**



Programarea Orientată pe Obiecte (POO)

Prelegere

*Particularități: clase și obiecte.
Funcții și clase friend*

SUMAR

1. Cuvântul cheie „**this**”
2. Date statice
3. Tablouri de obiecte
4. Funcții **friend**
5. Clase **friend**

Cuvântului cheie “**this**”

- § Semnificația cuvântului cheie “**this**” este de pointer către **obiectul curent**.
- § Utilizarea lui se justifică în două situații:
 1. De multe ori, constructorii au unul sau mai mulți parametri de același tip și cu același nume ca datele membru. În acest caz, pentru a face diferența între parametru și data membru, se utilizează cuvântul cheie “**this**”.
De exemplu, dacă data membru este x, și parametrul constructorului este tot x, atunci data membru se adresează prin **this->x**.
 2. Uneori se dorește ca anumite metode să întoarcă:
 - obiectul curent
 - un pointer către acesta
 - o referință către obiectul curent
- § În toate aceste cazuri, valoarea returnată se obține cu ajutorul operatorului “**this**”.

dr. Silviu GÎNCU

Exemplu de utilizare

```
class num{  
    int x;  
public:  
    num (int x);  
    num& incr();  
    void print();  
};  
num::num(int x){ this->x = x; }  
num& num::incr() {x++; return *this; }  
void num::print(){ cout<<x<<endl; }  
int main(){  
    num a(3);      a.print(); //afis 3  
    a.incr(); a.print(); //afis 4  
    a.incr().incr();  
    a.print(); //afis 6  
}
```

- § O prima utilizare a lui “**this**” se observă în alcătuirea constructorului. Parametrul este tot x, aşa că, pentru a accesa data membru, utilizăm expresia **this->x**.
- § O a doua utilizare a lui “**this**” este în cadrul metodei membru **incr()** care incrementează valoarea reținută de data membru x și returnează o referință către obiectul curent, cu ajutorul căreia se poate din nou, incrementa x.
- § Astfel, dacă obiectul a are inițial valoarea 3, x va reține 3, a.**incr()** are ca efect faptul că x va reține 4, dar în același timp și referința către x, căreia i se poate aplica din nou **incr()**.

Cuvântului cheie “**this**”

- § Accesul membrilor clasei prin pointerul this se realizează
 - operatorul săgeată -> pentru pointerul this la obiect: **this->x**
 - operatorul punct . pentru pointerul this dereferențiat: (***this**).x
- § Folosirea parantezelor care încadrează pointerul dereferențiat ***this** este obligatorie pentru că operatorul . are precedență mai mare decât *
- § Pointerul **this** este folosit implicit pentru a referi datele membre și funcțiile membre ale unui obiect. El poate fi folosit și explicit
- § Tipul pointerului **this** depinde de tipul obiectului și de caracterul const sau non-const al funcției membre apelate de obiect
- § Pointerul **this** al unui obiect nu este parte a obiectului, dar el este introdus de compilator ca prim argument în fiecare apel al funcțiilor nestatice realizat de obiect
- § Fiecare obiect are acces la propria adresă de memorie prin intermediul pointerului numit **this**

dr. Silviu GÎNCU

Problema 1

Să se creeze o clasă care să simuleze funcționalitatea funcției putere (pow).

```
class putere{
    double b; int e; double valoare;
public:
    putere(double baza, int exponent);
    double calculeaza() {return this->valoare ;}
};
putere::putere(double baza, int exponent){
    this->b=baza; this->e=exponent; this->valoare=1;
    if(exponent==0) return;
    for(;exponent>0;exponent--)
        this->valoare=this->valoare * this->b;
}
```

```
int main(){
    putere x(4,3); putere y(2.5,1); putere z(5.7,0);
    cout<<x.calculeaza()<<endl; // afiseaza 64
    cout<<y.calculeaza()<<endl; // afiseaza 2.5
    cout<<z.calculeaza()<<endl; // afiseaza 1
}
```

Date membru de tip **static**

- § O dată membră a unei clase poate fi declarată static în declarația clasei. În acest caz, va exista o singură copie a unei date de tip static, care nu aparține nici unuia dintre obiectele (instanțele) clasei, dar este partajată de toate acestea.
- § O variabilă membră de tip static a unei clase există înainte de a fi creat un obiect din clasa respectivă și, dacă nu este inițializată explicit, este inițializată implicit cu 0.
- § Declararea unei date statice se face în interiorul unei clase, iar definiția trebuie să se facă în afara clasei și este permisă doar o singură definiție. Sintaxa declarări unei date statice:

```
class A {  
    static int i;  
    public: // ....  
};
```

iar ulterior, definiția: **int A::i=1;**

- § Este asemănătoare cu declararea unei variabile globale: **int i=1;**

dr. Silviu GÎNCU

Problema 2

Se consideră următorul Program

```
class S{
    int v;
    static int s;      // declaratia var. statice s
public:
    S() { v = 0; }
    int gets() {return s;}    int getv() {return v;}
    void incs() {s++;}        void incv() {v++;}
};
int S::s;      // definitia var. statice s a clasei S
int main (){
    S x, y;
    cout << "Inainte de incrementare\n";
    cout <<"x.s: "<<x.gets()<<"y.s: "<<y.gets()<< endl;
    cout <<"x.v: "<<x.getv()<<"y.v: "<<y.getv()<< endl;
    x.incs();  x.incv();    cout << "Dupa incrementare\n";
    cout <<"x.s: "<<x.gets()<<"y.s: "<<y.gets()<< endl;
    cout <<"x.v: "<<x.getv()<<"y.v: "<<y.getv()<< endl;
}
```

Date membru de tip **static**

- § La execuția programului se afișează conținutul variabilelor **s** și **v** pentru obiectele **x** și **y**.
- § Diferența între comportarea unei date membre de tip **static** și a unei date normale este: după incrementarea variabilelor **s** și **v** pentru obiectul **x**, obiectele **x** și **y** văd aceeași valoare a variabilei statice **s** și valori diferite ale variabilei normale **v**.
- § Astfel, rezultatul afișat este după cum urmează:

Inainte de incrementare

x.s: 0 **y.s: 0**

x.v: 0 **y.v: 0**

Dupa incrementare

x.s: 1 **y.s: 1**

x.v: 1 **y.v: 0**

Problema 3

De la tastatură se citesc coordonatele a n puncte. Să se elaboreze un program, cu utilizarea obiectelor de tip punct, prin intermediul căruia se va determina cele mai îndepărtate două puncte și se va afișa distanța dintre ele.

```
class Point{
    int x, y;
public:
    Point (int x, int y){this->x=x;this->y=y;}
    void print();
    int X()const {return x;}
    int Y()const {return y;}
    static int n;
};
int Point::n=4;
void Point::print(){cout<<"x="<<x<<" " <<"y="<<y<<endl;
}
```

```
float distance(Point a, Point b){  
    return pow(pow((b.X()-a.X()),2)+pow((b.Y()-a.Y()),2),0.5);  
}  
int main(){  
    Point *t[Point::n];  
    t[0]=new Point(2,4); t[1]=new Point(1,6); t[2]=new Point(5,3); t[3]=new Point(0,0);  
    cout<<"Coordonatele punctelor"<<endl;  
    for(int i=0;i<(Point::n);i++) t[i]->print();  
    cout<<"Punctele cele mai indepartate"<<endl;  
    float m=distance(t[0],t[1]); int p1=0,p2=1;  
    for(int i=0;i<(Point::n)-1;i++)  
        for(int j=i+1;j<(Point::n);j++)  
            if(m<distance(t[i],t[j])){ m=distance(t[i],t[j]); p1=i; p2=j; }  
    cout<<"Distanta maxima "<<setprecision(2)<<m<<endl;  
    cout<<"Coordonatele punctelor sunt"<<endl;  
    t[p1]->print(); t[p2]->print();  
}
```

Problema 4

De la tastatură se citesc datele despre n ($n < 100$) sportivi, care au participat la o competiție sportivă. Despre fiecare sportiv se citește numele, prenumele, data nașterii (an/luna/zi), și timpul (ore:min:sec) obținut la concurs. Să se afișeze la ecran datele despre fiecare sportiv, sportivul cu timpul minim și sportivul de vârstă maximă. Se consideră că fiecare lună are exact 30 de zile.

Obiectul data	
Atribute	zi, luna an
Metode	Constructor, inițializare, transformare_in_zile, citire, afisare

Obiectul timp	
Atribute	ora, minute, sec
Metode	Constructor, inițializare, transformare_in_secunde, citire, afisare

Obiectul sportiv	
Atribute	Nume, prenume, data_nașterii, timpul_probei
Metode	Constructor, destructor, citire, afisare

Clasa Data

```
class data{
    int zi, luna, an;
public:
    data(int=0, int=0, int=0);
    void set_data(int, int, int);
    int in_zile(){return an*360+luna*30+zi;}
    void citire();
    void afisare();
}
void data::set_data(int zi, int luna, int an){
    this->zi=zi; this->luna=luna; this->an=an;
    while(this->zi>30){this->zi=this->zi-30;this->luna++;}
    while(this->luna>12){this->luna=this->luna-12;this->an++;}
}
data::data(int zi, int luna, int an){set_data(zi,luna,an);}
void data::citire(){ int zi, luna, an;
    cout<<"Scrieti data (zi/luna/an): ";
    cin>>zi>>luna>>an; set_data(zi,luna,an);
}
void data::afisare(){cout<<zi<<"/"<<luna<<"/ "<<an<< " " ;}
```

```
class timp{
    int ora, minute, sec;
public:
    timp(int=0, int=0, int=0);
    void set_timp(int, int, int);
    int in_sec(){return ora*360+minute*30+sec; }
    void citire();
    void afisare();
}
void timp::set_timp(int ora, int minute, int sec){
    this->ora=ora; this->minute=minute; this->sec=sec;
    while(this->sec>59){this->sec=this->sec-60;this->minute++;}
    while(this->minute>59){
        this->minute=this->minute-60;this->ora++; }
    timp::timp(int ora, int minute, int sec){set_timp(ora,minute,sec); }
void timp::citire(){int ora, minute, sec;
    cout<<"Scrieti timpul (ora:minute:sec): ";
    cin>>ora>>minute>>sec; set_timp(ora,minute,sec); }
void timp::afisare(){cout<<ora<<": "<<minute<<": "<<sec<< " ; }
```

```
class sportiv{
    char *nume, *prenume;
public:
    sportiv();
    ~sportiv();
    void citire();
    void afisare();
    data d;    timp t;
};

sportiv::sportiv(){nume=new char [15]; prenume=new char [15];}
sportiv::~sportiv(){delete [] nume; delete [] prenume;}
void sportiv::citire(){
    cout<<"Dati datele despre sportiv"<<endl;
    cout<<"Nume=";cin>>nume;
    cout<<"Prenume=";cin>>prenume; d.citire(); t.citire();
}
void sportiv::afisare(){
    cout<<nume<< " "<<prenume<< " ";
    d.afisare(); t.afisare(); cout<<endl;}
```

Funcția principală

```
int main(){
    sportiv s[10];
    timp tmin;
    data dmax;
    int p=0,i,n,q=0;
    cout<<"Dati nr. de sportivi "; cin>>n;
    for(i=0;i<n;i++) s[i].citire();
    dmax=s[0].d; tmin=s[0].t;
    cout<<"Datele despre sprotivi "<<endl;
    for(i=0;i<n;i++){
        s[i].afisare();
        if(dmax.in_zile()<s[i].d.in_zile()) {dmax=s[i].d;p=i;}
        if(tmin.in_sec()>s[i].t.in_sec()) {tmin=s[i].t;q=i;}
    }
    cout<<"Sportivul cu timpul minim: "<<endl;
    s[q].afisare();
    cout<<"Sportivul cu vîrstă maxima: "<<endl;
    s[p].afisare();
}
```

Funcții **friend**

- § Funcțiile prietene (**friend**) sunt funcții ne-membre ale unei clase, care au acces la datele membre private ale unei clase. Ele se declară în interiorul clasei.
- § Prototipurile unor astfel de funcții sunt precedate de cuvântul cheie **friend**.
- § Spre deosebire de funcțiile membre, funcțiile prietene ale unei clase nu posedă pointerul implicit **this**.
- § Accesul la datele clasei se realizează prin intermediul parametrului de tip clasă (nu prin numele obiectului clasei ca în cazul metodelor clasei).
- § Declararea unei funcții prietene f() clasei X se realizează astfel:

```
class X{  
    friend tip_returnat f(parametru_de_tip_clasă,lista_de_parametri);  
};  
tip_returnat f(parametru_de_tip_clasă, lista_de_parametri){  
    // corpul functiei  
}
```

dr. Silviu GÎNCU

Problema 5

Se consideră clasa Array, în care toate metodele, cu excepția celor speciale sunt declarate la nivel privat. Să se creeze o funcție de tip friend prin intermediul căreia se va realiza accesul la metodele protejate din cadrul clasei.

```
class Array{
public:
    Array(){c=0;t=NULL; }
    Array(int const);
    Array(Array const & );
    ~Array();
private:
    void print();
    void put_n(int);
    friend void prieten(Array &c);
    int *t,c;
};
```

```
void prieten(Array &c){
    c.put_n(50); c.print();
}
int main(){
    Array a;
    prieten(a);
}
```

Problema 6

Să se elaboreze un program prin intermediul căruia se va determina suma elementelor a unui obiect de tip vector cu un obiect de tip matrice.

```
#define dim 8
class Matrix;
class Array{
public:
    Array();
    ~Array();
    void print();
friend Matrix suma(Matrix &,Array &);
private:
int *t,c;
};
Array::Array(){c=dim; t=new int*[c];
for(int i=0;i<c;i++) t[i]=rand()%99;
}
```

```
class Matrix{
public:
    Matrix();
    ~Matrix();
    void print();
friend Matrix suma(Matrix &,Array &);
private: int **t,c;
};
Matrix::Matrix(){c=dim;t=new int*[c];
for(int i=0;i<c;i++) t[i]=new int[c];
for(int i=0;i<c;i++)
for(int j=0;j<c;j++)
t[i][j]=rand()%99;
}
```

```
Array::~Array(){delete t;}
void Array::print(){
    cout<<"Elementele vectorului: ";
    for(int i=0;i<c;i++)
        cout<<setw(5)<<t[i]; cout<<endl;
}
void Matrix::print(){
    cout<<"Elementele matrici: ";
    for(int i=0;i<c;i++){
        for(int j=0;j<c;j++)
            cout<<setw(5)<<t[i][j];
    }
    cout<<endl;
}
Matrix::~Matrix(){
    for(int i=0;i<c;i++) [] delete t[i];
    delete t;
}
```

```
Matrix suma(Matrix &m, Array &a){
    Matrix k;
    for(int i=0;i<dim; i++)
        for(int j=0;j<dim; j++)
            k.t[i][j]=m.t[i][j]+a.t[i];
    return k;
}
int main(){
    Array ob_1; ob_1.print();
    Matrix ob_2; ob_2.print();
    cout<<"suma elementelor"<<endl;
    Matrix ob_3=suma(ob_2,ob_1);
    ob_3.print();
}
```

Clase **friend**

§ În cazul în care se dorește ca toate funcțiile membre ale unei clase să aibă acces la membrii privați ai altor clase (să fie funcții prietene), prima clasă poate fi declarată clasa prietenă pentru cea de-a doua clasă conform următorului model:

```
class cls1;  
class cls2{  
    friend cls1;  
};
```

§ Relația de clasa prietenă nu este tranzitivă. Astfel, dacă clasa cls1 este clasa prietenă a clasei cls2, iar clasa cls2 este clasa prietenă a clasei cls3, aceasta nu implică faptul că cls1 este clasa prietenă pentru cls3.

Problema 7

În cadrul acestui program se prezintă modalitatea de crearea a claselor prietene

```
class Patrat;
class Dreptunghi {
    int lungime, latime;
public:
    int aria(){return (lungime*latime); }
    void conv (Patrat a);
};
class Patrat {
    int latura;
public:
    void seteaza_latura(int a){latura=a;}
    friend class Dreptunghi;
};
void Dreptunghi::conv(Patrat a){
    lungime=a.latura; latime=a.latura;
}
```

```
int main() {
    int lp; Patrat ptrt;
    Dreptunghi drpt;
    cout<<"Dati lat. patratului:" ;
    cin>>lp;
    ptrt.seteaza_latura(lp);
    drpt.converteste(ptrt);
    cout<<"Aria dreptunghiului:" ;
    cout<<drpt.aria();
}
```

Problema 8

Se consideră clasa Punct, definită prin coordonatele x și y. Să se creeze clasa segment, care va avea în calitate de date, două puncte (definite prin intermediul clasei Punct), să se implementeze o metodă de determinare a lungimii segmentului și de afișare a coordonatelor acestuia.

```
class Segment;
class Punct{ double x, y;
void afisare();
public:
Punct(double x=0, double y=0);
friend class Segment;
};
class Segment{ Punct o,v;
public:
Segment (Punct o, Punct v);
double lungime();
void afisare();
};
Punct::Punct(double x, double y){
this->x=x; this->y=y;
}
void Punct::afisare(){
cout<<" (" <<x<< ", "<<y<< ")" ;}
```

```
Segment::Segment (Punct o, Punct v){
this->o=o; this->v=v;
}
double Segment::lungime() {
return sqrt((o.x-v.x)*(o.x-v.x)+(o.y-v.y)*(o.y-v.y));
}
void Segment::afisare(){
cout<<" [ " ; o.afisare(); cout<<"," ;
v.afisare(); cout<< " ] " ;
}
int main(){ Punct o(1,0), v(4,4);
Segment s(o,v);
s.afisare();
cout<< "\nLungime = ";
cout << s.lungime();
}
```

Teme pentru acasă

- § A învăță și repeta conceptele prezentate în cadrul lecției
- § A depana programele prezentate în cadrul lecției

Prelegerea următoare

1. Suprâncărcarea operatorilor. Privire de ansamblu
2. Reguli privind suprâncărcarea operatorilor
3. Suprâncărcarea operatorilor binari
4. Suprâncărcarea operatorilor unari
5. Conversii



Programarea Orientată pe Obiecte (POO)

Prelegere

Suprîncărcarea operatorilor

SUMAR

1. Privire de ansamblu
2. Reguli privind supraîncărcarea operatorilor
3. Supraîncărcarea operatorilor binari
4. Supraîncărcarea operatorilor unari
5. Conversii

Privire de ansamblu

- § Un tip de date (obiect) este caracterizat prin:
- Domeniul de valori admisibile;
 - Set de operații caracteristice tipului.

Spre exemplu, tipul de date **număr întreg** (în C++ int)

- Domeniul de valori: -32768 ... +32767
- operații caracteristice: adunare, scădere, înmulțire și.a.

- § De regulă, operațiile caracteristice sunt implementate la nivel de operatori, astfel pentru variabile de tipul de dată întreg:

int a,b,c;

- § Operațiile caracteristice sunt implementate la nivel de operatori **c+a*b**

- § Care este mult mai simplă decât prin funcții: **c.adunare(a.produs(b))**

- § Un **operator** poate fi privit ca o funcție, în care termenii sunt argumente.

În lipsa operatorului + expresia **a+b** s-ar calcula apelând funcția **aduna(a,b)**.

Să analizăm următoare sevență de cod

```
#include "complex.h"
main(){
    complex x(3,4.4),y(2.5,4.87),z;
    double a=4.25;
    cout<<"Valorile introduse" << endl;
    cout<<"x=" <<x<< endl;
    cout<<"y=" <<y<< endl;
    cout<<"a=" <<a<< endl;
    cout<<"Operatii cu numere complexe" << endl;
    z=x+y; cout<<x<< "+" <<y<< "=" <<z<< endl;
    z=x+a; cout<<x<< "+" <<a<< "=" <<z<< endl;
    z=a+y; cout<<a<< "+" <<y<< "=" <<z<< endl;
```

```
z=x/y;
cout<<x<< "/" <<y<< "=" <<z<< endl;
z=x-y;
cout<<x<< "-" <<y<< "=" <<z<< endl;
z=x*y;
cout<<x<< "*" <<y<< "=" <<z<< endl;
cout<<" (x+y)-(a+y)++x---y";
z= (x+y)-(a+y)++x---y;
cout<< endl << "=" <<z<< endl;
}
```

Noțiuni fundamentale (I)

- § Operația de adunare + funcționează pentru variabile de tip **int**, **float**, **double** etc.
 - Operatorul + a fost supraîncărcat chiar în limbajul de programare C++
- § Pentru ca un operator să poată lucra asupra obiectelor unei noi clase, el trebuie să fie supraîncărcat pentru acea clasă
- § Operatorii se supraîncarcă scriind o definiție de funcție obișnuită, având un nume special - cuvântul cheie **operator** urmat de simbolul operatorului care urmează să fie supraîncărcat
- § **Exemplu**
 - **operator+** este numele funcției prin care se supraîncarcă operatorul +

Noțiuni fundamentale (II)

§ Scopul supraîncărcării operatorilor

- Scrierea expresiilor concise pentru obiecte care fac parte din clase definite de programatori în același fel în care se scriu pentru tipurile predefinite

§ Greșeli la supraîncărcarea operatorilor

- Operația implementată de operatorul supraîncărcat nu corespunde din punct de vedere semantic operatorului

§ Exemple

- Supraîncărcarea operatorului + printr-o operație similară scăderii
- Supraîncărcarea operatorului / ca să implementeze o înmulțire

Reguli pentru supraîncărcarea operatorilor

- § Setul de operatori ai limbajul C++ nu poate fi extins prin asocierea de semnificații noi unor caractere, care nu sunt operatori (de exemplu nu putem defini operatorul „**”).
- § Prin supraîncărcarea unui operator nu i se poate modifica paritatea (astfel operatorul “!” este unar și poate fi supraîncărcat numai ca operator unar).
- § Nu se poate modifica precedența și asociativitatea operatorilor.
- § Operatorii supraîncărcați într-o clasă sunt moșteniți în clasele derivate excepție face operatorul de atribuire „=”.
- § Unui operator i se poate atribui orice semnificație, însă se recomandă ca aceasta să fie cât mai apropiată de semnificația naturală.

dr. Silviu GÎNCU

Supraîncărcarea operatorilor

- § Limbajul C++ permite programatorului să definească diverse operații cu obiecte ale claselor, folosind simbolurile operatorilor standard. Un tip clasă se poate defini împreună cu un set de operatori asociați, obținuți prin supraîncărcarea operatorilor existenți.
- § În acest fel, se efectuează operații specifice cu noul tip la fel de simplu ca în cazul tipurilor standard.
- § Pot fi supraîncărcați următorii operatori: + - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= >>= <<= == != <= >= && || ++ -- ->*, -> << >> [] () **new new[] delete delete[]**
- § Nu pot fi supraîncărcați operatorii: :: . .* ?: sizeof.
- § Procesul de supraîncărcare a unui operator poate fi realizat prin :
 - intermediul unei funcții membru;
 - intermediul unei funcții prietene.

dr. Silviu GÎNCU

Supraîncărcarea operatorilor prin funcții membru

§ Sintaxa:

```
class nume_clasa{
    Tip_returnat operator S(lista_de_parametri);
};

Tip_returnat nume_clasa::operator S(lista_de_parametri){
    // descrierea procesului de supraîncărcare
} unde S este operatorul supraîncărcat, tip – tipul rezultatului operației S, iar lista_de_parametri reprezintă
```

§ !!! **Observație** primul argument va fi întotdeauna un operand de tip clasă.

```
class Complex{           // operația a+b   a este primul operand, b operandul doi
    double x,y;
public:
    complex operator+(complex arg_doi);
};
```

Supraîncărcarea operatorilor prin funcții friend

§ Sintaxa:

```
class nume_clasa{
    friend Tip_returnat operator S(lista_de_parametri);
};

Tip_returnat operator S(lista_de_parametri){
    // descrierea procesului de supraîncărcare
} unde S este operatorul supraîncărcat, tip – tipul rezultatului operației S, iar lista_de_parametri reprezintă
```

```
class Complex{           // operația a+b   a este primul operand, b operandul doi
    double x,y;
public:
    friend complex operator+(complex arg_1, complex arg_doi);
};
```

SUMAR

Supraîncărcarea operatorilor binari:

- § Supraîncărcarea operatorilor aritmetici
- § Supraîncărcarea operatorilor de comparație
- § Supraîncărcarea operatorilor compuși
- § Supraîncărcarea operatorilor << și >>
- § Supraîncărcarea operatorului de atribuire
- § Supraîncărcarea operatorului de indexare („[]”)
- § Supraîncărcarea operatorului apel de funcție(„()”)

Crearea tipului de date complex

- § Așa cum adunarea a două numere complexe este comutativă, vom examina cazurile:
 - complex + complex
 - double + complex
 - complex + double
- § În primul și al treilea caz supraîncărcarea poate fi efectuată prin intermediul funcțiilor membru, însă în cazul doi supraîncărcarea este posibilă numai prin intermediul unei funcție prieten, deoarece primul parametru al unei funcții membru operator este implicit de tipul clasei și nu poate fi de alt tip.

Suprîncărcarea operatorilor aritmetici

```
class complex{
    double x,y; //complex=x+yi
public:
    complex(){x=y=0;}      complex(double a, double b){x=a;y=b;}
    void afis() {cout<<x<<"+"<<y<<"i\n";}
    complex operator +(complex);
    friend complex operator +(double,complex);
    complex operator+(double);
};

complex complex::operator+(complex b){
    complex c; c.x=x+b.x;  c.y=y+b.y;      return c;
}
complex complex::operator+(double b){
    complex c; c.x=x+b;   c.y=y;           return c;
}
complex operator+(double a, complex b){
    complex c; c.x=a+b.x; c.y=b.y;         return c;
}
```

Supraîncărcarea operatorilor de comparație

Așa cum limbajul de programare C++ nu are implementat tipul de date logic, se va considera valoarea 1 echivalentă valorii true, iar valoare 0 echivalentă valorii false. În acest context tipul rezultatului unei operații de comparare a două valori numerice poate fi int, având valorile posibile 1 sau 0.

```
class complex{
    double x,y;
public:
    int operator==(complex);
};
int complex::operator==(complex ob){
    if(x==ob.x && y==ob.y) return 1;
    else return 0;
}
```

```
class complex{
    double x,y;
public:
    friend int operator==(complex,complex);
};
int operator==(complex ob1,complex ob2){
    if(ob1.x==ob2.x && ob1.y==ob2.y)
        return 1;    else return 0;
}
```

```
int main(){    complex a, b;
    if(a==b) cout<<"Numere egale";
    else cout<<"Numere diferite";
}
```

Supraîncărcarea operatorilor compuși

```
class complex{
    double x,y;
public:
    complex( ){x=y=0; }
    complex(double a, double b){x=a;y=b; }
    void afis() {cout<<x<<"+"<<y<<"i\n"; }
    complex operator += (complex &);
    complex operator += (double);
};

complex complex::operator+=(complex &a){ x+=a.x; y += a.y; return *this; }
complex complex::operator+=(double d){
    x+= d; return *this;
}

int main(){    complex a, b(2,3);
    a+=10; a.afis();          a+=b;   a.afis();
}
```

Supraîncărcarea operatorilor << și >>

- § Operațiile de intrare/ieșire în C++ se efectuează prin intermediul operatorilor de inserție << și respectiv operatorul de extragere >> din streamuri. Spre deosebire de alți operatori, operatorii de intrare/ieșire pot fi supraîncărcați doar prin intermediul funcțiilor friend.
- § Supraîncărcarea operatorilor de intrare/ieșire se poate face respectând sintaxa:

```
class nume_clasa {  
    friend ostream & operator<<(ostream& os, nume_clasa nume);  
    friend istream & operator>>(istream& is, nume_clasa &nume);  
};  
ostream& operator<<(ostream& os, tip_clasa nume){  
    // corpul functiei  
    return os; }  
istream& operator>>(istream& is, tip_clasa &nume){  
    // corpul functiei  
    return is; }
```

Supraîncărcarea operatorilor intrare/ieșire

```
class complex {    double x, y;  
public:  
    friend ostream& operator << (ostream& os, complex z);  
    friend istream& operator >>(istream& is, complex& z);  
};  
ostream& operator<<(ostream& os, complex z){  
    os <<z.x;    if(z.y==0) os<<endl;  
    if(z.y<0) os<<"-"<<z.y<<"i"<<endl;    if(z.y>0) os<<"+"<<z.y<<"i"<<endl;  
    return os;  
}  
istream& operator>>(istream& is, complex& z){  
    cout<<"Dati partea reala "; is >> z.x;  
    cout<<"Dati partea imaginara"; is>> z.y;  
    return is;  
}
```

Supraîncărcarea operatorului de atribuire

- § Atunci când se utilizează operatorul “=”, fără supraîncarcare, se realizează o “copiere bit cu bit”. Aceasta nu înseamnă, că întotdeauna obținem rezultatele dorite. Acest lucru se întâmplă, deoarece cele două obiecte care formează atribuirea: ex. `a = b`, primesc aceeași adresa.
- § În cazul în care obiectul `b` se distrugе, atunci, obiectul `a` va avea o adresa spre “un obiect care nu mai există”. Pentru a putea înlătura acest neajuns, trebuie să supraîncărcăm operatorul “=”.
- § **Operatorul “=” este binar**
- § Dacă acesta este supraîncărcat prin utilizarea unei funcții membru, atunci primul operand este obiectul curent, iar al doilea operand este parametrul funcției.
- § **Exemplu** Considerăm clasa `sir`, în care fiecare obiect reține adresa unui sir de caractere.
Astfel, data membru `adr` reține adresa unui pointer către sir, iar sirul va fi alocat dinamic cu ajutorul operatorului `new` într-o zonă de memorie disponibilă.

dr. Silviu GÎNCU

Supraîncărcarea operatorului de atribuire

```
class sir{    char *adr;
public:
    sir(char s[]);        ~sir();
    void afis(){ cout<<adr<<endl; }
    void operator=(sir& sirul);
};

sir::sir(char s[ ]) {adr = new char[strlen(s) + 1]; strcpy(adr, s);}
sir::~sir() {delete[ ] adr; adr = 0;}
void sir::operator=(sir& sirul) {cout<<"Operator = \n"; delete[ ] adr;
    adr = new char[strlen(sirul.adr)+1]; strcpy(adr,sirul.adr);}
}

int main() {
    sir s("un sir"), t("alt sir"); s.afis(); t.afis();
    s = t; t.~sir(); s.afis(); cout<<"Sf. Program\n";
}
```

Supraîncărcarea operatorului de indexare „[]”

§ Operatorul **de indexare []** este binar și are forma:

expresie_1[expresie_2]

§ Pentru supraîncărcarea acestui operator trebuie să folosim o funcție **operator[]**, care trebuie să fie membră a clasei, să fie nestatică și să aibă forma:

x[n] sau x.operator[] (n)

§ Al doilea operand, care are rolul indexului, poate fi la supraîncărcare de orice tip.

§ **Exemplu** *Să elaborăm un program prin intermediul căruia vom crea un tablou de înregistrări, ce conțin informații de tip medical despre persoane.*

Accesul la înregistrări urmează a realizat după nume, după greutate, după numărul de ordine.

În prima căutare - supraîncărcarea operatorului de indexare având ca parametru un sir de caractere care indică numele persoanei căutate.

*În al doilea caz, vom face supraîncărcarea operatorului de indexare având ca parametru de tip **float**.*

dr. Silviu GÎNCU

Suprîncărcarea operatorului de indexare „[]”

```
class analize;
class pers{
    char nume[35];    double greutate;    int varsta;
    friend class analize;
public:
    void init(char *s, double gr, int v);
    void tipar();
};

void pers::tipar(){
cout<<"\nPersoana: "<<nume<<"\tGreutatea: "<<greutate<<"Varsta: "<<varsta;
}

void pers::init(char *s, double gr, int v){
    strcpy(nume, s);    greutate=gr;    varsta=v;
}
```

```
class analize{    pers *sir;    int n;
public:
analize(){ n=5; sir=new pers[n]; }
analize(int nr){n=nr; sir=new pers[n]; }
pers *operator[] (char *);
pers *operator[] (double);
pers *operator[] (int);
void introd();
};
void analize::introd(){
for(int i=0;i<n;i++){
    cout<<endl;    cout<<"Persoana "<<i+1<<": ";
    cin>>sir[i].nume;    cout<<"Greutatea: ";
    cin>>sir[i].greutate;    cout<<"Varsta: ";
    cin>>sir[i].varsta;
}
}
```

```
pers *analize::operator[ ](char *nume) {
    for(int i=0;i<n;i++)
        if(strcmp(sir[i].nume, nume)==0) return &sir[i];
    return NULL;
}

pers *analize::operator[](double g) {
    for(int i=0;i<n;i++)
        if(sir[i].greutate==g) return &sir[i];
    return NULL;
}

pers *analize::operator[] (int index) {
    if(index<=n) return &sir[index-1];
    else return NULL;
}
```

```
int main(){
    char c;  int nr;
    cout<<"Cate analize efectuati ? ";
    cin>>nr;  analize t(nr); t.intro();
    while(1){
        cout<<"\nOptiunea [g]reutate, [n]ume, [i]ndex, [e]xit ? ";  cin>>c;
        switch(c){
            case 'g' : double g; cout<<"Greutatea: ";cin>>g; t[g]->tipar(); break;
            case 'n' : char n[100]; cout<<"Numele: ";cin>>n; t[n]->tipar(); break;
            case 'i' : int i;  cout<<"Nr. de index: ";cin>>i;t[i]->tipar(); break;
            case 'e': return 0;
        }
    }
}
```

Supraîncărcarea operatorului **apel de funcție „()”**

§ Când este realizată supraîncărcarea operatorului “apel de funcție”, aceasta nu va însemna o nouă variantă de supraîncărcare a unei funcții, ci supraîncărcarea unui operator binar nestatic de forma:

expresie (lista_expresiei);

§ Specificul utilizării acestei funcții operator:

- evaluarea și verificarea listei de argumente se face întocmai ca pentru o funcție obișnuită;
- chiar dacă operatorul este binar, nu are limită de argumente, deoarece al doilea argument este considerat o lista de argumente.
- atunci când lista de parametrii este vidă, al doilea argument poate lipsi.

§ **Exemplu** *Să elaborăm un program prin intermediul căruia vom utiliza operatorul “()” pentru accesarea unui element al tabloului bidimensional*

Supraîncărcarea operatorului **apel de funcție „()”**

```
#define dim 8
class Matrix{
    public:
        void print();
        Matrix();
        int & operator(int i,int j){
            return t[i][j];
        }
    private:
        int t[dim][dim];
}
Matrix::Matrix(){
    for(int i=0;i<dim;i++)
        for(int j=0;j<dim;j++)
            t[i][j]=rand()%100;
}
```

```
void Matrix::print(){
    for(int i=0;i<dim;i++){
        for(int j=0;j<dim;j++)
            cout<<setw(3)<<t[i][j];
        cout<<endl;
    }
}
int main(){
    Matrix a;
    cout<<"Elementele matricei"<<endl;
    a.print();
    cout<<"pe pozitia 1 2:";
    cout<<a(1,2)<<endl;
}
```

Teme pentru acasă

- § A învăță și repeta conceptele prezentate în cadrul lecției
- § A depana programele prezentate în cadrul lecției

Prelegerea următoare

1. Suprîncărcarea operatorilor unari
2. Conversii
3. Crearea tipului de date fracție

dr. Silviu GÎNCU



Programarea Orientată pe Obiecte (POO)

Prelegere

Suprîncărcarea operatorilor

SUMAR

1. Privire de ansamblu
2. Reguli privind supraîncărcarea operatorilor
3. Supraîncărcarea operatorilor binari
4. Supraîncărcarea operatorilor unari
5. Conversii
6. Crearea tipului de date fractie

Supraîncărcarea operatorilor unari ++ --

Operatorii unari pot fi supraîncărcați printr-o funcție membră nestatică (fără parametri expliciti) sau printr-o funcție prietenă cu un parametru explicit de tipul clasă.
Ca exemplu se prezintă supraîncărcarea operatorilor ++ și --

```
class complex{
    double x,y;
public:
    complex operator++();
    complex operator--();
};

complex complex::operator ++ () {
    x++; return *this;
}

complex complex::operator -- () {
    x--; return *this;
}
```

```
class complex{
    double x,y;
public:
    friend complex operator++(complex);
    friend complex operator--(complex);
};

complex operator ++ (complex a) {
    a.x++; return a;
}

complex operator -- (complex a) {
    a.x--; return a;
}
```

```
int main(){ complex a; ++a; --a; }
```

Supraîncărcarea operatorilor unari !

