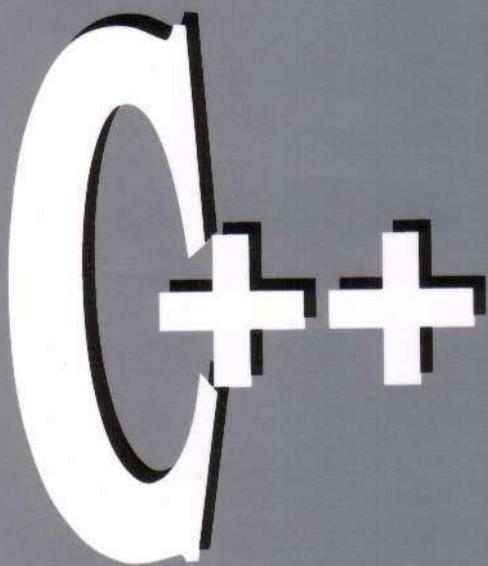


ION SMEUREANU MARIAN DÂRDALĂ

# Programarea orientată obiect în limbajul



**PROGRAMAREA  
ORIENTATĂ OBIECT  
în  
LIMBAJUL C++**

Editor:  
**NICULAE CHIRU**  
Coperta și prezentarea grafică  
**GHEORGHE CHIRU**

©Toate drepturile sunt rezervate editurii CISON

**Lucrarea poate fi comandată la:**

- Editura CISON tel. 0740092349 sau 021/7787193
- Catedra de Informatică Economică - ASE, str. Calea Dorobanți, nr. 15-17, et. 4 cam. 2413, tel. 021/2112650 int 336, 318 sau 0722747786



ISBN 973-8301-06-8

**1. ABSTRACTIZAREA DATELOR. CONCEPTUL DE CLASĂ**

1.1 Tipul abstract de date (Abstract Data Type – ADT) .....	10
1.2 Conceptele de clasă și obiect .....	11
• Clase și obiecte .....	11
• Separarea interfeței de partea de implementare .....	16
• Constructori și destrutor .....	17
• Obiecte cu extensii în memoria dinamică .....	22
• Pointerul <i>this</i> .....	26
• Funcțiile de acces .....	27
1.3 Pointeri la obiecte. Masive de obiecte .....	28
1.4 Clase incluse. Componerea obiectelor .....	30
1.5 Tipologia membrilor unei clase .....	33
• Clase cu membri constanți .....	33
• Specificatorul <i>static</i> aplicat membrilor unei clase .....	33
1.6 Transferul obiectelor în / din funcții .....	36
1.7 Pointeri de date și funcții membre .....	37
• Pointeri de membri, membri în clasă .....	42
1.8 Clase și funcții prietene. Privilegii în sistemul de acces .....	44
1.9 Modificatorul <i>const</i> în contextul obiectelor .....	46
• Obiecte constante .....	46
• Pointeri constanți de obiecte și pointeri de obiecte constante .....	48

**2. SUPRĂINCĂRCAREA OPERATORILOR ȘI FUNCȚIILOR**

2.1 Suprăîncărcarea funcțiilor independente și a funcțiilor membre .....	52
2.2 Aspecte generale și restricții privind suprăîncărcarea operatorilor .....	54
2.3 Suprăîncărcarea operatorilor .....	56

Pag

• Supraîncărcarea operatorilor unari <code>++</code> și <code>--</code> .....	56
• Supraîncărcarea operatorilor binari <code>+</code> de adunare și <code>+=</code> .....	58
• Supraîncărcări ale operatorilor <code>&gt;&gt;</code> și <code>&lt;&lt;</code> .....	62
• Supraîncărcarea operatorului <code>[ ]</code> .....	64
• Supraîncărcarea operatorilor <code>new</code> și <code>delete</code> .....	68
• Supraîncărcarea operatorului <code>cast</code> .....	71
• Supraîncărcarea operatorului virgulă .....	74
• Supraîncărcarea operatorului funcție .....	75
• Supraîncărcarea operatorului <code>-&gt;</code> .....	77
2.4 Conversii între obiecte de diferite tipuri .....	79
2.5 Aplicații .....	86
<b>3. CLASE DERIVATE. MOȘTENIRI. FUNCȚII VIRTUALE</b>	
3.1 Derivarea claselor. Moștenirea unor caracteristici .....	108
• Funcții care nu se moștenesc integral .....	114
• Moștenire versus includere de clase .....	116
3.2 Funcții virtuale .....	116
3.3 Moșteniri multiple .....	124
• Ambiguități la adresarea membrilor moșteniți, care se numesc la fel în două dintre clasele de bază .....	125
• Moșteniri multiple din clase cu o bază comună. Ambiguități la moștenirile din clase de bază cu o bază comună .....	126
• Derivare virtuală .....	129
• Componente virtuale și nevirtuale în aceeași clasă .....	130
• Ambiguități la funcții virtuale moștenite din clase derivate virtual .....	131
<b>4. OPERAȚII DE INTRARE / IEȘIRE ORIENTATE PE STREAM-URI</b>	
4.1 Lucru cu obiectele <code>cin</code> și <code>cout</code> .....	134
4.2 Intrări / ieșiri cu formatarea datelor .....	137
4.3 Detectarea erorilor apărute în operațiile de intrare / ieșire .....	144
4.4 Intrări / ieșiri pe fișiere nestandard .....	147
4.5 Formatarea datelor în memoria internă .....	158
<b>5. IMPLEMENTAREA OBIECTUALĂ A STRUCTURILOR DE DATE DINAMICE</b>	
5.1 Structuri de date dinamice liniare .....	162
• Lista simplu înlățuită .....	162
• Stiva .....	172
• Coada .....	174
5.2 Structuri arborescente .....	176
<b>6. SABLOANE DE CLASE (Clase <i>template</i>)</b>	
6.1 Funcții și clase <i>template</i> .....	186

• Funcții <i>template</i> .....	186
• Clase <i>template</i> .....	188
6.2 Instantierea sabloanelor. Constante în clasele <i>template</i> .....	192
6.3 Specializări .....	197
6.4 Relații între sabloane .....	203
• Derivarea claselor <i>template</i> .....	204
• Compunerea claselor <i>template</i> prin includere .....	209
• Compunerea claselor <i>template</i> prin parametrizare cu altă clasă <i>template</i> .....	211
<b>7. DOMENII DE NUME - NAMESPACE</b>	
7.1 Definirea și actualizarea domeniilor de nume .....	214
7.2 Utilizarea domeniilor de nume .....	215
<b>8. BIBLIOTECA DE SABLOANE STANDARD C++ (Standard Template Library - STL)</b>	
8.1 Structura de ansamblu a bibliotecii .....	220
8.2 Containere secvențiale .....	222
• Vectorul ca sablon .....	222
• Sablon pentru containerul secvențial <i>list</i> .....	225
• Sablonul <i>deque</i> .....	226
8.3 Containerele asociative .....	228
• Containerele asociative <i>set</i> , <i>multiset</i> .....	228
• Containerele asociative <i>map</i> , <i>multimap</i> .....	232
8.4 Containere adaptive .....	234
• Adaptorul <i>stack</i> .....	235
• Adaptorul <i>queue</i> .....	235
• Adaptorul <i>priority_queue</i> .....	236
8.5 Iteratori .....	238
• Iteratori predefiniți .....	243
8.6 Algoritmi .....	247
8.7 Aplicații .....	254
<b>9. IDENTIFICAREA TIPULUI LA MOMENTUL EXECUȚIEI (RTTI - Run-Time Type Identification)</b>	
9.1 Cadrul general .....	260
9.2 Operatorul <i>typeid</i> .....	261
9.3 Operatorul <i>dynamic_cast</i> .....	264
<b>10. FIȘIERUL CA OBIECT</b>	
10.1 Fișierul în acces secvențial și direct .....	270
10.2 Fișierul în acces indexat .....	275
• Cadrul conceptual .....	275
• Structura de index .....	276

Cuprins

• Implementarea accesului indexat la fișier .....	282
<b>Teste grilă</b>	
Întrebări .....	288
Răspunsuri .....	319
<b>Index</b> .....	329
<b>Bibliografie</b> .....	333



## **ABSTRACTIZAREA DATELOR. CONCEPTUL DE CLASĂ**

- Tipul abstract de date (Abstract Data Type – ADT)
- Conceptele de clasă și obiect
- Pointeri la obiecte. Masive de obiecte
- Clase incluse. Compunerea obiectelor
- Tipologia membrilor unei clase
- Transferul obiectelor în / din funcții
- Pointeri de date și funcții membre
- Clase și funcții prietene. Privilegii în sistemul de acces
- Modifierul *const* în contextul obiectelor

## 1.1 Tipul abstract de date (Abstract Data Type – ADT)

Primul lucru pe care-l facem când scriem un program care să ne șureze munca este să găsim un model ce simplifică realitatea, prin separarea detaliilor care interesează, de cele care nu afectează problema pe care o rezolvăm. Datele cu care se lucrează, operațiile care se execută și aşadar de specificul fiecărei probleme tratate. Spre exemplu, pentru un program de calculul automat al cotelor de întreținere, sunt importante anumite atrbute ale unei persoane (nume, salar, număr de persoane aflate în întreținere, suprafața locuibilă pe care o deține etc.), în timp ce pentru calculul salariului primit de o persoană sunt necesare informații precum: nume, categoria de încadrare, vechimea etc. Acest proces de grupare a datelor și metodelor de prelucrare specifice rezolvării unei probleme se mai numește **abstractizare**. Putem considera deci **tipul abstract persoana**, conținând atrbute și metode specifice, așa cum vorbim despre tipul *int* sau *float*.

Primul pas în această grupare l-au reprezentat structurile; ele permit să declarăm unor ansambluri eterogene de date ce erau manipulate unitar. Aplicarea operatorului de *typedef* unei structuri introducea practic un nou tip de dată, construit de utilizator, tip ce putea fi aplicat unei variabile la alocarea și inițializarea ei:

```
typedef struct
{
    char nume[20];
    int vîrstă;
    float salar;
} persoana;

persoana p1 = { "Popa Ion", 25, 85000.}, p2, *pp;
```

**Includerea în structură a unor pointeri**, către alte variabile sau zone dinamice conținând date de același tip, sau către diverse alte entități, a permis relativizarea controlului acțiunii la un context dat. Era, spre exemplu, mult mai greu să se gestioneze la nivel central toate meniurile și submeniurile cu care lucra un program, dar s-a dovedit a fi relativ simplu, dacă organizăm meniul ca o structură, iar în fiecare structură punem și pointeri care să ne indice unde ne putem îndrepta (stânga, dreapta, sus, jos) sau ce putem executa, când am atins un context oarecare, selectând o opțiune.