Pentru clasa complex să se supraîncarce operatorul !, astfel încât la apelarea acestuia să se determine modulul unui număr complex

```
class complex{
    double x,y;
public:
    complex(double a=0,double b=0)
    {x=a;y=b;}
    double operator !();
};

double complex::operator ! (){
    return pow(x*x+y*y,0.5);
}
```

```
class complex{
    double x,y;
public:
    complex(double a=0,double b=0)
    {x=a;y=b;}
    friend double operator !(complex);
};

double operator !(complex a){
    return pow(a.x*a.x+a.y*a.y,0.5);
}
```

```
int main(){ complex a(6,8);
cout<<!a; }
```

Conversii (I)

§ Există următoarele tipuri de conversii

- Conversii **implicite**
- Conversii **explicite**

§ **Conversiile implicite** au loc în următoarele situații:

- În cazul aplicării operatorului de atribuire
- La apelul unei funcții: Dacă tipul parametrilor efectivi (de apel) diferă de tipul parametrilor formali, se încearcă conversia tipului parametrilor efectivi la tipul parametrilor formali
- La revenirea dintr-o funcție: Dacă funcția returnează o valoare în programul apelant, la întâlnirea instrucțiunii return expresie; se încearcă conversia tipului expresiei la tipul funcției

§ **Conversiile explicite** pot fi:

- tip_predefinit_1 -> tip_predefinit_2
- tip_predefinit -> tip_definit_de_utilizator (clasă)
- clasă -> tip_predefinit
- clasă_1 -> clasă_2

dr. Silviu GÎNCU

Conversii (II)

- § Conversii **tip_predefinit_1 -> tip_predefinit_2**
- § Pentru realizarea unor astfel de conversii, se folosește operatorul unar de conversie explicită (**cast**), de forma:
(tip) operand
- § Exemplu:

```
int k; double x; x = (double) k / (k+1);
```
- § Conversii **tip_predefinit ->tip_definit_de_utilizator (clasa)**
- § Astfel de conversii se pot realiza atât implicit, cât și explicit, în cazul în care pentru clasa respectivă există un constructor cu parametri implicați, de tipul predefinit.
- § **Exemplu:** `class Fracție{`

```
    int nrt, nmt; Fracție( int nrt = 0, int nmt = 1);
```

`}`
- § **f = fractie(20);** conversie explicită se convertește întregul 20 într-un obiect al clasei **fracție (nrt=20 și nmt=1)**
- § **fracție f; f = 20;** conversie implicită se convertește operandul drept (de tip **int**) la tipul operandului stâng (tip **fracție**)

dr. Silviu GÎNCU

- ## Conversii clasă -> tip_predefinit
- § Acest tip de conversie se realizează printr-un operator special (cast) care convertește obiectul din clasă la tipul predefinit. La aplicarea operatorului se folosește una din construcțiile:
- (nume_tip_predefinit)obiect
 - nume_tip_predefinit (obiect)

```
class fractie{
    long nrt, nmt;
public:
    fractie(int n=0, int m=1) {nrt=n; nmt=m;}
    operator int(){return nrt/nmt;}
}
int main(){
    fractie a(5,4), b(3), c;
    int i=7, j=14; c=a; c=7;
```

```
//conversii explicite
cout<<"(int)a=<<(int)a<<'\n';
cout<<"int(a)=<<int(a)<<'\n';

//conversie implicita
int x=b;
}
```

Conversii clasă_1 -> clasă_2

```
class Point;
class Complex{
    double re, im;
public:
    Complex(){re = 0; im = 0;}
    Complex(double r, double i) {
        re = r; im = i;}
    operator Point();
};

class Point {
    int x,y;
public:
    friend Complex::operator Point();
};
```

```
Complex::operator Point(){
    Point tmp;
    tmp.x = re;
    tmp.y = im;
    return tmp;
}

int main(){
    Complex c1(4,7);
    Point p1;
    p1 = c1;
}
```

Problemă

- § Să se elaboreze un program prin intermediul căruia se vor gestiona tablouri unidimensionale, care să conțină în calitate de elemente ale sale fracții. Elementele tabloului se vor citi din fișier.
- § Tipul de date **fracție** se va defini prin intermediul unei clase și va avea în calitate de

Date membru:

- Numitorul și numărătorul

Metode:

- Constructor cu parametri implicați
- supraîncărcați operatorii aritmetici;
- supraîncărcați operatorii compuși;
- supraîncărcați operatorii de intrare/ieșire;
- supraîncărcați operatorii de comparație.

- § Tipul de date **tablou de fracții** se va crea prin intermediul unei clase și va avea în calitate de

Date membru:

- Numărul de elemente și un pointer pentru memorarea elementelor

Metode:

- Constructori și destructorul
- supraîncărcați operatorii de intrare/ieșire;
- pentru determinarea elementului maximal;
- pentru determinarea sumei elementelor;
- Pentru sortarea elementelor tabloului

fractie.in - Notepad

File Edit Format View Help

6
1 2
1 3
1 4
2 5
3 7
4 5

```
class fractie{
    int nrt, nmt;
public:
    fractie(int nrti=0, int nmti=1); ~fractie() {};
    void simplifica();
    friend fractie operator+(fractie&, fractie&);
    friend fractie operator-(fractie&, fractie&);
    friend fractie operator*(fractie&, fractie&);
    friend fractie operator/(fractie&, fractie&);
    fractie & operator+=(fractie&);
    fractie & operator-=(fractie&);
    fractie & operator*=(fractie&);
    fractie & operator/=(fractie&);
    int operator==(fractie);
    friend int operator!=(fractie, fractie);
    int operator>(fractie);
    int operator>=(fractie);
    int friend operator<(fractie, fractie);
    int friend operator<=(fractie, fractie);
    friend ostream & operator<<(ostream &, fractie);
    friend istream & operator>>(istream &, fractie &);
};
```

Funcții de suport

```
int cmmdc(int x,int y){  
    int z;  
    if (x==0 || y==1)  return 1;  
    if (x<0)  x=-x;  
    if (y<0)  y=-y;  
    while (x!=0){  
        if (y>x){  
            z=x;  x=y;  y=z;  
        }  
        x%=y;  
    }  
    return y;  
}
```

```
void fractie::simplifica(){  
    int cd;  
    if (nmt<0){  
        nrt=-nrt;  
        nmt=-nmt;  
    }  
    if (nmt>1){  
        cd=cmmdc(nrt,nmt);  
        if (cd>1) { nrt/=cd; nmt/=cd; }  
    }  
}
```

Constructorul și operatorii + -

```
fractie::fractie(int nri, int nmi) {
    nrt=nri; nmt=nmi; simplifica();
}
fractie operator +(fractie &f1, fractie &f2){
    int dc=cmmdc(f1.nmt,f2.nmt); fractie f;
    f.nmt=(f1.nmt/dc)*f2.nmt;
    f.nrt=f1.nrt*(f2.nmt/dc)+f2.nrt*(f1.nmt/dc);
    f.simplifica(); return f;
}
fractie operator -(fractie &f1, fractie &f2){
    int dc=cmmdc(f1.nmt,f2.nmt); fractie f;
    f.nmt=(f1.nmt/dc)*f2.nmt;
    f.nrt=f1.nrt*(f2.nmt/dc) - f2.nrt*(f1.nmt/dc);
    f.simplifica(); return f;
}
```

Operatorii * /

```
fractie operator * ( fractie &f1,  fractie &f2){  
    int dc=cmmdc(f1.nrt,f2.nmt);  
    fractie f;  
    if (dc>1){f1.nrt/=dc;  f2.nmt/=dc;  }  
    dc=cmmdc(f2.nrt,f1.nmt);  
    if (dc>1){f2.nrt/=dc;  f1.nmt/=dc;  }  
    f.nrt=f1.nrt*f2.nrt;      f.nmt=f1.nmt*f2.nmt;  
    return f;  
}  
fractie operator / (fractie &f1,  fractie &f2){  
    int dc=cmmdc(f1.nrt,f2.nrt);  fractie f;  
    if (dc>1){f1.nrt/=dc;  f2.nrt/=dc;  }  
    dc=cmmdc(f2.nmt,f1.nmt);  
    if (dc>1){f2.nmt/=dc;  f1.nmt/=dc;  }  
    f.nrt=f1.nrt*f2.nmt;  f.nmt=f1.nmt*f2.nrt;  
    return f;  
}
```

Operatorii compuși += -= *=

```
fractie& fractie::operator+=(fractie &f1){  
    int dc=cmmdc(nmt,f1.nmt);  
    nmt=(nmt/dc)*f1.nmt;    nrt=nrt*(f1.nmt/dc)+f1.nrt*(nmt/dc);  
    simplifica();    return *this;  
}  
fractie& fractie::operator-=(fractie &f1){  
    int dc=cmmdc(nmt,f1.nmt);  
    nmt=(nmt/dc)*f1.nmt;    nrt=nrt*(f1.nmt/dc)-f1.nrt*(nmt/dc);  
    simplifica();    return *this;  
}  
fractie& fractie::operator *=(fractie &f1){  
    fractie f1=f11;    int dc=cmmdc(nrt,f1.nmt);  
    if (dc>1){nrt/=dc;    f1.nmt/=dc; }    dc=cmmdc(f1.nrt,nmt);  
    if (dc>1){f1.nrt/=dc;    nmt/=dc; }  
    nrt=nrt*f1.nrt;    nmt=nmt*f1.nmt;    simplifica();  
    return *this;  
}
```

Operatorii de intrare/ieșire și /=

```
fractie& fractie::operator /=(fractie &f11){  
    fractie f1=f11;  int dc=cmmdc(nrt,f1.nrt);  
    if (dc>1){nrt/=dc;  f1.nrt/=dc; }  
    dc=cmmdc(f1.nmt,nmt);  
    if (dc>1){f1.nmt/=dc;  nmt/=dc; }  
    nrt=nrt*f1.nmt;    nmt=nmt*f1.nrt;  
    return *this;  
}  
ostream& operator<<(ostream &os, fractie f){  
    os << '(' << f.nrt << '/' << f.nmt << ')';  
    return os;  
}  
istream& operator>>(istream &is, fractie &f){  
    is >> f.nrt>>f.nmt;  f.simplifica();  
    return is;  
}
```

Operatorii de comparație

```
int fractie::operator ==(fractie f){
    if(nrt==f.nrt && nmt==f.nmt) return 1; else return 0;
}
int operator !=(fractie f1, fractie f2){
    if(f1==f2) return 0; else return 1;}
int fractie::operator >(fractie f){
    if(nrt*f.nmt>f.nrt*nmt) return 1; else return 0;
}
int fractie::operator >=(fractie f){
    if(nrt*f.nmt>=f.nrt*nmt) return 1; else return 0;
}
int operator <(fractie f1, fractie f2){
    if(f1>=f2) return 0; else return 1;}
int operator <=(fractie f1, fractie f2){
    if(f1>f2) return 0; else return 1;
}
```

Clasa tablou de fracții

```
class FArray{
    fractie *t;
    int n;
public:
    FArray(int);
    FArray();
    ~FArray();
    friend ostream & operator <<(ostream &, FArray);
    friend istream & operator >>(istream &, FArray &);
    fractie suma();
    fractie maxim();
    void sortare();
};

FArray::FArray(int k){n=k; t=new fractie[n];}
FArray::FArray(){n=0; t=0;}
FArray::~FArray(){delete []t; n=0;}
```

Funcția principală

```
ostream& operator<<(ostream &os, FArray f){
    for(int i=0;i<f.n;i++) os<<setw(8)<<f.t[i];  os<<endl;
    return os;}
istream& operator>>(istream &is, FArray &f){
    is>>f.n; f.t=new fractie[f.n];
    for(int i=0;i<f.n;i++){is>>f.t[i]; f.t[i].simplifica();}
    return is;}
fractie FArray::suma(){fractie y=t[0];
    for(int i=1;i<n;i++) y=y+t[i]; return y;}
fractie FArray::maxim(){fractie y=t[0];
    for(int i=1;i<n;i++) if(y<t[i]) y=t[i]; return y;}
void FArray::sortare(){fractie aux=t[0]; int i,j;
    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
            if(t[i]>t[j]){aux=t[i];t[i]=t[j];t[j]=aux;}
    }
```

Implementarea metodelor clasei (1)

```
int main() {
fractie f(1,10), f1, f2;
cout << "Introduceti prima fractie:\n"; cin >> f1;
cout << "\nIntroduceti a doua fractie:\n"; cin >> f2;
cout << "\nf1=" << f1; cout << "\nf2=" << f2 << "\n\n";
cout << f1 << " + " << f2 << " = " << (f1+f2) << endl;
cout << f1 << " - " << f2 << " = " << (f1-f2) << endl;
cout << f1 << " * " << f2 << " = " << (f1*f2) << endl;
cout << f1 << " / " << f2 << " = " << (f1/f2) << endl;
cout<<"Compararea fractilor "<<f1<<" si "<<f2<<endl;
cout<<"== - "<<(f1==f2)<<endl;
cout<<"!= - "<<(f1!=f2)<<endl;
cout<<"> - "<<(f1>f2)<<endl;
cout<<">= - "<<(f1>=f2)<<endl;
cout<<">< - "<<(f1<f2)<<endl;
cout<<">== - "<<(f1<=f2)<<endl;
```

Implementarea metodelor clasei (2)

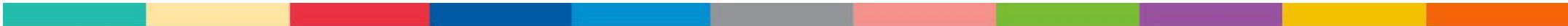
```
cout << endl; cout << f1 << " += " << f << " = ";
f1+=f; cout << f1 << endl; cout << f1 << " -= " << f << " = ";
f1-=f; cout << f1 << endl; cout << f1 << " *= " << f << " = ";
f1*=f; cout << f1 << endl; cout << f1 << " /= " << f << " = ";
f1/=f; cout << f1 << endl;
f1=f2; cout << "\nDupa instructiunea f1=f2, f1 = " << f1;
f1=5; cout << "\nDupa instructiunea f1=5, f1 = " << f1;
FArray x;
ifstream file("fractie.in"); file>>x;
cout<<"Fractiile din fisier"<<endl<<x;
cout<<"Elementele sortate:"<<endl;
x.sortare(); cout<<x;
cout<<"Elementul maximal=" <<x.maxim() << endl;
cout<<"Suma elementelor=" <<x.suma() << endl;
file.close();
}
```

Teme pentru acasă

- § A învăță și repeta conceptele prezentate în cadrul lecției
- § A depana programele prezentate în cadrul lecției

Prelegerea următoare

1. Moștenire. Privire de ansamblu
2. Derivarea simplă a claselor
3. Rolul modifierilor de acces
4. Constructori și destructori în relația de moștenire
5. Gestiunea funcțiilor membru în relația de moștenire



Programarea Orientată pe Obiecte (POO)

Prelegere

Moștenire. Derivarea simplă a claselor

SUMAR

1. Moștenire. Privire de ansamblu
2. Derivarea simplă a claselor
3. Rolul modifierilor de acces
4. Constructori și destructori în relația de moștenire
5. Gestiunea funcțiilor membru în relația de moștenire

Analiză comparativă

Obiectul student		Obiectul angajat	
Atribute	Nume, Prenume, Instituția, Grupa; Data nașterii; Bursa; Media_notelor.	Atribute	Nume, Prenume, Instituția, Functia; Data nașterii; Nr_ore_lucrate; Plata_oră.
Metode	Constructor/destructor, initializare, citire, afisare	Metode	Constructor/destructor, initializare, citire, afisare, salariu

Informații comune despre obiectele student și angajat

Atribute	Nume, Prenume, Instituția, Data nașterii
Metode	Constructor/destructor, initializare, citire, afisare

dr. Silviu GÎNCU

Moștenire, noțiuni generale (I)

- § Prin **moștenire** se înțelege acea proprietate a claselor prin care un tip nou construit poate prelua datele și metodele unui tip mai vechi
- § În C++ acest mecanism mai este cunoscut sub numele derivarea claselor
- § Moștenirea este o formă de reutilizare a codului în care noile clase sunt create din clase existente prin:
 - absorbirea atributelor și comportamentelor lor
 - prin înlocuirea unor comportamente
 - prin adăgarea unor attribute și comportamente noi
- § Într-o relație de moștenire dintre două clase nu se moștenesc:
 - Constructorii și destructorii
 - Supraîncărcarea operatorului de atribuire = dr. Silviu GÎNCU

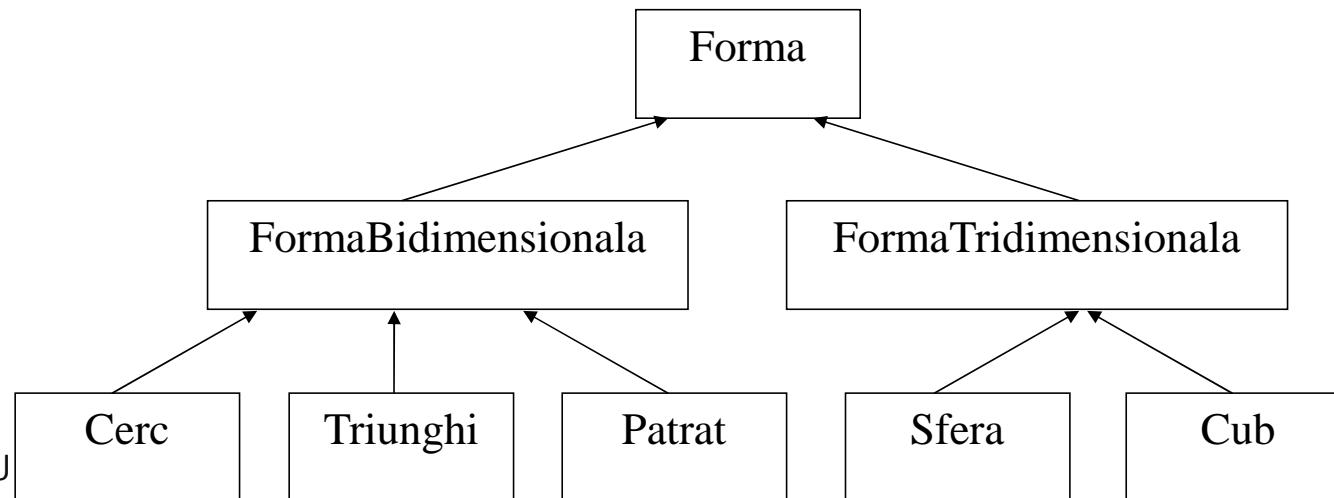
Moștenire, noțiuni generale (II)

- § O clasă moștenește datele membre și funcțiile membre dintr-o clasă de bază definită anterior – **clasă de bază**
- § Noua clasă, este o **clasă derivată**
- § Moștenirea se folosește la crearea unei noi clase asemănătoare uneia existente pentru a nu rescrie complet datele membre și funcțiile membre
- § Fiecare clasă derivată poate deveni, mai departe, un candidat pentru derivări ulterioare
- § **Moștenirea simplă** o clasă este derivată dintr-o singură clasă de bază
Ex.: persoană – **angajat** – **director**
- § **Moștenirea multiplă** o clasă este derivată din mai multe clase de bază
Ex.: imprimantă, scaner – **printer multifuncțional**

dr. Silviu GÎNCU

Exemple de moștenire simplă

- § Adeseori, un obiect al unei clase este un obiect al altei clase în același timp, spre exemplu:
 - Un dreptunghi *este un* patrulater, la fel ca și un pătrat, un paralelogram sau un trapez
 - Clasa **Dreptunghi** moștenește clasa **Patrulater**
- § Moștenirea conduce la relații între clase care pot fi reprezentate prin structuri arborescente



Sintaxă

§ Derivarea unei clase se realizează, după cum urmează:

```
class nume_baza{
    public:
        //membrii clasei de bază publici
    protected:
        //membrii clasei de bază protected
    private:
        //membrii clasei de bază publici
    };
    class nume_derivata: specifikator_acces nume_baza{
        // corpul clasei derivate
    };
}
```

§ Specifikatorul de acces poate fi unul din cuvintele-cheie: **public, private, protected**

Problema 1

Se consideră clasa **cerc**, definită prin rază (date) și următoarele metode: constructor cu parametru, determină lungimii discului și a suprafeței cercului, precum și supraîncărcat operatorul de ieșire <<. Să se creeze clasa derivată **con**, definită ca clasă derivată a clasei cerc prin datele și metodele corespunzătoare.

```
class cerc{
protected:
double raza;
public:
cerc(double a=0){raza=a;}    double l_disc(){return 2*M_PI*raza;}
double supr(){return M_PI*raza*raza;}
friend ostream& operator <<(ostream &, cerc &);
};
ostream& operator<<(ostream &ecran, cerc &f){
ecran<<"Raza=" <<f.raza<<" Lung. discului=" <<f.l_disc();
ecran<<" Suprafata discului=" <<f.supr()<<endl;
return ecran;
}
```

```
class con: public cerc{
    double gen, h;
public:
    con(double a=0, double b=0, double c=0){raza=a; gen=b; h=c; }
    double volum(){return supr()*h/3;}
    double supr_t(){return supr()+h*gen*M_PI;}
    friend ostream& operator <<(ostream &, con &);
};

ostream& operator<<(ostream &ecran, con &f){
    ecran<<"Raza=" <<f.raza<<" Inaltimea=" <<f.h<<" Generatoarea=" <<f.gen<<endl;
    ecran<<"Supraf. discului=" <<f.supr();
    ecran<<" Lung. discului=" <<f.l_disc()<<endl;
    ecran<<"Supraf. conului=" <<f.supr_t()<<" Volumul=" <<f.volum()<<endl;
    return ecran;
}
int main(){
    cerc b(3);        cout<<"Datele despre cerc\n" <<b;
    con d(4,5,6);    cout<<"Datele despre con\n" <<d;
}
```

Pentru membrii **public** din clasa de bază

§ Moștenirea membrilor declarați la nivelul **public**:

- **public** în clasa derivată
- Poate fi accesat direct prin orice funcție membră, funcție **friend**

```
class baza{  
    public:  
        // corpul clasei de baza  
};
```

§ Moștenirea membrilor declarați la nivelul **protected**:

- **protected** în clasa derivată
- Poate fi accesat direct prin orice funcție membră sau funcție **friend**

```
public  
class derivat: protected baza{  
    private  
        // corpul clasei derivate  
};
```

§ Moștenirea membrilor declarați la nivelul **private**:

- **private** în clasa derivată

Pentru membrii **protected** din clasa de bază

§ Moștenirea membrilor declarați la nivelul **public**:

- **protected** în clasa derivată
- Poate fi accesat direct prin orice funcție membră sau funcție **friend**

```
class baza{  
    protected:  
        // corpul clasei de baza  
};
```

§ Moștenirea membrilor declarați la nivelul **protected**:

- **protected** în clasa derivată
- Poate fi accesat direct prin orice funcție membră sau funcție **friend**

```
public  
class derivat: protected baza{  
    private  
        // corpul clasei derivate  
};
```

§ Moștenirea membrilor declarați la nivelul **private**:

- **private** în clasa derivată

Pentru membrii **private** din clasa de bază

§ Moștenirea membrilor declarați la nivelul **public**:

- **inaccesibil** în clasa derivată
- Poate fi accesat direct prin funcții **friend** sau prin funcții membre **public** sau **protected** din clasa de bază

§ Moștenirea membrilor declarați la nivelul **protected**:

- **inaccesibil** în clasa derivată
- Poate fi accesat direct prin funcții **friend** sau prin funcții membre **public** sau **protected** din clasa de bază

§ Moștenirea membrilor declarați la nivelul **private**:

- **inaccesibil** în clasa derivată
- Poate fi accesat direct prin funcții **friend** sau prin funcții membre **public** sau **protected** din clasa de bază

dr. Silviu GÎNCU

```
class baza{  
    private:  
        // corpul clasei de baza  
};  
  
public  
class derivat: protected baza{  
    private  
        // corpul clasei derivate  
};
```

Modificatori de acces - Concluzii

Clasa de bază	Modifierul de acces	Ce se poate accesa în clasa derivată	Ce se poate accesa în exterior
private	private	nu este accesibil	nu este accesibil
protected	private	private	nu este accesibil
public	private	private	nu este accesibil
private	protected	nu este accesibil	nu este accesibil
protected	protected	protected	nu este accesibil
public	protected	protected	nu este accesibil
private	public	nu este accesibil	nu este accesibil
protected	public	protected	nu este accesibil
public	public	public	public

Constructori și destructori în clasele derivate

- § În general, clasele utilizează constructori definiți de programator. În cazul în care aceștia lipsesc, compilatorul generează automat un constructor implicit pentru clasa respectivă
- § **Constructorii și destructorii** sunt funcții membre care **nu se moștenesc**
- § La instanțierea unui obiect din clasa derivată se apelează mai **întâi constructorii** claselor de bază, în ordinea în care aceștia apar în lista din declararea clasei derivate
- § La distrugerea obiectelor, se apelează **întâi destructorul** clasei derivate, apoi destructorii claselor de bază

Constructori și destructori în clasele derivate

```
class baza{
public:
baza (){cout<<"Constructor cls. baza\n";}
~baza(){cout<<"Destructor baza\n";}
};

class derivat: public baza{
public:
derivat(){cout<<"Constructor derivat\n";}
~derivat(){cout<<"Destructor derivat\n";}
};

int main(){
baza ob1;
derivat ob2;
}
```

După execuția secvenței de program, la consolă se vor afișa următoarele mesaje:

Constructor cls. Baza
Constructor cls. Baza
Constructor derivat
Destructor derivat
Destructor baza
Destructor baza

Constructori și destructori în clasele derivate

- § Transmiterea argumentelor unei funcții constructor din clasa de bază se face folosind o formă extinsă a declarației constructorului clasei derivate, care transmite argumentele unui sau mai multor constructori din clasa de bază.
- § Forma extinsă a declarației constructorului clasei derivate

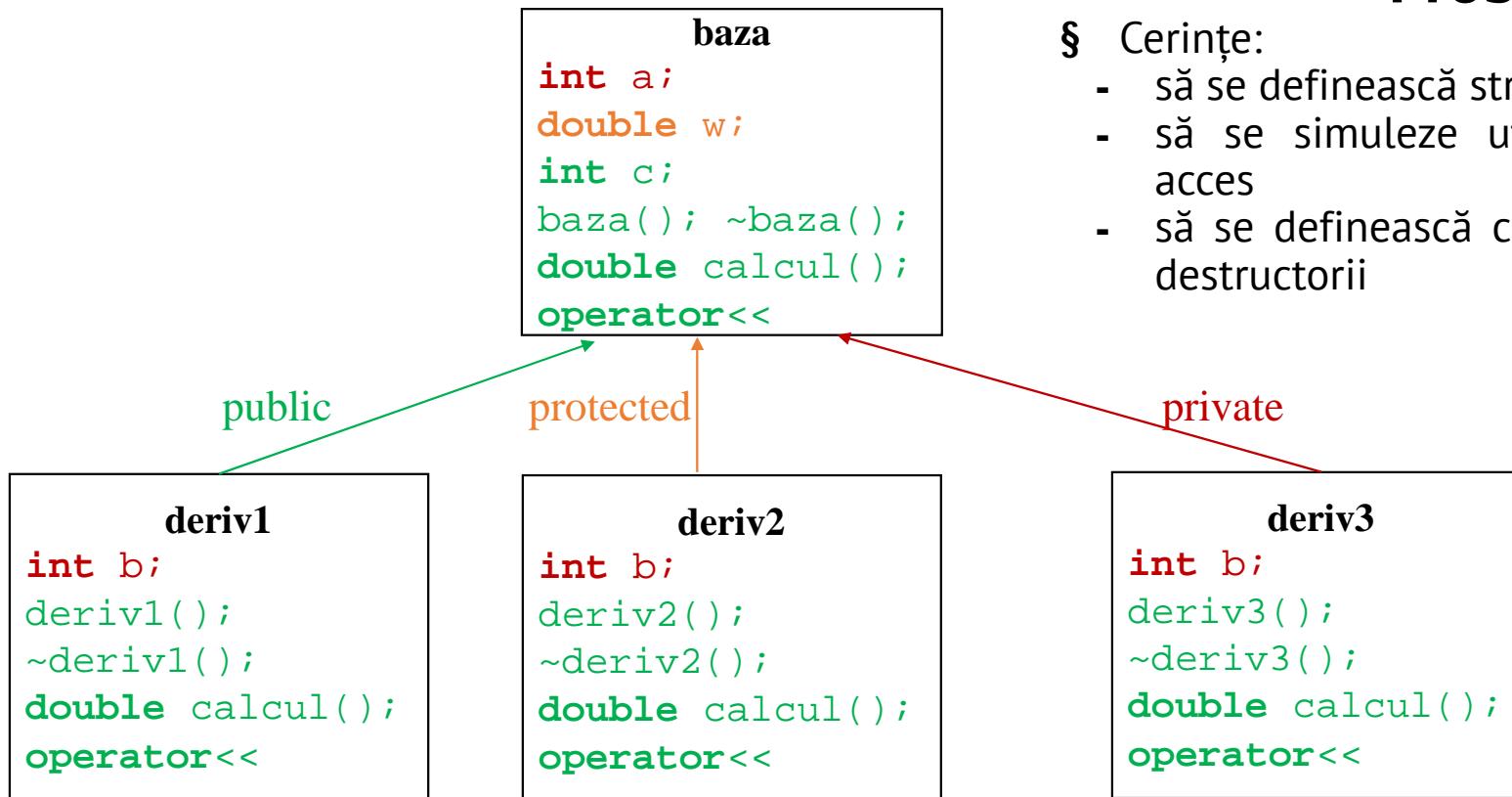
```
class baza{  
    baza(lista par_1){}  
};  
class derivat: public baza{  
    derivat(lista par_1, lista par_2): baza(lista par_1){}  
};
```

```
class CD : public CB1, public CB2, ...,  
public CBn{  
...  
public:  
    CD() : CB1(), CB2(), ..., CBn(){  
        //initializarea noilor date ale CD  
    }  
};
```

Problema 2

§ Cerințe:

- să se definească structura claselor
- să se simuleze utilitatea modificatorilor de acces
- să se definească constructorii cu parametri și destructorii



```
class baza{
    int a;
protected:
double w;
public:
int c;
baza (int a1, double w1, int c1){
    a=a1; w=w1; c=c1;
    cout<<"Constructor cls. baza\n";
}
~baza(){cout<<"Destructor baza\n";}
double calcul(){return a+w+c;}
friend ostream & operator<<(ostream &, const baza &);
};
ostream &operator<<(ostream &ies, const baza &b){
    ies<<b.a<<' '<<b.w<<' '<<b.c<<'\n';
    return ies;
}
```

Clasele deriv1 și deriv2

```
class deriv1: public baza{ int b;
public:
deriv1(int a1,double w1,int c1,int b1):baza(a1,w1,c1){
    b=b1; cout<<"Constructor deriv1\n";
}
~deriv1(){cout<<"Destructor deriv1\n";}
double calcul(){return w+c+b;} //a nu poate fi folosit
friend ostream &operator<<(ostream &, const deriv1 &);
};
class deriv2: protected baza{ int b;
public:
deriv2(int a1,double w1,int c1,int b1):baza(a1,w1,c1){
    b=b1; cout<<"Constructor deriv2\n";
}
~deriv2(){cout<<"Destructor deriv2\n";}
double calcul(){return w+c+b;}
friend ostream &operator<<(ostream &, const deriv2 &);
};
```

Clasa deriv3

```
class deriv3: private baza{
    int b;
public:
    deriv3(int a1,double w1,int c1,int b1):baza(a1,w1,c1){
        b=b1; cout<<"Constructor deriv3\n";
    }
    ~deriv3(){cout<<"Destructor deriv3\n";}
    double calcul(){return w+c+b;}
    friend ostream &operator<<(ostream &, const deriv3 &);
}
ostream &operator<<(ostream &ies, const deriv1& d1){
    ies<<d1.w<<' '<<d1.c<<' '<<d1.b<<'\n'; // a private
    return ies;
}
ostream &operator<<(ostream &ies, const deriv2& d2){
    ies<<d2.w<<' '<<d2.c<<' '<<d2.b<<'\n'; // a private
    return ies;
}
```

```
ostream &operator<<(ostream &ies, const deriv3& d3){  
    ies<<d3.w<<' ' <<d3.c<<' ' <<d3.b<<'\n'; // a private  
    return ies;  
}  
int main(){  
    baza x(1, 1.23, 2);  
    deriv1 y(2, 2.34, 3, 4);  
    deriv2 z(3, 3.45, 4, 5);  
    deriv3 v(4, 5.67, 6, 7);  
    cout<<"x=" <<x << '\n' <<"z=" <<z << '\n' <<"v=" <<v << '\n' ;  
    cout<<"x.calcul()=" <<x.calcul() << '\n' ;  
    cout<<"y.calcul()=" <<y.calcul() << '\n' ;  
    cout<<"z.calcul()=" <<z.calcul() << '\n' ;  
    cout<<"v.calcul()=" <<v.calcul() << '\n' ;  
    cout<<"x.c=" <<x.c << '\n' ;  
    cout<<"y.c=" <<y.c << '\n' ;  
}
```

Observații

- § La construirea unui obiect dintr-o clasă derivată din clasa bază, se apelează mai întâi constructorul din clasa de bază, apoi constructorul clasei derive. Astfel, un obiect y din clasa deriv2 incorporează un obiect deja inițializat cu ajutorul constructorului din clasa bază.
- § Dacă pentru clasa deriv1 dacă defineam un constructor de forma:

```
deriv1(int a1, double b1, int c1, int d1){  
    a=a1; b=b1; c=c1; d=d1;  
}
```