## 1.2 Concepțele de clasă și obiect

### Clase și obiecte

Dacă într-un astfel de tip de dată **introducem chiar și funcții** (gestionate flexibil prin pointeri) care să ne spună la ce prelucrări specifice sunt supuse datele structurii, obținem așa numitul **tip încapsulat** (*user built-in*) suplinit în limbajul C++ prin noțiunea de **clasă**. Ea implementează un concept abstract, indicând *natura datelor* ce-l compun, precum și *metodele* (funcțiile și operatorii specifici) ce-i pot fi aplicate.

Din punct de vedere sintactic o clasă în C++ se definește sub forma:

```
class nume_c
{
    // date + funcții membre
};
```

unde, *nume\_c* reprezintă numele dat de programator clasei respective.

Avantajele unei asemenea abordări sunt imense:

- specializarea prelucrărilor și adaptarea lor de la un caz la altul;
- localizarea facilă a erorilor;
- surprinderea într-un mod specific a tipologiei relațiilor dintre entități;
- gestionarea accesului prin "ascunderea" unor date, restricționarea folosirii unor funcții, obținerea unor informații doar prin funcții de acces specializate.
- perfectarea unui mod de comunicare între entități.

Vom încerca să exemplificăm concepțele propuse, pe obiecte concrete ale lumii reale, amintind o dată în plus, că proiectarea și programarea orientate obiect au valoare doar atunci când structura internă a obiectului și interfața acestuia cu celelalte obiecte își găsesc o reflectare în lumea reală, altfel ramân doar o teorie frumoasă! Tocmai aceste legături, pe care utilizatorul le cunoaște deja, facilitează înțelegerea programelor, manipularea și dezvoltarea lor ulterioară.

Funcțiile și datele unei clase pot fi grupate, din punct de vedere al dreptului de acces, în trei categorii, demarcate prin etichetele cheie *private*,

*public* și *protected* și care surprind drepturile de acces ale unui terț la respectivele resurse ale structurii:

**class** persoana

{

**private:**

    int virsta;

**protected:**

    float salariu;

**public:**

    char nume[20];

    void init(char \*n="Anonim", int v=0, float s=0.)  
        { strcpy(nume,n); virsta=v; salariu=s; }

    char \*spune\_nume();

    int spune\_virsta() { return virsta; }

};

    char \* persoana::spune\_nume() { return nume; }

Din definirea clasei *persoana* se pot observa câteva aspecte:

- datele membre se declară obișnuit ca și în cazul structurilor;
- funcțiile membre se pot defini direct în clasă fără a necesita o sintaxă specială și se numesc funcții *inline* (în cazul nostru *init()* și *spune\_virsta()*) sau se pot doar declara în clasă (îi scrie doar prototipul) și se definesc în afara ei folosindu-se sintaxa:

*tip nume\_c::nume\_m( lista\_p\_f ) {.....}*

unde:

- *tip* – reprezintă tipul funcției membre;
- *nume\_c* – este numele clasei;
- *nume\_m* – este numele metodei sau al funcției membru;
- *lista\_p\_f* – reprezintă lista parametrilor formali;

este cazul metodei *spune\_nume()*.

O dată definit acest tip, el poate fi folosit la declararea unor variabile de acest gen (fără a mai folosi cuvântul cheie *class*). Ca și la structuri, dacă numele de clasă lipsește, declaratiile de variabile se pot face numai odată cu definirea clasei, deoarece nu mai dispuneam de nume de şablon pentru descriere:

```
class
{
    // date și funcții membre
} v1,v[5], *pv;
```

Indiferent că se declară sau nu variabile, este obligatoriu, lipsa lui fiind poate cea mai răspândită eroare de sintaxă, antrenând neințelegeri majore pentru compilator.

O variabilă de tip clasă poate stoca un set complet al datelor clasei respective. Acest set concret de date reprezintă o **instantiere a clasei**, adică o manifestare concretă a clasei. În teoria programării orientată obiect, o astfel de instanță poartă numele de **obiect**.

Din programul următor:

```
#include <iostream.h>
void main()
{
    persoana p;
    p.init();
    cout<<p.spune_nume();
}
```

se desprind următoarele:

- *p* – este un obiect de tip *persoana*;
- apelul unei metode se face în legătură cu un obiect concret, forma generală este: *obiect.metoda(parametri\_de\_apel)*;
- expresia *cout<<p.spune\_nume()*; are drept consecință afișarea pe monitor a variabilei *nume*.

Se impune în acest moment să se facă o scurtă paranteză cu privire la modul de realizare a operațiilor de intrare / ieșire pe dispozitive standard (tastatură / monitor). Aceste operații le vom face folosind două obiecte predefinite (*cout* și *cin*) specializate în acest sens.

Afișarea pe monitor se face construind o expresie de genul: *cout<<var\_c*; unde, *var\_c* este o variabilă sau o constantă. Într-un literal secvențele de escape sunt recunoscute, efectul fiind identic ca și în cazul folosirii lor în funcția *printf()*. De exemplu secvența :

```
int a=17;
cout<<a;
cout<<"\n a="<<a;
```

va afișa:

```
17
a=17
```

Se observă că putem afișa mai multe date într-o singură expresie doar că ele trebuie legate cu ajutorul operatorului *<<*. În constanta literal secvența *\n* determină trecerea la linie nouă.

Pentru citirea de la tastatură a unei variabile se folosește obiectul *cin* în forma: *cin>>var;* unde, *var* este numele unei variabile. De exemplu secvența:

```
int a;
cin >> a;
```

determină citirea variabilei *a* de tip întreg; dacă se dorește a se citi într-o singură expresie mai multe variabile, ele se vor înlăntui cu ajutorul operatorului *>>*:

```
int a;
double t;
cin>>a>>t;
```

Folosirea obiectelor *cin* și *cout* implică includerea fișierului header *iostream.h*.

Mai multe detalii despre realizarea operațiilor de intrare / ieșire folosind aceste obiecte găsiți în Capitolul 4 - *Operații de intrare / ieșire orientate pe stream-uri*.

Revenind la exemplul nostru putem concluziona că un membru al clasei este calificat:

- prin **operatorul de rezoluție ::** în raport cu clasa căreia aparține;
- cu **. sau ->,** în raport cu instanțierile acesteia (obiect sau pointer la obiect).

Astfel, la definire, funcția *spune\_nume()* poartă în față calificativul *persoana::*, în timp ce la apel particularizează obiectul pentru care lucrează *p1.spune\_nume()*.

Domeniul **public** cuprinde datele și funcțiile membre ce sunt văzute prin intermediul obiectului și pot fi folosite sau apelate de oricare alte funcții din cadrul programului. Domeniul **private** cuprinde datele și funcțiile membre ce pot fi folosite doar de către celălalte funcții aparținând clasei respective.

În afara celor două domenii, în C++ poate fi delimitată și o zonă denumită **protected**, similară cu **private**, dar care dă drepturi de acces funcțiilor membre ale claselor derivate (obținute) din clasa respectivă. Așadar, **protected** este un atribut mai slab decât **private**, dar mai restrictiv decât **public**.

Față de teoria programării orientate obiect (POO), în C++ lucrurile sunt oarecum simplificate; două sunt deosebirile majore între cele două abordări:

- în teoria POO nu se admit date de domeniu *public*, toate fiind *private* ;
- obiectele sunt active ( primesc în permanență și tratează mesaje), adică cel puțin o funcție de interfață este în permanență activă pentru a sesiza mesajele adresate obiectului, implementarea fiind deci una de

tip multitasking. Din aceasta cauză, conceptul de clasă din C++ mai este denumit prin **ADT (Abstract Data Type)** pentru a-l distinge de obiect.

Eticheta **private** poate lipsi, subînțelegându-se ca private resursele ce nu apar în domeniul public. Când nu apare nici eticheta **public**, întreaga clasă este deci privată, accesul asigurându-se în exclusivitate prin interfață (funcțiile de acces menționate de clasă). Prin contrast cu clasele, din punct de vedere al accesului, structurile se consideră entități cu acces implicit public. În C++ domeniile de acces pot fi menționate explicit și pentru structuri, struct devenind un echivalent aproape perfect al lui *class*.

Când sunt menționate explicit toate domeniile de acces, ele se dau ușual în ordinea **private**, **public**, **protected**, dar practic pot apărea în orice succesiune, putându-se reveni de mai multe ori asupra unui domeniu:

**class persoana**

```
{
    private:
        int vîrstă;
    protected:
        char nume[20];
    public:
        void init(char *n="Anonim", int v=0, float s=0.)
            { strcpy(nume,n); vîrstă=v; salariu=s; }
        char *spune_nume();
        int spune_vîrstă() { return vîrstă; }
    private:
        float salariu;
};
```

În concluzie, putem spune că un tip de date se distinge printr-un mod de reprezentare și prin operațiile pe care le suportă; pe lîngă **tipurile predefinite** (de bază sau fundamentale) se pot introduce **tipuri de utilizator** folosind *struct*, *union* sau *class*, al căror mod de reprezentare și operații recunoscute se dau la definire.

Încapsularea permite:

- un transfer mai simplu al datelor în / din funcții (transferăm obiectul în ansamblu, nu elementele sale);
- controlul accesului la datele membre;
- scutește utilizatorul de cunoașterea unor detalii tehnice, percepția obiectului fiind una orientată spre client.

După rolul pe care-l joacă în cadrul clasei, funcțiile membre ale unei clase se împart în patru categorii:

- **constructori**, responsabili cu crearea obiectelor;
- **destructor**, responsabil cu distrugerea obiectelor și eliberarea memoriei ocupate de acestea;
- **funcții de acces**, care mediază legătura obiectului cu exteriorul;
- **metode**, funcții care introduc operațiile și prelucrările specifice obiectului.

În realitate un obiect conține numai datele specifice, compilatorul ținând un singur exemplar din funcțiile membre, pe care-l particularizează în momentul apelului, când va cunoaște și obiectul proprietar al funcției; această modalitate de lucru se explică prin faptul că o funcție membră are același cod executabil pentru toate obiectele clasei, dar execută acest cod pe datele specifice fiecărui obiect.

### Separarea interfeței de partea de implementare

Uzual, **partea de definire** a unei clase se pune într-un fișier de tip header, ce va fi preluat cu **#include** de programatorii ce dezvoltă aplicații folosind clase deja existente; prototipurile descrise în definirea clasei sunt suficiente în faza de compilare; **partea de implementare** unde se dă codul executabil al fiecărei funcții membre, se pune în fișiere de tip CPP și se poate prelua abia în faza de linkeditare, deja compilată.

Programatorii care folosesc clasa (numiți și clienți) nu au acces la detaliile clasei, cunoscute doar de proprietari, ci doar la **interfața publică** a clasei. Pe de altă parte, proprietarul clasei este obligat să respecte stabilitatea acestei interfețe, iar modificările pe care le fac nu trebuie să antreneze modificări în codul client.