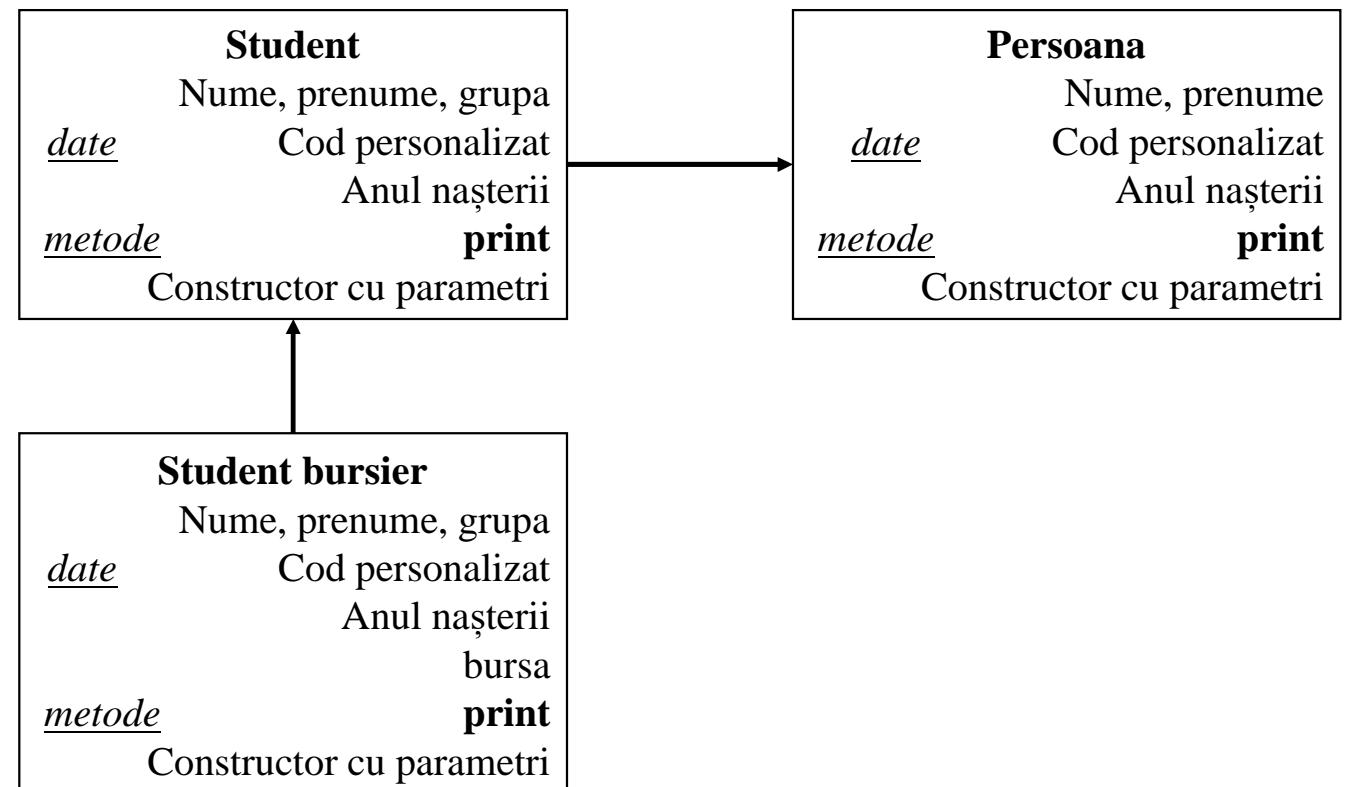
nu era corect, deoarece

1. clasa bază nu are constructori fără parametri, deci nu există constructor implicit
 2. data a este înaccesibilă în deriv1
- § Prin urmare apelarea constructorului se realizează apelând explicit constructorul din clasa bază.

dr. Silviu GÎNCU

Gestiunea funcțiilor membru

- § Se consideră următoarea ierarhie de clase
- § Să se elaboreze un program care să implementeze cele 3 clase, cu redefinirea metodei print pentru fiecare dintre clase



Clasa persoana

```
class persoana{
    char cod[5];
protected:
    char nume[15], prenume[15];
    int anul_n;
public:
    persoana(char *, char *, int);
    void print();
}
persoana::persoana(char *a,char *b, int c){
    strcpy(nume,a); strcpy(prenume,b); anul_n=c;
    for(int i=0;i<4;i++) cod[i]=(rand()%10)+48; cod[4]='\0';
}
void persoana::print(){
    cout<<"Codul: "<<cod<<endl<<"Nume: "<<nume<<endl;
    cout<<"Prenume: "<<prenume<<endl;
    cout<<"Anul nasterii: "<<anul_n<<endl;
}
```

```
class student : public persoana{
protected:
char grupa[10];
public:
student(char *, char *, int, char *);
void print();
};
student::student(char *a, char *b, int c, char *d):persoana(a,b,c){
strcpy(grupa,d);
}
void student::print(){
persoana::print();
cout<<"Grupa: "<<grupa<<endl;
}
```

Clasa student bursier

```
class st_bursier : public student{
protected:
double bursa;
public:
st_bursier(char *, char *, int, char *,double);
void print();
};
st_bursier::st_bursier(char *a, char *b, int c, char *d,double e):
student(a,b,c,d){
bursa=e;
}
void st_bursier::print(){
student::print();
cout<<"Bursa: "<<bursa<<endl;
}
```

Funcția principală

```
int main(){
persoana a( "Afanas" , "Ion" , 1990 );
cout<<"Datele despre persoana" << endl;
a.print();

student b( "Manole" , "Maria" , 1995 , "TI-175" );
cout<<"\nDatele despre student" << endl;
b.print();

st_bursier c( "Ciubotaru" , "Maxim" , 2000 , "SI-149" , 1000 );
cout<<"\nDatele despre studentul bursier" << endl;
c.print();
}
```

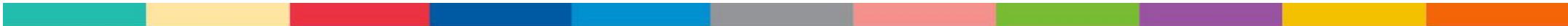
Teme pentru acasă

- § A învăță și repeta conceptele prezentate în cadrul lecției
- § A depana programele prezentate în cadrul lecției

Prelegerea următoare

Funcții virtuale și polimorfism

1. Derivarea simplă a claselor
2. Polimorfism. Tipuri de polimorfism
3. Polimorfismul dinamic
4. Funcții virtuale pure. Clase abstracte
5. Destructori virtuali
6. Implementarea polimorfismului în C++



Programarea Orientată pe Obiecte (POO)

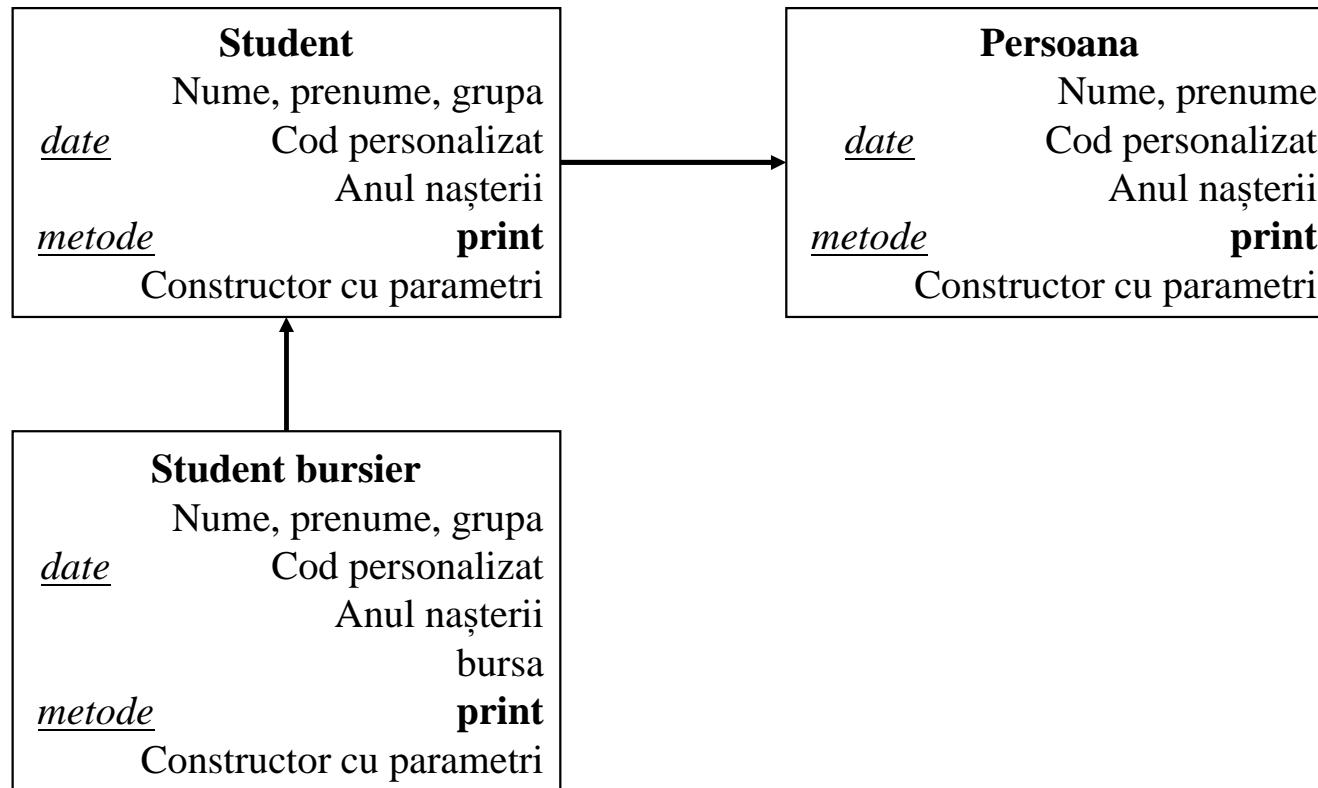
Prelegere

Funcții virtuale și polimorfism

SUMAR

1. Context
2. Polimorfism. Tipuri de polimorfism
3. Polimorfismul dinamic
4. Funcții virtuale pure. Clase abstracte
5. Destructori virtuali
6. Implementarea polimorfismului în C++

Se consideră ierarhia de clase



Și apelul de în funcția principală

```
int main(){
    persoana *a;
    a=new persoana("Afanas", "Ion", 1990);
    cout<<"Datele despre persoana"<<endl;
    a->print(); delete a;

    a=new student("Manole", "Maria", 1995, "TI-175");
    cout<<"\nDatele despre student"<<endl;
    a->print(); delete a;

    a=new st_bursier("Ciubotaru", "Ana", 2000, "SI-149", 1000);
    cout<<"\nDatele despre studentul bursier"<<endl;
    a->print(); delete a;
}
```

Funcții virtuale

- § O funcție **virtuală** este o funcție membru care în clasa de bază are specificat cuvântul cheie **virtual**, în clasele derivate cuvântul **virtual** nu se utilizează:

```
class nume_baza{
    virtual tip_returnat nume_metoda();
};

class derivat : public nume_baza{
    tip_returnat nume_metoda();
};
```

- § În cazul funcțiilor **virtuale**, urmare a creării obiectelor claselor derivate în baza unui pointer al clasei de bază, apelul funcțiilor derivate se va efectua în corespondere cu constructorul în baza căruia acesta a fost creat, adică se va apela funcția din clasa derivată

dr. Silviu GÎNCU

Pentru ierarhia de clase persoana-student-st_bursier

```
class persoana{  
    // datele și metodele descrise anterior  
    virtual void print(); //este virtuală pentru toate cele 3 clase  
};  
class student : public persoana{  
    // datele și metodele descrise anterior  
    void print();  
};  
class st_bursier : public student{  
    // datele și metodele descrise anterior  
void print();  
};
```

Apelul în funcția principală

```
int main(){
persoana *a; student *b;
a=new persoana("Afanas", "Ion", 1990);
a->print(); delete a;
a=new student("Manole", "Maria", 1995, "TI-175");
a->print(); delete a;
a=new st_bursier("Ciubotaru", "Ana", 2000, "SI-149", 1000);
a->print(); delete a;
b=new student("Manole", "Maria", 1995, "TI-175");
b->print(); delete b;
b=new st_bursier("Ciubotaru", "Ana", 2000, "SI-149", 1000);
b->print(); delete b;
}
```

Concluzii

- § Cu ajutorul ***funcțiilor virtuale*** este posibilă proiectarea și implementarea sistemelor software care sunt mult mai ușor extensibile
- § Programele pot fi concepute să proceseze în mod generic, sub forma obiectelor din clasele de bază, a obiectelor tuturor claselor dintr-o ierarhie
- § Clasele care nu există în timpul dezvoltării inițiale a programului pot fi adăugate ulterior cu modificări minore sau chiar fără a face modificări părții generice a programului, atâtă timp cât clasele sunt părți ale ierarhiei procesate generic
- § Singurele părți din program care trebuie modificate sunt cele care folosesc informații specifice despre o clasă adăugată în ierarhie

dr. Silviu GÎNCU

Ce este polimorfismul?

- § Polimorfismul este capacitatea unor entități de a lua forme diferite. Termenul provine de la:
polimorm *Poly = multe + Morfm = forma*
- § Este unul din concepțile esențiale din POO

Tipuri de polimorfism

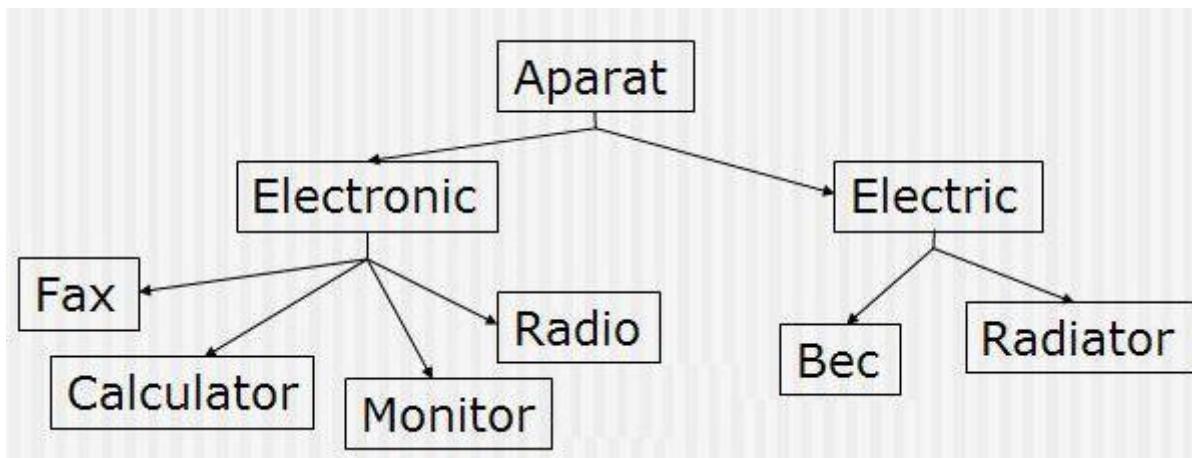
- § **Polimorfismul dinamic** – mecanismul prin care o metodă din clasa de bază este redefinită cu aceeași parametri în clasele derivate. În limbajul C++ este implementat cu ajutorul funcțiilor virtuale.
- § **Polimorfismul parametric** – mecanismul prin care putem defini mai multe metode cu același nume în aceeași clasă, dar să difere prin numărul și/sau tipul parametrilor formali

De exemplu: complex a, b, c; a=b+c; a=b+2.45;

Operația + se realizează între două obiecte de tip complex: b, c și dintre un obiect de tip complex și o constată de tip număr real

dr. Silviu GÎNCU

Polimorfismul dinamic



```
Aparat *aparate[6] ;  
aparate[0] = new Fax(1440, 230);  
aparate[1] = new Calculator(1200, 256, 350);  
aparate[2] = new Monitor(17, 130);  
aparate[3] = new Radio("Panasonic", 80);  
aparate[4] = new Bec(200);  
aparate[5] = new Radiator(2000);  
for (i=0; i<6; i++) aparate[i]->porneste();
```

Funcții virtuale pure și clase abstracte

- § O **funcție virtuală pură** este o funcție care nu are definiție în clasa de bază, iar declarația ei are următoarea formă:

```
class nume_baza{  
    //...  
    virtual tip_returnat nume_metoda() {};//sau return 0;  
    ...  
};
```

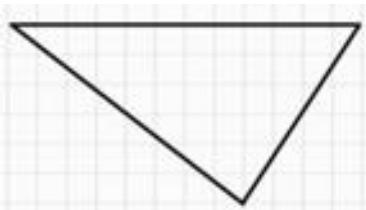
- § Dacă o clasă conține cel puțin o funcție **virtuală pură**, atunci ea se numește **clăsă abstractă**.

Problema 1

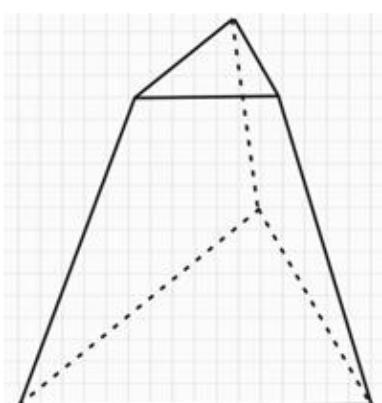
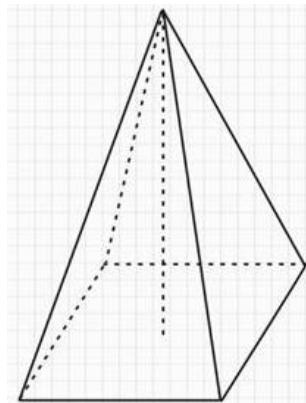
Să elaborăm un program în care vom defini clasa de bază **triunghi** (corespunzătoare triunghiului echilateral) și clasele derivate **piramida** și **prisma** (corespunzătoare piramidei triunghiulare regulate și prismei triunghiulare regulate). Programul va citi datele despre **n** astfel de figuri și corpuri geometrice, apoi:

- va afișa denumirile celor cu aria maximă, respectiv cu volumul maximal.
- va afișa fiecare tip de figură;
- va ordona crescător figurile, conform suprafeței lor.

Numărul **n** de figuri/corpuri geometrice, de asemenea, se va citi de la tastatură.



dr. Silviu GÎNCU



```
class triunghi{
protected:
double lat;
public:
virtual int tip(){return 1;}          // 1- triunghi
virtual void citire();
virtual double arie();
virtual void afis();
virtual double volum(){return 0;} //virtuala pura
};
void triunghi::citire(){ lat=rand()%10+1;}
double triunghi::arie(){return lat*lat*sqrt(3)/4;}
void triunghi::afis(){
cout<<"triunghi lat=" << lat << " supr=" << arie();
cout<<endl;
}
```

Clasa piramida

```
class piramida: public triunghi{
protected:
double a,h;
public:
int tip(){return 2;}           // 2 - piramida
void citire();                double arie();
double volum();               void afis();
};

void piramida::afis(){
cout<<"Piramida lat=" <<lat <<" h=" <<h <<" a=" <<a <<" supr=" <<arie();
cout<<" Volumul " <<volum() <<endl;
}
void piramida ::citire() { triunghi::citire();
a=rand()%10+1; h=rand()%10+1;}
double piramida ::arie(){ double s=triunghi::arie();
s=s+triunghi::lat*a/2*3; return s;}
double piramida::volum(){return triunghi::arie()*h/3;}
```

```
class prisma: public triunghi{
protected:
double H;
public:
int tip(){return 3;}           // 3- prisma
void citire();                 double arie();
double volum();                void afis();
};
void prisma::citire(){triunghi::citire(); H=rand()%10+1;}
double prisma::arie(){ double s=2*triunghi::arie();
s=s+triunghi::lat*H; return s; }
double prisma::volum(){double s=triunghi::arie()*H; return s; }
void prisma::afis(){
cout<<"Prisma lat=<<lat<<" H=<<H;
cout<<" supr=<<arie()<<" Volumul ";
cout<<volum()<<endl;
}
```

Citire și afișare (1)

```
triunghi *fig[100];  int n;
void citire(){  int q;  cout<<"Introdu num. Fig. ";cin>>n;
for(int i=0;i<n;i++){
    q=rand()%3;  switch(q){
        case 1:  fig[i]=new prisma; break;
        case 2:  fig[i]=new piramida;break;
        default: fig[i]=new triunghi;
    }fig[i]->citire();}
void afisare(){
    int arie, volum;  double amax=0,vmax=0;
    for(int i=0;i<n;i++){ fig[i]->afis();
        if(amax<fig[i]->arie()){amax=fig[i]->arie(); arie=i;}
        if(vmax<fig[i]->volum()){vmax=fig[i]->volum();volum=i;}
    }
    cout<<"Figura cu aria maxima"<<endl;    fig[arie]->afis();
    cout<<"Figura cu vol. maxim"<<endl;    fig[volum]->afis();
}
```

Citire și afișare (2)

```
void af_tip(int t){  
    for(int i=0;i<n;i++) if(fig[i]->tip()==t) fig[i]->afis();}  
void lista_fig(){  
    cout<<"Lista triunghiurilor"<<endl; af_tip(1);  
    cout<<"Lista piramidelor"<<endl; af_tip(2);  
    cout<<"Lista prismelor"<<endl; af_tip(3);}  
void sortare(){ triunghi *aux; int i,j; aux=fig[0];  
    for(i=0;i<n-1;i++)  
        for(j=i+1;j<n;j++)  
            if(fig[i]->arie()>fig[j]->arie()){aux=fig[i];fig[i]=fig[j];fig[j]=aux;}  
    cout<<"Sortare dupa suprafata"<<endl;  
    for(int i=0;i<n;i++) fig[i]->afis();}  
int main(){  
    citire(); afisare(); lista_fig(); sortare();  
    for(int i=0;i<n;i++) delete fig[i];  
}
```

Destructorii virtuali

- § Atunci când se folosește **polimorfismul** pentru procesarea obiectelor alocate dinamic dintr-o ierarhie de clase apar probleme legate de distrugerea acestora.
- § Dacă un obiect cu un destructor **non-virtual** este distrus explicit prin aplicarea operatorului **delete** unui pointer la clasa de bază care pointează către obiect, se apelează destructorul clasei de bază, indiferent de tipul actual al obiectului. O soluție la această problemă este declararea **virtual** a destructorului clasei de bază.
- § Această declarare face ca toți destructorii claselor derivate să devină **virtuali**.
- § Spre deosebire de destructori, constructorii nu pot fi declarați virtual.
- § Dacă acum un obiect din ierarhie este distrus explicit prin aplicarea operatorului **delete** unui pointer către clasa de bază care a fost inițializat cu adresa unui obiect dintr-o clasă derivată, se apelează, în mod corect, destructorul clasei derivate corespunzătoare pointerului cu care a fost inițializat pointerul la clasa de bază.

dr. Silviu GÎNCU

Problema 2

Prin acest exemplu se prezintă modalitatea de declarare a destructorului virtual

```
class baza{
public:
    virtual ~baza() {cout<<"Destructor baza"<<endl; }
};

class derivat: public baza{
public:
    ~derivat() {cout<<"Destructor derivat\n"; }
};

int main(){
    baza *p; p=new derivat;
    delete p;
}
```

Urmare a depărării programului, la consolă se vor afișa mesajele:
Destructor derivat
Destructor baza

Problema 3

- § O implementare a metodei reluării (tehnici de programare): algoritmul este format dintr-o secvență unică de instrucțiuni cu același nume, dar care se modifică diferit în dependentă de problemă.

Schema generală a algoritmului recursiv bazat pe metoda reluării

```
void reluare(int k){  
    if(k<=n) {  
        x[k]=PrimulElement(k);  
        if(Continuare(k)) reluare(k+1);  
        while(ExistaSuccesor(k)) {  
            x[k]=Succesor(k);  
            if(Continuare(k)) reluare(k+1);  
        } }  
    else PrelucrareaSolutiei();  
}
```

dr. Silviu GÎNCU

Implementare

- § Se va crea o clasă, în care funcțiile ***init()*** ***am_succesor()*** ***e_valid()*** ***solutie()*** ***tipar()*** să fie definite inițial cu valori simple, iar apoi în clasele derivate să poată fi redefinite astfel încât să rezolve fiecare problemă în parte.
- § De aceea ele vor fi definite din start – în clasa de bază – ca fiind virtuale, pentru ca apoi în clasele derivate să poată fi redefinite corespunzător fiecărei clase în parte care rezolvă un anumit tip de problemă.
- § Astfel, se va utiliza:
 - n - numărul de elemente;
 - as - are succesor (se mai poate adăuga un element pe nivelul curent)
 - ev - este valid (se îndeplinesc cerințele problemei).
 - k - elementul curent;

Implementarea clasei de bază

```
class bkt{
protected:
int st[10],n,k;
public:
virtual void init() {}
virtual int am_succesor(){return 0;}
virtual int e_valid(){return 0;}
virtual int solutie(){return 0;}
virtual void tipar() {};
void run();
};
void bkt::run(){ int as; k=1; init();
while(k>0){
do{ }while((as=am_succesor())&&!e_valid());
if(as) if(solutie()) tipar();
else{ k++; init(); } else k--;
}
}
```

Implementarea clasei permutări

```
class permut : public bkt{
public:
    permut (int v) { n=v; }
    void init();           int am_succesor();
    int e_valid();         int solutie();
    void tipar();
};

void permut::init(){ st[k]=0; }
int permut::am_succesor(){
    if(st[k]<n){ st[k]++; return 1; } else return 0; }
int permut::e_valid(){
    for(int i=1; i<k; i++) if( st[i] == st[k] ) return 0;
    return 1; }
int permut::solutie(){return (k==n); }
void permut::tipar(){ for(int i=1;i<=k;i++)
    cout<<st[i]<<" ";   cout<<endl; }
```

Implementarea clasei regine

```
class dame : public permut{
public:
dame(int v) : permut(v) { };
int e_valid();
}
int dame::e_valid(){
for(int i=1; i<k; i++)
if((st[i]==st[k]) || abs(st[k]-st[i])==abs(k-i)) return 0;
return 1;
}
int main(){
cout<<"Permutari"<<endl;
permut x(3); x.run();
cout<<"Plasarea reginelor"<<endl;
dame y(4); y.run();
}
```

Interpretarea rezultatelor

Permutări

1 2 3

1 3 2

2 1 3

2 3 1

3 1 2

3 2 1

Plasarea reginelor

2 4 1 3

3 1 4 2

	1	2	3	4
1	■		●	
2		■		●
3	●		■	
4		●		■

Problema 4

- § Se propune simularea unui joc prin care utilizatorului i se va propune să determine suma numerelor. Jocul se va organiza pe nivele, astfel încât corespunzător nivelului 1 utilizatorului i se va propune să determine suma a două numere, nivelului 2, respectiv suma a 3 numere, ... , nivelului n, suma a n+1 numere.
- § În cazul în care utilizatorul a oferit un răspuns greșit i se va propune să repete nivelul, iar dacă au fost depășite toate nivelele atunci se va afișa un mesaj corespunzător.

Organizarea problemei în clase / obiecte

Obiectul joc	
Atribute	i, niv
Metode	joc() show() suma() nivel()
Obiectul joc_1	
Atribute	joc::i, niv, i
Metode	joc::show() joc::suma() nivel() show() suma()
Obiectul joc_2	
Atribute	joc::i, joc_1::i, niv, i
Metode	joc::show() joc::suma() joc_1::show() joc_1::suma() nivel() show() suma()

Clasa de bază joc	
Atribute	Primul număr, nivelul
Metode	Constructor, afișează numărul, suma, returnează nivelul
Clasa derivată joc_1	
Atribute	al doilea număr
Metode	Constructor, afișează numărul, suma
Clasa derivată joc_2	
Atribute	al treilea număr
Metode	Constructor, afișează numărul, suma

Implementarea claselor

```
class joc{      int i;
protected:      int niv;
public:         joc(){i=rand()%10; niv=0;}
virtual void show(){cout<<setw(3)<<i;}
virtual int suma(){return i;}    int nivel(){return niv;}
};
class joc_1: public joc{  int i;
public:         joc_1(){i=rand()%10; niv=1;}
void show(){joc::show(); cout<<setw(3)<<i;}
int suma(){return joc::suma()+i;}
};
class joc_2: public joc_1{ int i;
public:         joc_2(){i=rand()%10; niv=2;}
void show(){joc_1::show(); cout<<setw(3)<<i;}
int suma(){return joc_1::suma()+i;}
};
class joc_3: public joc_2{ int i;
public:         joc_3(){i=rand()%10; niv=3;}
void show(){joc_2::show(); cout<<setw(3)<<i;}
int suma(){return joc_2::suma()+i;}
};
```

```
int main(){ joc *a; int suma,n=1,p=1;
cout<<"Jocul afla suma numerelor"<< endl;
do{
    switch(n){
        case 1: delete a; a=new joc_1; break;
        case 2: delete a; a=new joc_2; break;
        case 3: delete a; a=new joc_3; break;
        default: cout<<"Joc final"; getch(); delete a; exit(1);
    }
    cout<<"Determina suma numerelor"<<endl;
    a->show(); cout<<endl; cin>>suma;
    if(suma == a->suma()){
        cout<<"Corect ai trecut nivelul " <<a->nivel()<<endl;
        n++;
    }
    else cout<<"raspuns gresit\n mai incercati"<<endl;
    cout<<"Continuati ?\n 1 - Da 0 - Nu"<<endl; cin>>p;
}while(p);
delete a;
}
```

```
Jocul afla suma numerelor
Determina suma numerelor
  1  7
8
Corect ai trecut nivelul  1
Continuati ?
  1 - Da 0 - Nu
1
Determina suma numerelor
  4  0  9
13
Corect ai trecut nivelul  2
Continuati ?
  1 - Da 0 - Nu
1
Determina suma numerelor
  4  8  8  2
22
Corect ai trecut nivelul  3
Continuati ?
  1 - Da 0 - Nu
1
Joc final
```

Rezultatul execuției

Concluzii

§ Polimorfism

- Posibilitatea ca obiecte din diverse clase care sunt legate prin relații de moștenire să răspundă diferit la același mesaj

§ Polimorfismul este implementat prin funcțiile **virtuale**

- Atunci când programul cere folosirea unei funcții printr-un pointer sau o referință la o clasa de bază, C++ alege suprascrierea corectă din clasa derivată corespunzătoare

§ Când este util polimorfismul și funcțiile virtuale?

- Când într-o fază intermediară a proiectării și implementării unei aplicații nu sunt cunoscute toate clasele care vor fi folosite
- Atunci când noile clase care sunt adăugate sistemului sunt integrate prin legare dinamică
- În situațiile în care tipul unui obiect care apelează o funcție **virtuală** nu este nevoie să fie cunoscut la compilare
- Când la rulare, funcția apelată **virtual** este identificată cu funcția membră din clasa căreia îi aparține obiectul

dr. Silviu GÎNCU

Teme pentru acasă

- § A învăță și repeta conceptele prezentate în cadrul lecției
- § A depana programele prezentate în cadrul lecției

Prelegerea următoare

1. Moștenire multiplă. Generalități
2. Exemplu de moștenire multiplă
3. Constructori și destructori în relația de moștenire multiplă
4. Polimorfismul vs moștenirea multiplă
5. Exemplu de implementare a polimorfismului în relația de moștenire multiplă



Programarea Orientată pe Obiecte (POO)

Prelegere



SUMAR

1. Moștenire multiplă. Generalități
2. Exemplu de moștenire multiplă
3. Constructori și destructori în relația de moștenire multiplă
4. Polimorfismul vs moștenirea multiplă
5. Exemplu de implementare a polimorfismului în relația de moștenire multiplă

Problemă 1 – dispozitive periferice

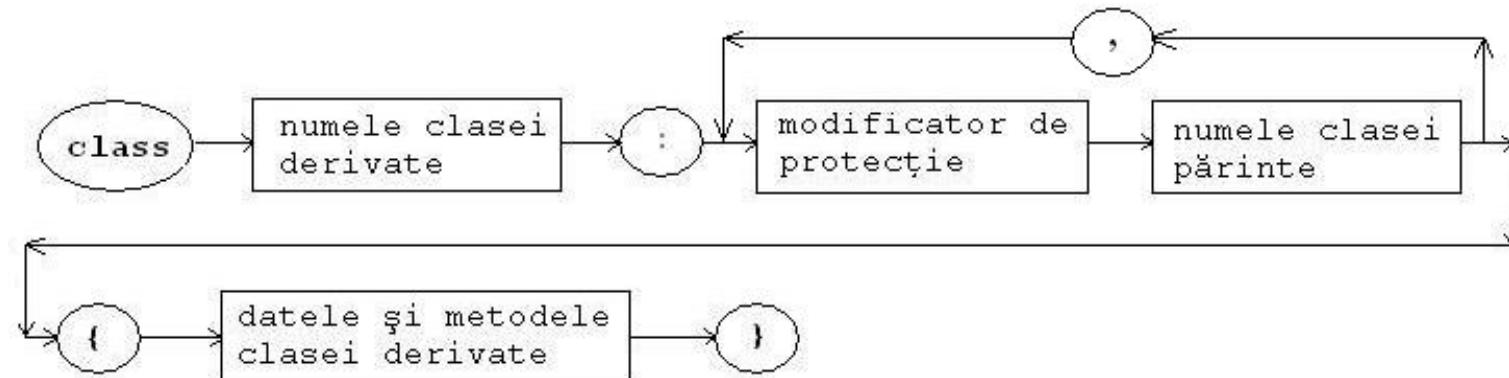
- § Într-un magazin de calculatoare sunt propuse spre vânzare următoarele categorii de dispozitive periferice: imprimantă, scaner și printer multifuncțional.
- § Se solicită a prezenta un model de organizare a informației în vederea gestionării ei despre dispozitivele periferice.



dr. Silviu GÎNCU

Generalități

- § Conceptul de moștenire multiplă permite crearea de clase noi care moștenesc datele și metodele mai multor clase de bază. Moștenirea multiplă aduce mai multă flexibilitate în construirea claselor, rezultatul fiind obținerea unor structuri de clase complexe. Astfel, sintaxa generală pentru declararea unei clase este:



Reprezentarea grafică a moștenirii multiple

```
class Baza_1{  
    int a1;  
protected: int b1;  
public: int c1;  
Baza_1(){a1=b1=c1=0;}  
void print_1();  
};
```

```
class Baza_2{  
    int a2;  
protected: int b2;  
public: int c2;  
Baza_2(){a2=b2=c2=0;}  
void print_2();  
};
```

```
class Baza_3{  
    int a3;  
protected: int b3;  
public: int c3;  
Baza_3(){a3=b3=c3=0;}  
void print_3();  
};
```

public

protected

private

```
class derivat : public baza_1, protected baza_2, private baza_3{  
public:  
void print();  
};
```

Constructori și destructori în clasele derivate

§ În relațiile de moștenire multiplă Constructorii și destructorii se comportă după aceleași reguli și proceduri ca și în cazul relație de moștenire simplă.

Forma extinsă a declarației constructorului clasei derivate

```
class Baza1{           Baza1(tip1 arg1,tip2 arg2,...,tipk argk);  
};  
class Baza2 {         Baza2(tipk+1 argk+1, tipk+2 argk+2,...,tipk+p argk+p);  
};  
class Derivata : public Baza1,public Baza2 {  
    Derivata(tip1 arg1,...,tipk argk,...,tipn argn);  
};  
Derivata::Derivata(tip1 arg1,..,tipk argk,..,tipn argn):  
    Baza1(arg1,..,argk),  
    Baza2(argk+1,..,argk+p){  
    //definirea constructorului clasei derivate  
}
```

dr. Silviu GÎNCU

Rezolvarea problemei - dispozitive periferice

```
class Imprimanta{  
protected:  
int rezolutie;  
public:  
Imprimanta(int rez=600){rezolutie=rez;cout<<"Apel Constr. Imprimanta\n";}  
~Imprimanta(){cout<<"Apel Destr. Imprimanta\n";}  
void print(char *text){cout<<"Print: "<<text<<"\n";}  
};  
class Scaner{  
protected:  
int rezolutie;  
public:  
Scaner(int rez=1200){rezolutie = rez;  
cout<<"Apel Constr. Scaner\n";}  
~Scaner(){cout<<"Apel Destr. Scaner\n";}  
void scan(){cout<<"Scanez\n";}  
};
```

Rezolvarea problemei - dispozitive periferice

```
class MultiFunct: public Imprimanta, public Scaner{
public:
    MultiFunct(int rezI,int rezS):Imprimanta(rezI),Scaner(rezS){
        cout<<"Apel Constr. MultiFunctională\n";
    }
    ~MultiFunctională(){cout<<"Apel Destr. MultiFunctională\n"; }
};

int main(){ MultiFunct m(300,600); m.print("hello"); m.scan(); }
```

Apel Constr. Imprimanta

Apel Constr. Scaner

Apel Constr. MultiFunctională

Print: hello

Scanez

Apel Destr. MultiFunctională

Apel Destr. Scaner

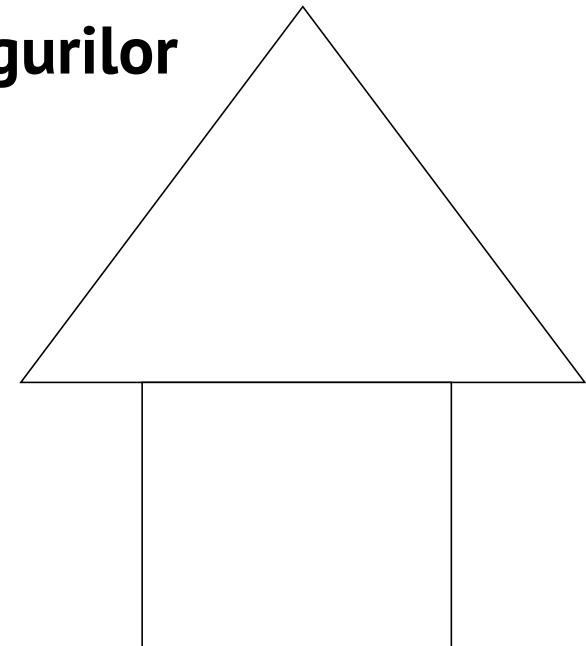
Apel Destr. Imprimanta

Problemă 2 – gestiunea figurilor

- § În calitate de clase de bază se consideră clasele **pătrat** și **triunghi echilateral**.

Aceste clase derivă clasa figura, formată dintr-un pătrat și un triunghi echilateral.

- § Creați clasa figura, derivată claselor pătrat și triunghi echilateral, dacă se cunoaște că lungimea laturii triunghiului echilateral este egală cu $a+0.3*a$, unde a este lungimea laturii pătratului. Pentru clasa figura vor fi implementate metodele de determinare a suprafeței și a perimetrului.



```
class patrat{
protected:
double lp;
public:
patrat(){lp=0;}
patrat(double a){lp=a;}
void cit_patrat();    void afis_patrat();
double per_patrat(){return 4*lp;}
double arie_patrat(){return lp*lp;}
};
void patrat::cit_patrat(){cout<<"Dati latura patratului"<<endl; cin>>lp;}
void patrat::afis_patrat(){
cout<<"Patrat: lat=" <<lp;
cout<<" Aria=" <<arie_patrat();
cout<<" Perimetru=" <<per_patrat()<<endl;
}
```

```
class tr_echil{
protected:
double lt;
public:
tr_echil(){lt=0;}      tr_echil(double a){lt=a;}
void cit_triunghi();  void afis_triunghi();
double per_triunghi(){return 3*lt;}
double arie_triunghi(){return sqrt(3)*0.25*lt*lt;}
};
void tr_echil::cit_triunghi(){
cout<<"Dati latura tr. echilateral"<<endl;  cin>>lt;}
void tr_echil::afis_triunghi(){
cout<<"Triunghi echilateral: lat=" <<lt;
cout<<" Aria=" <<arie_triunghi();
cout<<" Perimetru=" <<per_triunghi()<<endl;
}
```

Clasa figura

```
class figura : public patrat, public tr_echil{
public:
figura(){};
figura(double a):patrat(a),tr_echil(a+a*0.3){};
void cit_figura();
void afis_figura();
double arie_figura(){return arie_triunghi()+arie_patrat();}
double per_figura(){return per_triunghi()+per_patrat()/2;}
};
void figura::cit_figura(){
cit_patrat();
lt=lp+lp*0.3;
}
```

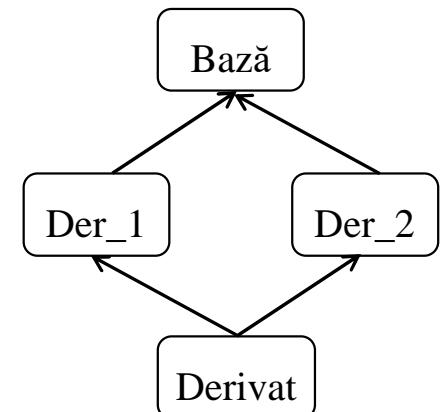
Funcția principală

```
void figura::afis_figura(){
    cout<<"Figura este formata din:"<<endl;
    afis_patrat();    afis_triunghi();
    cout<<"Aria figurii="<<arie_figura();
    cout<<" Perimetru figurii=";
    cout<<per_figura()<<endl;
}
main(){
    figura a,b(10);
    a.afis_figura();
    b.afis_figura();
    a.cit_figura();
    a.afis_figura();
}
```

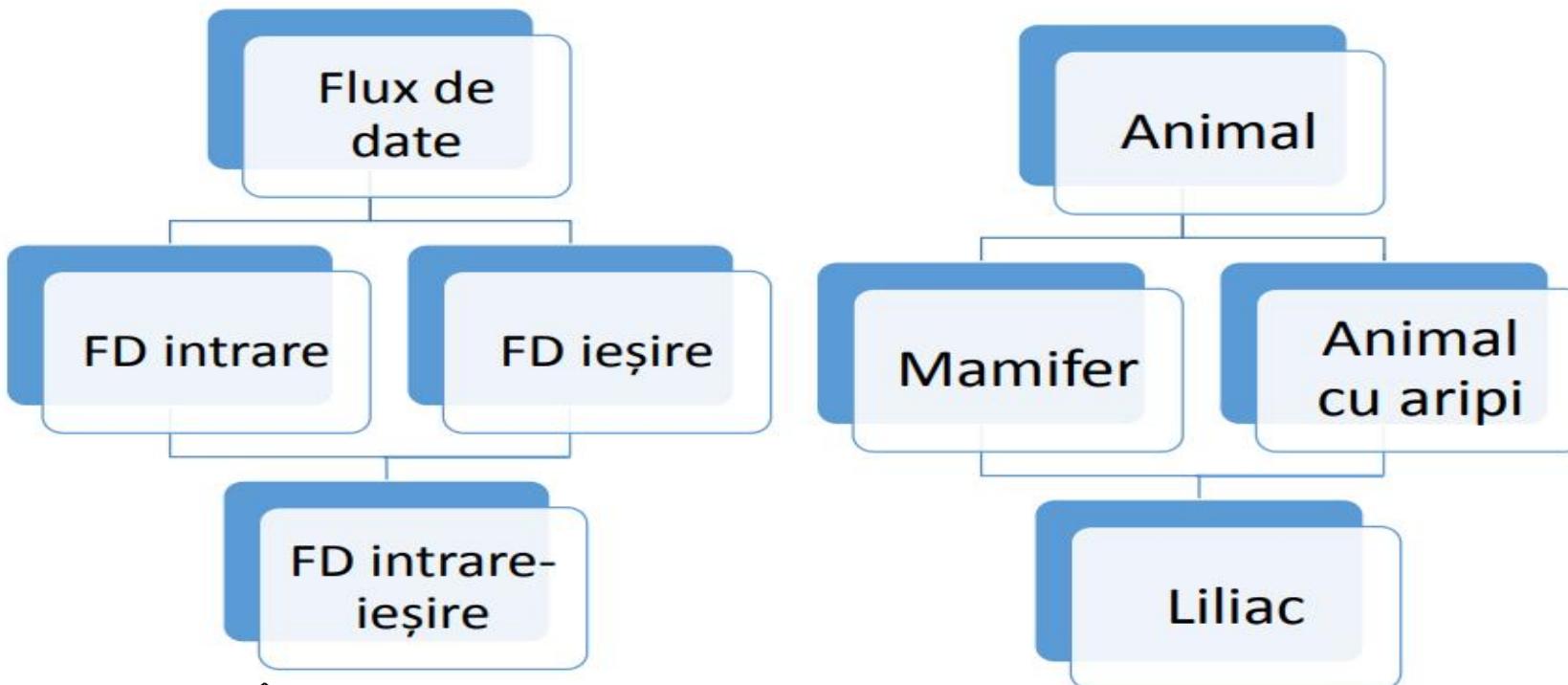
Polimorfismul dinamic vs moștenire multiplă

- § Într-o relație de moștenire multiplă este posibil ca o clasă să fie moștenită indirect de mai multe ori, prin intermediul unor clase care moștenesc, fiecare în parte, clasa de bază. De exemplu:
- Un obiect din clasa **Derivat** va conține membrii clasei Bază de două ori, o dată prin clasa Der_1 și o dată prin clasa Der_2. În această situație, accesul la un membru al unui obiect de tip Derivat moștenit din clasa Bază este interzis (este semnalat ca eroare la compilare – numită **ambiguitate**).
- § O soluție pentru eliminarea ambiguităților în moștenirile multiple este de a impune crearea unei singure copii a clasei de bază în clasa derivată. Pentru aceasta este necesar ca acea clasă care ar putea produce copii multiple prin moștenire indirectă să fie declarată clasă de bază de tip **virtual**.
- § O **clasă de bază virtuală** este moștenită o singură dată și creează o singură copie în clasa derivată.

dr. Silviu GÎNCU



Moștenire multiplă: ambiguități



dr. Silviu GÎNCU

Clase virtuale

§ Sintaxă:

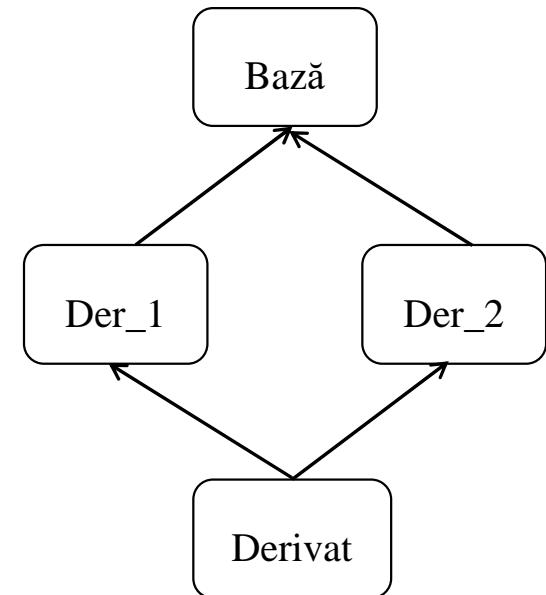
```
class Bază{ //date și metode
};

class Der_1: virtual public Bază{           //date și metode
};

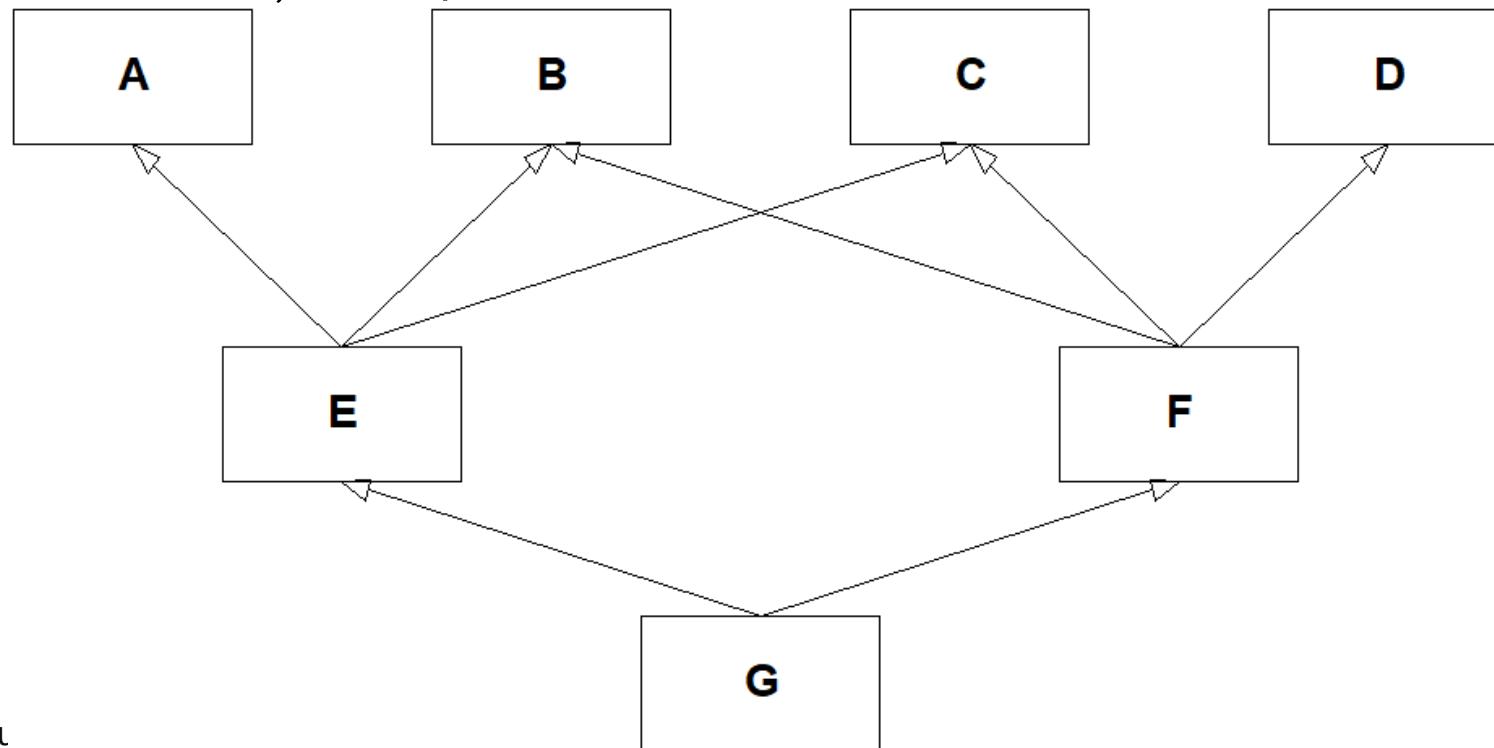
class Der_2: virtual public Bază{           //date și metode
};

class Derivat: public Der_1, public Der_2{ //date și metode
};
```

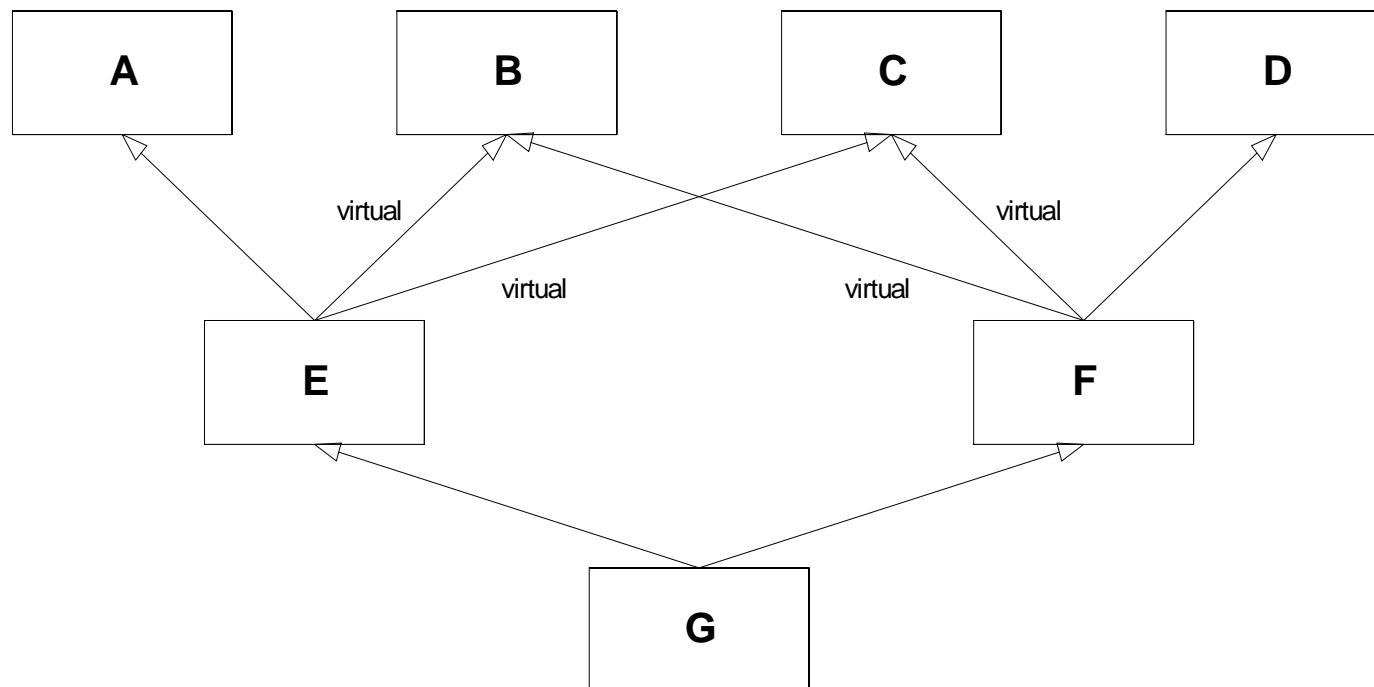
§ În acest caz clasa **Bază** este moștenită o singură dată și creează o singură copie în clasa derivată, deoarece ea a fost moștenită în clasele Der_1 și Der_2 prin utilizarea cuvântului **virtual**.



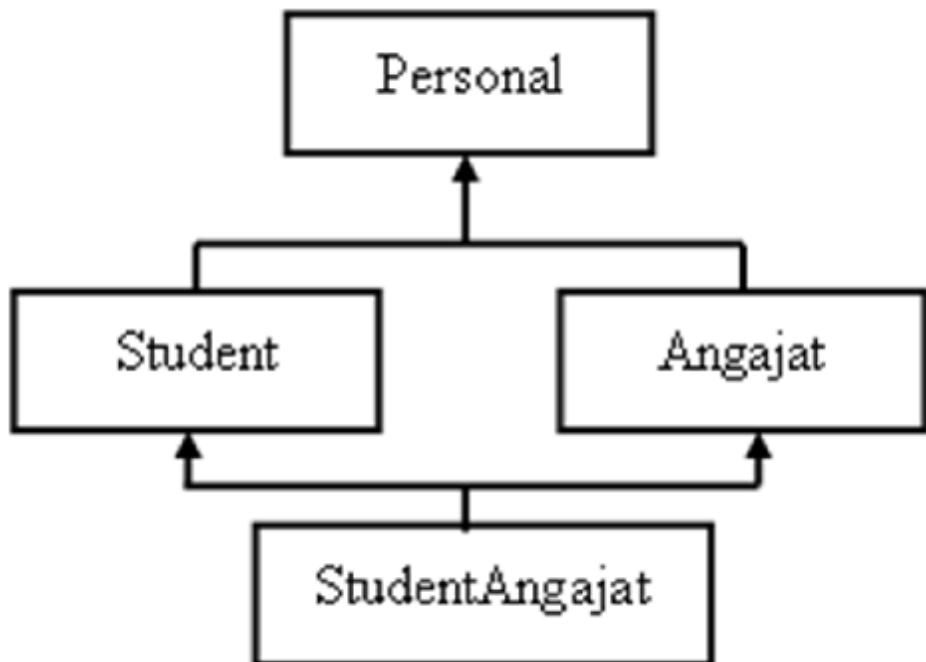
Pentru relația de mai jos, precizați numele claselor care necesită a fi moștenite prin intermediul cuvântului **virtual**



Moștenirea multiplă prin intermediul cuvântului **virtual**



Problemă 3 – gestiunea personalului



- § Elaborați un program, prin intermediul căruia se vor prelucra datele despre tipuri de date precum: **Angajat**, **Student** și **Student_angajat**.
- Despre un angajat se cunoaște: *numele, prenumele, anul nașterii, nr_ore_lucrare, plata_ora*;
 - Despre un student se cunoaște *numele, prenumele, anul nașterii, grupa, media*.

```
class Persoana{
    char *idnp;           static int an;
protected:          char *nume,*prenume;     int anul;
public:
    Persoana( char [], char [] , int ); virtual void afisare();
    virtual int salariu(){return 0;}      virtual int bursa(){return 0;}
    int virsta(){return an-anul;}        virtual ~Persoana();
};

class Student : virtual public Persoana{
protected:
    char *grupa; float media;
public:
    Student( char [], char[] , int , char [] , int );
    void afisare();    int bursa();
    ~Student();
};
```

```
class Angajat : virtual public Persoana{
protected:
int ore, pl_ora;
public:
Angajat( char [ ], char [ ], int, int, int );
void afisare();
int salariu();
~Angajat(){cout<<"Destructor Angajat"<<endl;}
};
class Stud_Ang : public Student, public Angajat{
public:
Stud_Ang(char[],char[],int, char[], float, int, int);
void afisare();
~Stud_Ang(){cout<<"Destructor Stud_Ang"<<endl;}
};
int Persoana::an=2021;
```

```
void Persoana::afisare(){
    cout<<setw(15)<<idnp<<setw(10)<<nume<<setw(10)<<prenume<<setw(4)<<virsta();
}
void Angajat::afisare(){
    Persoana::afisare(); cout<<setw(8)<<setprecision(2)<<salariu();
}
void Student::afisare(){
    Persoana::afisare();
    cout<<setw(10)<<grupa<<setw(12)<<setprecision(2)<<bursa();
}
void Stud_Ang::afisare(){
    Student::afisare(); cout<<setw(12)<<setprecision(2)<<salariu();
}
int Student::bursa(){ if(media<7.5) return 0;
    else if(media<8.5) return 300;
    else if(media<9.5) return 400;
    else return 500;
}
```

Descrierea constructorilor

```
Persoana::Persoana( char a[], char b[], int c){  
    nume=new char[strlen(a)+1]; prenume=new char[strlen(b)+1];  
    idnp=new char[14]; anul=c; strcpy(nume,a); strcpy(prenume,b);  
    for(int i=0;i<13;i++) idnp[i]=(rand()%10)+48; idnp[13]='\0';  
}  
Angajat::Angajat(char nume[],char prenume[],int anul,int ore int pl_ora):  
    Persoana(nume, prenume, anul){this->ore=ore; this->pl_ora=pl_ora;  
}  
Student::Student(char nume[],char prenume[],int anul,char grupa[],float  
    media): Persoana(nume, prenume, anul){  
    this->grupa=new char[strlen(grupa)+1];  
    strcpy(this->grupa,grupa); this->media=media;  
}  
Stud_Ang::Stud_Ang(char nume[],char prenume[],int anul,char grupa[],  
float media,int ore,int pl_ora):Student(nume, prenume, anul, grupa, media),  
    Angajat(nume, prenume, anul, ore, pl_ora), Persoana(nume, prenume, anul){};
```

Funcția principală

```
Persoana::~Persoana() {delete idnp; delete nume; delete prenume;  
cout<<"Destructor Persoana"<<endl;  
}  
Student::~Student() {delete grupa; cout<<"Destructor Student"<<endl;  
}  
int Angajat::salariu() { return ore*pl_ora; }  
int main()  
{  
    Persoana *t[3];  
    t[0]=new Student("Mocanu","Victor",1990,"TA-132",8.5);  
    t[1]=new Angajat("Moraru","Ana",1991, 40,200);  
    t[2]=new Stud_Ang("Ciubota","Ion",1992,"SQ-131",9.01,20,50);  
    for(int i=0;i<3;i++) {t[i]->afisare(); cout<<endl;}  
    for(int i=0;i<3;i++) delete t[i];  
}
```

Studiu de caz

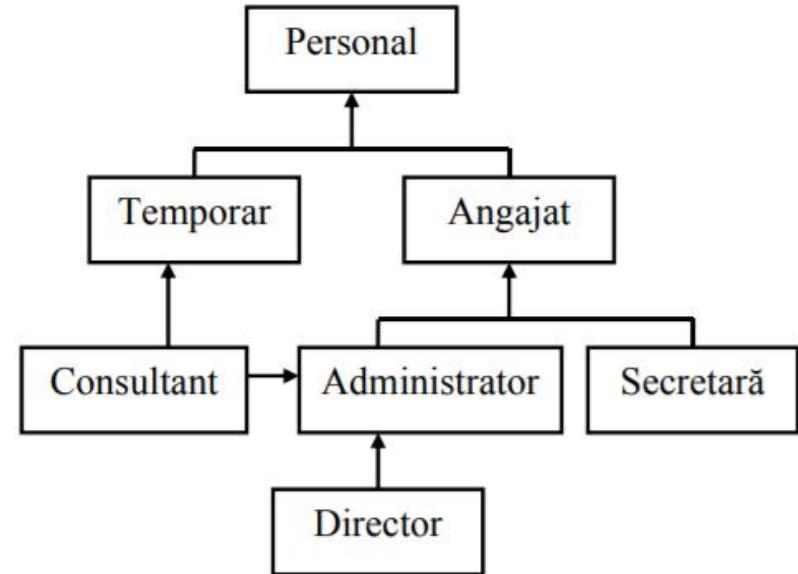
- § Se consideră personalul unei instituții, formată din următoarele tipuri de salariați: secretar, administrator, director, cumular (angajat temporar), consultant. Elaborați un program prin intermediul căruia se va efectua evidența personalului din instituție.

Soluție

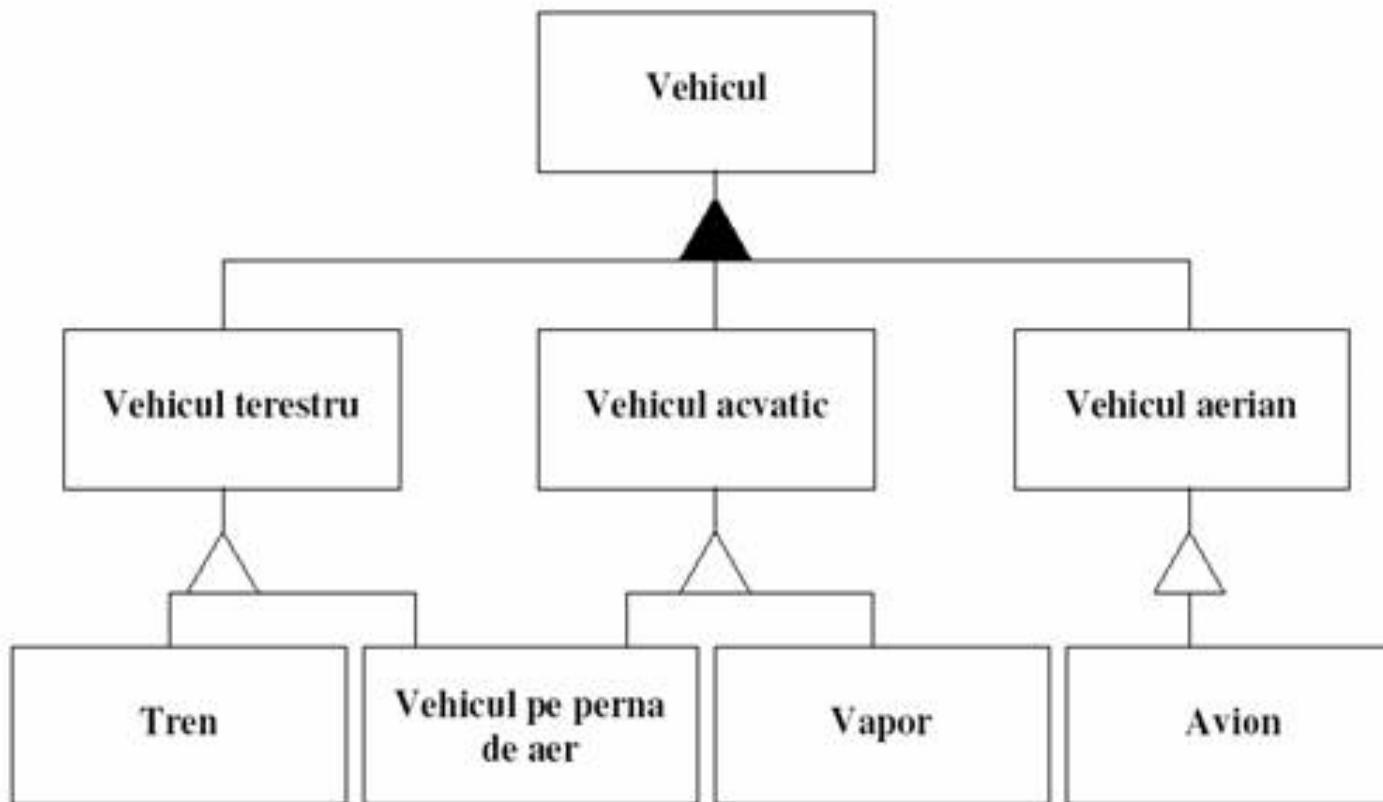
- § Pentru efectuare evidenței personalului instituției este necesar de defini cinci tipuri de date (conform condiției problemei). Observăm însă că aceste tipuri conțin în structura sa, elemente comune:

date : nume, prenume, anul etc;

metode : citire, afișare, salariu etc.



Propuneti o implementare a claselor în baza figurii



Teme pentru acasă

- § A învăță și repeta conceptele prezentate în cadrul lecției
- § A depana programele prezentate în cadrul lecției

Prelegerea următoare

Ierarhii de clase

1. Relații între clase
2. Agregarea
3. Exemplu de agregare
4. Gestiunea structurilor dinamice de date prin intermediul claselor
5. Agregare vs moștenire
6. Studiu de caz



Programarea Orientată pe Obiecte (POO)

Prelegere

lerarhii de clase

SUMAR

1. Relații între clase
2. Asociere (colaborare)
3. Agregarea
4. Exemplu de agregare
5. Gestiunea structurilor dinamice de date prin intermediul claselor
6. Agregare vs moștenire
7. Studiu de caz

Relații între clase

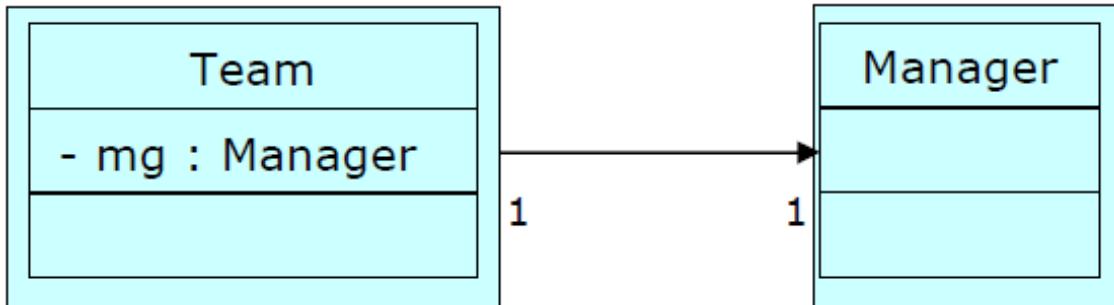
- § Concepțele POO nu există izolate. Ele coexistă și interacționează. La elaborarea modelului obiectual al unei aplicații se disting următoarele etape:
 1. Identificarea claselor → corespund conceptelor aplicației
 2. Stabilirea relațiilor dintre clase → corespunde specificațiilor aplicației
- § Astfel, în cadrul cursului analizăm următoarele tipuri de relații:
 - **Moștenire** – relație în care obiectele unei clase sunt create prin extinderea/dezvoltarea unei clase existente
 - **Asociere** - relație în care obiectele unei clase comunică/cunosc cu/despre de obiectele altor clase
 - **Componere/Agregare** – relație în care obiectele unei clase se regăsesc în cadrul altor obiecte

dr. Silviu GÎNCU

Relația de asociere

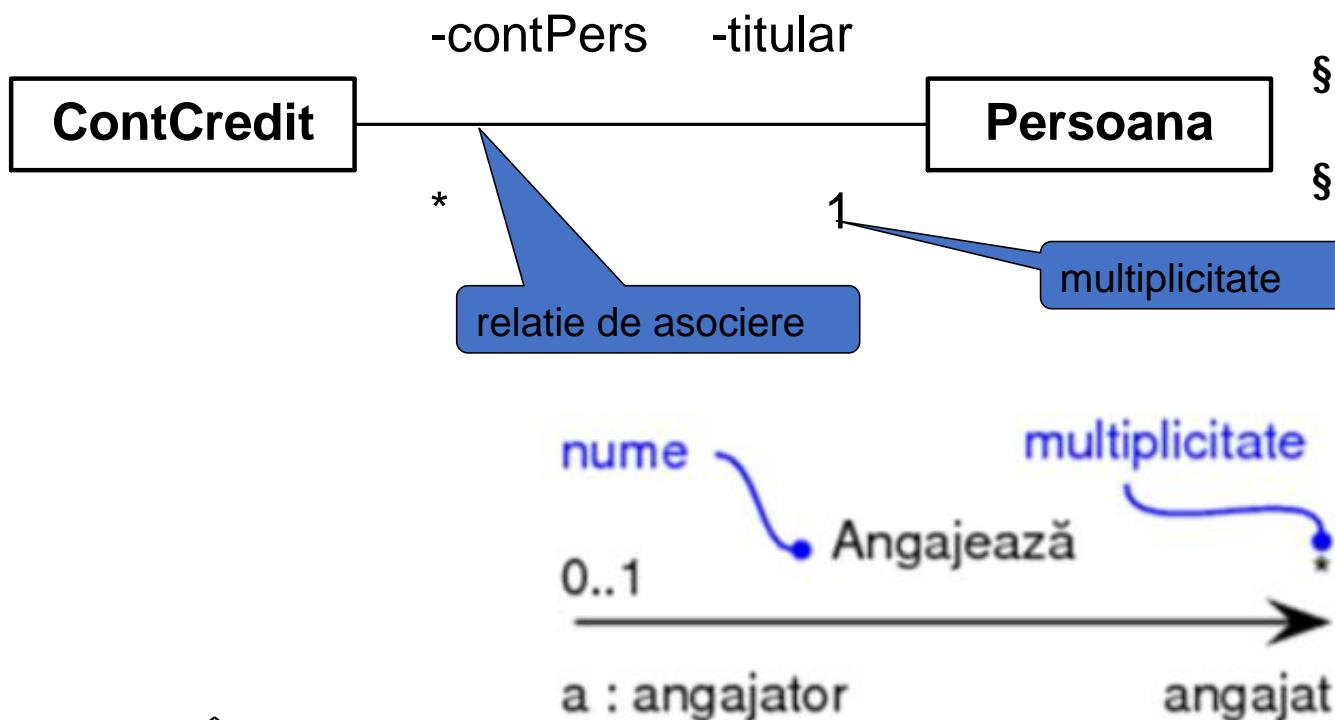
§ **Asocierea** = presupune o relație dintre două sau mai multe obiecte. De regulă, aceasta se implementează ca o instanță a unei clase (în altă clasă).

Exemplu: O echipă are un **manager**



§ Multiplicitate – în acest caz este în relație de 1: 1,
aceasta poate fi diferită, de exemplu *una la mai multe*(*), sau chiar fixe 2..4

Relația de asociere. Exemple



§ clasa ContCredit are o dată membru titular;
clasa Persoana are o data membra contPers

Problema 1

Următorul program implementează o relație de asociere între clasele **Manager** - **Team**

```
class Manager{
private:
char* name;
public:
void setName(char * );
char* getName();
};

class Team{
private:
Manager m;
public:
void setManager(Manager & );
Manager& getManager();
};

void Manager::setName(char* n){
name = new char[strlen(n) + 1]; strcpy(name,n); }
char* Manager::getName(){return name; }
void Team::setManager(Manager &m){this->m = m; }
Manager& Team::getManager(){return m; }
int main(){Team t; Manager m; m.setName("Ion");
t.setManager(m); cout<<"Manager:"<<t.getManager().getName();
}
```

Agregare

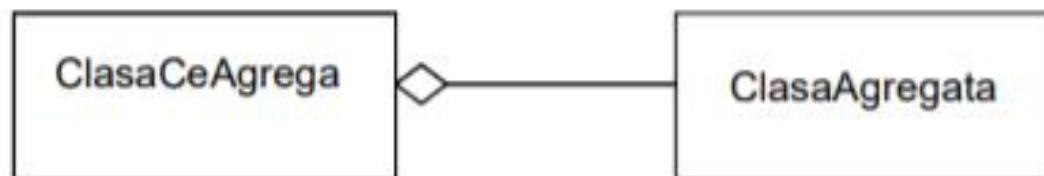
- § **Agregare** = definirea unei noi clase ce include una sau mai multe date membre care au ca tip clase deja definite. Permite construirea de clase complexe pornind de la clase mai simple.
- § Relațiile de **agregare** pot fi:
 - **fixe** - constau dintr-un număr fix de componente;
clasa data are 3 obiecte componente: zi, luna, an.
 - **variabile** - constau dintr-un număr variabil de elemente;
clasa carte are mai multe obiecte componente pagină.
 - **recursive** – acest tip de compunere presupune existența de componente de același tip cu clasa agregată;
clasa **arbore_binar** are ca și componente tot obiecte **arbore_binar**

Tipuri de agregare

- § Există **două modele** de agregare diferențiate de relația dintre **agregat** și **subiecte**:
- § **Compoziția** – subiectele aggregate aparțin exclusiv aggregatului din care fac parte, iar durata de viață coincide cu cea a aggregatului;



- § **Agregarea** propriu-zisă - subiectele au o existență independentă de aggregatele ce le partajază. Mai mult, un obiect poate fi partajat de mai multe aggregate.