Cum am văzut în exemplele anterioare, unele funcții reduse ca volum, frecvent apelate în lucru cu obiectele clasei, pot fi definite chiar în interiorul clasei, devenind implicit **inline**, adică apelul funcției este înlocuit pretutindeni cu codul întregii funcții, evitându-se pierderea timpului pentru transmiterea parametrilor de apel, pe stivă. Aceste funcții nu admit folosirea instrucțiunilor repetitive în cadrul lor. Deducem că funcțiile **inline** sunt în același timp cele mai stabile (nemodificabile). Dacă definirea lor **inline** urmărește numai considerente de eficiență a execuției și nu se bazează pe stabilitatea implementării, se recomandă plasarea codului în afara clasei, deci în zona de implementare și menționarea explicită a modificatorul **inline**.

După cum ați văzut în clasa *persoana*, metoda *spune\_virsta()* era implicit **inline** pentru că era definită în interiorul clasei; dacă dorim să o definim explicit **inline**, se va proceda astfel:

```
class persoana
{
    private:
        int virsta;
        // .....
    public:
        // .....
        int spune_virsta();
};

inline int persoana::spune_virsta() { return virsta; }
```

### Constructori și destructor

După cum se observă, funcțiile ce apar frecvent în structură au fie rolul de inițializare a datelor membre din obiect, fie rolul de extragere a datelor și comunicare a lor prin interfață cu exteriorul clasei. Pentru inițializare se definesc metode specializate în acest sens, numite **constructori**, care au același nume cu numele clasei. Iată principalele motivații pentru care se utilizează constructori :

- complexitatea structurii obiectelor dată de existența variabilelor și funcțiilor, de existența secțiunilor privată, publică și protejată, de posibilitatea descrierii intercalate a metodelor și a datelor, face dificilă inițializarea directă a obiectelor după modelul structurilor;
- există situații în care doar unele date membre trebuie inițializate, altele sunt încărcate în urma apelării unor metode;
- datele de obicei sunt declarate în secțiunea privată și deci nu pot fi accesate direct din exterior ci prin intermediul metodelor;
- degrevarea programatorului de a reține și apela explicit metode care realizează inițializarea obiectelor; de exemplu noi am definit în clasa *persoana* metoda *init()* pentru a face inițializarea obiectelor.

Spre deosebire de celelalte funcții membre, constructorul nu are tip (dat de variabila returnată). O clasă poate menționa mai mulți constructori, prin supraîncărcare, folosirea uneia dintre ei la declararea unei variabile de clasă dată, fiind dedusă în funcție de numărul și tipul parametrilor de apel.

Vom redefini clasa *persoana* pentru care vom defini și doi constructori astfel:

```
class persoana
{
    int virsta;
    float salariu;
    char nume[20];
public:
    persoana() { strcpy(nume,"Anonim"); virsta=0; salariu=0; }
    persoana(char *n, int v, float s)
        { strcpy(nume,n); virsta=v; salariu=s; }
    char *spune_nume() { return nume; }
    int spune_virsta() { return virsta; }
};
```

Se observă că s-au definit doi constructori :

- unul care nu are nici un parametru (*persoana()*) și inițializează datele membre cu valori constante; un astfel de constructor (fără parametri) se mai numește și **constructor implicit**;
- celălalt constructor primește trei parametri și are ca scop inițializarea datelor membre din variabile elementare.

Constructorii astfel definiți în clasa *persoana* se apelează astfel:

```
persoana p1, p2("Gigi",45,89999);
```

Simpla definire a unui obiect (*p1*) determină apelul constructorului implicit, în cazul definirii obiectului *p2*, prin parametri de apel s-a sugerat care constructor să fie apelat.

Trebuie de subliniat faptul că un constructor (cel implicit) este definit automat de compilator, dacă nu se definește explicit nici un constructor, și nu face nimic, doar se folosește pentru a putea genera obiecte ale clasei respective. A se vedea clasa *persoana* care inițial a fost definită fără a-i explicita vreun constructor; totuși pentru inițializare s-a definit metoda *init()*.

Cei doi constructori ai clasei *persoana* pot fi combinați în unul singur care primește parametri cu valori implicate:

```
persoana(char *n="Anonim", int v=0, float s=0) : virsta(v), salariu(s)
    { strcpy(nume,n); }
```

Acest constructor are o **listă de inițializatori** (cea separată prin : de prototipul funcției), prin care primesc valori implicate de inițializare o parte din datele clasei; construcția *virsta(v)* este echivalentă funcțional cu *virsta=v*; scrisă în blocul constructorului. În capitolul 3 privind derivarea claselor, vom vedea că

forma cu : anunță o conlucrare între constructori înruditi; aşadar sensul celor de mai sus indică o colaborare între constructorul *persoana* și *constructorii* tipurilor de bază, invocați prin tipurile *int* și *float* ai variabilelor *virsta* și *salariu*. De exemplu o declarație de forma *int a=5*; este echivalentă cu *int a(5)*; care sugerează apelul *constructorului* clasei *int*.

Indiferent de varianta aleasă pentru definirea constructorilor, programul:

```
void main()
{
    persoana p1, p2("Gigi",45,89999);
    cout<<"\n"<<p1.spune_nume()<<p1.spune_virsta();
    cout<<"\n"<<p2.spune_nume()<<p2.spune_virsta();
```

are ca rezultat al execuției:

*Anonim 0*  
*Gigi 45*

Este util a se defini mai mulți constructori pentru o clasă deoarece modurile de inițializare a datelor membre sunt diverse :

- inițializarea membrilor cu constante;
- inițializarea din date elementare;
- inițializare prin citire de la tastatură;
- inițializare prin citire din fișier;
- inițializare din datele unui alt obiect.

Pentru un obiect este selectat totdeauna un singur constructor (în funcție de parametri din definire) care este apelat o singură dată; programatorul nu mai poate ulterior apela alt constructor pentru respectivul obiect.

Tipul introdus printr-o **clasă poate avea și constante**, numai că, spre deosebire de tipurile de bază, recunoscute implicit de compilator din construcție, o constantă asociată unei clase poartă numele clasei asociate. Pentru clasa *persoana* o constantă apare sub forma *persoana(" Vasile Liviu", 30, 89000.)* și poate fi folosită la inițializarea în bloc a unei variabile de clasă *persoana*, astfel:

```
persoana p1 = persoana(" Vasile Liviu", 30, 89000.);
```

ce diferă fundamental de declarația:

```
persoana p1(" Vasile Liviu", 30, 89000.);
```

în care constructorul se apelează implicit, prin definirea obiectului. În prima variantă, denumirea clasei apare de două ori, o dată pentru obiect și o dată

pentru constanta de inițializare. Așadar, constructorul poate fi interpretat la modul general, ca transformând tipuri în alte tipuri noi (**char[ ]**, **int** și **float** în **persoana**).

Faptul că este implicat tot constructorul și în declararea constantei, o dovedește și ordinea constantelor de inițializare, care corespunde parametrilor de apel al constructorului și nu corespunde ordinii câmpurilor din definiția clasei. Încărcarea valorilor se poate face chiar combinat, constructorul preluând constante din tipurile de bază sau valori returnate de funcțiile de acces la variabilele unor clase de tip similar:

```
persoana p1("Popa Ion", 25, 85000.);  
persoana p2 = p1, p3;  
p3 = persoana( p1.spune_nume(), p2.spune_vîrstă, 100000.);
```

Pentru a putea fi apelat în orice context al programului, constructorul trebuie declarat în domeniul public al clasei pentru a se autoriza accesul în folosirea lui, ca metodă generală. De asemenea, constructorul este singura funcție membră care este văzută în afara clasei fără a specifica un obiect sau pointer la un obiect al clasei (formele **p.f()** sau **pp->f()**); acest lucru se asigură prin faptul că el poartă numele clasei și deci este și **funcție și tip**, în același timp.

Ne putem pune întrebarea dacă pot exista constructori **private** și la astfel apelați din orice funcție a programului, care are nevoie de obiectele unei clase. După cum se poate constata ușor, există destul de multe situații în care sunt necesare obiecte temporare, pentru o gestiune internă; la construirea acestora poate fi folosită o supraîncărcare de constructor, chiar din domeniul private, cu parametrii care să o individualizeze de celelalte supraîncărcări. Un astfel de constructor nu poate fi invocat însă decât de funcțiile membre ale clasei, adică tocmai de funcțiile care folosesc obiecte temporare.

Ca și în cazul tipurilor de bază, tipul introdus printr-o clasă suportă **constante cu nume**, în vederea utilizării lor ulterioare:

```
const persoana seful("Anghel Victor", 49, 12500.);  
const persoana fictiv;  
// ...  
persoana p1=fictiv;
```

S-au apelat forme distincte ale constructorului, cu implicații asupra datelor generate, în fiecare caz în parte.

Pentru a rezolva cerințe de genul inițializării obiectelor din altele de același tip, deja existente (**persoana ob1 = ob2;**) sau pentru transmiterea obiectelor prin valoare în / din funcții (**f(ob)**; sau **return ob;**), fiecare clasă dispune și de un **constructor de copiere** implicit, pus de compilator, care realizează copierea obiectului sursă în destinație bit cu bit, sau explicit, dat de utilizator. Un constructor de copiere se recunoaște prin faptul că operează cu două obiecte: unul recunoscut implicit, printr-un pointer (datorită faptului că fiecare funcție nestatică a clasei, deci și constructorul are un obiect proprietar) și unul explicit, primit ca parametru prin referință:

```
class cls  
{  
    // .....  
public:  
    cls(cls &);  
};  
cls::cls(cls &x) /* definire constructor de copiere */
```

Constructorul de copiere poate face o copie *controlată* a datelor preluate dintr-un obiect existent. Astfel, ne putem imagina că la crearea unui nou cont bancar pentru o persoană care are deja un alt tip de cont, putem prelua informații precum: numele titularului și datele de identificare, dar inițializăm noul cont și pe baza altor informații cum ar fi :

- data creării contului, preluând data sistemului de operare;
- parola, cerută de tastatură etc.

Nu vor fi copiate celelalte date precum tipul contului (care este altul, pentru noul cont), soldul etc.

Dacă programatorul **preia controlul copierii prin constructor**, atunci trebuie să furnizeze valori sau să initializeze toate câmpurile. Programul de mai jos, în care s-a dorit marcarea trecerii prin constructorul de copiere, omite copierile propriu-zise, lăsând cîmpul **x** neinițializat; în acest fel, tocmai obiectele pentru care s-a dorit inițializarea cu un obiect nestandard (cu valoarea lui **x** alta decât cea stabilită implicit), rămân neinițializate!

```
#include <iostream.h>  
class cls  
{  
public:  
    int x;  
    cls(int a = 0) : x(a)  
    { cout << "\n Constructor de clasa"; }
```

```

cls(cls & c) { cout << "\n Constructor de copiere"; }

void main() { cls c1(1), c2 = c1; cout << "\n " << c2.x; }

```

Complementul acțiunii constructorului este îndeplinit de o altă funcție implicită sau explicită la nivelul unei clase, **destructorul**. Când este declarat explicit în cadrul clasei, destructorul poartă numele clasei, precedat de semnul `~: ~persoana();` Spre deosebire de constructor, el este unic și nu are niciodată parametri de apel. Destructorul este apelat implicit, la ieșirea din blocul în care a fost declarat obiectul definit de o clasă.

De cele mai multe ori, destructorul nu conține cod executabil scris de programator și deci nu este nevoie să-l menționăm în structura clasei. Când este nevoie (spre exemplu, prin constructor se fac și alocări dinamice de către utilizator, iar destructorul trebuie să elibereze prin cod explicit acest tip de memorie) destructorul va conține cod.

Ca funcții, constructorii și destructorul nu pot fi moșteniți de alte clase, dar pot fi apelați de clasele derivate. Când lipsesc, ei vor fi generați automat de către compilator, în zona *public*.

### Obiecte cu extensii în memoria dinamică

Din considerante legate de folosirea eficientă a memoriei este posibilă definirea unor clase ale căror elemente să fie dimensionate dinamic. Spre exemplu, apare oarecum justificat să includem în clasa persoana un membru `char *pnume` în locul lui `char nume[20]` care ar ocupa totdeauna 20 bajți; `pnume` va ocupa doar 2 bajți și va pointa spre o zonă alocată dinamic în funcție de lungimea numelui fiecărei persoane. Clasele cu membri alocați dinamic se mai numesc și **clase cu membri pointeri**.

Este important de observat că pentru obiectele ce conțin membri rezervați în memoria dinamică (spre exemplu, clasa stivă sau o clasă persoana în care apare `char *pnume`), programatorul trebuie să furnizeze constructori, care să aloce explicit zona dinamică pointată de `pnume` și un destructor care să elibereze explicit această memorie.

Practic obiectul *persoana* va ocupa în acest caz două categorii de memorie:

- una alocată și eliberată implicit la definirea obiectului, respectiv la terminarea duratei lui de viață;
- alta alocată și eliberată explicit, prin codul pus de programator în constructor și destructor.

Acum ne vom opri mai mult asupra a încă două funcții existente la nivelul clasei, care sunt influențate puternic de alocarea unor extensii ale obiectelor, în memoria dinamică: constructorul de copiere și operatorul de atribuire `operator=( )`. Înțelegerea lor corectă este cu atât mai necesară cu cât cele două funcții sunt invocate de multe ori implicit, programatorul doar intuind folosirea lor. Prezente în orice clasă (sunt puse automat de compilator dacă programatorul omite scrierea lor), cele două funcții membre au aparent roluri apropiate, de multe ori confundate. Ambele fac copieri de obiecte, dar deosebirile sunt esențiale: în timp ce **constructorul de copiere copiază obiectul sursă într-o zonă neinitializată** în care își construiește un obiect nou, **operatorul de atribuire lucrează cu două obiecte deja existente**, sursă și destinație, având doar sarcina de copiere a informațiilor dintr-o zonă în alta.

Constructorul de copiere este invocat la:

- crearea de obiectelor cu inițializare, pornind de la un obiect care există (cazul **persoana p3 = p1;**);
- apelul unei funcții care lucrează cu obiecte transferate prin valoare, când este nevoie de crearea unei copii a obiectului pe stivă (cazul **f(p1);**);
- returnarea dintr-o funcție a unui obiect, prin valoare (**return p1;**).

Să revenim însă la **constructorul de copiere și operatorul de atribuire în situația obiectelor cu extensii în memoria dinamică**.

Dacă programatorul nu definește un constructor de copiere care să aloce explicit o zonă dinamică și pentru extensia noului obiect și să copieze și extensiile din una în alta, cel implicit pus de compilator va copia doar câmpurile membre (față vizibilă a obiectului), fără eventualele extensii ale acestora, prin trimiterile la memoria dinamică; în acest caz două sau mai multe obiecte vor partaja aceeași zonă dinamică prin pointerii membri pe care i-a duplicat prin copiere. Acest lucru este eficient din punctul de vedere al economiei de memorie, dar poate conduce la situații anormale; spre exemplu, când unul din obiecte este șters, el șterge și zona partajată cu alte obiecte, lasându-le pe acestea fără suportul fizic de păstrare al valorilor efective ale unor membri din clasă (figura 1.1).

Vom exemplifica acest caz definind clasa *persoana* în forma:

```

class persoana
{
    int vîrstă;
public:
    char *pnume;
    persoana(char *n, int v):vîrstă(v)
    {
        pnume=new char[strlen("Anonim")+1];
    }
}

```

```

    strcpy(pnume,n);
}
int spune_virsta( ) { return virsta; }
};

În urma rulării programului:

```

```

void main()
{
    persoana p1("Vasilache",45);
    persoana p2=p1; // apel constructor de copiere
    cout<<"\n"<<p1.pnume<<" are "<<p1.spune_virsta()<<" ani";
    cout<<"\n"<<p2.pnume<<" are "<<p2.spune_virsta()<<" ani";
    strcpy(p2.pnume,"Gigi");
    cout<<"\n"<<p1.pnume<<" are "<<p1.spune_virsta()<<" ani";
    cout<<"\n"<<p2.pnume<<" are "<<p2.spune_virsta()<<" ani";
}

```

se va afișa:

```

Vasilache are 45 ani
Vasilache are 45 ani
Gigi are 45 ani
Gigi are 45 ani

```

Se observă atât prin rularea programului cât și din figura 1.1 că ambele obiecte (*p1* și *p2*) lucrează pe aceeași zonă de memorie alocată câmpului *pnume* la construirea obiectului *p1*, deoarece la construirea obiectului *p2* s-a apelat constructorul implicit de copiere care a copiat membrul *pnume* (adresa) dar nu a făcut și alocarea memoriei.

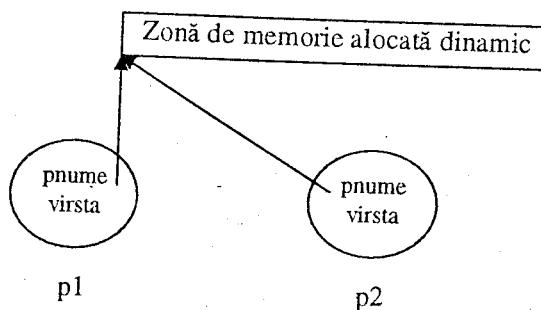


Fig. 1.1 Obiecte cu zonă de memorie comună

Însă dacă se definește un constructor de copiere în clasa *persoana* care să aloce explicit memorie, după care să facă copierea conținutului zonei de memorie a obiectului sursă la destinație:

```

class persoana
{
    int virsta;
public:
    char *pnume;
    persoana(char *n, int v) : virsta(v)
    {
        pnume=new char[strlen("Anonim")+1];
        strcpy(pnume,n);
    }
    persoana(persoana& pers) : virsta(pers.virsta)
    {
        pnume=new char[strlen(pers.pnume)+1];
        strcpy(pnume,pers.pnume);
    }
    int spune_virsta( ) { return virsta; }
};

```

rezultatul afișat prin rularea aceluiși program este:

```

Vasilache are 45 ani
Vasilache are 45 ani
Vasilache are 45 ani
Gigi are 45 ani

```

Dacă este nevoie ca o persoană să-și modifice lungimea șirului alocat (spre exemplu, la schimbarea numelui prin căsătorie), în obiect se va introduce o funcție membru *update()*, care actualizează o instanță. Ea va face delete pe vechiul pointer spre nume, va aloca prin new o zonă adecvată noii lungimi a numelui, actualizând totodată pointerul din obiect cu noua adresă și după aceea va încărca noul nume.

Similar stau lucrurile și la un apel *f(p)*, când se construiește pe stivă un obiect temporar care va partaja aceeași extensie cu obiectul transferat ca parametru actual. Se ajunge astfel la anomalie ca eventualele prelucrări ce afectează copia (partea de extensie a acesteia) să modifice în același timp și originalul, deși transferul s-a făcut prin valoare !

Anomalii și mai grave apar atunci când există un destructor scris de programator, care cum era și firesc, făcea și dezalocarea zonei dinamice pointate de un membru al clasei. La ieșirea dintr-o funcție care returnează un obiect prin valoare, de pe stivă va fi șters obiectul returnat, invocându-se destructorul de clasă. Cum obiectul partaja zonă dinamică cu un alt obiect, ștergerea lasă acest obiect cu un pointer invalid, care adreseză o zonă de memorie deja dezalocată. Din nefericire, eroarea nu este sesizată acum, ci abia

când obiectul rămas cu pointerul invalid este folosit sau șters, lucru care îngreunează teribil depanarea.

Situată stă aproximativ la fel în cazul operatorului de atribuire; nescriind o versiune proprie pentru **operator=**, care să dezaloce extensia obiectului destinație (care se distrugă prin copiere), și să aloce o extensie proprie în concordanță cu dimensiunea extensiei obiectului sursă, programatorul acceptă tacit ca obiectele sursă și destinație să partajeze aceeași zonă adresată de un pointer membru, duplicat prin copiere. În plus, obiectul destinație fiind suprascris prin copiere bait de bait, se suprascrie și pointerul membru, pierzându-se astfel legătura cu propria zonă dinamică și generând așanumitele pierderi de memorie (**memory leaks**). De aici în colo, anomaliiile se tîn lanț:

- încercarea de dezalocare repetată a aceleiași zone;
- modificarea membrilor unui obiect, fără ca acest lucru să fie dorit (dorim modificarea componentelor obiectului *x*, dar implicit le modificăm și pe cele ale lui *y*, adică elementele aflate în zona comună celor două obiecte).

Nu exemplificăm aici anomaliiile pentru operatorul de atribuire(=) pentru că subiectul capitolului 2 se referă la supraîncărcarea operatorilor.

În concluzie, pentru un obiect cu un membru pointer spre o zonă alocată dinamic programatorul va furniza:

- constructori adecvați de clasă, care să aloce extensia și să încarce pointerul prin care o gestionează;
- constructor de copiere care să aloce extensia pentru noul obiect, să încarce pointerul prin care o gestionează și să inițializeze extensia copiind extensia din obiectul de initializare.
- destrutor de clasă, care să dezaloce extensia adresată prin pointerul membru;
- operator de atribuire, care să dezaloce extensia adresată prin pointerul membru al obiectului destinație, să realoce și încarce conform dimensiunii și conținutului celei din obiectul sursă.