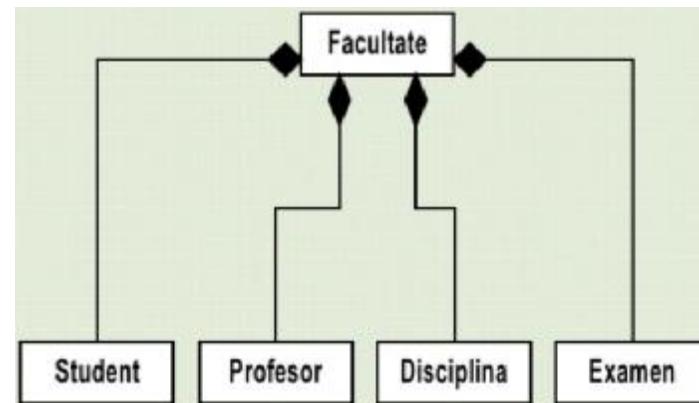


Compoziție/compunerea

- § În relația de compoziție, întregul (A) controlează durata de viață a părții (B), iar întregul (A) nu poate exista fără părți (B).
- A conține (1/mai multe) B-uri
 - B este creat de către A

Exemple

- § Floarea (A) este compusă din Petale (B)
§ Cartea (A) conține mai multe Pagini (B)

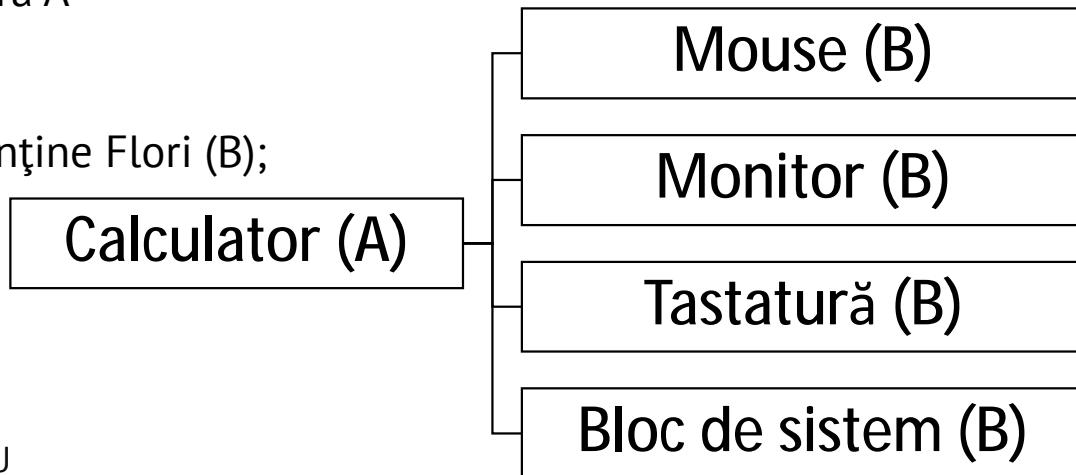


Agregare

- § O relație de tip întreg (A) – parte (B) între 2 tipuri de obiecte. Partea (B) și întregul (A) au diferite durate de viață.
- A conține (1/mai multe) B-uri;
 - B există fără A

Exemple

- § Grădina (A) conține Flori (B);



Asociere - Agregare – Compoziție

1. Asociere

§ Obiectele știu unul despre altul astfel încât pot lucra împreună

2. Agregare

§ Protejează integritatea obiectului
§ Funcționează ca un singur tot întreg
§ Controlul se realizează printr-un obiect

3. Compoziție

§ Fiecare parte poate fi membru al unui singur obiect agregat

1. O echipă are un manager

2. Grădina conține Flori

3. Floarea este compusă din Petale

Agregare vs moștenire

Când se foloseşte moștenirea şi când agregarea ?

- § **Agregarea** este folosită atunci când se dorește utilizarea trăsăturilor unei clase în interiorul altrei clase;
- § **Moștenirea** este folosită atunci când se dorește construirea unui tip de date care să reprezinte o implementare specifică (o specializare oferită prin clasa derivată) a unui lucru mai general.

Diferența dintre moștenire și agregare:

- § **moștenire** - *is a* (este un) - indică faptul că o clasă este derivată dintr-o clasă de bază (intuitiv, dacă avem o clasă Animal și o clasă Cîine, atunci ar fi normal să avem Cîine derivat din Animal, cu alte cuvinte Cîine este un Animal)
- § **agregare** - *has a* (are un) - indică faptul că o clasă are o clasă conținută în ea (intuitiv, dacă avem o clasă Mașină și o clasă Motor, atunci ar fi normal să avem Motorul referit în cadrul Mașinii, cu alte cuvinte Mașina are un Motor)

dr. Silviu GÎNCU

Constructorii în relația de agregare

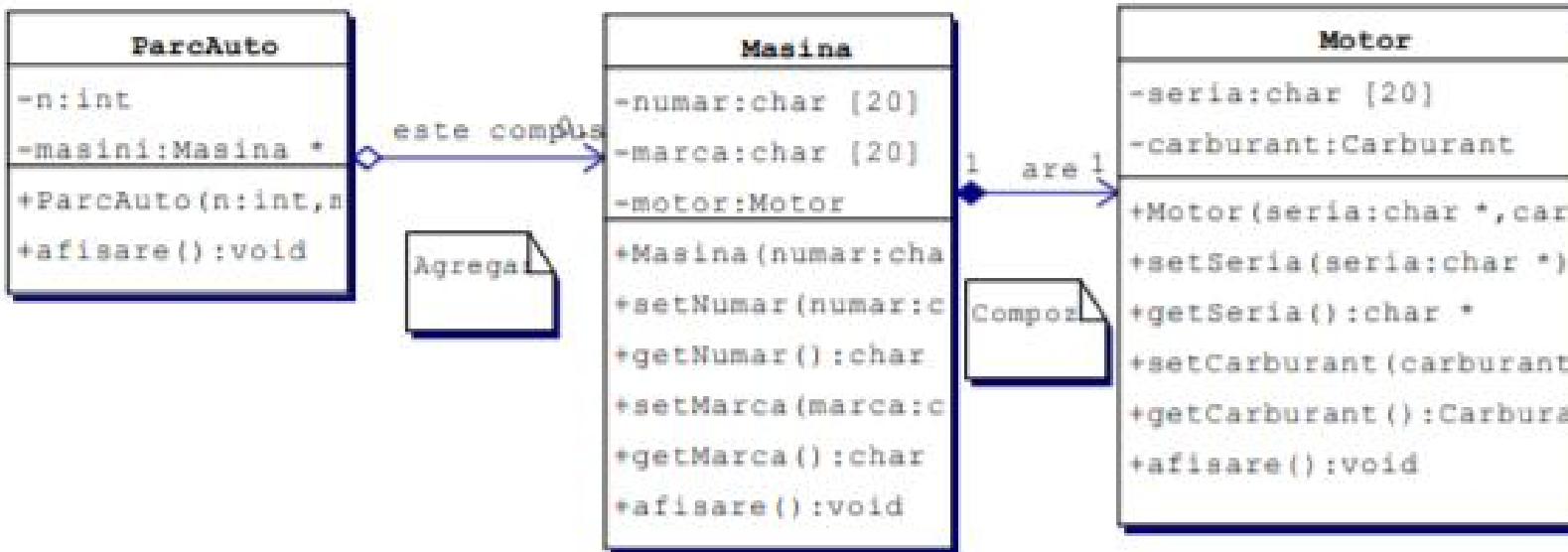
- § Dacă o clasă A conține ca date membru obiecte ale altor clase (C1, ..., Cn) atunci constructorul clasei A va avea în calitate de parametri datele membru ale claselor C1,..., Cn.
 - § La crearea unui obiect al clasei A se vor apela mai întâi constructori claselor C1,..., Cn pentru a se inițializa obiectele incluse în obiectul clasei A și apoi se vor executa instrucțiunile constructorului clasei A.
 - § Apelul constructorilor claselor C1,..., Cn se poate face:
 - **explicit** la definirea constructorului clasei A

```
class A{  
    C1 c1;    Cn cn;  
    A(lista de parametri);  
};  
A::A(lista de parametri):C1(sub_lista_param_1),..., Cn(sub_lista_param_n){  
    instructiuni;}
```
 - **automat** de către compilator – în acest caz se apelează constructorul implicit al acelor clase
- dr. Silviu GINCU

Problema 2

Un **parc auto** este format dintr-o mulțime de **mașini**. Pentru fiecare **mașină** din parc se cunosc următoarele detalii: numărul de înmatriculare, marca, seria **motorului** și tipul de caburant utilizat de acesta.

Reprezentarea problemei:



```
enum Carburant { benzina=0, motorina};  
class Motor{  
    char seria[20];           Carburant carburant;  
public:  Motor(char *seria, Carburant carburant);  
void setSeria(char *seria);   char* getSeria();  
void setCarburant(Carburant carburant);  
Carburant getCarburant();    void afisare();  
~Motor(){cout<<"Destructor Motor\n";}  
};  
void Motor::setCarburant(Carburant carburant){this->carburant=carburant;}  
Motor::Motor(char *seria, Carburant carburant){setSeria(seria);  
setCarburant(carburant);}  
void Motor::setSeria(char *seria){strcpy(this->seria,seria);}  
char* Motor::getSeria(){return seria;}  
Carburant Motor::getCarburant(){return carburant;}  
void Motor::afisare(){  
cout<<"Serie Motor:"<<seria<<endl<<"Tip Combustibil:";  
cout<<(carburant==motorina)?"motorina":"benzina")<<endl;}
```

```
class Masina{
    char numar[20],marca[20]; Motor motor;
public:
    Masina(char *numar="",char *marca="",char *seria="",Carburant carburant=benzina);
    void setNumar(char *numar);
    char* getNumar(); char* getMarca();
    void setMarca(char *marca); void afisare();
    ~Masina(){cout<<"Destructoar Masina\n";}
}
Masina::Masina(char *numar, char *marca, char *seria, Carburant carburant):
motor(seria,carburant){setNumar(numar); setMarca(marca);}
void Masina::setNumar(char *numar){ strcpy(this->numar, numar);}
char* Masina::getNumar(){return numar;}
void Masina::setMarca(char *marca){strcpy(this->marca, marca);}
char* Masina::getMarca(){return marca;}
void Masina::afisare(){cout<<"Numar:"<<numar<<endl;
    cout<<"Marca:"<<marca<<endl; motor.afisare(); }
```

```
class ParcAuto{
    int n;    Masina *masini;
public:
    ParcAuto(int n, Masina masini[]);
    void afisare();
    ~ParcAuto(){delete [] masini; cout<<"Destructor ParcAuto\n";}
};

ParcAuto::ParcAuto(int n, Masina masini[]){
    this->n = n;  this-> masini = new Masina[n];
    for(int i=0;i<n;i++) this->masini[i] = masini[i];
}

void ParcAuto::afisare(){for(int i=0;i<n;i++) masini[i].afisare();}
int main(){
    Masina m[]={Masina( "DJ-01-UV" , "Audi" , "1234" , motorina),
                Masina( "DJ-02-CV" , "Logan" , "2121" , benzina)};
    ParcAuto parc(2,m); parc.afisare();
}
```

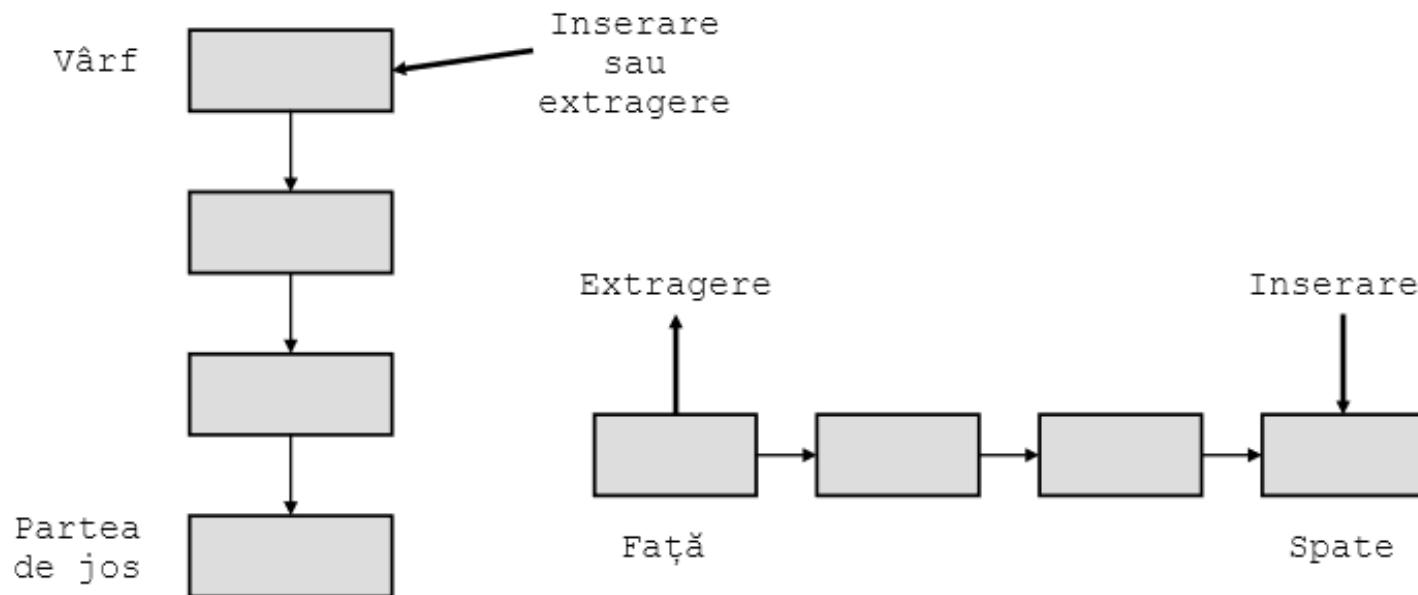
Gestiunea structurilor dinamice de date prin intermediul claselor

- § Structurile dinamice de date sunt date structurate ale căror componente se aloca pe măsura ce se creează. Pentru a crea o structură dinamică de date se impune folosirea unui câmp care să rețină adresa de memorie la care se află următorul element din structura, astfel se realizează o înlățuire după adrese.
- § În funcție de tipul înlățuirii realizate între componente, există următoarele tipuri de organizări:
 - structuri liniare: liste simplu înlățuite și liste dublu înlățuite, cu cazuri particulare: lista circulară, stiva, coada.
 - structuri arborescente ierarhice
 - structuri rețea
- § Caracteristic unei liste liniare sunt următoarele operații: creare, parcursere, prelucrarea informației utile (afișare, calcule, sortare, căutare, etc), inserarea unui nod în listă, ștergerea unui nod din listă.

dr. Silviu GÎNCU

Stiva și coada

- § **Stiva** este o listă în care toate operațiile (de inserare și de extragere) au loc printr-un singur capăt al listei.
- § **Coada** este o listă în care operațiile de inserare și extragere au loc prin capete diferite ale listei.



Problema 3

Următorul program demonstrează modul de implementare la nivel de obiecte a structurilor dinamice stiva și coada

```
class coada;
class stiva;
class TStudent{
    char *nume,*grupa;    int     anul,id;    double media;   TStudent *next;
public:
TStudent();      ~TStudent();
friend istream & operator>>(istream &, TStudent *&);
friend ostream & operator<<(ostream &, TStudent*);
friend class coada;   friend class stiva;
};
TStudent::TStudent(){ nume=new char[20];  grupa=new char[10];
anul=0;id=0; media=0; next=NULL;
}
```

Unele funcții ale clasei TStudent

```
TStudent::~TStudent(){
    delete nume; delete grupa;
    nume=grupa=NULL; next=NULL;
}
istream & operator>>(istream &is, TStudent *&t){
    cout<<"Dati datele despre student:";
    cout<<" Id Nume, Grupa, media, anul"<<endl;
    is>>t->id>>t->nume>>t->grupa>>t->media>>t->anul;
    return is;
}
ostream & operator<<(ostream &os, TStudent *t){
    os<<setw(4)<<t->id<<setw(20)<<t->nume<<setw(10);
    os<<t->grupa<<setw(6)<<t->anul<<setw(5)<<t->media;
    os<<endl;
    return os;
}
```

Structura claselor stiva și coada

```
class stiva{
    TStudent *current;
public:
    stiva(){current=NULL; }
    void creare();
    void parcurge();
    void inserare();
    void exclude();
    ~stiva();
};
```

```
class coada{
    TStudent *fata,*spate;
public:
    coada(){fata=spate=NULL; }
    void creare();
    void parcurge();
    void inserare();
    void exclude();
    ~coada();
};
```

```
stiva::~stiva(){while(current!=NULL) exclude(); }
coada::~coada(){while(fata!=NULL) exclude(); }
void stiva::creare(){ int c;
    cout<<"Introdu numarul de elemente din stiva"<<endl;
    cin>>c; if(c!=0) for(int i=0;i<c;i++) inserare(); }
```

Definirea metodelor claselor stiva și coada

```
void stiva::parcurge(){
    TStudent *p; p=curent;
    if(curent==NULL)
        cout<<"Stiva vida\n";
    while(p!=NULL) {
        cout<<p; p=p->next;
    }
    void stiva::inserare(){
        TStudent *q;
        q=new TStudent; cin>>q;
        q->next=curent;
        curent=q;
    }
    void stiva::exclude(){
        TStudent *q; q=curent;
        curent=curent->next;
        delete q;
    }
```

```
void coada::parcurge(){
    TStudent *p; p=fata;
    if(fata==NULL)
        cout<<"Coada vida"<<endl;
    while(p!=NULL){
        cout<<p; p=p->next;
    }
    void coada::inserare(){
        TStudent *q;
        q=new TStudent; cin>>q;
        spate->next=q; spate=q;
    }
    void coada::exclude(){
        TStudent *q; q=fata;
        fata=fata->next;
        delete q;
    }
```

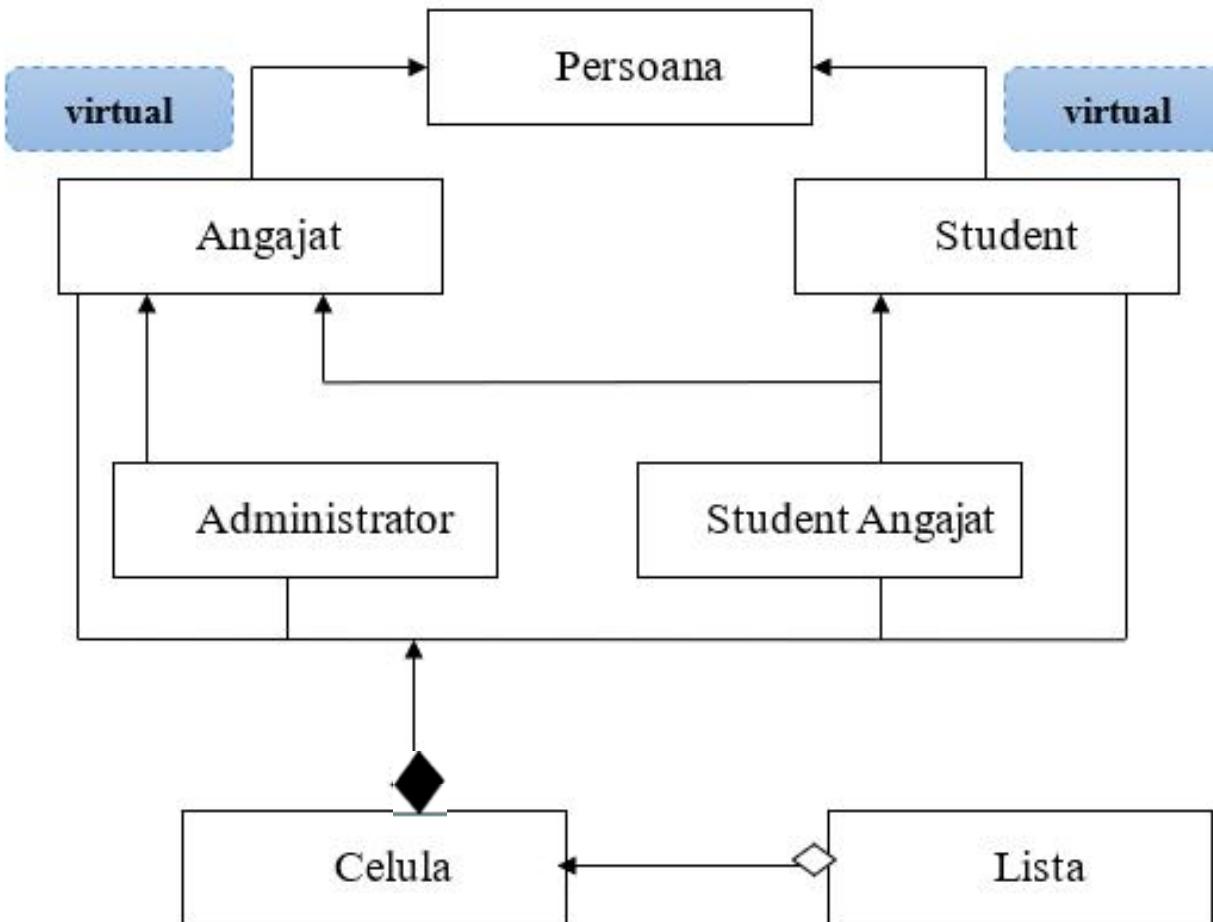
Unele funcții ale clasei TStudent

```
void coada::creare(){ int c; TStudent *p,*q;
cout<<"Introdu numarul de elemente din coada" << endl; cin>>c;
if(c!=0) for(int i=0;i<c;i++) {
    if(fata==NULL) { fata=new TStudent; cin>>fata; p=fata; }
    else{ q=new TStudent; cin>>q; p->next=q; p=q; }
} spate=p;}
int main(){
cout<<"prelucrarea stivei" << endl;
stiva s; s.creare(); s.parcurge();
cout<<"Inserare" << endl; s.inserare(); s.parcurge();
cout<<"exclude" << endl; s.exclude(); s.parcurge();
cout<<"Prelucrare cozii" << endl;
coada c; c.creare(); c.parcurge();
cout<<"Inserare" << endl; c.inserare(); c.parcurge();
cout<<"exclude" << endl; c.exclude(); c.parcurge();
}
```

Studiu de caz

- § Se solicită elaborarea unui program, prin intermediul căruia se vor prelucra datele despre personalul unei universități, pentru următoarele categorii de personal: *Angajat*, *Student* și *Student_angajat*, *Administrator*.
- § În funcție de tipul înlănțuirii realizate între componente, există următoarele tipuri de organizări:
 - Despre un **angajat** se cunoaște: numele, prenumele, anul nașterii, nr_ore_lucrate, plata_ora;
 - Despre un **student** se cunoaște numele, prenumele, anul nașterii, grupa, media;
 - Despre un **administrator** se cunoaște: numele, prenumele, anul nașterii, nr_ore_lucrate, plata_ora, anul angajării și gradul de calificare.
- § Programul va oferi posibilitatea de a gestiona (include/exclude, afișa, calcula, etc) datele corespunzătoare tipurilor existente, precum și va oferi posibilitatea de a completa programul cu noi tipuri de date fără a face modificări majore în codul sursă.

dr. Silviu GÎNCU



Reprezentarea grafică a relațiilor dintre clase

Structura clasei Persona

```
class Persoana{
    char *idnp;
    static int an;
protected:
    char *nume, *prenume;
    int anul;
public:
    Persoana(char [], char [], int);
    virtual void afisare();
    virtual void citire();
    virtual int salariu(){return 0;}
    virtual int bursa(){return 0;}
    virtual int tip(){return 0;}
    int virsta(){return an-anul;}
    virtual ~Persoana();
};
```

Structura clasei Angajat și Administrator

```
class Angajat : virtual public Persoana{
protected:
    int ore, pl_ora;
public:
    Angajat( char [ ], char [ ], int, int, int );
    void afisare();    void citire();
    int salariu();    int tip(){return 2;}
};

class Administrator : public Angajat{
protected:
    int an_ang,grad;
Administrator(char [ ],char [ ],int, int, int, int, int);
void afisare();    void citire();
double salariu();    int tip(){return 4;}
};
```

Structura clasei Student și Student angajat

```
class Student : virtual public Persoana{
protected:
char *grupa;
float media;
public:
Student(char [], char[], int, char [], int);
void afisare();      void citire();
int bursa();          int tip(){return 1;}
~Student();
};

class Stud_Ang : public Student, public Angajat{
public:
Stud_Ang(char[], char[], int, char[], float, int, int);
void afisare();      void citire();
int tip(){return 3;}
};
```

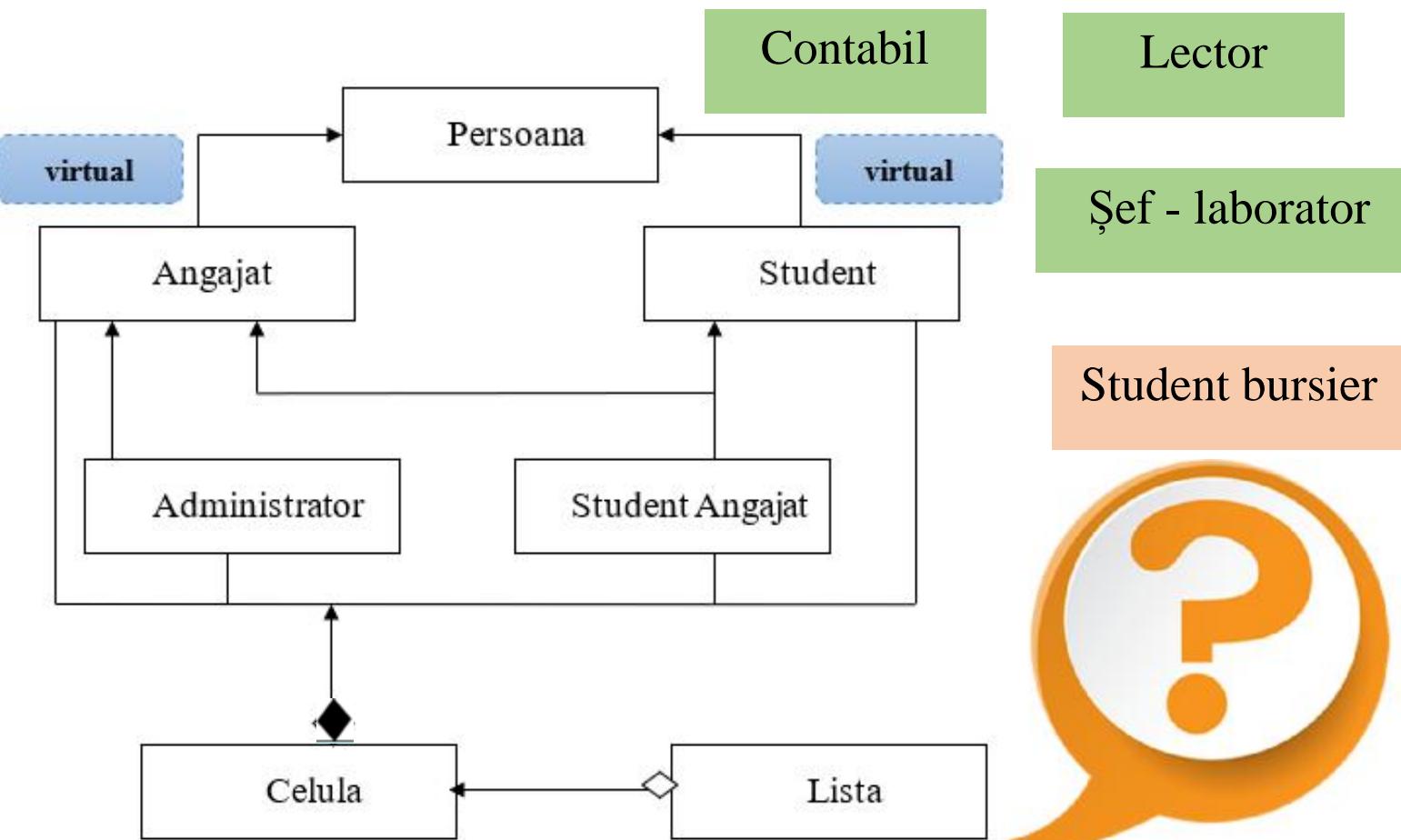
Structura clasei celula și lista

```
class celula{
public:
Persoana *p;
celula *next;
celula(){next=NULL; }
void cit();
void afis();
double consum();
};
```

```
class lista{
celula *prim;
public:
lista(){prim=NULL; }
public:
void creare(); void afisare();
void inserare(); void exclude();
double bani();
~lista();
};
```

```
int main(){
lista p;
// apelul metodelor clasei lista, conform condițiilor
}
```

Completarea problemei



Consultant
part - time

Jurist
part - time

Student bursier

Teme pentru acasă

- § A învăța și repeta conceptele prezentate în cadrul lecției
- § A depana programele prezentate în cadrul lecției
- § A dezvolta și depana problema propusă în cadrul studiului de caz

Prelegerea următoare

1. Programarea **generică**
2. Funcții **template**
3. Clase **template**
4. Utilizarea șabloanelor în relația de **agregare**
5. Utilizarea șabloanelor în relația de **moștenire**
6. Studiu de caz



Programarea Orientată pe Obiecte (POO)

Prelegere

Programarea generică

SUMAR

1. Programarea **generică**
2. Funcții **template**
3. Clase **template**
4. Utilizarea şablonelor în relația de **agregare**
5. Utilizarea şablonelor în relația de **moștenire**
6. Studiu de caz

Programarea generică

- § **Programarea generică** este o metodă de programare în care funcțiile și clasele au parametri formali cu tip nedefinit. În C++ programarea generică poate fi realizată prin mecanismul **template**.
- § **Template-ul** (sau clasa parametrizată) implementează conceptul de tip parametrizat. Parametrizarea datelor permite definirea unor clase care conțin tipuri de date nespecificate complet.
- § O clasă parametrizată reprezintă un **șablon** (sau container) ce definește o mulțime de clase. Deci, un container este o colecție de obiecte, în care poate fi accesat un singur obiect înt-un anumit moment.
- § Obiectivul principal al containerelor este instanțierea tipului de date al obiectelor componente. Un container definește operațiile ce se pot efectua asupra unor elemente componente, fără să fie precizat tipul acestora.
- § Șabloanele se realizează fie prin funcții **template** sau prin clase **template**.

dr. Silviu GÎNCU

Funcții template

- § În cazul când se utilizează același algoritm pentru diferite tipuri de date, deseori se creează o funcție **template**, care are un parametru de tip formal, ce urmează să precizeze tipul ei.
- § Declararea unei funcții template se realizează conform sintaxei:
template <class T> T nume_functie(lista param formali) {}
unde **T** reprezintă parametrul de tip formal, adică tipul funcției.
- § Dacă funcția are doi parametri tip atunci declarația template-ului se realizează după cum urmează:
template <class T, class M >
 tip nume_functie(lista param formali){
 // corpul funcției
 }
unde **T, M** reprezintă parametrii tipurilor de date; tipul funcției poate fi **T, M** sau orice tip de date compatibil și descris în cadrul programului.
dr. Silviu GÎNCU

Problema 1

Următorul program implementează două funcții parametrizate pentru determinarea numărului maximal dintre două tipuri de date și o funcție pentru ordonarea elementelor unui tablou unidimensional

```
template <class T> T maxim(T a,T b){if(a>b) return a; else return b; }
template<class M> void sort(M* vect, int n){int i,j; M x;
for(i=0; i<n; i++){ x = *(vect + i); j = i - 1;
while((j >= 0) && (x < *(vect + j))){ 
*(vect + j + 1) = *(vect + j); j--; } *(vect + j + 1) = x; }
int main(){
cout<<"int : "<<maxim<int>(4,7)<<endl;
cout<<"double : "<<maxim<double>(4.7,2.3)<<endl;
cout<<"char : "<<maxim<char>('c','h')<<endl;
double dvect[]={4.7,0.66,7.0,1.8,3.0}; sort(dvect,5);
for (int i=0; i<5; i++) cout << dvect[i] << " ";
int ivect[] = {10, 9, 5, 3, -2, 4}; sort(ivect,6);
for (int i=0; i<6; i++) cout << dvect[i] << " "; }
```

Rezultate

- § Compilatorul creează câte o funcție pentru fiecare tip de dată folosit ca argument de apel:
- § **maxim** - o funcție pentru determinarea valorii maximale dintre două valori de tip **char** și, respectiv pentru numere întregi de tip **int** și **double**
- § **sort** - o funcție de sortare pentru vectori de numere întregi și, respectiv pentru vectori de numere de tip **double**
- § După apelul funcției main rezultate sunt afișate la consolă:

```
int : 7
double : 4.7
char : h
0.66 1.8 3.0 4.7 7.0
-2 3 4 5 9 10
```

Problema 2

Următorul program implementează două funcții template, fiecare dintre ale având câte două tipuri de date parametrizate.

```
template <class T, class M>
void f(T a, M b){ cout << a << "\t" << b; }
template <class T, class M> T g(T a, M b){
return a+(T)b; }
int main(){
f(40000, 6); cout<<"\n";
f(1, 2.3); cout<<"\n";
f(1, 'a'); cout<<"\n";
f(2.3, 1); cout<<"\n";
cout<<"g(2, 3.4)="\"><<g(2, 3.4)<<endl;
cout<<"g(10, 'A')="\"><<g(10, 'A')<<endl;
}
```

Rezultatele execuției

```
40000 6
1 2.3
1 a
2.3 1
g(2, 3.4)=5
g(10, 'A')=75
```

Clase template

- § O **clasă template** specifică modul în care pot fi construite clase individuale, diferite prin tipul sau tipurile de date asupra cărora se operează.
- § Prefixul **template <class T>** specifică declararea unui template cu un argument T. După această introducere, T este folosit exact la fel ca orice tip de date, în tot domeniul clasei template declarate.
- § Numele unei clase template urmat de numele tipurilor de date folosite ca argumente, încadrate între parantezele < și > este numele unei clase (definite aşa cum specifică template-ul) și poate fi folosită la fel ca oricare altă clasă.
- § Utilizarea template-urilor implică generarea de către compilator a fiecărei clase care corespunde tipului (sau tipurilor) de date folosit la declararea unui obiect.

Clase template

§ Sintaxa pentru **clasele template** este aceeași ca template-urile funcțiilor:

```
template <class T> class nume_clasa{  
    .....  
}
```

§ Antetul unei metode se declară în interiorul clasei la fel ca la clasele obișnuite, iar definiția metodei care se construiește în afara clasei se face adăugând clasa din care face parte.

```
template <class T> tip_met nume_clasa<T>::nume_met(parametri){  
    .....  
}
```

§ Pentru a defini un obiect al unei clase template procedăm în același fel ca și la definirea obiectelor de până acum.