### **Pointerul *this***

După cum ați putut observa metodele unei clase invocau datele membre din clasă fără ca acestea să fie transmise ca parametri în funcții; o explicație de suprafață se referă la faptul că și datele și funcțiile aparțin clasei respective (sunt încapsulate într-o construcție unitară). Pe de altă parte

la scrierea funcțiilor membre, se face deseori apel la datele clasei, fără a se menționa la care set de date (obiect) se referă prelucrările.

Privind lucrurile în profunzime, la apel, o funcție membru va primi pe lângă parametri de apel expliciti și un parametru implicit care este tocmai adresa obiectului vizat, adică a obiectului asupra căruia se efectuează prelucrările. Această adresă, deși transparentă pentru utilizator, există realmente memorată într-un pointer numit **this** (cuvânt cheie). El poate fi și explicit folosit când se dorește folosirea adresei obiectului. Ca exemplu vom defini metoda **adresa()** în clasa *persoana* care va afișa pointerul *this*:

```
void persoana::adresa()
    { cout<<"\n Adresa fizica a obiectului este:"<<this; }
```

Urmărind codul asembler generat de un apel de funcție membră, putem vedea că *this* este primul parametru de apel, deși nu apare explicit în sursa C++.

Datele membre sunt referite direct când se definesc funcții membre, de exemplu metoda **spune\_virsta()** referă direct membrul *virsta*:

```
int persoana::spune_virsta() { return virsta; }
```

dar de fapt referirea se face prin intermediul pointerului *this*, ceea ce vom face și noi explicit definind metoda **spune\_virsta()** după cum urmează:

```
int persoana::spune_virsta() { return this->virsta; }
```

### **Funcțiile de acces**

Teoria recomandă trecerea în domeniul *private* a cât mai multor date și funcții membre; acest lucru nu înseamnă că viitorii beneficiari ai clasei nu au acces la aceste resurse, ci că accesul este controlat prin funcții specializate, prevenind pierderea intergrității datelor și simplifică procesul de depanare a programelor.

Consultând multe exemple din domeniul programării orientate obiect observăm că majoritatea funcțiilor de acces propuse nu fac decât să returneze sau să modifice valoarea unei date membre private. Ele pentru a se distinge de celealte funcții membre au un prefix sugestiv și anume **get\_xxx()** dacă returnează o valoare sau **set\_xxx()** dacă o modifică.

Am definit și noi în clasa *persoana* funcții de acces care returnau diferite valori, cum ar fi: **spune\_virsta()** sau **spune\_nume()**; se observă că am optat pentru prefixul **spune\_xxx()**. Definim în continuare și o funcție de acces care modifică vârsta unei persoane:

```
void persoana::set_virsta(int k) { virsta=k; }
```

Care este atunci sensul existenței funcțiilor de acces ? Ce rost are să punem o variabilă pe domeniul privat, ca mai apoi tot noi să o furnizăm în afară prin intermediul unei funcții de acces ? Răspunsul este puțin mai nuanțat.

În primul rând, la o privire mai atentă, vedem că și în acest fel accesul este totuși restricționat, el fiind de tip **read only**, adică o funcție de tip `get_xxx()` nu modifică o dată privată ci o comunică în exterior.

În al doilea rând, trebuie să percepem o funcție de acces ca pe o posibilitate introdusă de creatorul clasei de a interveni ulterior asupra accesului acordat clienților. Oricând funcția de acces poate fi rescrisă astfel încât să returneze o valoare prelucrată în locul celei reale.

Un exemplu concluzient de prelucrare îl poate reprezenta salariul unei persoane, la nivel de individ, el are caracter privat, dar la nivelul unei colectivități statistice el trebuie cunoscut, fiind necesar unor programe economice sau unor analize statistice. Ne putem imagina acces controlat la această informație privată, în sensul că pe dispozitivul de mesaje de eroare (*cerr*) este afișat mesajul privind caracterul privat al salariului, iar funcția returnează valoarea medie a salariului pentru toate persoanele cu aceeași profesie ca a persoanei vizate.

### 1.3 Pointeri la obiecte. Masive de obiecte

Pointerul la obiect este prin analogie cu ceea ce știm despre un pointer în general, o variabilă care conține adresa unui obiect. Astfel se pot manipula și obiecte prin intermediul adreselor lor. Luând în considerare structura complexă a obiectelor pot fi subliniate anumite particularități în folosirea pointerilor la obiecte.

Definirea unui obiect declanșă și execuția unui constructor al clasei; declarația: `persoana p;` determină apelul constructorului implicit al clasei *persoana*, pe când declararea unui pointer la obiect nu declanșează execuția unui constructor al clasei, deoarece se alocă memorie doar pentru o variabilă capabilă să stocheze o adresă.

Încărcarea unui pointer se poate face pornind de la adresa unui obiect deja definit:

```
persoana p, *pp;
```

```
pp=&p;
```

iar accesarea membrilor (publici) se face cu ajutorul operatorului `>` ca la structuri:

- `pp->nume` – accesarea membrului *nume*;
- `pp->spune_virsta()` – apelul metodei *spune\_virsta()*.

Alocarea dinamică a memoriei pentru obiecte se face utilizând operatorul *new* construit special pentru varianta obiectuală a limbajului C (C++). Acest operator când se aplică în legătură cu tipul *class* pe lângă alocarea de memorie, determină și apelul unui constructor; astfel expresia:

```
persoana *pp = new persoana;
```

alocă zonă de memorie capabilă să stocheze un obiect *persoana*; adresa zonei este stocată în variabila pointer *pp*, dar în plus, se execută și constructorul implicit al clasei *persoana*.

Se poate apela și un constructor explicit la momentul alocării, ca în expresia:

```
persoana *pp = new persoana("Liviu", 9, 250000.);
```

Dezalocarea se face folosind operatorul *delete*, care apelează automat și destructorul clasei; de exemplu *delete pp*;

Funcțiile de bibliotecă *malloc()* și *free()*, pentru alocare / dezalocare de memorie pot fi folosite pentru obiecte doar că nu determină apelul constructorului / destructorului clasei respective.

Faptul că o clasă desemnează de fapt un tip înglobat, definit de utilizator, o dovedește și posibilitatea declarării de masive din tipul respectiv. Inițializarea elementelor masivului nu diferă cu nimic de mecanismul inițializării masivelor din tipurile de bază, constantele aparținând de data aceasta nouului tip:

```
persoana grup[ ]= {
```

```
persoana("Adam D. ", 35, 75000.),
persoana("Bucur I. ", 25, 95000.),
persoana(),
persoana("Costea A.", 29, 97000.)
```

```
};
```

Dimensiunea masivului este dedusă automat de compilator, din lista de inițializatori, sau poate fi dată explicit.

Constructorul clasei va fi apelat repetat, pentru fiecare element în parte.

Așa cum aminteam, vom denumi prin instanțiere, valorile unui element al acestui vector, adică un set de valori pentru o clasă sau ceea ce denumeam prin *realizarea unei entități*, în cazul bazelor de date.

O funcție de acces la elementele masivului, va trebui să fie corect specificată, precizându-se în acest caz și indexul elementului:

```
cout << grup[1].spune_virsta();
```

Când tipul obiect introdus prin definirea unei clase suportă ca instanțe doar o mulțime limitată de valori, este foarte flexibil și sugestiv să se lucreze cu enumerări de obiecte ale clasei respective:

```
enum echipa { sef, inginer, zidar, fierar_betonist } e1;
```

Elementele enumerării sunt întregi (mai corect, pot fi convertite la întregi), mediind localizarea datelor într-un vector de persoane:

```
cout << "\n Inginerul este " << grup[ inginer ].nume;
```

#### 1.4 Clase incluse. Componerea obiectelor

La o privire superficială, componerea claselor și implicit lucrul cu obiecte conținute în alte obiecte, nu ridică probleme deosebite. Putem oricând defini:

```
class copil
{ /* date și funcții specifice */ };
```

și apoi cita în structura clasei persoana și un membru *copil c[5]*; prin aceasta înzestrăm fiecare obiect persoană cu un vector de maxim cinci obiecte, de tip copil.

Puteam să declarăția clasei *copil* chiar în interiorul clasei *persoana*, dar declarăția rămâne cunoscută numai la nivelul acestei clase, neputând produce deci „instantieri” copil, în afara părintilor (obiect persoana), decât menționând de fiecare dată și clasa persoana, ca și cum clasa interioară se numește *persoana:: copil*.

```
#include <iostream.h>
#include <string.h>
class persoana
{
private: int virsta;
```

```
public:
char nume[20];
float salariu;
class copil
{
public:
char prenume[10], virsta; persoana *parinte;
copil (persoana *parent=NULL, char pren[10] = "Puiu", int v=0):
virsta(v), parinte(parent)
{ strcpy(prenume,pren); cout << "\n Constructor copil"; }
char * spune_nume();
copil (copil &c){ cout << "\n Copy constructor copil"; }
} c[5];

persoana(char n[20] = "Anonim ", int v=0, float s=0.):
virsta(v), salariu(s) { strcpy(nume,n); }

int spune_virsta() { return virsta; }

char * persoana::copil::spune_nume()
{
static char nume_pren[50];
strcpy(nume_pren, strtok ( parinte->nume, " ") );
strcat(nume_pren, " ");
strcat(nume_pren, prenume);
return nume_pren;
}

void f(persoana p) {}

void main()
{
persoana p1("Popescu Ion"), p2("DOI ", 22, 222222.);
p1.c[0]=persoana::copil(&p1, "Viorel", 10);
cout << "\n Ma cheagă " << p1.c[0].spune_nume();
f(p1); // trecere prin ambii constructori de copiere
}
```

În general, constructorii de clasă exterioară (*persoana*) primesc ca parametri și datele necesare clasei interioare (*copil*), dar ele vor fi pasate constructorilor de clasă *copil*.

În cazul nostru putem admite că la instantierea clasei *persoana* nu dispunem de datele tuturor copiilor, astfel încât ulterior vom putea invoca explicit constructorul *copil* pentru a inițializa cu constante, elementele vectorului *c[ ]* din obiectele părinte deja existente, sub forma:

**p1.c[0] = copil( &p1, "Viorel", 10);**

Alteori, constructorii clasei interioare primesc în intrare și **adresa unui obiect părinte**, adică de tipul clasei exterioare. Acesta este singurul mod de a lega obiectele interioare, atunci când au fost definite în afara clasei care le includea de obicei, adică s-a folosit clasa *persoana::copil*, pentru a defini obiecte *copil*, și în afara clasei *persoana*.

La o privire mai atentă sesizăm că prin includerea unei clase în altă clasă, implementăm și o **relație ierarhică între două clase diferite**, respectiv între obiectele acestora. De multe ori, în tehnologia **client-server**, clasa exterioară joacă rol de clasă client, iar cea interioară de clasă server; în acest sens o clasă fișier poate îngloba o clasă index, care îi permite autoindexarea la scrierea obiectelor în fișier; clasele pentru structuri autoreferite (listă, stivă, coadă etc.) conțin clase iterator incluse, care să permită descrierea unitară a algoritmilor ce presupun traversarea structurilor.

**Includerea claselor nu presupune și acordarea unor drepturi de acces speciale clasei exterioare.** Spre exemplu, încercarea unei funcții din clasa persoana de a accesa zone private din clasa copil ar eșua încă din faza de compilare. Restricția este valabilă și reciproc: o funcție din clasa *copil* nu poate accesa date private din clasa *persoana*. Acest lucru face ca și conlucrarea între clase să se realizeze totdeauna numai prin intermediul funcțiilor membre specialize ale fiecărei clase. În exemplul nostru, numele unui copil n-ar putea fi recompus din prenumele său atașat numelui tatălui, dacă numele n-ar apărea ca public.

În schimb, clasa exterioară poate introduce noi restricții de acces asupra informațiilor obținute via funcțiile sale, chiar dacă informațiile se refereau la clasa inclusă, iar acolo erau de natură publică. Această control al accesului se practică în mod curent:

- declarând constante informațiile provenind din clasa inclusă, returnate de funcții ale clasei exterioare (acces de tip *read-only*);
- introducând o cooperare controlată explicit prin program, între constructorii celor două clase, sau în general între funcțiile celor două clase.

Merită de observat că la transferul prin valoare, spre exemplu, constructorul implicit de copiere pus de compilator crează și inițializează pe stivă un obiect persoana temporar; pentru a realiza acest lucru, este invocat tot implicit, de cinci ori, constructorul de copiere al clasei copil, corespunzător dimensiunii vectorului de obiecte incluse.

Pentru a ușura depanarea programelor și pentru a beneficia de verificări competente făcute de compilator este bine ca funcțiile ce lucrează cu obiecte sără a le modifica, să fie marcate cu *const*. Spre exemplu un constructor de copiere este mai bine declarat sub forma:

**persoana (*const* persoana &p );**

atenționând că obiectul sursă nu este alterat prin copiere, pentru inițializarea altui obiect.

## 1.5 Tipologia membrilor unei clase

### Clase cu membri constanți

Presupunem clasa:

```
class muncitor
{
public:
    const double tarif;
    int nr_piese;
    muncitor(double t=0.0, int np=0) : tarif(t) { nr_piese = np; }
};
```

Unul din membrii clasei este constant (tariful odată stabilit, nu mai poate fi renegotiat). Atributul de *const* durează de la terminarea construirii obiectului pînă la începerea distrugerii lui. În faza de construire pot fi inițializați unii membri prin atribuire, dar nu cei declarați cu specificatorul *const*. Constructorul de clasă va avea deci obligatoriu în lista de inițializatori și un inițializator pentru membrul constant.

În exemplul nostru, programatorul are de ales dacă va folosi inițializator sau atribuire pentru *nr\_piese*, dar inițializatorul *tarif(t)* este obligatoriu.

### Specificatorul *static* aplicat membrilor unei clase

Din punct de vedere al clasei de memorie, datele și funcțiile unei clase pot fi automate, statice sau alocate explicit în memoria dinamică.

Specificatorul **static** dobândește în contextul claselor semnificații particulare, surprinzând atribute specifice clasei în ansamblu. Astfel, **datele statice nu se regăsesc în fiecare set de valori ale clasei** (obiecte), ci într-un singur exemplar, pentru toate obiectele clasei. Datele ce apar în toate obiectele se alocă de către constructor, dar un membru static nu face parte din nici un obiect, deci nu se aplocă prin constructor. Din aceasta cauză, la definirea clasei, o dată statică nu se consideră definită, ci doar declarată, urmând a avea o definiție externă clasei. Legătura cu declarația din interiorul clasei se face prin operatorul de rezoluție, variabila purtând atât tipul ei de bază, cât și cel al clasei căreia îi aparține:

```
class persoana
{
    ...
    static int total_pers;
    ...
};

int persoana::total_pers = 0;
```

Variabila **total\_pers** va fi unică pentru toate persoanele. La definire nu se mai reia specificatorul *static* care ar avea în afara clasei altă semnificație; odată cu definirea s-a făcut și o inițializare, care după cum se observă, surclasază accesul privat, ceea ce nu s-ar permite în cazul unei atribuirii ulterioare, de forma:

```
persoana::total_pers = 0;
```

Variabila statică fiind unică pentru toate obiectele, nu necesită în calificare precizarea obiectului, ci doar a clasei, în general.

În ceea ce privesc **funcțiile de clasă static**, ele efectuează prelucrări ce pot să nu fie individualizate pe obiecte, ci să se refere la nivelul clasei. Din această cauză, funcțiile statice nu aparțin unui obiect anume și deci nu **beneficiază de referință implicită a obiectului asociat** (pointerul *this*). Când operează pe o dată nestatică a clasei, obiectul trebuie transmis explicit ca parametru funcției membre statice, în timp ce cu datele membre statice lucrează în mod direct.

Înțeținerea variabilei **total\_pers** din exemplul nostru, cade în sarcina constructorilor și destructorului, care incrementează sau decrementează această variabilă. Tot statică este și o variabilă ce contorizează numărul total de bărbați (*total\_b*), din clasa respectivă. Funcția de numărare a persoanelor de fiecare persoană în parte. Deoarece **numara\_b()**, a fost declarată funcție

statică, ea are nevoie de adresa fiecărei persoane, pentru a actualiza variabila statică *total\_b*.

```
#include <iostream.h>

class persoana
{
private:
    static int total_pers;
    static int total_b;
    char sex;

public:
    persoana( char sx = 'B' ) { total_pers++; sex = sx; }
    ~persoana() { total_pers--; }
    static int spune_total() { return total_pers; }
    static int spune_total_b() { return total_b; }
    static void numara_b( persoana *pp )
        { if( pp->sex == 'B' ) total_b++; }

} p[10];
```

```
int persoana::total_pers=0;
```

```
int persoana::total_b=0;
```

```
void main()
```

```
{
    int i;
    p[0]=persoana('F'); p[3]=persoana('F');
    cout << "\n Avem " << persoana::spune_total() << " persoane";
    for(i=0; i< persoana::spune_total(); i++) persoana::numara_b(&p[i]);
    cout << ", din care " << persoana::spune_total_b() << " barbati!";
}
```

Programul prezentat exemplifică conceptul de membru static lucrând pe un vector de persoane, care prin valorile implicate ale constructorului sunt de sex masculin. Rezultatul rulării este:

*Aven 10 persoane, din care 8 barbati!*

confirmă modul de lucru al variabilelor și funcțiilor statice.

Un apel corect ar fi fost și **p[0].spune\_total()**, care ar fi returnat același rezultat, referință la primul obiect din clasă (p[0]), nefiind necesară; ea înlocuiește aici clasa în general, pentru calificarea funcției, acțiunea funcției fiind independentă de un obiect concret.

## 1.6 Transferul obiectelor în / din funcții

Ca și pentru tipurile fundamentale, compilatorul alocă în anumite situații spațiu temporar pe stivă. Aceste obiecte temporare sunt în cazul claselor și ascunse, adică inaccesibile prin program. Un exemplu concludent îl reprezintă transferul și returnarea unui obiect în / din funcții. Obiectele, fiind producători ale unor structuri, pot fi transferate prin adresă, prin valoare sau prin referință. De crearea obiectelor temporare pe stivă, necesare transferului, responsabil este constructorul de copiere, nu constructorul de clasă. Constructorul de copiere primește ca parametru de intrare, prin referință, un obiect al clasei pentru a-l folosi la inițializarea altui obiect.

În programul de mai jos, constructorul de clasă și de copiere se apelează fiecare câte o dată:

```
class cls
{
public:
    cls() { cout << "\n Constructor"; }
    cls( cls &c) { cout << "\n Constructor copiere"; }

    int f( cls c) { return 1; }
}

void main() {    cls c; f(c); }
```

Dacă modificăm programul, optând pentru transferul obiectului prin referință, constatăm că se apelează o dată constructorul de clasă, iar cel de copiere nu mai este apelat deloc:

```
class cls
{
public:
    cls() { cout << "\n Constructor"; }
    cls( cls &c) { cout << "\n Constructor copiere"; }

    int f( cls &c) { return 1; }
}

void main() {    cls c; f(c); }
```

La fel stau lucrurile și dacă transferul s-ar face prin adresă, nemaifiind necesare obiecte temporare pe stivă. Trebuie avut în atenție și că eventualele

modificări aduse obiectelor în funcție, se regăsesc sau nu și în apelant, după cum transferul s-a făcut prin adresă sau referință, respectiv prin valoare.

Atenție sporită trebuie acordată și transferului prin referință când el impune conversii de obiecte pentru adaptare la prototip; aceste conversii crează tot obiecte temporare și deși transferul se face prin referință, modificările din funcție nu se regăsesc la revenire, în obiectul trimis spre actualizare. Spre exemplu, după execuția programului:

```
#include <iostream.h>
class persoana
{
public: int virsta; persoana(int v=30) : virsta(v){ } }

class profesor
{
public:
    int virsta; profesor(int v = 20) : virsta(v) { }
    operator persoana() { persoana p; p.virsta = virsta; return p; }

};

persoana f(persoana &p) { p.virsta++; return p; }

void main()
{
    persoana p; f(p); cout << endl << p.virsta;
    profesor prof; f(prof); cout << endl << prof.virsta;
}
```

pe multe versiuni de compilatoare, datorită conversiei obiectului profesor în persoana, pentru adaptare la prototip, vârstele afișate sunt 31, respectiv 20, deși se execută aceeași funcție !

Nu este cazul compilatorului Visual C++ 6.0, care la crearea obiectelor temporare necesare adaptării la prototip, le pune și specificatorul const, ceea ce previne modificările, care oricum nu puteau fi preluate de obiectul original.

## 1.7 Pointeri de date și funcții membre

Manipularea obiectelor unei clase se poate face și folosind pointeri, atât la nivelul obiectului unitar, cât și / sau la nivelul unui membru al clasei (funcție sau dată).