Problema 3

Următorul program implementează clasa parametrizată Array, fiecare dintre ale având câte două tipuri de date parametrizate.

```
template <class T> class vector{T *v;  int n;
public:
vector(int c){n=c; v=new T[n];}
void citire();    void afisare();    void sortare();
T& operator[] (int);
~vector(){delete []v;}
};

template <class T> void vector<T>::citire(){
for(int i=0;i<n;i++){cout<<"Introdu elem. "<<i<<" ";
cin>>v[i];}}
template <class T> T& vector <T>::operator[](int k){
return v[k];}
template <class T> void vector<T>::afisare(){
for(int i=0;i<n;i++) cout<<v[i]<<" ";  cout<<endl;}
```

Apelul metodelor

```
template <class T> void vector<T>::sortare(){ int i,j; T x;
for(i=1; i<n; i++){x = v[i]; j = i - 1;
while((j >= 0) && (x < v[j])){v[j+1] = v[j]; j--;} v[j+1] = x;}
int main(){
vector<int> v1(5);
vector<double> v2(5);
vector<char> v3(5);
cout<<" 5 valori int "<<endl; v1.citire();
cout<<" 5 valori double "<<endl; v2.citire();
cout<<" 5 valori char "<<endl; v3.citire();
cout<<"valori citite"<<endl;
v1.afisare(); v2.afisare(); v3.afisare();
cout<<"vectorii sortati"<<endl;
v1.sortare(); v1.afisare(); v2.sortare();
v2.afisare(); v3.sortare(); v3.afisare();
cout<<"pozitia 3 in tabele"<<endl;
cout<<v1[3]<<v2[3]<<v3[3]<<endl;
}
```

Problema 4

Scrieti un template pentru o clasă **Echipa** care contine:

- § Două variabile membru de tipuri neprecizate
- § O metoda de tip "void display()" care să afișeze conținutul celor două variabile membru sub forma (a, b)
- § Definiti o structura Persoana cu doi membri de tip string :
 - nume;
 - prenume
- § Instantiați un obiect de tip Echipa <Persoana, Persoana>

```
template<class T, class V>
class Echipa{
T Tdata;
V Vdata;
public:
Echipa(T,V);
void display();
};
```

```
template<class T, class V>
Echipa<T,V>::Echipa(T Tdata, V Vdata){
    this->Tdata=Tdata;    this->Vdata=Vdata; }
template<class T, class V>
void Echipa<T,V>::display(){
    cout << "(" << Tdata;
    cout<< ", " << Vdata << " )\n";
}
```

```
class Pers{
    string nume, prenume;
public:
    Pers(string, string);
    ostream & operator << (ostream& output, Persoana& p){
    };
    Pers::Pers(string nume, string prenume){
        this->nume=nume;      this->prenume=prenume;
    }
    Pers::ostream & operator << (ostream& output, Persoana& p){
        output << p.nume << " " << p.prenume;
        return output;
    }
    int main(){
        Echipa<Pers,Pers> team(Pers("Ion","Manole"),Pers("Ana","Sandu"));
        t.display();
    }
}
```

Utilizarea şabloanelor în relaţia de agregare

Se cere să se scrie o clasă numită stiva care gestionează o stivă memorată ca lista liniară simplu înlăntuită.

Fiecare nod al stivei reține:

§ Date membru:

- o valoare de tipul formal T;
- adresa următorului element din listă;

§ Metode:

- constructorul – va atribui valoarea NULL pentru fiecare nod;
- funcții pentru realizarea operațiilor de intrare/ieșire pentru fiecare nod

Stiva va conține:

§ Date membru:

- o valoare de tipul formal T;
 - adresa următorului element din listă;
- #### § Metode:
- constructorul – constructor de initializare a listei;
 - funcții pentru realizarea operațiilor specifice unei liste: parcurgere, creare, excludere, inserare a elementelor stivei;
 - destructorul – pentru eliberarea memoriei ocupată de elementele stivei.

Componența claselor

```
template <class T> class celula{
    T elem;
public:
    celula *next;
    celula(){next=NULL; }
    void citire();
    void afisare();
};
```

```
template <class T> class stiva{
public:
    celula<T> *current;
    stiva(){current=NULL; }
    void creare();
    void parcurge();
    void inserare();
    void exclude();
    ~stiva();
};
```

```
template <class T> void celula<T>::citire(){cin>>elem; }
template <class T> void celula<T>::afisare(){cout<<setw(6)<<elem; }
template <class T> stiva<T>::~stiva(){while(current!=NULL) exclude(); }
```

Implementarea metodelor

```
template <class T> void meniu( stiva<T> a){  
    char c;  
    a.creare(); system("cls");  
    do{  
        cout<<"Alegeti optiunea:"<<endl;  
        cout<<"1-Parcurge"<<endl;  
        cout<<"2-Inserare"<<endl;  
        cout<<"3-Exclude"<<endl;  
        cout<<"0-iesire"<<endl;  
        c=getch(); system("cls");  
        switch(c){  
            case '1':a.parcurge(); getch(); break;  
            case '2':a.inserare(); break;  
            case '3':a.exclude(); break;  
        } system("cls");  
    }while(c!='0');  
}
```

```
template <class T> void  
stiva<T>::creare(){  
    int c;  
    cout<<"Introdu nr de elemente "<<endl;  
    cin>>c;  
    for(int i=0;i<c;i++) {  
        if(curent==NULL){curent=new celula<T>;  
        curent->citire();}  
        else inserare(); }  
    }  
template <class T> void stiva<T>::parcurge(){  
    celula<T> *p; p=curent;  
    while(p!=NULL){  
        p->afisare(); p=p->next;  
    }  
    cout<<endl;  
}
```

Obiecte parametrizate

```
template <class T> void stiva<T>::inserare(){
    celula<T> *q; q=new celula<T>;
    q->citire(); q->next=curent; curent=q;
}

template <class T> void stiva<T>::exclude(){
    celula<T> *q; q=curent;
    curent=curent->next; delete q;
}

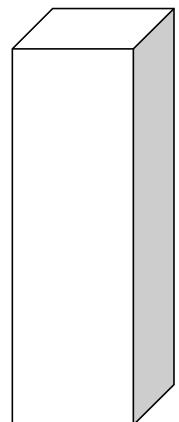
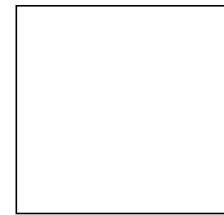
int main(){
    cout<<"Stiva de numere intregi"<<endl;
    stiva<int> sn;
    meniu(sn);

    cout<<"Stiva de caractere"<<endl;
    stiva<char> sc;
    meniu(sc);
}
```

Utilizarea şabloanelor în relaţia de moştenire

- § Clasele **template** ca şi clasele obişnuite susţin mecanismul de moştenire. Toate principiile de bază ale moştenire rămân neschimbate. Astfel se oferă posibilitatea de a construi modele ierarhice de clase.

§ Fie dată ierarhia:



- § Se consideră drept bază clasa *dreptunghi*, iar în calitate de derivată clasa *prisma*. Pentru această ierarhie va fi realizat polimorfismul pentru metodele citire, afisare, suprafata şi volum. Se va descrie şi constructorii ambelor clase.

Clase template – moștenire

```
template <class T> class drept {  
protected: T a,b;  
public:  
drept(){}; drept(T,T);  
virtual void citire(); virtual void afisare();  
virtual T suprafata(); virtual T volum(){return 0;} //metodă virtuala pură  
};  
template <class T> class prisma : public drept< T>{  
protected: T h;  
public:  
prisma(){};  
prisma(T,T,T);  
void citire(); void afisare();  
T suprafata(); T volum();  
};
```

Implementarea metodelor

```
template <class T> drept<T>::drept(T x, T y){a=x;b=y; }
template <class T> prisma<T>::prisma(T x,T y,T z) : drept<T>(x,y){h=z; }
template <class T> void drept<T>::citire(){cout<<"a="; cin>>a;
cout<<"b=";cin>>b; }
template <class T> void prisma<T>::citire(){drept<T>::citire();
cout<<"h=";cin>>h; }
template <class T> T drept<T>::suprafata(){return a*b; }
template <class T> T prisma<T>::suprafata(){return 2*(a*b+a*h+b*h); }
template <class T> T prisma<T>::volum(){return drept<T>::suprafata()*h; }
template <class T> void drept<T>::afisare(){
cout<<"Dreptunghi lungimile laturilor: "<<a<<" " <<b<<endl;
cout<<"Suprafata: "<<suprafata()<<endl; }
template <class T> void prisma<T>::afisare(){
cout<<"Prisma lungimile laturilor bazei: "<<a;
cout<<" " <<b<<"Inaltimea: "<<h<<endl<<"Suprafata: ";
cout<<suprafata()<<" Volumul: "<<volum()<<endl; }
```

Crearea obiectelor parametrizate

```
int main(){
    int i; double st,vt;
    drept<int> *p[4];
    p[0]=new drept<int>(2,3);
    p[1]=new prisma<int>(4,2,7);
    p[2]=new drept<int>; p[2]->citire();
    p[3]=new prisma<int>;p[3]->citire();
    cout<<"Datele introduse de tipul int" << endl;
    for(i=0;i<4;i++) p[i]->afisare();
    drept<double> *t[4];
    t[0]=new drept<double>(2.5,3);
    t[1]=new prisma<double>(4.3,2,7.4);
    t[2]=new drept<double>; t[2]->citire();
    t[3]=new prisma<double>;t[3]->citire();
```

Apelul metodelor

```
cout<<"Datele introduse de tipul double"=><endl;
for(i=0;i<4;i++) t[i]->afisare();
prisma<double> b[3];
cout<<"Dati datele a 3 prisme"=><endl;
for(i=0;i<3;i++) b[i].citire();
vt=st=0.0;
cout<<"Datele introduse"=><endl;
for(i=0;i<3;i++){
b[i].afisare();
vt+=b[i].volum();
st+=b[i].suprafata();
}
cout<<"Volumul total:="=><vt<<endl;
cout<<"Suprafata totala:="=><st<<endl;
}
```

Studiu de caz

Branduri

SAMSUNG Canon hp BRAUN SONY ORazor

Samsung Canon Hp Braun Sony Razor

FURY IDEA neffos UniFi MikroTik montebolla

Fury Idea Neffos UniFi MikroTik Montebolla

Synology UBIQUITI MikroTik Valera+ EcoCity

Synology Ubiquiti Mikrotik Valera EcoCity

MEGOGO eurogold celly Playseat smeg

Categorii

-  Telefoane și gadget-uri
-  Tehnică computerizată
-  Televizoare
-  Tehnică audio-video
-  Electrocasnice mici
-  Electrocasnice mari
-  Încorporabile
-  Tehnică de climatizare

Teme pentru acasă

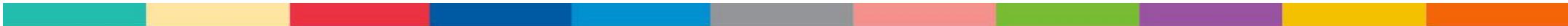
- § A învăță și repeta conceptele prezentate în cadrul lecției
- § A depana programele prezentate în cadrul lecției

Prelegerea următoare

Şabloane STL

1. Standard Template Library (STL)
2. Containeri
3. Iteratori

dr. Silviu GÎNCU



Programarea Orientată pe Obiecte (POO)

Prelegere

*Programarea generică:
containeri și iteratori*

SUMAR

1. Standard Template Library (STL)
2. Containeri
3. Iteratori

Noțiuni generale

- § Anii '70 - componentele folosite în programe erau sub forma structurilor de control și a funcțiilor
- § Anii '80 - au început să se folosească componente sub formă de clase dintr-o gamă largă de biblioteci dependente de platformă
- § Ultima parte a anilor '90 - odată cu apariția STL se introduce un nou nivel de folosire a componentelor prin clase independente de platformă
- § **Standard Template Library (STL)** o bibliotecă standard care cuprinde o colecție foarte cuprinzătoare de componente reutilizabile. Componentele cheie ale acestei biblioteci
 - Containerii (structuri de date sub formă de template-uri);
 - Iteratorii - sunt obiecte care se comportă asemănător pointerilor și care sunt utilizați pentru a accesa elementele unui container;
 - Algoritmii - furnizează funcționalități de acces și prelucrare asupra elementelor containerelor.

dr. Silviu GÎNCU

Containerele

- § Un **container** este un **obiect** care stochează o colecție de alte obiecte (elementele sale).
 - § **Containerul:**
 - gestionează spațiul pentru elemente
 - oferă funcții de acces la elemente, direct sau prin intermediul iteratorilor
 - § Unele containere au funcții comune și împart aceleași funcționalități
 - § Alegerea unui anumit tip de container depinde de:
 - funcționalitățile oferite de container
 - eficiența (complexitatea) acestor funcționalități
- Exemplu:**
Tipul de date tablou (vector) ar putea fi extins astfel încât să fie posibil implementarea `vector<int>` `vector<char>` `vector<double>` `vector<Student>` sau, în general, tablouri de orice tip de dată.
- dr. Silviu GÎNCU

Clasificarea containerelor

secvențiale – colecții liniare și ordonate de date în care accesul se face pe baza poziției elementului în cadrul containerului

vector, list, deque

adaptate – adaugă funcționalități containerelor secvențiale

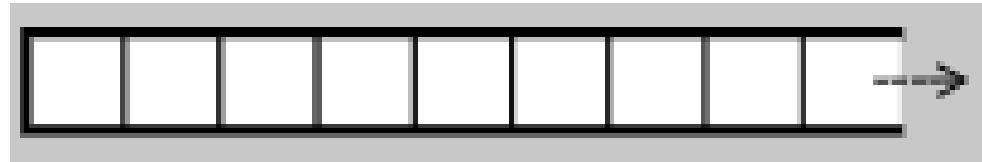
stack, queue, priority_queue

asociate – se diferențiază de celelalte prin faptul că stocarea elementelor se face pe baza unor chei. Accesul în acest caz se poate realiza după cheie, deci în mod direct.

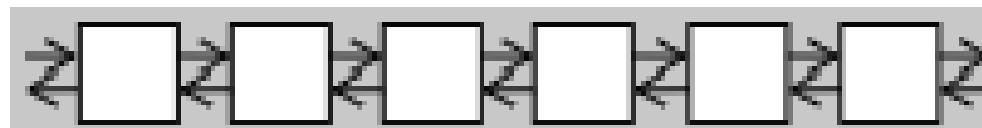
set, multiset, map, multimap

Containerele sevențiale

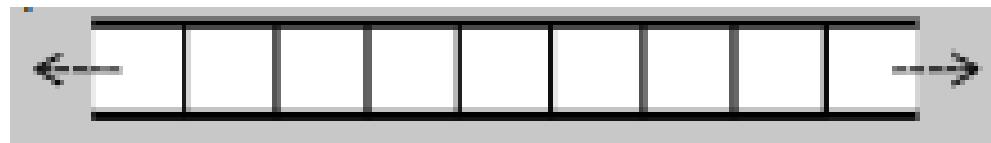
- § **vector** – tablou dinamic, cu redimensionare automată la inserarea unui nou element sau la ștergerea unui element



- § **list** – listă dublu înlănțuită cu inserare și ștergere rapidă

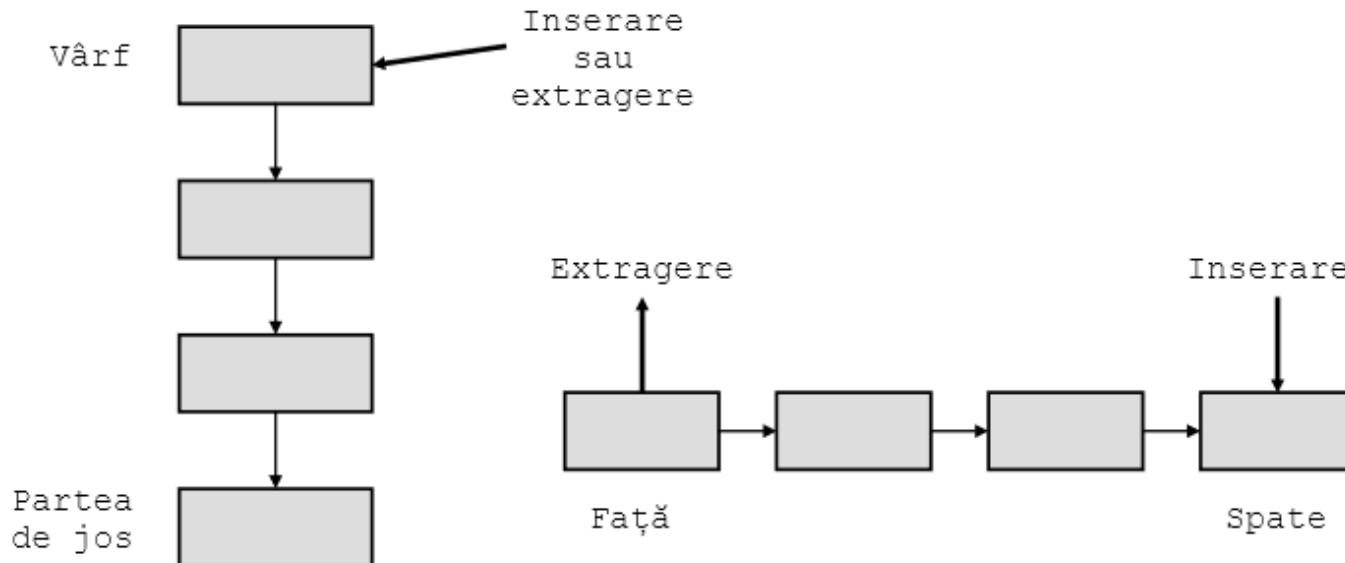


- § **deque** (coadă cu 2 capete) – coadă în care elementele pot fi adăugate sau șterse din ambele capete; diferă de coada obișnuită prin faptul că acolo adăugarea și ștergerea elementelor se face la un singur capăt



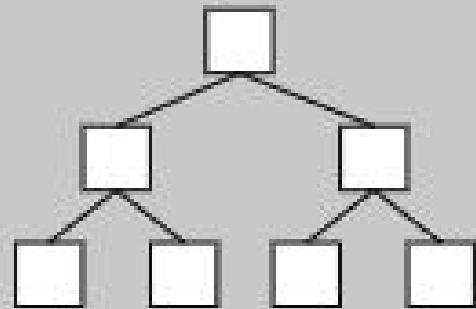
Containerele adaptate

- § **stack** – stivă: last-in-first-out (LIFO)
- § **queue** – coadă: first-in-first-out (FIFO)
- § **priority_queue** – coadă în care elementul cu prioritatea cea mai mare este întotdeauna primul element extras

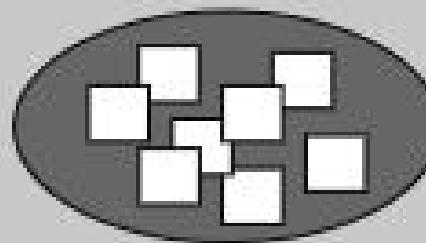


Containerele asociate

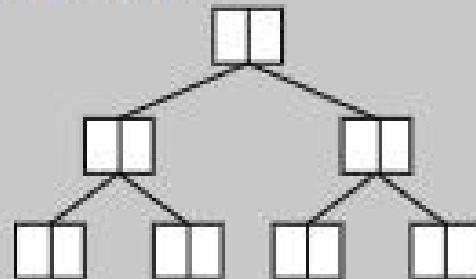
Set/Multiset:



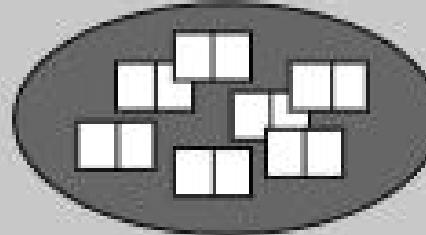
Unordered Set/Multiset:



Map/Multimap:



Unordered Map/Multimap:



- § **map** – asocieri între perechi de valori și chei, mapare unu-la-unu
- § **multimap** – similar cu map, dar acceptă duplicate
- § **set** – cheile sunt chiar valorile păstrate
- § **multiset** – este similar cu set, dar acceptă duplicate

Containeri – elemente comune

- § Un container este un obiect care stochează o colecție de alte obiecte (elementele sale).
- § Funcții membre pentru toate containerele:
 - gestionează spațiul pentru elemente;
 - Constructor implicit (default), constructor de copiere, destructor
 - empty, max_size, size
 - = < <= > >= == !=
- § Funcții pentru containere:
 - begin, end
 - rbegin, rend
 - erase, clear

<http://www.cplusplus.com/reference/stl/>

dr. Silviu GÎNCU

Containerul **vector**

§ Se folosesc vectori atunci când elementele lor trebuie parcurse de mai multe ori.

§ Header vector:

```
#include <vector>
using namespace std
```

§ Declarare:

```
vector <elemType> vectorName;
```

§ Operații de inserare a elementelor în container se realizează prin intermediul funcțiilor:

- *push_back*(val)
- *insert*(iterator, valoare)
- *insert*(iterator, numar_elemente_de_inserat, valoare)

§ Operații de excludere a elementelor în container se realizează prin intermediul funcțiilor:

- *erase*(iterator)
- *pop_back*()

dr. Silviu GÎNCU

Containerul **vector**

- § Operații de referire la elementele containerului se realizează prin intermediul:
 - ***push_back(val)***
 - ***operator[] - index***
 - Metoda ***at(index)***
 - Metodele ***front()*** și ***back()***
 - ***iteratorilor***
- § Dimensiunea actuală ***size_t size()***
- § Numărul maxim de elemente ***size_t capacity()***
- § Ștergerea tuturor elementelor: ***void clear()***
- § Verificare existență elemente ***bool empty()***
- § Interschimbul elementelor a doi vectori de același tip: ***void swap(vector&)***
- § Operatori relaționali ***<, <=, >, >=, ==, !=***

Problema 1

Următorul program demonstrează modalitatea de utilizare a containerului vector

```
#include <vector>
void print(const vector<int> &n){for (int j = 0; j < n.size(); j++)
    cout << "n[ " << j << " ] = " << n[j] << endl;}
int main(){
    vector<int> v1, tab = {50, 45, 47, 65, 80};
    cout << "lungimea vectorului: "<<tab.size()<<endl;    print(tab);
    cout<<"Primul element: "<<tab.front()<<endl;           //50
    cout<<"Ultimul element: "<<tab.back()<<endl;            //80
    cout<<"Redimensionarea vectorului"<<endl;
    tab = {50, 47, 60};
    cout << "lungimea vectorului: "<<tab.size()<<endl;    //3
    print(tab); tab.clear();
    cout <<"Capacity : "<<tab.capacity()<<endl;           //5
    cout << "lungimea vectorului: "<<tab.size()<<endl;    //0
```

Problema 1

```
if(v1.empty()) cout<<"Vectorul v1 nu contine elemente"<<endl;
if(tab.empty()) cout<<"Vectorul tab nu contine elemente"<<endl;
tab.resize(10); cout<<"vectorul redimensionat "<<endl; print(tab);
tab.at(7)=100; print(tab);
cout<<"numarul maxim de elemente ce pot fi memorate ";
cout<<tab.max_size()<<endl;
tab.assign(8, 40); print(tab);
cout<<"Insereaza pe ultima pozitie 51: "<<endl;
tab.push_back(51); print(tab);
cout<<"Exclude ultimul element "<<endl;
tab.pop_back(); print(tab);
v1={50, 45, 47, 65, 80};
tab.swap(v1); print(tab);
tab.erase(tab.begin()+2); print(tab); //exclude 47
v1.erase(v1.begin()+1,v1.begin()+3);print(tab); //45 47
}
```

Cazuri de utilizare ale metodei insert

```
vector<int> v = {0,1};
```

0	1
---	---

```
v.insert(v.begin(),2);
```

2	0	1
---	---	---

```
v.insert(v.begin(),3,333);
```

333	333	333	2	0	1
-----	-----	-----	---	---	---

```
vector<int> v1 = {555,555};
```

555	555
-----	-----

```
v.insert(v.begin()+4, v1.begin(), v1.end());
```

333	333	333	2	555	555	0	1
-----	-----	-----	---	-----	-----	---	---

Problema 2

Următorul program implementează containerul vector bidimensional

```
void print(const vector<vector<int>> &v) {
    for(int i=0; i<v.size(); i++){
        for(int j = 0; j<v[i].size(); j++) {
            cout << v[i][j] << "\t";
        }cout << endl;
    }
}

int main() {
    vector<vector<int>> tab = { {{1,2,3},
                                {4,5,6},
                                {7,8,9}}};
    print(tab);
}
```

dr. Silviu GINCU

Containerul secvență **<list>**

- § Containerul secvență **<list>** este o implementare a listei dublu înlăncuite. Acest container suportă și iteratori bidirecționali care permit parcurgerea sa în ambele sensuri.
- § Pe lângă funcțiile membre specifice tuturor containerilor și cele ale containerelor secvență, containerul list dispune și de funcții membre:

push_front	pop_front
splice	sort
unique	remove
reverse	merge
- § În vederea prezentării funcționalității funcțiilor membru ale containerului secvență **<list>** se consideră următoarea declarație:

```
list<int> values, o_Values = {2, 4, 6, 8};
```

Funcții - containerul <list>

§ Instrucțiunea ***push_front*** inserează o valoare la începutul listei, iar instrucțiunea ***push_back*** adaugă valori la sfârșitul său:

`values.push_front(1);`

`values.push_back(4);`

`values.push_front(2);`

`values.push_back(3);`

după aceste patru operații de inserare, lista values conține patru valori ordonate astfel: **2 1 4 3**.

§ Funcția `values.splice(values.end(), o_Values)`; șterge elementele din `o_Values` și le inserează în `values` înainte de poziția specificată (primul argument).

În exemplul redat, poziția specificată este `values.end()`, iar valorile din `o_Values` sunt adăugate la sfârșitul listei `values`.

§ După această operație, lista `values` va conține valorile:

`1 2 3 4 2 4 6 8`

dr. Silviu GÎNCU

Funcții - containerul <list>

- § Funcția membră **sort** fără nici un argument realizează o ordonare crescătoare a valorilor din listă:
`values.sort();`
după apelul acestei funcții, elementele din listă se vor regăsi în ordinea 1 2 2 3 4 4 6 8
- § Prin apelul `values.unique()`; se exclud elementele duplicate din listă.
după apelul acestei funcții, elementele din listă se vor regăsi în ordinea 1 2 3 4 6 8
- § **Notă !!!** Pentru a garanta ștergerea tuturor duplicatelor din listă, aceasta trebuie sortată în prealabil pentru ca valorile care se repetă să se găsească pe poziții alăturate.
- § Prin apelul `values.remove(4)`; sunt șterse toate aparițiile lui 4 din lista values.
după apelul acestei funcții, elementele din listă se vor regăsi în ordinea 1 2 3 6 8

Funcții - containerul <list>

- § Funcția **reverse()**; se utilizează fără parametri, iar prin apelul ei sunt inversate elementele din listă. după apelul `values.reverse()`; elementele din listă se vor regăsi în ordinea 8 6 3 2 1
- § Funcția **merge()** este utilizată pentru concatenarea a două liste. După apelul funcției elementele liste vor fi sortate. Sunt uzuale două metode de utilizare a funcției:

Sintaxa de utilizare:

```
list_container1.merge(list_container2);  
List_container1.merge(list_container2, comparator);
```

Exemplu :

```
list<int> myList_1 = {2, 4, 6, 8 };
```

```
list<int> myList_2 = {1, 3, 5, 7 };
```

după apelul `myList_2.merge(myList_1)`; obiectul `myList_2` va conține următoarele elemente 1 2 3 4 5 6 7 8, iar elementele din obiectul `myList_1` vor fi exlcuse

dr. Silviu GÎNCU

Problema 3

- § Pentru tipul de date **int** și tipul definit **Student** să se creeze a câte un container de tip **vector**, **list** și **deque**. În fiecare dintre acestea să fie adăugate elemente de tip **int** sau **Student**, după caz, utilizând metode specifice acestor şabloane.
- § Tipul de date **Student** se va defini prin intermediul unei clase și va avea în calitate de

Date membru:

- câmp numeric de tipul int
- câmp pentru păstrarea numelui studentului

Metode:

- constructorul cu parametri și constructorul de copiere
- funcții specifice (get/set)
- Operatori supraîncărcați:
 - 1) Operatorul de comparație <
 - 2) Operatorul de ieșire <<

```
class Student{
    int nrMat;
    char nume[20];
public:
    Student(int nr = 0, char* n = "Student"){strcpy(nume, n); nrMat=nr; }
    void setNume(char* n){ strcpy(this->nume,n); }
    Student(const Student& s){this->nrMat = s.nrMat; strcpy(this->nume, s.nume); }
    bool operator<(Student& s) {return (nrMat < s.nrMat); }
    friend ostream& operator<<(ostream& o, Student s) {
        o<<s.getNrMat()<<" "<<s.getNume()<<endl;
        return o;
    }
    int getNrMat() { return nrMat; }
    char* getNume(){ return nume; }
};
```

Containerul vector

```
int main(){
    Student s1(1, "Maria");
    Student s2(2, "Ionel");
    Student s3(3, "Anton");
    cout<<"vector de tip de date predefinit"<<endl;
    vector<int> vectInt;
    vectInt.push_back(10); vectInt.push_back(1);
    vectInt.push_back(25); vectInt.push_back(12);
    for(int i=0; i<vectInt.size(); i++)
    cout<<setw(5)<<vectInt[i]<<endl;

    cout<<"vector de date definit de programator"<<endl;
    vector<Student> vectStud;
    vectStud.push_back(s3);
    vectStud.push_back(s2);
    vectStud.push_back(s1);
    for(int i=0; i<vectStud.size(); i++)
    cout<<setw(5)<<vectStud[i];
```

Containerul list

```
cout<<"lista de tip de data predefinit"<<endl;
list<int> listInt; list<int>::iterator itInt;
listInt.push_back(10);    listInt.push_front(1);
listInt.insert(listInt.end(),25);  listInt.insert(listInt.begin(),12);
for(itInt=listInt.begin(); itInt!=listInt.end(); itInt++)
cout<<setw(5)<<*itInt;  cout<<endl;
listInt.sort();  cout<<"Lista sortata"<<endl;
for(itInt=listInt.begin(); itInt!=listInt.end(); itInt++)
cout<<setw(5)<<*itInt;  cout<<endl;
cout<<"lista de data definit de programator"<<endl;
list<Student> listStud; listStud.push_back(s3);  listStud.push_front(s1);
listStud.insert(listStud.end(),s2);  list<Student>::iterator itStud;
for(itStud=listStud.begin();itStud!=listStud.end();itStud++)
cout<<setw(5)<<*itStud;  listStud.sort();
cout<<"Lista de studenti sortata"<<endl;
for(itStud=listStud.begin();itStud!=listStud.end();itStud++)
cout<<setw(5)<<*itStud;
```

Containerul deque

```
cout<<"deque pentru tipuri de date predefinite" << endl;
deque<int> deqInt;
deqInt.push_front(10); deqInt.push_back(1);
deqInt.push_front(25); deqInt.push_back(12);

deque<int>::iterator iInt;
for(iInt = deqInt.begin(); iInt!= deqInt.end(); iInt++)
cout<<setw(5)<<*iInt; cout<< endl;

cout<<"deque pentru date definite de programator" << endl;
deque<Student> deqStud;
deqStud.push_front(s1); deqStud.push_back(s3); deqStud.push_front(s2);

deque<Student>::iterator iStud;
for(iStud=deqStud.begin();iStud!=deqStud.end();iStud++)
cout<<setw(5)<<*iStud; cout<< endl;
}
```

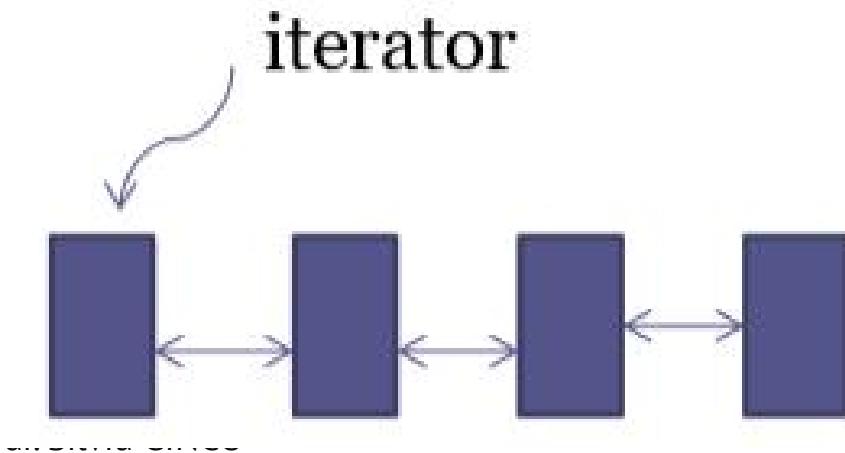
Containere: Vector sau deque?

- § **Deque** permite inserarea / ștergerea elementelor de la începutul / sfârșitul cozii
- § **Vectorul** permite inserarea / ștergerea elementelor doar la sfârșit
- § Accesul aleator la elemente este mai rapid la **vector** decât la **deque**
- § Pentru secvențe mari de elemente **vectorul** va aloca o zona mare de memorie continuă, pe când **deque** va aloca mai multe blocuri de memorie de dimensiuni mai mici – mai eficient din punctul de vedere al sistemelor de operare
- § **Deque** se comportă atât ca o stivă, cât și ca o coadă

Iteratori STL

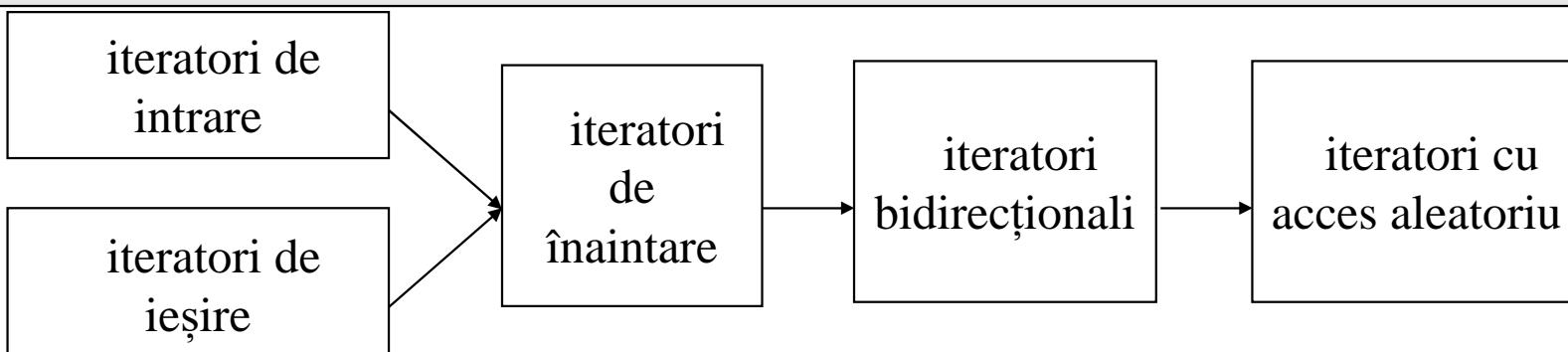
- § Iteratorii permit accesul la elementele unui container, independent de modul în care aceste elemente sunt stocate.
- § Iteratorii sunt asemănători cu pointerii, altfel spus aceștia sunt o generalizare a pointerilor, fiind obiecte care indică alte obiecte.
- § Fiecare clasă container are definiți iteratori proprii.