### ● declararea pointerilor de membru în clasă.

Un membru va putea fi adresat printr-un pointer care poartă în declarație atât tipul membrului, cât și al clasei căreia aparține:

```
int persoana:: *pdm;
```

declară un pointer de data membru, întreg al clasei *persoana*, în timp ce

```
int ( persoana :: *pfm )( );
```

declară pointer de funcție membră a clasei *persoana*, care întoarce un întreg. Confuzii frecvente se pot produce între un pointer membru în clasă și un pointer nemembru în clasă, dar care pointează un membru din clasă. Să examinăm comparativ declarațiile celor doi pointeri:

```
int persoana:: *p; // pointer de membru
```

```
int *persoana:: p; // pointer membru
```

Pozitia *\** în raport cu rezoluția de clasă (*persoana::*) este cea care face distincția; pentru pointer de membru, *\** se pune lângă identificator, sugerând că *p este în primul rând pointer* și mai apoi apare legat de clasa *persoana*, prin aceea că nu pointează întregi oarecare, ci numai întregi din clasa *persoana*.

În cel de-al doilea caz, *\** se pune lângă *int*, iar identificatorul devine *persoana::p*, arătând că *p este în primul rând membru în clasa persoana* și mai apoi pointer de *int*.

Similar stau lucrurile în legătură cu funcțiile membre:

- membrii pointeri de funcții (spre exemplu funcția de calculul salariului se tot schimbă, astfel încât programatorul a scris mai multe versiuni, dar numai una este cea valabilă la un moment dat și ea este legată de clasă printr-un pointer, pe care programatorul îl poate comuta de pe o funcție pe alta);
  - pointerii de funcții membre (un pointer nemembru în clasă, în principiu independent, dar care pointează pe câte una din funcțiile membre ale clasei *persoana*, cu prototipuri identice).
- Declarațiile lor comparative sunt:

```
int ( persoana :: *pf )( ); // pointer de membru
```

```
int (*persoana :: pf )( ); // membru pointer
```

Prima declarație arată că *pf este în primul rând pointer de funcție*, *\** fiind plasat lângă identificator și mai apoi legat de *persoana::* prin faptul că funcțiile pointate de el nu sunt independente, ci aparțin clasei *persoana*.

Cea de-a doua declarație plasează persoana:: lângă identificator sugerând că *pf este în primul rând membru* al clasei *persoana* și mai apoi pointer.

### ● încărcarea pointerului cu adresa unui membru al clasei, care respectă prototipul din declarația pointerului:

```
pdm = &persoana::virsta; // incarcare pointer la data membru
pfm=persoana::spune_vechime; // incarcare pointer functie membru
```

● folosirea efectivă a pointerilor de membri presupune precizarea obiectului ai căror membri sunt deja pointați în faza anterioară:

```
v = p2.*pdm;
v = (p2.*pfm)( );
```

când se pornește de la un obiect, respectiv

```
v = pp->*pdm;
v = (pp->*pfm)( );
```

când se foloșește un pointer de obiect.

Se observă că pointerul la membru nu se asociază unei instanțe, ci clasei în sine. El nu conține adresa efectivă a câmpului pointat, ci adresa lui relativă la începutul clasei. Acest lucru conferă particularități pointerilor de membri în clasă, așa cum se poate observa din programul de mai jos.

Concluzionând putem spune că un pointer de membru se încarcă în două etape:

- încărcarea offset-ului (deplasamentul membrului în cadrul clasei);
- la folosire, încărcarea adresei de început a obiectului, la care se va adăuga deplasamentul anterior; cele două componente **bază:offset** sunt suficiente pentru a localiza univoc datele membre ale unei anumite persoane. Din modul de declarare, încărcare și folosire un pointer de membru apare ca și cum odată este independent, iar altă dată este legat de o clasă sau obiecte ale acesteia.

În prima parte, programul afișează vârsta unei persoane folosind pointer de dată membră și vechimea folosind pointer de funcție membră. Apoi se reîncarcă pointerii de membri, comutând cu pointerul de dată de pe *virsta* pe *vechime* și cu pointerul de funcție de pe *spune\_vechime()* pe *spune\_virsta()*; se afișează din nou vârsta și vechimea, dar folosind acum pointer de funcție membră pentru vârstă și pointer de dată membră pentru vechime.

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

```

class copil
{
public:
    char prenume[10];
    copil(char *pren = "Puiu")
        { strcpy(prenume,pren); }

class persoana
{
public:
    int vîrstă;
    char nume[20];
    int vechime;
    copil c;
    persoana(char *n="Anonim ", int v=0, int vv=0) :
        vîrstă(v), vechime(vv) { strcpy ( nume, n); }
    int spune_vîrstă() { return vîrstă; }
    int spune_vechime() { return vechime; }
};

void main()
{
    persoana p1, p2("Barbu Doru ",22,3), *pp;
    int persoana::*pdm; // declarare pointer la data membru
    int (persoana::*pfm)(); // declarare pointer la funcție membru
    copil persoana::*pc; // declarare pointer la membru structura

    pdm = &persoana::vîrstă; // incarcare pointer la data membru
    pfm=persoana::spune_vechime; // incarcare pointer functie membru

    cout << "\n " << p2.nume << " are vîrstă " << p2.*pdm;
    cout << " si o vechime de " << (p2.*pfm)();
    cout << "\n " << p2.nume << " are vîrstă " << (p2.*pfm)();
    cout << " si o vechime de " << p2.*pdm << " ani";

    // Refolosirea pointerilor, prin reincarcare cu alte adrese
    pdm = &persoana::vechime; // comuta de pe vîrstă pe vechime
    pfm = persoana::spune_vîrstă; // schimba functia pointata
    cout << "\n " << p2.nume << " are vîrstă " << (p2.*pfm)();
    cout << " si o vechime de " << p2.*pdm << " ani";

    // acces prin pointer de obiect
    pp=&p2;
    cout << "\n " << pp->nume << " are vîrstă " << (pp->*pfm)();
    cout << " si o vechime de " << pp->*pdm << " ani";

    // Incarcarea si folosirea pointerului de membru de tip structura
    pc=&persoana::c; cout << "\n " << (p2.*pc).prenume;
}

```

```

cout << "\n " << (pp->*pc).prenume;
}

```

Programul afișează:

```

Barbu Doru are vîrstă 22 si o vechime de 3 ani
Barbu Doru are vîrstă 22 si o vechime de 3 ani
Barbu Doru are vîrstă 22 si o vechime de 3 ani
Puiu
Puiu

```

confirmându-ne localizarea corectă a elementelor pointate prin pointerii de membri. Totdeauna la folosirea efectivă a unui pointer de membru, este nevoie de adresa unui obiect concret, care să constituie baza în adresarea efectivă a unei date sau funcții, pointerul în sine conținând doar offset-ul în cadrul obiectului.

Dacă specificarea se face pornind de la un obiect al clasei, accesul prin pointer de membru se face sub forma **obiect.\*pointer\_membru**, iar dacă specificarea pornește cu pointer chiar de la nivel de obiect, calificarea se va face sub forma: **pointer\_object -> \*pointer\_membru**.

Deoarece numele funcțiilor sunt tratate ca pointeri (constantă), la încărcarea pointerilor de funcții nu mai este obligatorie folosirea operatorului **\***, de extragere adresă, deși este acceptată în limbaj și această posibilitate. În schimb, nu trebuie pus niciodată operatorul **( )**, deoarece **spune\_vechime()** ar însemna apel de funcție.

După cum s-a observat, este posibilă declararea unei clase ce conține printre datele membre și obiecte de tipul altor clase. În ce privește membrii mai *speciali*, care sunt la rândul lor structuri sau clase, lucrurile nu diferă cu nimic de pointerii de membri simpli; **copil persoana::\*pc;** declară pointer la membru de clasă copil, **pc = &persoana::c;** încarcă acest pointer cu adresa unui copil, în eventualitatea că o persoana ar avea mai mulți copii, iar pointerul ar putea adresa pe rând câte unul, iar

```

cout << "\n " << (p2.*pc).prenume;
cout << "\n " << (pp->*pc).prenume;

```

ne arată cum se folosește un astfel de pointer pentru a accesa efectiv informațiile specifice obiectului inclus.

Ar mai fi de discutat un aspect legat de pointerii de membri și anume ce se întâmplă atunci când **membrul pointat este unul privat**. În mod logic, un pointer de membru privat poate fi încărcat, dar nu și folosit. La ce mai este însă util acest pointer? Transmis ca parametru unei funcții friend sau unei funcții membre, el devine operațional! Realizăm astfel o adaptare a funcției la

context; spre exemplu, putem avea în clasa persoana un salariu de încadrare, unul efectiv realizat și unul mediu, pe ultimul an. Dacă printr-o opțiune meniu se stabilește ca toate prelucrările să se facă pe mărimi medii, aceleași funcții de interfață se adaptează automat contextului, deoarece pointerul primit încorporează contextul, fiind actualizat la momentul setării opțiunii din meniu.

Există versiuni de compilatoare (inclusiv Visual C++ 6.0) care nu permit nici încărcarea unui pointer de membru privat.

### Pointeri de membri, membri în clasă

Interesant că pointerii de membri pot fi făcuți la rândul lor membri în clasă; se ajunge astfel ca un obiect să se „autoconțină”, adică să încapsuleze informație despre sine. Acest lucru permite generalizări dintre cele mai interesante; să presupunem că scriem o funcție care primește un obiect persoana și-l introduce într-un arbore de indexare, în funcție de CNP (cod numeric personal), considerat cheie alfanumerică; alteori vrem să construim un index similar, dar care să permită o regăsire rapidă după numele persoanei sau după altă cheie tot alfanumerică. Cum putem realiza acest lucru, refolosind funcția de indexare fără nici o modificare? Vom include în clasa persoana un **pointer de membru char \***, care să pointeze pe cheia activă la un moment dat; funcția de indexare nu știe cu ce cheie lucrează, ci se bazează în exclusivitate pe ce-i arată acest pointer de cheie!

Ilustrăm în continuare acest aspect de folosire a pointerilor de membri ca membri în clasă, pe un exemplu mult mai simplu. Funcția de premiere a unui salariat este făcută mai general, în sensul că aplică un procent de premiere la un salariu care poate fi cel de încadrare, cel efectiv realizat în acea perioadă, sau de ce nu, unul mediu pe ultimul an. Ea nu știe căruia salariul **psal**, adică un pointer de salariu, unul din salariile membre în clasa *persoana*; urmărind programul următor se observă că *psal* este la rândul lui membru în clasa *persoana*, adică este un pointer de membru, membru în clasă! Cu o altă funcție membră, **set\_cheie()**, comutăm de pe un salariu pe altul, activând diferite, deși procentul de premiere este același.

```
#include <iostream.h>
class persoana
{
    double salariu_incadrare;
    double salariu_efectiv;
```

public:

```
double persoana::*psal;
void set_cheie(int c)
{
    switch(c)
    {
        case 1: psal= &persoana::salariu_incadrare; break;
        case 2: psal= &persoana::salariu_efectiv; break;
    }
}
double premiere( float procent)
{
    return this->*psal * procent/100;
}
persoana(double si=0, double se=0):
    salariu_incadrare(si), salariu_efectiv(se) {}

void main()
{
    persoana p(1000, 1500);
    p.set_cheie(1);
    cout << "\n La salariu incadrare: " << p.premiere(2);
    p.set_cheie(2);
    cout << "\n La salariu efectiv: " << p.premiere(2);
}
```

De remarcat în funcție prezența lui *this* pentru a indica obiectul folosit în faza a doua a încărcării pointerului de membru:

```
this->*psal* procent/100;
```

aparent *this* ar putea lipsi căci ne aflăm într-o funcție membră a clasei care oricum raportează membrii clasei la adresa obiectului *\*this*; nu încercă să-l scoateți căci veți fi sancționați cu eroare de compilare, pentru că aici *this* ajută la calificarea lui **psal**, spunând că este pointer de membru din obiectul curent.