<http://www.cplusplus.com/reference/iterator/>



Categorii de iteratori

Iteratori	Operații permise
iteratori de intrare	Acces pentru citire. Parcurgere înainte
iteratori de ieșire	Acces pentru scriere. Parcurgere înainte
iteratori de înaintare	Acces pentru citire și scriere. Parcurgere înainte
iteratori bidirectionali	Acces pentru citire și scriere. Parcurgere în ambele sensuri
iteratori cu acces aleatoriu	Acces pentru citire și scriere. Acces direct (aleatoriu)



Iteratori

§ Majoritatea containerelor acceptă iteratori, cu excepția stivei, cozii și cozii cu priorități

Header : `#include <iterator>`

§ Declarare

`<container>::iterator nume iterator;`

De ex: `vector<int>::iterator it;`

§ Accesul la elementul curent - se realizează prin intermediul operatorului * sau, după caz ->

§ Parcursere prin intermediul iteratorilor:

```
vector<int> v;
for (int i=1; i<= 10; i++) v.push_back(i);
vector<int>::iterator it;
for (it = v.begin(); it != v.end(); ++it) cout << *it << " ";
```

dr. Silviu GÎNCU

Iteratori: intrare/ieșire

- § Un iterator de **intrare** servește pentru a citi dintr-un container, iar cel de **ieșire** transferă informația în container
- § Operațiile permise pentru această categorie de iteratori sunt incrementarea (`it++` sau `++it`) și derefențierea (`*it=...`)

```
#include <iterator>
int main(){
    int number1, number2;
    cout << "Introduceti doua numere intregi: ";
    istream_iterator<int> inputInt(cin);
    number1 = *inputInt; //citeste primul int
    ++inputInt; //muta iteratorul pe urmatoarea valoare
    number2 = *inputInt; //citeste urmatorul int
    cout << "Suma este: "; ostream_iterator<int> outputInt(cout);
    *outputInt = number1 + number2; cout << endl;
}
```

Iteratori pe streamuri

```
int main () {
vector<int> myvector;    int value;
cout << "Introduceți valori: (CTRL+Z to stop) ";
istream_iterator<int> eos;      // end-of-stream iterator
istream_iterator<int> iit (cin);   // stdin iterator
while (iit!=eos){myvector.push_back(*iit);  iit++;}
ifstream fin("date.in");
istream_iterator<int> fiit (fin);      //file iterator
while (fiit!=eos){myvector.push_back(*fiit);  fiit++;}
ostream_iterator<int> out_it (cout, ", ");
copy (myvector.begin(), myvector.end(), out_it );
ofstream fout("date.out");
ostream_iterator<int> fout_it(fout, " | ");
copy (myvector.begin(), myvector.end(), fout_it );
}
```

Iteratori pe containere reversibile

- § Container reversibil – produce iteratori care parcurge un container de la sfârșit spre început
- § Toate containerele standard permit existența iteratorilor reversibili
- § Pentru inițializare se utilizează funcțiile: **rbegin()**, **rend()**

```
#include <vector>
int main() {
    vector<int> v;
    v.push_back(3);  v.push_back(4); v.push_back(5);
    vector<int>::reverse_iterator rit=v.rbegin();
    while (rit<v.rend()){
        cout<<*rit<<endl;
        ++rit;
    } // va afișa 5 4 3
}
```

```
#include <iterator>
#include <list>
int main () {
    list<int> firstlist, secondlist;
    for(int i=1; i<=5; i++){
        firstdeque.push_back(i);           // 1 2 3 4 5
        seconddeque.push_back(i*10);      // 10 20 30 40 50
    }
    list<int>::iterator it;    it = firstlist.begin();
    advance (it,3);
    insert_iterator< list<int> > insert_it (firstlist,it);
    copy (secondlist.begin(),secondlist.end(),insert_it);
    for(it = firstlist.begin();it!= firstlist.end();++it)
        cout << *it << " ";
        cout << endl;
    // 1 2 3 10 20 30 40 50 4 5
}
```

back_insert_iterator

```
#include <iterator>
#include <vector>
int main () {
    vector<int> firstvector, secondvector;
    for (int i=1; i<=5; i++){
        firstvector.push_back(i);          // 1 2 3 4 5
        secondvector.push_back(i*10); // 10 20 30 40 50
    }
    back_insert_iterator< vector<int> > back_it (firstvector);
    copy(secondvector.begin(),secondvector.end(),back_it);

    ostream_iterator<int> out_it (cout, " , ");
    copy(firstvector.begin(), firstvector.end(), out_it );
    // 1 2 3 4 5 10 20 30 40 50
}
```

front_insert_iterator

```
#include <iterator>
#include <deque>
int main () {
    deque<int> firstdeque, seconddeque;
    for(int i=1; i<=5; i++){
        firstdeque.push_back(i);           // 1 2 3 4 5
        seconddeque.push_back(i*10);      // 10 20 30 40 50
    }
    front_insert_iterator< deque<int> > front_it (firstdeque);
    copy(seconddeque.begin(),seconddeque.end(),front_it);
    deque<int>::iterator it;
    ostream_iterator<int> oit(cout, " , ");
    copy(firstdeque.begin(),firstdeque.end(),oit);

    // 50 40 30 20 10 1 2 3 4 5
}
```

Concluzii: Iteratori & Containere

§ Tipuri de iteratori suportați de containere:

1) **Containere secvențiale:**

- **vector** - iteratori cu acces aleatoriu
- **deque** - iteratori cu acces aleatoriu
- **list** - iteratori bidirecționali

2) **Containere asociative:**

- **set, multiset** - iteratori bidirecționali
- **map, multimap** - iteratori bidirecționali

3) **Containere adaptate:**

- **stack, queue, priority_queue** – nu au atașați iteratori

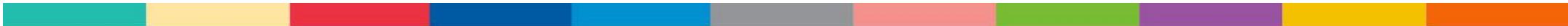
Teme pentru acasă

- § A învăță și repeta conceptele prezentate în cadrul lecției
- § A depana programele prezentate în cadrul lecției

Prelegerea următoare

Şabloane STL

1. Algoritmii
2. Exemplu de utilizare STL



Programarea Orientată pe Obiecte (POO)

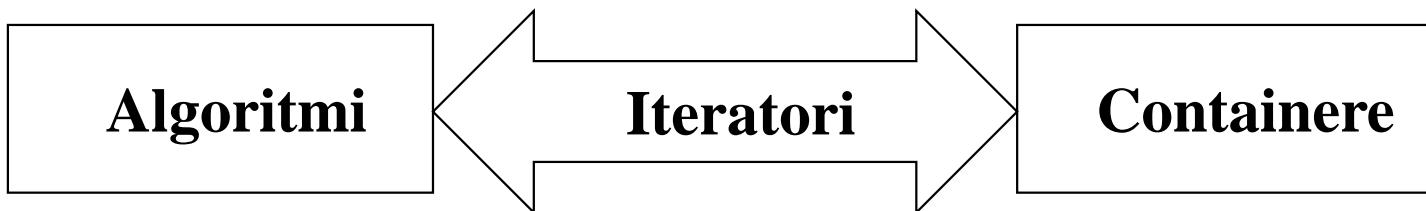
Prelegere

*Programarea generică:
algoritmi*

SUMAR

1. Algoritmii
2. Exemplu de utilizare STL

Algoritmi



- § Algoritmii operează asupra elementelor unei secvențe doar indirect, prin intermediul iteratorilor
- § STL include aproximativ 70 de algoritmi standard
- § Deoarece STL este extensibil, se pot adăuga cu ușurință noi algoritmi fără a opera nicio modificare asupra containerelor
- § Algoritmii:
 - sunt generici (nu depind de tipul de date)
 - sunt definiți în headerul `<algorithm>` sau `<numeric>`
 - În general, sunt implementați optim

dr. Silviu GÎNCU

Obiecte de tip funcție

- § Sunt utilizate în calitate de parametri la apelul algoritmilor/funcțiilor
- § Obiectele de tip funcție:
 - abstractizare care permite comportamentul de funcție
 - Instanță a unei clase care supraîncarcă operatorul () (apel de funcție)

```
class gt_n{  
    int val;  
public:  
    gt_n(int v){val=v;}  
    bool operator()(int n){return val>n;}  
};  
int main(){  
    gt_n f(4);  
    cout<<f(3)<<endl;//f.operator()(3); - true  
    cout<<f(5)<<endl;//f.operator()(5); - false  
}
```

Clasificarea obiectelor de tip funcție

- § **Generator** – obiect de tip funcție care nu ia nici un parametru și returnează o valoare de tip arbitrar (ex: rand(), <cstdlib>)
- § Funcție **unară** – funcție cu un parametru care returnează un tip arbitrar (inclusiv void)
- § Funcție **binară** – funcție cu 2 parametri care returnează un tip arbitrar (inclusiv void)
- § Predicat **unar** – funcție unară care returnează bool
- § Predicat **binar** – funcție binară care returnează bool
- § **Ordonare strictă** – predicat binar care permite o interpretare mai generală a egalității (2 elemente sunt egale dacă nici unul nu e strict mai mic decât altul – nr reale)
- § **LessThanComparable** – clasa care are supraîncărcat operatorul <
- § **Assignable** – clasa care are supraîncărcat operatorul =
- § **EqualityComparable** – clasa care are supraîncărcat operatorul ==

Algoritmii

- | | | |
|------------------|----------------|-------------------|
| 1. find | 9. swap | 17. reverse |
| 2. find_if | 10. transform | 18. sort |
| 3. equal | 11. replace | 19. binary_search |
| 4. count | 12. fill | 20. merge |
| 5. count_if | 13. generate | 21. includes |
| 6. mismatch | 14. generate_n | 22. min_element |
| 7. copy | 15. remove | 23. max_element |
| 8. copy_backward | 16. unique | |

find / find_if

- § **find** - returnează un iterator, care este primul element egal cu valoarea transmisă în calitate de parametru
- § **find_if** – similar find, iteratorul returnat va respecta condiția specificată

```
void myfunction (int i) {cout << " " << i ;}
int main () {
    vector<int> mv; mv.push_back(10); mv.push_back(20); mv.push_back(30);
    cout<< "Vectorul contine:"; for_each(mv.begin() ,mv.end() ,myfunction);
} // 10 20 30

bool positive (int i) { return (i>0); }
int t[] = { -10, -20, 30 ,40 };
vector<int> mv(t,t+4); vector<int>::iterator it;
it = find(mv.begin(), mv.end(), -20); ++it;
cout << "Succesorul lui -20 este " << *it << endl;
it = find_if(mv.begin(), mv.end(), positive);
cout << "Primul element >0 identificat este" << *it;
```

equal

§ **equal** - verifică dacă elementele din două domenii sunt egale facând compararea pe perechi de elemente corespunzătoare

```
bool egal (int i, int j) { return (i==j); }
int main () {
    int myints[] = {20,40,60,80,100};
    vector<int>myvector (myints,myints+5);
    if (equal (myvector.begin(), myvector.end(), myints))
        cout << "Containerele sunt egale" << endl;
    else cout << "Containerele nu sunt egale" << endl;

    myvector[3]=81;
    if(equal(myvector.begin(),myvector.end(),myints, egal))
        cout << "Containerele sunt egale" << endl;
    else cout << "Containerele nu sunt egale" << endl;
}
```

count/count_if

§ **count/count_if** - returnează numărul de apariții ale unui element într-o secvență sau / numărul elementelor pentru care o condiție este adevărată

```
bool poz(int i) { return (i>0); }
int main () {
    vector<int> mv;
    mv.push_back(-2);    mv.push_back(-3);
    mv.push_back(40);    mv.push_back(-2);
    mv.push_back(60);

    int nrPos=(int) count_if(mv.begin(),mv.end(), poz);
    cout << "Numarul de elemente >0 este " << nrPos;

    int nr =(int) count(mv.begin(), mv.end(), -2);
    cout << "numarul de aparitii ale lui -2 este "< nrPos;
}
```

mismatch

§ **mismatch** – compară două secvențe și returnează poziția primei diferențe

```
bool verif (int i, int j) {return (i==j);}  
int main () {  
    vector<int> mv;  
    for(int i=1; i<6; i++) mv.push_back (i*10);  
    int myints[] = {10,20,80,320,1024};  
    pair<vector<int>::iterator,int*> mp;  
    mp = mismatch (mv.begin(), mv.end(), myints);  
    cout << " Prima deosebire: " << *mp.first;  
    cout << " si " << *mp.second << endl;  
    mp.first++; mp.second++;  
    mp=mismatch(mp.first, mv.end(), mp.second, verif);  
    cout<< " a doua deosebire: " << *mp.first;  
    cout << " si " << *mp.second << endl;  
}
```

copy/copy_backward

§ **copy/copy_backward** – copiază elementele de pe un anumit diapazon în obiectul specificat

```
int my[ ]={10,20,30,40,50,60,70};  
vector<int> mv(7);  
vector<int>::iterator it;  
copy(my, my+7, mv.begin() );  
cout<<"elementele mv:";  
for(it=mv.begin(); it!=mv.end(); ++it) cout<< " "<<*it;  
cout << endl; // 10 20 30 40 50 60 70  
  
mv.resize(mv.size()+4);  
copy_backward(mv.begin(),mv.begin()+7, mv.end());  
cout<<"elementele mv:";  
for(it=mv.begin(); it!=mv.end(); ++it) cout<< " "<<*it;  
cout << endl; // 10 20 30 40 10 20 30 40 50 60 70
```

§ swap – interschimbă valorile a două obiecte

```
int x=10, y=20;
swap(x,y);
vector<int> first(5), second(5);
for(int i=0;i<5;i++){
    first.push_back(i*10); second[i]=i;
}
swap(first,second);
cout << "primul container:" ;
vector<int>::iterator it;
for(it=first.begin(); it!=first.end(); ++it)
cout << " " << *it;
cout << endl << "al doilea container:" ;
for(it=second.begin(); it!=second.end(); ++it)
cout << " " << *it;
```

transform

§ **transform** – aplică o transformare în acord cu parametri transmiși

```
int op_chSign (int i) { return (-1)*i; }
int op_dif (int i, int j) { return i-j; }

int main () {
    vector<int> f, s;    vector<int>::iterator it;
    for(int i=1; i<6; i++) f.push_back (i*10);
    s.resize(f.size());
    transform (f.begin(), f.end(), s.begin(),op_chSign);

    cout<<"Primul container:" ;
    for(it=f.begin(); it!=f.end(); ++it) cout<< " " << *it;
    transform(f.begin(),f.end(),s.begin(),f.begin(),op_dif);
    cout<<"După transformare:" ;
    for(it=f.begin(); it!=f.end(); ++it) cout<< " " << *it;
}
```

replace/fill

§ **replace** – înlocuiește pe un anumit interval valorile indicate

§ **fill** – completează cu elemente noi pe un anumit interval

```
int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };  
vector<int> myvector (myints, myints+8);  
replace(myvector.begin(), myvector.end(), 20, 99);  
vector<int>::iterator it; cout << "Elementele containerului:";  
for (it=myvector.begin(); it!=myvector.end(); ++it)  
cout << " " << *it; // 10 99 30 30 99 10 10 99  
  
vector<int> mv (8), mv1(8,10); vector<int>::iterator it;  
fill(mv.begin(),mv.begin()+4,5);  
fill(mv.begin()+3,mv.end()-2,8);  
cout<<"Elementele containerului mv:";  
for(it=mv.begin(); it!=mv.end();++it) cout << " "<<*it;  
fill_n(mv1.begin(),4,20); fill_n(mv1.begin()+3,3,33);  
cout<<" Elementele containerului mv1 :";  
for(it=mv1.begin(); it!=mv1.end(); ++it) cout<< " "<<*it;
```

generate / generate_n / remove

- § **generate/generate_n** – completează cu elemente un anumit interval
- § **remove** – elimină o valoare de pe un anumit interval

```
int RandomNumber(){ return (rand()%100); } int current=0;
int UniqueNumber () { return ++current; }
int main () {
    vector<int> mv (8); vector<int>::iterator it; int mr[9];
    generate(mv.begin(), mv.end(), RandomNumber);
    generate_n(mr, 9, UniqueNumber);
}
```

```
int myints[] = {10,20,30,40,50,20,70};
int* pbegin = myints;
int* pend = myints+sizeof(myints)/sizeof(int);
pend = remove(pbegin, pend, 20);
// 10 30 40 50 70
```

unique / reverse

- § **unique** – elimină elementele consecutive egale
- § **reverse** – inversează ordinea elementelor

```
bool myfunction (int i, int j) { return (i==j); }
int main () {
    int myints[] = {10,20,20,20,30,30,20,20,10};
    vector<int> mv (myints,myints+9);
    vector<int>::iterator it;
    it = unique (mv.begin(), mv.end());
    mv.resize( it - mv.begin() ) // 10 20 30 20 10
    unique (mv.begin(), mv.end(), myfunction);
}
```

```
vector<int> mv;
vector<int>::iterator it;
for (int i=1; i<10; ++i) mv.push_back(i);
reverse(mv.begin(),mv.end());
```

sort

§ **sort** – sortează elementele de pe un anumit interval în baza unui criteriu indicat

```
bool myfunction (int i,int j) { return (i<j); }
class myclass {
public:
bool operator() (int i,int j) { return (i<j); }
}myobject;
int main () {
int myints[] = {32,71,12,45,26,80,53,33};
vector<int> myvector (myints, myints+8); vector<int>::iterator it;
sort(myvector.begin(), myvector.begin()+4);
// 12 32 45 71 26 80 53 33
sort(myvector.begin()+4, myvector.end(), myfunction);
// 12 32 45 71 26 33 53 80
sort(myvector.begin(), myvector.end(), myobject);
// 12 26 32 33 45 53 71 80
}
```

binary_search

§ **binary_search** – verifică dacă un anumit element se regăsește într-un anumit diapazon

```
bool myfunction (int i,int j) { return (i<j); }
class myCmp{ public:
    int operator()(const int i, const int j){return i<j;}
}myComp;
int main () {
    int myints[] = {1,2,3,4,5,4,3,2,1};
    vector<int> v(myints,myints+9);    sort(v.begin(), v.end());
    cout << "cauta valoarea 3... ";
    if(binary_search (v.begin(), v.end(), 3))
        cout<<"gasit!\n"; else cout<<"nu e gasit\n";
    sort(v.begin(), v.end(), myfunction);
    cout << " cauta valoarea 6... ";
    if(binary_search (v.begin(), v.end(), 6, myComp))
        cout<<"gasit!\n"; else cout<<"nu e gasit\n";
}
```

merge / includes

- § **merge** – combină două secvențe (secvența finală este ordonată)
- § **includes** – verifică dacă elementele dintr-un domeniu se regăsesc în altul

```
int first[] = {5,10,15,20,25}; int second[] = {50,40,30,20,10};
vector<int> v(10); vector<int>::iterator it;
sort(first,first+5); sort(second,second+5);
merge(first,first+5,second,second+5,v.begin());
// 5 10 10 15 20 20 25 30 40 50
```

```
bool myfunction (int i, int j) { return i<j; }
int c[] = {5,10,15,20,25,30,35,40,45,50};
int c1[] = {40,30,20,10}; int c2[] = {40,30,20,10,9};
sort(c,c+10); sort(c1,c1+4); sort(c2,c2+5);
if(includes(c,c+10,c1,c1+4)) cout<<"c includes c1!";
if(includes(c,c+10,c1,c1+4, myfunction))
cout << "c includes c1!" << endl; if(includes(c,c+10,c2,c2+5))
cout << "include"; else cout << "nu include" << endl;
```

min_element/max_element

§ **min_element/max_element** – returnează un iterator la elementul minim/maxim dintr-un domeniu specificat

```
bool myfn(int i, int j) { return i<j; }
class myclass {
public:
bool operator() (int i, int j) { return i<j; }
} myobj;
int main () {
int m[ ] = {3,7,2,5,6,4,9};
cout<<"Elementul minimal "<<*min_element(m,m+7)<<endl;
cout<<"Elementul maximal "<<*max_element(m,m+7)<<endl;
cout<<"Elementul minimal "<<*min_element(m,m+7,myfn);
cout<<"Elementul maximal "<<*max_element(m,m+7,myfn);
cout<<"Elementul minimal "<<*min_element(m,m+7,myobj);
cout<<"Elementul maximal "<<*max_element(m,m+7,myobj);
}
```

Exemplu de utilizare STL

§ Se consideră tipul de date **Student** definit după cum urmează

Date membru:

- nume
- nr_o

Metode:

- constructorul cu/fără parametri
- destructorul
- funcții specifice (get/set)

Operatori supraîncărcați:

- de comparație: > < == !=
- de atribuire =
- de intrare/ieșire: >> <<

§ În baza tipului de date predefinit **int** și a tipului de date definit (**Student**) se vor aplica elemente din concepțele studiate (*containere, iteratori, algoritmi*)

dr. Silviu GÎNCU

Clasa student

§ Pentru realizarea programului se vor include bibliotecile: <iostream> <fstream> <string.h> <stdlib.h> <vector> <deque> <list> <stack> <queue> <set> <map> <algorithm>

```
class Student{
    char* name;
    int nr_o;
public:
    Student();
    Student(char* , int);
    Student(const Student &);
    Student(char* );
    ~Student();
    Student& operator=(const Student &);
    char* getName();
    int getNr_o();
    void setName(char* );
    void setNr_o(int);
};

bool operator<(const Student &) const;
bool operator>(const Student &) const;
bool operator==(const Student &) const;
bool operator!=(const Student &) const;
friend ostream& operator<<(ostream &, const Student &);
friend istream& operator>>(istream &, Student &);
```

Implementarea metodelor clasei student

```
Student::Student(){ name = NULL; nr_o = 0;}
Student::Student(char* name, int p){
    this->name = new char[strlen(name) + 1];
    strcpy(this->name, name);    nr_o = p;}
Student::Student(const Student &f){
    name = new char[strlen(f.name) + 1];
    strcpy(name, f.name); nr_o = f.nr_o;}
Student::Student(char* s){ name = new char[strlen(s) + 1];
strcpy(name, s);}
Student::~Student(){if (name){ delete[] name; name = NULL;}}
Student& Student::operator=(const Student &f){
if (this != &f){setName(f.name); nr_o = f.nr_o;}
return *this;}
int Student::getNr_o(){ return nr_o;}
char* Student::getName(){ return name;}
void Student::setName(char* n){ if (name){ delete[] name; }
name = new char[strlen(n) + 1]; strcpy(name, n);}
void Student::setNr_o(int p){ nr_o = p; }
```

```
bool Student::operator<(const Student &f) const{
    return (strcmp(name,f.name) == -1);}
bool Student::operator>(const Student &f) const{
    return (strcmp(name,f.name) == 1);}
bool Student::operator==(const Student &f) const{
    return (strcmp(name,f.name) == 0);}
bool Student::operator!=(const Student &f) const{
    return (strcmp(name,f.name) != 0);}
ostream& operator<<(ostream &os, const Student &f){
    os << f.name << " " << f.nr_o;    return os;}
istream& operator>>(istream &is, Student &f){
    char* s = new char[100];
    f.name = new char[strlen(s) + 1];
    strcpy(f.name, s);
    is >> f.nr_o;
    return is;
}
```

Containerul vector

```
void vectorOfIntegers(){
    vector<int> v; int n = 3; v.reserve(n);
    for(int i = 0; i < n; i++) v.push_back(i);
    cout << "vector cu numere intregi : " << endl;
    for(int i = 0; i < v.size(); i++){cout << v[i] << endl;
    }
    while(!v.empty()){ v.pop_back(); } // <=> v.clear();
}
void vectorOfStudents(){
    vector<Student> v; int n = 3;
    Student f1("Ion", 5); Student f2("Ana", 10);
    Student f3("Iulian", 2);
    v.reserve(n);
    v.push_back(f1); v.push_back(f2); v.push_back(f3);
    cout << "vector de Studenti : " << endl;
    for(int i = 0; i < v.size(); i++) cout << v[i] << endl;}
```

```
bool lessThan5(int val){ return (val < 5);}
bool lessThan10(Student f){
    return (f.getNr_o() < 10);}
class LessThan{
    int limit;
public:
    LessThan(int l){ limit = l; }
    LessThan(const LessThan &l){ limit = l.limit; }
    ~LessThan(){}
    LessThan& operator=(const LessThan &l){
        limit = l.limit; }
    int getLimit(){return limit;}
    bool operator()(int n){
        return (n < limit);}
};
```

Containerul vector cu iteratori

```
void vectorOfStudentAddresses(){
vector<Student*> v; int n = 3;
Student* f1 = new Student("Ion", 5);
Student* f2 = new Student("Ana", 10);
Student* f3 = new Student("Iulian", 2);
v.reserve(3);
v.push_back(f1);    v.push_back(f2); v.push_back(f3);
cout << "Adr: vector of Students is : " << endl;
n = v.size();
for (int i = 0; i < n; i++){
    cout << v[i] << endl;
    Student* f = v[i];    Student fc = *f;
    cout << fc.getName();
}
delete f3;  delete f2;  delete f1;
}
```

```
void vectorOfStudentsWithIterator(){
vector<Student> v; int n = 3;
Student f1("Ana", 5); Student f2("Ion", 10);
Student f3("Iulian", 2);
v.reserve(4);
v.push_back(f1);    v.push_back(f2);
v.push_back(f3);
cout << "vector de Studenti : " << endl;
vector<Student>::iterator it = v.begin();
while (it != v.end()){
    cout << *it << endl;
    it++;
}
v.clear();
}
```

Containerul vector cu iteratori

```
void vectorWithBidirectionalIterators(){  
    int a[] = {2, 5, 3};      int l = sizeof(a)/sizeof(int);  
    vector<int> v (a, a + sizeof(a)/sizeof(int));  
    cout << "vector cu numere intregi : " << endl;  
    vector<int>::reverse_iterator rit = v.rbegin();  
    while( rit != v.rend()){ cout << *rit << " "; ++rit;}  
    cout << endl;  
}
```

```
void dequeOfInteger(){  
    deque<int> d;  int n = 3;  
    for(int i = 0; i < n; i++)  d.push_back(i);  
    cout << "deque of integer: " << endl;  
    for(int i = 0; i < d.size(); i++) cout << d[i] << endl;  
    while(!d.empty()) d.pop_back();  
}
```

```
void dequeOfStudents(){  
    deque<Student> d;  int n = 3;  
    Student f1("Ion", 5);  Student f2("Ana", 10);  
    Student f3("Iulian", 2);  
    d.push_back(f1); d.push_back(f2);  
    d.push_back(f3);  
    cout << "deque of Students: " << endl;  
    for(int i = 0; i < d.size(); i++)  
        cout << d[i] << endl;  
}
```

Containerul vector/list cu iteratori

```
void listOfStudentsWithIterator(){
list<Student> l; int n = 3;
Student f1("Ion", 5); Student f2("Ana", 10);
Student f3("Iulian", 2);
l.insert(l.begin(), f3); l.insert(l.begin()++, f1);
l.insert(l.end(), f2);
cout << "lista este : " << endl;
list<Student>::iterator it = l.begin();
while (it != l.end()){ cout << *it << "; "; it++; }
cout << endl;
l.sort(); cout << "lista sortata: " << endl;
it = l.begin();
while (it != l.end()){cout << *it << "; "; it++; }
cout << endl;
}
```

```
void dequeOfStudentsWithIterator(){
deque<Student> d; int n = 3;
Student f1("Ion", 5); Student f2("Ana", 10);
Student f3("Iulian", 2);
d.push_back(f1); d.push_back(f2);
d.push_back(f3);
cout << "deque of Students : " << endl;
deque<Student>::iterator it = d.begin();
while (it != d.end()){
cout << *it << endl;
it++;
}
}
```

Containerul stack/queue/priority_queue

```
void stackOfStudents(){
stack<Student> s;    int n = 3;
Student f1("Ion", 5); Student f2("Ana", 10);
Student f3("Iulian", 2);
s.push(f1); s.push(f2); s.push(f3);
cout << "stack: " << endl;
while (!s.empty()){ cout << s.top() << endl; s.pop();}
cout << endl; }

void queueOfStudents(){
queue<Student> q; int n = 3;
Student f1("Ion", 5); Student f2("Ana", 10);
Student f3("Iulian", 2);
q.push(f1); q.push(f2); q.push(f3);
cout << "queue: " << endl;
while (!q.empty()){ cout << q.front() << endl; q.pop();}
cout << endl; }
```

```
void priorityQueueOfStudents(){
priority_queue<Student> pq;
int n = 3;
Student f1("Ion", 5); Student f2("Ana", 10);
Student f3("Iulian", 2);
pq.push(f1); pq.push(f2); pq.push(f3);
cout << "priority queue is : " << endl;
while (!pq.empty()){
cout << pq.top() << endl;
pq.pop();
}
cout << endl;
}
```

Containerul set

```
void setOfStudents(){
    set<Student> s; Student f5("Anton", 1);
    Student f1("Ion", 5); Student f2("Ana", 10);
    Student f3("Iulian", 2); Student f4("Maria", 20);
    s.insert(f1); s.insert(f2); s.insert(f3); s.insert(f4);
    cout << "set of Students: " << endl;
    set<Student>::iterator it = s.begin();
    while( it != s.end()){ cout << *it << " "; ++it;} cout << endl;
    set<Student>::iterator itf = s.find(f2);
    if (itf != s.end()) //<=> if (counter == 1)
        cout << "Student (" << f2 << ") este gasit..." << endl;
    else cout << "Student (" << f2 << ") nu este gasit \n";
    itf = s.find(f5);
    if (itf != s.end())
        cout << "Student (" << f5 << ") este gasit \n";
    else cout << "Student (" << f5 << ") nu este gasit \n";}
```

Containerul multiset

```
void multiSetOfStudents(){
multiset<Student> ms;
Student f1("Ion", 5); Student f2("Ana", 10); Student f3("Iulian", 2);
Student f4("Maria", 20); Student f5("Anton", 1);
ms.insert(f1); ms.insert(f2); ms.insert(f3); ms.insert(f4);
cout << "multiset of Students: " << endl;
multiset<Student>::iterator it = ms.begin();
while( it != ms.end()){cout << *it << " ";++it;} cout << endl;
multiset<Student>::iterator itf = ms.find(f2);
if (itf != ms.end()) //<=> if (counter != 0)
cout << "Student (" << f2 << ") este gasit \n";
else cout<<"Student (" << f2 << ") este gasit \n";
itf = ms.find(f5); if (itf != ms.end()) //if (counter != 0)
cout << "Student (" << f5 << ") este gasit \n";
else cout<< "Student" << "(" <<f5<< ")" nu este gasit \n";
int counter = ms.count(f2);
cout << "Student (" << f2 << ") apare de " << counter;
cout << " ori in multiset " << endl;}
```

Containerul map

```
void mapOfStudents(){
    map<char, Student> m;    Student f1("Ion", 5);
    Student f2("Ana", 10);    Student f3("Iulian", 2);    Student f4("Maria", 20);
    m.insert(pair<char, Student>(f1.getName()[0], f1));
    m.insert(pair<char, Student>(f2.getName()[0], f2));
    m.insert(pair<char, Student>(f3.getName()[0], f3));
    m.insert(pair<char, Student>(f4.getName()[0], f4));
    cout << "map of Students: " << endl;
    map<char, Student>::iterator it = m.begin();
    while( it != m.end()){
        cout<<"key = " << (*it).first << " value = " << (*it).second << endl; ++it;}
        cout<<endl; map<char, Student>::iterator itf = m.find(f2.getName()[0]);
        if (itf != m.end())
            cout<<"Student (" << f2 << ") este gasit ..." << endl;
        else cout << "Student (" << f2 << ") nu este gasit\n";
        int counter = m.count(f1.getName()[0]);
        cout<<"Student ("<<f1<<") apare "<<counter<<" ori"<<endl; }
```

Containerul multimap

```
void multiplemapOfStudents(){
multimap<char, Student> mm; Student f1("Ion", 5);
Student f2("Ana", 10); Student f3("Iulian", 2); Student f4("Maria", 20);
mm.insert(pair<char, Student>(f1.getName()[0], f1));
mm.insert(pair<char, Student>(f2.getName()[0], f2));
mm.insert(pair<char, Student>(f3.getName()[0], f3));
mm.insert(pair<char, Student>(f4.getName()[0], f4));
cout << "multiplemap of Students: " << endl;
multimap<char, Student>::iterator it = mm.begin(); while( it != mm.end()){
cout<<"key = "<< (*it).first << " value = " << (*it).second << endl; ++it; }
cout<<endl; multimap<char, Student>::iterator itf = mm.find(f2.getName()[0]);
if (itf != mm.end()) cout << "Student (" << f2 << ") este gasit \n";
else cout << "Student (" << f2 << ") nu este gasit" << endl;
int counter = mm.count(f1.getName()[0]);
cout<<"Student cu key = "<<f1.getName()[0]<<"apare "<<counter<<"ori ";
it = mm.equal_range(f1.getName()[0]).first;
while (it != mm.equal_range(f1.getName()[0]).second){
cout << (*it).second << ", "; it++; } cout << endl; }
```

Algoritmi de sortare

```
void sortVectorOfStudents(){
vector<Student> v;
Student f1("Ion", 5); Student f2("Ana", 10);
Student f3("Iulian", 2);
v.push_back(f1); v.push_back(f2); v.push_back(f3);
cout << "initial vector : ";
for(int i = 0; i < v.size(); i++) cout << v[i] << ", ";
cout << endl;
sort(v.begin(), v.end());
cout << "vectorul sortat crescator: ";
for(int i = 0; i < v.size(); i++) cout << v[i] << ", ";
cout << endl; reverse(v.begin(), v.end());
cout << "vectorul sortat descrescator: ";
for(int i = 0; i < v.size(); i++) cout << v[i] << ", ";
cout << endl;
}
```

```
void sortVectorOfIntegers(){
vector<int> v; v.push_back(9);
v.push_back(1); v.push_back(4);
cout << "initial vector : ";
for(int i = 0; i < v.size(); i++)
cout << v[i] << " "; cout << endl;
sort(v.begin(), v.end());
cout << "vectorul sortat > : ";
for(int i = 0; i < v.size(); i++)
cout << v[i] << " "; cout << endl;
reverse(v.begin(), v.end());
cout << "vectorul sortat < : ";
for(int i = 0; i < v.size(); i++)
cout << v[i] << " ";
cout << endl;}
```

Algoritmul copy

```
void copyVector(){
vector<int> v1; vector<int> v2(v1.size());
v1.push_back(9); v1.push_back(1); v1.push_back(4);
cout << "before copy " << endl;
for(int i = 0; i < v1.size(); i++) cout << v1[i] << ", ";
cout << endl; cout << "v2 = ";
for(int i = 0; i < v2.size(); i++) cout << v2[i] << ", ";
cout << endl;
copy(v1.begin(), v1.end(), v2.begin());
cout << "after copy " << endl; cout << "v1 = ";
for(int i = 0; i < v1.size(); i++) cout << v1[i] << ", ";
cout << endl;
cout << "v2 = ";
for(int i = 0; i < v2.size(); i++) cout << v2[i] << ", ";
cout << endl;
```

```
vector<int> v3;//v3 is empty
cout << "before copy " << endl;
cout << "v1 = ";
for(int i = 0; i < v1.size(); i++) cout << v1[i] << ", ";
cout << endl; cout << "v3 = ";
for(int i = 0; i < v3.size(); i++) cout << v3[i] << ", ";
cout << endl;
copy(v1.begin(), v1.end(), back_inserter(v3));
cout << "after copy " << endl; cout << "v1 = ";
for(int i = 0; i < v1.size(); i++) cout << v1[i] << ", ";
cout << endl; cout << "v3 = ";
for(int i = 0; i < v3.size(); i++) cout << v3[i] << ", ";
cout << endl;
}
```

Swap și Predicate

```
void vectorWithIteratorsAndPredicates(){
int MAX = 10;    vector<int> v;
for(int i = 0; i < MAX; i++)  v.push_back(i);
vector<int> copyv(MAX);
remove_copy_if(v.begin(), v.end(), copyv.begin(),
lessThan5);
cout << "copy vector : ";
for(int i = 0; i < copyv.size(); i++) cout << copyv[i] << " ";
cout << endl;
//cu iterator
cout << endl << "original vector : ";
vector<int>::iterator it = v.begin();
while( it != v.end()){cout << *it << " ";++it;}
cout << endl;
}
```

```
void swapVector(){
vector<int> v1; vector<int> v2;
v1.push_back(9); v1.push_back(1);
v1.push_back(4);
cout << "pina la schimb " << endl << " v1 = ";
for(int i = 0; i < v1.size(); i++) cout << v1[i] << ", ";
cout << endl;   cout << " v2 = ";
for(int i = 0; i < v2.size(); i++) cout << v2[i] << ", ";
cout << endl;   swap(v1, v2);
cout << "dupa schimb " << endl << " v1 = ";
for(int i = 0; i < v1.size(); i++) cout << v1[i] << ", ";
cout << endl;   cout << " v2 = ";
for(int i = 0; i < v2.size(); i++) cout << v2[i] << ", ";
cout << endl;
}
```

Swap și Predicate

```
void vectorWithIteratorsAndObjectFunctions(){
    LessThan lt1(5); cout << lt1(2); //false/0
    cout << lt1(7); //true/1
    int MAX = 10; vector<int> v;
    for(int i = 0; i < MAX; i++) v.push_back(i);
    vector<int> copyv(MAX);
    remove_copy_if(v.begin(), v.end(), copyv.begin(), lt1);
    cout << endl << "copy vector : ";
    for(int i = 0; i < copyv.size(); i++)
        cout << copyv[i] << " ";cout << endl;
    //cu iterator
    cout << endl << "original vector : ";
    vector<int>::iterator it = v.begin();
    while( it != v.end()){ cout << *it << " ";++it;}
    cout << endl;
}
```

```
int main(){
    vectorOfIntegers(); vectorOfStudents();
    vectorOfStudentAddresses();vectorOfStudentsWithIterator();
    vectorWithBidirectionalIterators();
    dequeOfInteger();dequeOfStudents();
    dequeOfStudentsWithIterator();
    listOfStudentsWithIterator();
    stackOfStudents(); queueOfStudents();
    priorityQueueOfStudents();
    setOfStudents(); multiSetOfStudents();
    mapOfStudents();multiplemapOfStudents();
    sortVectorOfIntegers(); swapVector();
    sortVectorOfStudents(); copyVector();
    vectorWithStreamIterators();
    vectorWithIteratorsAndPredicates();
    vectorWithIteratorsAndObjectFunctions();
}
```

Teme pentru acasă

- § A învăță și repeta conceptele prezentate în cadrul lecției
- § A depana programele prezentate în cadrul lecției

Prelegerea următoare

Tratarea Excepțiilor

1. Excepții. Noțiuni generale
2. Tratarea excepțiilor în C++
3. Exemple de tratare a excepțiilor în situații elementare
4. Ierarhii de clase în tratarea excepțiilor
5. Polimorfism
6. Tipuri de excepții predefinite
7. Exemplu de tratare a excepțiilor predefinite

```

#include <iostream>
#include <fstream>
#include <string.h>
#include <stdlib.h>
#include <vector>
#include <deque>
#include <list>
#include <stack>
#include <queue>
#include <set>
#include <map>
#include <algorithm>
using namespace std;
class Student{
private:
    char* name;
    int nr_o;
public:
    Student();
    Student(char* , int);
    Student(const Student &);
    Student(char* );
    ~Student();
    Student& operator=(const Student &);
    char* getName();
    int getNr_o();
    void setName(char* );
    void setNr_o(int);
    bool operator<(const Student &) const;
    bool operator>(const Student &) const;
    bool operator==(const Student &) const;
    bool operator!=(const Student &) const;
    friend ostream& operator<<(ostream &, const Student &);
    friend istream& operator>>(istream &, Student &);
};

class LessThan{
    int limit;
public:
    LessThan(int l){ limit = l; }
    LessThan(const LessThan &lt){ limit = lt.limit; }
    ~LessThan(){}
    LessThan& operator=(const LessThan &lt){ limit = lt.limit; }
    int getLimit(){ return limit; }
    bool operator() (int n){ return (n < limit); }
};

Student::Student() { name = NULL; nr_o = 0; }
Student::Student(char* name, int p){
    this->name = new char[strlen(name) + 1];
    strcpy(this->name, name);
    nr_o = p;
}
Student::Student(const Student &f){
    name = new char[strlen(f.name) + 1];
    strcpy(name, f.name); nr_o = f.nr_o;
}

```

```

Student::Student(char* s) {
    name = new char[strlen(s) + 1];
    strcpy(name, s);
}
Student::~Student(){if (name){ delete[] name; name = NULL;}}
Student& Student::operator=(const Student &f) {
    if (this != &f){setName(f.name); nr_o = f.nr_o; }
    return *this;
}
int Student::getNr_o(){ return nr_o; }
char* Student::getName(){ return name; }
void Student::setName(char* n){
    if (name){ delete[] name; }
    name = new char[strlen(n) + 1];
    strcpy(name, n);
}
void Student::setNr_o(int p){ nr_o = p; }
bool Student::operator<(const Student &f) const{
    return (strcmp(name, f.name) == -1);
}
bool Student::operator>(const Student &f) const{
    return (strcmp(name, f.name) == 1);
}
bool Student::operator==(const Student &f) const{
    return (strcmp(name, f.name) == 0);
}
bool Student::operator!=(const Student &f) const{
    return (strcmp(name, f.name) != 0);
}
ostream& operator<<(ostream &os, const Student &f){
    os << f.name << " " << f.nr_o;
    return os;
}
istream& operator>>(istream &is, Student &f){
    char* s = new char[100];
    f.name = new char[strlen(s) + 1];
    strcpy(f.name, s);
    is >> f.nr_o;
    return is;
}
void vectorOfIntegers(){
    vector<int> v;
    int n = 3;
    v.reserve(n);
    for(int i = 0; i < n; i++) v.push_back(i);
    //afisarea elementelor de la 0 la n
    cout << "vector cu numere intregi : " << endl;
    for(int i = 0; i < v.size(); i++){
        cout << v[i] << endl;    //=> cout << v.at(i) << endl;
    }
    while(!v.empty()){ v.pop_back(); } // => v.clear();
}
void vectorOfStudents(){
    vector<Student> v;
    int n = 3;
    Student f1("Ion", 5);
    Student f2("Ana", 10);
}

```

```

Student f3("Iulian", 2);
v.reserve(n);
v.push_back(f1);
v.push_back(f2);
v.push_back(f3);
cout << "vector de Studenti : " << endl;
for(int i = 0; i < v.size(); i++) cout << v[i] << endl;
}

void vectorOfStudentAddresses() {
    vector<Student*> v;
    int n = 3;
    Student* f1 = new Student("Ion", 5);
    Student* f2 = new Student("Ana", 10);
    Student* f3 = new Student("Iulian", 2);
    v.reserve(3);
    v.push_back(f1);
    v.push_back(f2);
    v.push_back(f3);
    cout << "Adr: vector of Students is : " << endl;
    n = v.size();
    for (int i = 0; i < n; i++) {
        cout << v[i] << endl;
        Student* f = v[i];
        Student fc = *f;
        cout << fc.getName();
    }
    delete f3;
    delete f2;
    delete f1;
}

void vectorOfStudentsWithIterator() {
    vector<Student> v;
    int n = 3;
    Student f1("Ana", 5);
    Student f2("Ion", 10);
    Student f3("Iulian", 2);
    v.reserve(4);
    v.push_back(f1);
    v.push_back(f2);
    v.push_back(f3);
    cout << "vector de Studenti : " << endl;
    vector<Student>::iterator it = v.begin();
    while (it != v.end()) {cout << *it << endl; it++; }
    v.clear();
}

void vectorWithBidirectionalIterators() {
    int a[] = {2, 5, 3};
    int l = sizeof(a)/sizeof(int);
    vector<int> v (a, a + sizeof(a)/sizeof(int));
    cout << "vector cu numere intregi : " << endl;
    vector<int>::reverse_iterator rit = v.rbegin();
    while( rit != v.rend()) { cout << *rit << " ";++rit; }
    cout << endl;
}

bool lessThan5(int val) { return (val < 5); }

```

```

bool lessThan10(Student f) { return (f.getNr_o() < 10); }
void vectorWithStreamIterators() {
    vector<Student> v;
    int n = 3;
    Student f1("Ion", 5);
    Student f2("Ana", 10);
    Student f3("Iulian", 2);
    v.push_back(f1);
    v.push_back(f2);
    v.push_back(f3);
    cout << "vector cu Studenti : " << endl;
    vector<int> v1;
    ifstream fin("dataInt.in");
    if (!fin.fail()) {
        cout << "vector of integers is : " << endl;
        fin.close();
    }
}
void dequeOfInteger() {
    deque<int> d;
    int n = 3;
    for(int i = 0; i < n; i++) d.push_back(i);
    cout << "deque of integer: " << endl;
    for(int i = 0; i < d.size(); i++) cout << d[i] << endl;
    while(!d.empty()) d.pop_back();
}
void dequeOfStudents() {
    deque<Student> d;
    int n = 3;
    Student f1("Ion", 5);
    Student f2("Ana", 10);
    Student f3("Iulian", 2);
    d.push_back(f1);
    d.push_back(f2);
    d.push_back(f3);
    cout << "deque of Students: " << endl;
    for(int i = 0; i < d.size(); i++) cout << d[i] << endl;
}
void dequeOfStudentsWithIterator() {
    deque<Student> d;
    int n = 3;
    Student f1("Ion", 5);
    Student f2("Ana", 10);
    Student f3("Iulian", 2);
    d.push_back(f1);
    d.push_back(f2);
    d.push_back(f3);
    cout << "deque of Students : " << endl;
    deque<Student>::iterator it = d.begin();
    while (it != d.end()) { cout << *it << endl; it++; }
}
void listOfStudentsWithIterator() {
    list<Student> l;
    int n = 3;
    Student f1("Ion", 5);
    Student f2("Ana", 10);
    Student f3("Iulian", 2);
}

```

```

l.insert(l.begin(), f3);
l.insert(l.begin()++, f1);
l.insert(l.end(), f2);
cout << "list is : " << endl;
list<Student>::iterator it = l.begin();
while (it != l.end()) { cout << *it << " "; it++; }
cout << endl;
l.sort();
cout << "sorted list: " << endl;
it = l.begin();
while (it != l.end()) { cout << *it << " "; it++; }
cout << endl;
}
void stackOfStudents() {
    stack<Student> s;
    int n = 3;
    Student f1("Ion", 5);
    Student f2("Ana", 10);
    Student f3("Iulian", 2);
    s.push(f1); s.push(f2); s.push(f3);
    cout << "stack: " << endl;
    while (!s.empty()) { cout << s.top() << endl; s.pop(); }
    cout << endl;
}
void queueOfStudents() {
    queue<Student> q;
    int n = 3;
    Student f1("Ion", 5);
    Student f2("Ana", 10);
    Student f3("Iulian", 2);
    q.push(f1); q.push(f2); q.push(f3);
    cout << "queue: " << endl;
    while (!q.empty()) { cout << q.front() << endl; q.pop(); }
    cout << endl;
}
void priorityQueueOfStudents() {
    priority_queue<Student> pq;
    int n = 3;
    Student f1("Ion", 5);
    Student f2("Ana", 10);
    Student f3("Iulian", 2);
    pq.push(f1); pq.push(f2); pq.push(f3);
    cout << "priority queue is : " << endl;
    while (!pq.empty()) { cout << pq.top() << endl; pq.pop(); }
    cout << endl;
}
void setOfStudents() {
    set<Student> s;
    Student f1("Ion", 5);
    Student f2("Ana", 10);
    Student f3("Iulian", 2);
    Student f4("Maria", 20);
    s.insert(f1); s.insert(f2);
    s.insert(f3); s.insert(f4);
    cout << "set of Students: " << endl;
    set<Student>::iterator it = s.begin();
    while (it != s.end()) { cout << *it << " "; ++it; }
}

```

```

cout << endl;
set<Student>::iterator itf = s.find(f2);
if (itf != s.end()) //<=> if (counter == 1)
cout << "Student (" << f2 << ") este gasit..." << endl;
else cout <<"Student (" << f2 << ") nu este gasit \n";
Student f5("Anton", 1);
itf = s.find(f5);           //int counter = s.count(f2);
if (itf != s.end())       //if (counter == 1)
cout << "Student (" << f5 << ") este gasit \n";
else cout << "Student (" << f5 << ") nu este gasit \n";
}
void multiSetOfStudents() {
multiset<Student> ms;
Student f1("Ion", 5);
Student f2("Ana", 10);
Student f3("Iulian", 2);
Student f4("Maria", 20);
ms.insert(f1); ms.insert(f2);
ms.insert(f3); ms.insert(f4);
cout << "multiset of Students: " << endl;
multiset<Student>::iterator it = ms.begin();
while( it != ms.end()) {cout << *it << " ";   ++it;}
cout << endl;
multiset<Student>::iterator itf = ms.find(f2);
if (itf != ms.end())      //<=> if (counter != 0)
cout << "Student (" << f2 << ") este gasit \n";
else cout <<"Student (" << f2 << ") este gasit \n";
Student f5("Anton", 1);
itf = ms.find(f5);           //int counter = ms.count(f2);
if (itf != ms.end())       //if (counter != 0)
cout << "Student (" << f5 << ") este gasit \n";
else cout << "Student" << "(" << f5 << ") nu este gasit \n";
int counter = ms.count(f2);
cout << "Student (" << f2 << ") apare de " << counter;
cout << " ori in multiset " << endl;
}
void mapOfStudents() {
map<char, Student> m;
Student f1("Ion", 5);
Student f2("Ana", 10);
Student f3("Iulian", 2);
Student f4("Maria", 20);
m.insert(pair<char, Student>(f1.getName() [0], f1));
m.insert(pair<char, Student>(f2.getName() [0], f2));
m.insert(pair<char, Student>(f3.getName() [0], f3));
m.insert(pair<char, Student>(f4.getName() [0], f4));
cout << "map of Students: " << endl;
map<char, Student>::iterator it = m.begin();
while( it != m.end()) {
cout << "key = " << (*it).first << " value = " << (*it).second << endl;
++it;
}
cout << endl;
map<char, Student>::iterator itf = m.find(f2.getName() [0]);
if (itf != m.end())
cout << "Student (" << f2 << ") este gasit ..." << endl;
else cout << "Student (" << f2 << ") nu este gasit\n";
}

```

```

int counter = m.count(f1.getName()[0]);
cout << "Student (" << f1 << ") apare " << counter << " ori" << endl;
}
void multimapOfStudents() {
    multimap<char, Student> mm;
    Student f1("Ion", 5);
    Student f2("Ana", 10);
    Student f3("Iulian", 2);
    Student f4("Maria", 20);
    mm.insert(pair<char, Student>(f1.getName()[0], f1));
    mm.insert(pair<char, Student>(f2.getName()[0], f2));
    mm.insert(pair<char, Student>(f3.getName()[0], f3));
    mm.insert(pair<char, Student>(f4.getName()[0], f4));
    cout << "multimap of Students: " << endl;
    multimap<char, Student>::iterator it = mm.begin();
    while( it != mm.end()){
        cout << "key = " << (*it).first << " value = " << (*it).second << endl;
        ++it;
    }
    cout << endl;
    multimap<char, Student>::iterator itf = mm.find(f2.getName()[0]);
    if (itf != mm.end())
        cout << "Student (" << f2 << ") este gasit \n";
    else cout << "Student (" << f2 << ") nu este gasit" << endl;
    int counter = mm.count(f1.getName()[0]);
    cout << "Student cu key = " << f1.getName()[0] << " apare " << counter << " ori";
    it = mm.equal_range(f1.getName()[0]).first;
    while (it != mm.equal_range(f1.getName()[0]).second) {
        cout << (*it).second << ", ";
        it++;
    }
    cout << endl;
}
void sortVectorOfIntegers(){
    vector<int> v;
    v.push_back(9);      v.push_back(1);      v.push_back(4);
    cout << "initial vector : ";
    for(int i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
    sort(v.begin(), v.end());
    cout << "vectorul sortat crescator : ";
    for(int i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
    reverse(v.begin(), v.end());
    cout << "vectorul sortat descrescator: ";
    for(int i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
}
void sortVectorOfStudents(){
    vector<Student> v;
    Student f1("Ion", 5);
    Student f2("Ana", 10);
    Student f3("Iulian", 2);
    v.push_back(f1);    v.push_back(f2);    v.push_back(f3);
    cout << "initial vector : ";
    for(int i = 0; i < v.size(); i++) cout << v[i] << ", ";
    cout << endl;
}

```

```

sort(v.begin(), v.end());
cout << " vectorul sortat crescator: ";
for(int i = 0; i < v.size(); i++) cout << v[i] << ", ";
cout << endl;
reverse(v.begin(), v.end());
cout << " vectorul sortat descrescator: ";
for(int i = 0; i < v.size(); i++) cout << v[i] << ", ";
cout << endl;
}
void swapVector(){
vector<int> v1;
v1.push_back(9);    v1.push_back(1);    v1.push_back(4);
vector<int> v2;
cout << "pina la schimb " << endl << " v1 = ";
for(int i = 0; i < v1.size(); i++) cout << v1[i] << ", ";
cout << endl;
cout << " v2 = ";
for(int i = 0; i < v2.size(); i++) cout << v2[i] << ", ";
cout << endl;
swap(v1, v2);
cout << "dupa schimb " << endl << " v1 = ";
for(int i = 0; i < v1.size(); i++) cout << v1[i] << ", ";
cout << endl;
cout << " v2 = ";
for(int i = 0; i < v2.size(); i++) cout << v2[i] << ", ";
cout << endl;
}
void copyVector(){
vector<int> v1;
v1.push_back(9);    v1.push_back(1);    v1.push_back(4);
vector<int> v2(v1.size());
cout << "before copy " << endl;
for(int i = 0; i < v1.size(); i++) cout << v1[i] << ", ";
cout << endl;
cout << " v2 = ";
for(int i = 0; i < v2.size(); i++) cout << v2[i] << ", ";
cout << endl;
copy(v1.begin(), v1.end(), v2.begin());
cout << "after copy " << endl;
cout << " v1 = ";
for(int i = 0; i < v1.size(); i++) cout << v1[i] << ", ";
cout << endl;
cout << " v2 = ";
for(int i = 0; i < v2.size(); i++) cout << v2[i] << ", ";
cout << endl;
vector<int> v3;//v3 is empty
cout << "before copy " << endl;
cout << " v1 = ";
for(int i = 0; i < v1.size(); i++) cout << v1[i] << ", ";
cout << endl;
cout << " v3 = ";
for(int i = 0; i < v3.size(); i++) cout << v3[i] << ", ";
cout << endl;
copy(v1.begin(), v1.end(), back_inserter(v3));
cout << "after copy " << endl;
cout << " v1 = ";
for(int i = 0; i < v1.size(); i++) cout << v1[i] << ", ";
cout << endl;
cout << " v3 = ";
for(int i = 0; i < v3.size(); i++) cout << v3[i] << ", ";
cout << endl;
}
void vectorWithIteratorsAndPredicates(){
int MAX = 10;
vector<int> v;
for(int i = 0; i < MAX; i++) v.push_back(i);
vector<int> copyv(MAX);
remove_copy_if(v.begin(), v.end(), copyv.begin(), lessThan5);
cout << "copy vector : ";
}

```

```

for(int i = 0; i < copyv.size(); i++) cout << copyv[i] << " ";
cout << endl;
    //cu iterator
cout << endl << "original vector : ";
vector<int>::iterator it = v.begin();
while( it != v.end()) {cout << *it << " ";++it;}
cout << endl;
}
void vectorWithIteratorsAndObjectFunctions () {
    LessThan lt1(5);
    cout << lt1(2);      //false/0
    cout << lt1(7);      //true/1
    int MAX = 10;
    vector<int> v;
    for(int i = 0; i < MAX; i++) v.push_back(i);
    vector<int> copyv(MAX);
    remove_copy_if(v.begin(), v.end(), copyv.begin(), lt1);
    cout << endl << "copy vector : ";
    for(int i = 0; i < copyv.size(); i++) cout << copyv[i] << " ";
    cout << endl;
        //cu iterator
    cout << endl << "original vector : ";
    vector<int>::iterator it = v.begin();
    while( it != v.end()) { cout << *it << " ";++it;}
    cout << endl;
}
int main() {
    //containere secrete
    vectorOfIntegers();
    vectorOfStudents();
    vectorOfStudentAddresses();
    vectorOfStudentsWithIterator();
    vectorWithBidirectionalIterators();
    dequeOfInteger();
    dequeOfStudents();
    dequeOfStudentsWithIterator();
    listOfStudentsWithIterator();

    //containere adaptate
    stackOfStudents();
    queueOfStudents();
    priorityQueueOfStudents();

    //containere asociate
    setOfStudents();
    multiSetOfStudents();
    mapOfStudents();
    multiplemapOfStudents();

    //algoritmi
    sortVectorOfIntegers();
    sortVectorOfStudents();
    swapVector();
    copyVector();
    vectorWithStreamIterators();
    vectorWithIteratorsAndPredicates();
    vectorWithIteratorsAndObjectFunctions();
}

```



Programarea Orientată pe Obiecte (POO)

Prelegere

Tratarea Exceptiilor

SUMAR

1. Excepții. Noțiuni generale
2. Tratarea excepțiilor în C++
3. Exemple de tratare a excepțiilor în situații elementare
4. Ierarhii de clase în tratarea excepțiilor
5. Polimorfism
6. Tipuri de excepții predefinite
7. Exemplu de tratare a excepțiilor predefinite

Context

- § Ideal, toate situațiile neobișnuite ce pot să apară pe parcursul execuției unui program trebuie detectate și tratate corespunzător de acesta.
- § De exemplu, un program ar putea cere utilizatorului să introducă de la tastatură un număr, dar acesta să introducă un sir de caractere ce nu reprezintă un număr. Un program bine gândit va detecta această problemă, va anunța greșeala și probabil va cere utilizatorului reintroducerea numărului.
- § Un program care se oprește subit în momentul în care utilizatorul face o greșală, fără nici un avertisment și fără nici un mesaj clar care să anunțe problema, va “băga în ceată” orice utilizator.
- § Un alt exemplu ar fi depășirea limitelor tabloului - eroare de programare. Tratarea unei astfel de situații ar putea implica de exemplu crearea unui raport de eroare de către utilizator la cererea programului. Tot este mai bine decât oprirea subită a programului.

dr. Silviu GÎNCU

Ce este o excepție?

- § O excepție este o eroare care poate să apară la rularea unui program, sau altfel spus un eveniment deosebit ce poate apărea pe parcursul execuției unui program și care necesită o deviere de la cursul normal de execuției al programului
 - § **Exemple:**
 - încercarea de deschidere a unui fișier ce nu există;
 - depășirea limitelor unui tablou;
 - încercarea de alocare a unui spațiu de memorie ce depășește dimensiunea heap-ului;
 - Încercarea de a efectua calcule fără sens (împărțirea la 0)
 - etc.
 - § Erorile se pot fi tratate în timpul rulării unui program. Cea mai folosită metodă de tratare a erorilor este aceea de a introduce în program secvențe de cod adaptate prevenirii unor posibile situații nedorite
- dr. Silviu GÎNCU

Avantaje / Dezavantaje

§ **Avantajul** acestei abordări este că:

- persoana care citește codul poate vedea modalitatea de prelucrare a erorii în vecinătatea codului
- poate determina dacă metoda de tratare a excepției este implementată corect

§ **Dezavantajul** este că prin această schemă codul:

- este oarecum „poluat” cu secvențe de procesare a erorilor
- devine mult mai greu de citit de un programator care este concentrat pe aplicația însăși

§ Această metodă face codul mult mai greu de înțeles și de menținut

§ Stilul C++ de tratare a excepțiilor permite interceptarea tuturor excepțiilor sau a tuturor excepțiilor de un anumit tip

- face programul mult mai robust, reducând probabilitatea de întrerupere neplanificată a programului

§ Tratarea excepțiilor se folosește în situația în care programul poate să își continue rularea după ce depășește eroarea care provoacă excepția

dr. Silviu GÎNCU

Tratarea excepțiilor în C++

- § C++ are un mecanism avansat de tratare a erorilor, codul ce tratează erorile fiind decuplat de logica aplicației:
 - Inițial scriem codul care dorim să se execute, apoi separat,
 - Se scrie codul ce tratează situațiile neprevăzute
- § O **excepție** în C++ este un obiect al unei clase ce este generat la apariția unei erori și cuprinde de regulă informații sumare despre cauza erorii.
- § Tratarea **excepțiilor** - mod organizat de a gestiona situațiile excepționale ce apar în timpul execuției. Mecanismul de tratare presupune:
 - Suspendarea execuției funcției curente în momentul în care s-a detectat o eroare, generarea unei excepții care conține informații referitoare la eroare și “aruncarea” (throw) ei către funcția apelantă;
 - Reluarea execuției programului în altă zonă, prinderea (catch) excepției și executarea acțiunilor necesare tratării erorilor

dr. Silviu GÎNCU

Elemente specifice

Elemente:

- § Blocul **try** - marchează blocul de instrucțiuni care poate arunca excepții.
- § Blocul **catch** - bloc de instrucțiuni care se executa în cazul în care apare o excepție (tratează excepția).
- § Instrucțiunea **throw** - mecanism prin care putem arunca (genere excepții) pentru a semnala codului client apariția unei probleme.

Sintaxa:

```
try{  
    // secvență ce poate genera excepții  
}  
catch (TipExcepție1 ex1){ //tratare TipExcepție1  
}  
catch (TipExcepție2 ex2){ //tratare TipExcepție2 . . .  
}  
catch (TipExcepțieN exN){ //tratare TipExcepțieN  
}  
catch (...) { //tratare orice alta excepție ce poate apărea  
}
```

Prinderea excepțiilor: try/catch

Mod de execuție:

- § **try** marchează începutul unui bloc de instrucțiuni care pot genera (arunca) excepții
- § dacă la execuția unei instrucțiuni este aruncată o excepție atunci se abandonează execuția instrucțiunilor din bloc și se încearcă (prinderea) identificarea tipului obiectului aruncat cu unul dintre tipurile specificat în ramurile **catch**: TipExceptie1, TipExceptie2 etc.
- § dacă se identifică o ramură **catch** pentru care tipul excepției coincide se va executa secvența de instrucțiuni pentru tratarea acelui tip de excepție
- § În cazul în care tipul excepției aruncate nu coincide cu nici unul din tipurile specificate în ramurile **catch** și există ramura **catch(...)** atunci se vor executa instrucțiunile din acel bloc

dr. Silviu GÎNCU

Tratarea excepțiilor în C++

- § Clauza catch nu trebuie neapărat să fie în același metodă unde se aruncă excepția. Excepția se propagă (transmite).
- § Când se aruncă o excepție, se caută cel mai apropiat bloc de catch care poate trata excepția.
- § Dacă nu avem cluză catch în funcția în care a apărut excepția, se caută clauza **catch** în funcția care a apelat funcția.
- § Căutarea continuă până se găsește o cluză **catch** potrivită. Dacă excepția nu se tratează (nu există o cluză catch potrivită) programul se oprește semnalând eroarea apărută.

Excepții - obiecte

- § Când se aruncă o excepție se poate folosi orice tip de date. Tipuri predefinite (**int**, **char**, etc) sau tipuri definite de utilizator (obiecte).
- § Este recomandat să se creeze clase pentru excepții.
- § Obiectul excepție este transmis ca referință sau pointer.
- § Obiectul excepție este folosit pentru a transmite informații despre eroarea apărută.

dr. Silviu GÎNCU

Problema 1

Se cere să se scrie un program care va genera excepții pentru valorile introduse de la tastatură 1 - 4, după cum urmează:

- § Pentru valoarea 1 se va afișa:
 - Mesajul **eroare de tip double**
- § Pentru valorile 2 și 3 se va afișa:
 - afișat la nivel de obiect al clasei,
 - Mesaj careva preciza că valoarea introdusă este 2 sau 3
- § Pentru valoarea 4 se va afișa:
 - Mesajul **eroare necunoscută**
- § Pentru gestionarea mesajului de eroare la nivel de clasă:
 - Se va crea clasa **Err**;
 - Clasa **Err** va conține o dată în care se va păstra mesajul de eroare;
 - Se va descrie constructorul cu parametri al clasei

dr. Silviu GÎNCU

Clasa Err

```
class Err{
public: const char* mesaj;
Err(const char* m){mesaj=m; }
};
void f(int n){ switch (n){
case 1: throw 0.5;
case 2: case 3: throw Err("Eroare n este 2 sau 3");
case 4: throw n;
} cout<<"Terminare functie f()"<
```

Rezultate

n=1

Execut functia f(1)

Tratare eroare de tip double: 0.5 n=1

n=3

Execut functia f(3)

Tratare eroare de tip E: Eroare n este 2 sau 3

n=4

Execut functia f(4)

Tratare eroare necunoscuta

n=5

Execut functia f(5)

Terminare funtie f()

Executie cu succes

Problema 2

§ Tratarea excepției împărțire la 0

```
class Divide_0{
    char * message;
public:
    Divide_0(): message( "Impartire la zero" ) {}
    char * print() {return message; }};
double imparte( int numarator, int numitor ){
    if ( numitor == 0 ) throw Divide_0();
    return (double) numarator / numitor;
}
int main(){ int n1=6, n2=0; double r;
try{ r = imparte(n1,n2);
    cout << "Impartirea este: "<<r<<endl;
} catch (Divide_0 Exc){cout <<"Exceptia: "<<Exc.print(); }
}
```

Excepții. Ierarhii de clase

- § Pentru determinarea unei clauze catch care se va executa dintr-o secvență de mai multe clauze, compilatorul determină prima clauză din secvență la care tipul parametrului se potrivește cu tipul obiectului generat de excepție. Această potrivire nu trebuie însă să fie perfectă, deoarece se utilizează principalele reguli ale derivării claselor.
- § Din acest motiv, pentru mai multe excepții înrudite, se pot forma ierarhii de clase pentru tratarea lor, iar clasele dintr-o asemenea ierarhie pot apărea în antetul clauzelor catch.
- § Un obiect al unei clase derivate, sau o referință spre un obiect al unei clase derivate, poate selecta un handler de tratare care are ca parametru un obiect al clasei de bază, sau o referință spre acesta.
- § Din acest motiv, în cazul unei secvențe de handle, specificarea acestora trebuie făcută de la clasa cea mai de jos a ierarhiei spre clasa de bază.
- § Folosind ierarhiile de excepții putem beneficia și de polimorfism

dr. Silviu GÎNCU

Problema 3

§ Ierarhii de trei clase pentru tratarea excepțiilor

```
class Avertisment {};
class Eroare: public Avertisment {};
class EroareFatala: public Eroare {};
void f(){ int n; cout<<"Nivelul erorii: (1) (2) (3)" ; cin>>n;
switch(n){
    case 1: throw Avertisment();
    case 2: throw Eroare();
    case 3: throw EroareFatala();
}
int main() {
try { f(); }
catch(EroareFatala){cerr<<"Exceptie EroareFatala"; }
catch(Eroare){cerr<<"Exceptie Eroare"; }
catch(Avertisment){cerr<<"Exceptie Avertisment"; }
}
```

Polimorfismul

- § O variantă frecvent utilizată în cazul ierarhiilor de clase pentru tratarea excepțiilor o constituie polimorfismul. În acest caz, clasa de bază a ierarhiei este indicat să fie o clasă abstractă.
- § Deoarece se utilizează pointeri pentru transmiterea informațiilor asupra excepției curente, trebuie ca în cadrul handlerelor de tratare, obiectele să fie distruse în mod explicit.

```
class Exceptie {  
protected: char m[20]; // Mesajul exceptiei  
public:  
virtual ~Exceptie() {}  
virtual void ProcesareExceptie() = 0;  
friend ostream& operator<<(ostream& os, Exceptie& e){return os << e.m; }  
};  
class Avertisment: public Exceptie{  
public:  
Avertisment(char* s){strcpy(m,s);}  
void ProcesareExceptie(){cout << m; exit(1);}  
};
```

Polimorfismul

```
class Eroare: public Exceptie{
public:          Eroare(char* s){strcpy(m, s);}
void ProcesareExceptie(){cout << m; exit(1);}
};

class EroareFatala: public Exceptie{
public:    EroareFatala(char* s){strcpy(m, s);}
void ProcesareExceptie(){cout << m; exit(1); }
};

void f() {int n; cout<<"Nivelul erorii: (1) (2) (3) ";
cin>>n; Exceptie *e; switch(n){
case 1: e=new Avertisment("Avertisment !"); throw e;
case 2: e=new Eroare("Eroare !!");           throw e;
case 3:e=new EroareFatala("Eroare fatala!!!");throw e;
}}
int main() { try {f(); }
catch(Exceptie* e){e->ProcesareExceptie(); delete e; }
}
```

Structura claselor

<http://www.cplusplus.com/reference/exception/exception/>

```
class exception{
public:
    exception () throw();
    exception (const exception&) throw();
    exception& operator= (const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
};

class out_of_range : public logic_error{
public:
    explicit out_of_range (const string& what_arg);
};
```

Tipuri de excepții predefinite

- § Din clasa exception sunt derivate două clase:
 - **logic_error** - pentru raportarea erorilor detectabile înainte de execuția programului și
 - **runtime_error** - pentru raportarea erorilor din timpul execuției programului.
- § Principalele clase derivate din **logic_error** sunt următoarele:
 - **domain_error** – raportează violarea unei precondiții;
 - **invalid_argument** – argument invalid pentru o funcție;
 - **length_error** – un obiect cu o lungime mai mare decât valoarea maximă reprezentabilă;
 - **out_of_range** – în afara domeniului;
 - **bad_cast** – operație dynamic_cast eronată.
- § Principalele clase derivate din **runtime_error** sunt următoarele:
 - **range_error** – violarea unei postcondiții;
 - **overflow_error** – operație aritmetică eronată (depășire);
 - **bad_alloc** – eroare de alocare.

Problema 4

Exemplu de tratare a excepțiilor `out_of_range`

Se cere să se scrie un program prin intermediul căruia se va gestiona erori de tipul `out_of_range` prin prisma unei clase `Vector`.

§ Date membru:

- Un tablou unidimensional de lungime Maximă MAX;
- Numărul de elemente ale tabloului;

§ Metode:

- Constructor - numărul de elemente ale tabloului și inițializarea cu 0 a acestora – va genera o excepție pentru cazul când $n > MAX$;
- Metodă pentru afișarea elementelor vectorului;
- Metodă care returnează numărul de elemente ale vectorului;
- Metodă care va returna valoarea de pe o anumită poziție, va genera o eroare în cazul când poziția solicitată nu este pe segmentul $[0; n-1]$;
- Metodă care va scrie o valoarea de pe o anumită poziție, va genera o eroare în cazul când poziția solicitată nu este pe segmentul $[0; n-1]$.

dr. Silviu GÎNCU

Exceptii de tipul `out_of_range`

```
#include <stdexcept>
#define MAX 10
using namespace std;
class Vector{
    float t[MAX];
    int n;
public:
    Vector(int);
    int getNrElemente();      void setElement(int, float);
    float getElement(int);   void afisare();
};
Vector::Vector(int n){
    this->n = n;
    if(n>MAX)  throw out_of_range("Depasire dimensiune ");
    for (int i = 0; i < n; i++) {  this->t[i] = 0;  }
```

Exceptii de tipul `out_of_range`

```
void Vector::setElement(int i, float val){  
    if (i >= 0 && i < n){ t[i] = val; }  
    else { throw out_of_range("Depasire limite"); } }  
float Vector::getElement(int i){  
    if (i >= 0 && i < n) return t[i];  
    else throw out_of_range("Depasire limite");}  
void Vector::afisare(){  
    for( int i=0;i<n;i++) cout<<setw(5)<<t[i]; cout<<endl; }  
int main(){  
    int n;  
    Vector v(2); v.afisare();  
    v.setElement(0,4); v.afisare();  
    try{ v.setElement(3,9); }  
    catch (exception &e){ cerr<<"Eroare:"<<e.what()<<endl; }  
    v.afisare();  
}
```

Rezultate

§ Pentru datele de intrare specificate în program la ieșire vor fi prezentate următoarele informații:

```
0      0
4      0
```

Eroare: Depasire limite

```
4      0
```

§ În cazul în care se intenționează a crea un vector format din 12 elemente

```
vector v(12);
```

§ Rezultatul afișat va fi:

```
terminate called after throwing an instance of 'std::out_of_range'
what():  Depasire dimensiune
```

Teme pentru acasă

- § A învăță și repeta conceptele prezentate în cadrul lecției
- § A depana programele prezentate în cadrul lecției



Prelegerea următoare

Studiu de caz

Sistemul informatic al facultății

dr. Silviu GÎNCU