**psal** este membru nestatic al clasei *persoana* și ne arată că putem fixa baza de premiere pentru fiecare persoană în parte, căci odată încărcat pointerul de membru e purtat cu obiectul pretutindeni, el fiind în același timp și membru al clasei. Ce se întâmplă dacă dorim ca baza de premiere să fie unică pentru toate persoanele? Evident trebuie să facem membru static; lăsăm în sarcina cititorului modificarea programului de mai sus, încât să răspundă noilor ipoteze.

## 1.8 Clase și funcții prietene. Privilegii în sistemul de acces

Accesul la membrii unei clase, deși restricționat prin domeniile private și protected, poate fi îngăduit și unor funcții externe clasei (deseori numite independente) sau aparținând altor clase. În acest caz, este nevoie de garantarea drepturilor de acces prin declararea funcțiilor respective drept funcții prietene, folosind cuvântul cheie friend.

Funcțiile rămân externe, nefiind legate de clasă și cu atât mai puțin de un obiect anume. Calificarea lor se face în mod normal, fără a necesita referiri la clasă sau la vreun obiect. Pentru a accede însă la datele unui obiect individual, funcția trebuie să primească drept parametru de intrare și nu ca element în propria calificare, referința la obiectul respectiv.

Vom simplifica și completa clasa persoana cu două funcții, una de încredințare a informației de vîrstă, prietenilor, alta de comunicare a ei prin funcția consult(), aparținând clasei medicilor.

```
class persoana;
class medic
{
    public: int consult( persoana & );
    // ...
};

class persoana
{
    int vîrstă;
    friend int spune_prieten( persoana & );
    friend int medic::consult(persoana & );
public:
    persoana( int v=0 ) { vîrstă=v; }

int spune_prieten( persoana & p ) { return p.vîrstă; }
int medic::consult( persoana & p ) { return p.vîrstă; }

#include <iostream.h>
void main()
{
    persoana p(32); medic m;
```

```
cout << "\nPrietene, am " << spune_prieten(p) << " ani !";
cout << "\nDoctore, am " << m.consult(p) << " ani !";
```

{}

Declarația *friend* se face în clasa care acordă drepturile de acces (clasa *persoana*), nu în cea care beneficiază de aceste drepturi (clasa *medic*). După cum se observă declararea funcțiilor friend se poate face și în zona privată.

În ce privește funcția independentă *spune\_prieten()*, lucrurile sunt simple, deoarece ea nu aparține unei clase. Pentru funcția *consult()*, care trebuie să specifică două clase, una căreia aparține și alta, sursa ei de date, declarăriile se dau într-o anumită ordine.

Succesiunea de declarare este impusă de faptul că declarația friend trebuie să precizeze clasa căreia aparține funcția (*medic*), care clasă nici ea nu poate fi declarată prima, deoarece lista de parametri a funcției *consult()* trebuie să includă și o referire la clasa *persoana*, nedefinită încă! Astfel, se acceptă doar o **declarație formală** a tipului *persoana*, urmând ca ulterior să se dea structura completă a clasei. În momentul compilării funcției *consult()* se vor cunoaște deci complet structurile celor două clase implicate în dialog.

Când se dorește ca toate funcțiile unei clase să fie funcții prietene ale altor clase, atunci **întreaga clasă** se poate declara *friend*.

Spre exemplu, funcțiile de consultare ale clasei *medic* sunt în bloc declarate prietene ale persoanei. Se observă că nu se face distincție între *private*, *protected* sau *public* în ce privesc drepturile de acces prin funcții prietene.

```
class medic;
class persoana
{
    private: int vîrstă;
    public: persoana(int v=20) {vîrstă=v;}
            char nume[20];
            friend medic;

};

class medic
{
    public:
        void cere_nume(persoana &p) { cout << "\n " << p.nume; }
        void cere_vîrstă(persoana &p) { cout << " " << p.vîrstă; }

};

void main()
{
    persoana p(15); medic m; strcpy(p.nume,"Petrescu V.");
```

```
m.cere_nume(p);      m.cere_virsta(p);
```

Când nu există declarația formală **class medic** în față, în clasa **persoana** trebuie dat **friend class medic**, altfel compilatorul n-ar știi ce reprezintă identificatorul medic.

Funcțiile prietene pot nu numai să consulte datele unei clase, ca în exemplul nostru, ci să și le modifice, constituind un punct dificil în gestiunea unitară a informațiilor unei clase.

Dincolo de încălcarea unor reguli de acces, generator de scăpări de sub control ale unor informații, mecanismul **friend**, modelează o realitate indiscutabilă, facilitând comunicarea între obiecte.

## 1.9 Modificatorul **const** în contextul obiectelor

### Obiecte constante

Am văzut deja că o clasă acceptă declararea de obiecte constante sub forma:

```
const persoana p1;
```

sau:

```
persoana const p1;
```

Un obiect constant **nu este Ivaloare** (nu poate fi scris în stânga unei atribuirii), dar poate fi sursa unei atribuirii. El poate fi transferat prin referință sau adresă ca parametru în funcție, doar dacă funcția a declarat și tratat referința obiectului drept constantă.

La transferul prin valoare, obiectul constant folosește doar la crearea copiei parametrului actual pe stivă; ca urmare nu există restricții. În schimb dacă și copia este declarată constantă, prelucrările din funcție trebuie să respecte acest lucru. Pentru testare au fost construite funcțiile:

```
void f_a( persoana * );
void f_r( persoana & );
void f_v( persoana p );
```

care transferă obiecte de tip **persoana** prin adresă, prin referință, respectiv prin valoare.

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

class persoana
{
private:    int virsta;
public:
    char nume[20];
    double salariu;
    persoana(char n[20] = "Anonim ", int v=0, float s=1000000. )
        : virsta(v), salariu(s) { strcpy(nume,n); }
    int get_virsta() const { return virsta; }
    double get_salariu() { return salariu; }
    void set_salariu(double s) { salariu=s; }
    void f_m(const persoana p) { }

};

void f_a( persoana *p)      {}
void f_r( persoana &p)      {}
void f_v( persoana p)      {}

void main()
{
    persoana const p1;
    persoana p2, p3;
    p2 = p1;
    // p1=p2;      - Eroare: obiectul const nu e l_valoare
    // f_r(p1);   - Eroare: f_r nu a declarat const obiectul referit,
    //               // sub forma void f_r( const persoana &p ) {}
    // f_a(&p1);  - Eroare: f_a nu a declarat const obiectul adresat,
    //               // sub forma void f_a(const persoana *p ) {}
    f_v(p1);
    cout << p1.get_virsta(); cout << p1.salariu;
    // cout << p1.get_salariu(); - Eroare: functie nedeclarata const
    // cout << p1.set_salariu(); - Eroare: functie care modifica obiectul
    cout << p1.get_virsta(); cout << p1.salariu;
    // cout << p1.get_salariu(); - Eroare: functie nedeclarata const
    // cout << p1.set_salariu(); - Eroare: functie care modifica obiectul
    // p1.f_m(p1);
    // - Eroare: unul din obiecte, cel implicit nu e declarat const
    // sub forma void f_m(const persoana p) const {}
```

După cum se observă în exemplu de mai sus, `f_a` și `f_r`, care folosesc obiectul `p1` transferat prin adresă, respectiv referință, nu declară că nu-l vor altera, astfel încât compilatorul semnalează eroare la apelul lor. În comentariu apar declarațiile complete, purtând și modificatorul `const`, astfel încât funcțiile să poată lucra și cu obiecte constante.

De reținut că acest lucru e obligatoriu, chiar dacă din corpul funcțiilor (vid, de altfel) se putea vedea că funcțiile nu modifică efectiv obiectul primit.

**În ce privește datele membre ale unui obiect constant**, ele pot fi consultate, dar nu și modificate. Domeniile de acces public, private și `protected` sunt de asemenea respectate și în cadrul obiectelor constante.

**În ce privește funcțiile membre**, pot fi activate pornind de la un obiect constant doar dacă au declarat obiectul primit prin pointerul `this` drept constant. Apelul `p1.get_salariu()`; eșuează deoarece `p1` este `const`, iar `get_salariu()` n-a declarat explicit că nu va modifica obiectul primit implicit.

Obiectele primite explicit pot fi declarate drept constante în prototipul funcției, dar cel transferat implicit nu apare descris undeva, astfel încât mențiunarea lui drept constant se face înainte de acolada de început a corpului funcției:

```
int get_vîrstă() const { return vîrstă; }
```

Acest lucru este echivalent ca efect cu a declara pointerul `this` drept pointer spre un conținut constant. Declarația de **const nu este permisă pe constructor și pe destructor**, deoarece se consideră implicit că aceștia modifică datele obiectului, în faza de creare, respectiv distrugere a obiectului.

În exemplu de mai sus se observă că obiectul constant `p1` poate activa `get_vârstă()` care a fost declarată corespunzător, dar nu și `get_salariu()`, deși aceasta nu modifică obiectul, dar n-a declarat acest lucru !

În privința funcției membre `f_m` lucrurile sunt mai subtile; ea lucrează cu două obiecte `p1`, unul explicit primit ca parametru de intrare și unul implicit primit prin `this`, funcția fiind membră nestatică a clasei. Primul obiect este declarat `const`, dar cel de-al doilea nu !

### **Pointeri constanți de obiecte și pointeri de obiecte constante**

Spre deosebire de obiecte, pointeri de obiecte pot indica la definire că sunt ei însăși constanți și/sau că pointează o zonă constantă.

Un **pointer** este **constant** în sensul că o dată încărcat cu o adresă, rămâne fixat pe această adresă, pe toată durata existenței lui; de altfel la declarare el trebuie să și menționeze adresa de care este legat, mai târziu nemaiputându-se reîncărca. Zona pointată poate fi însă modificată, fie direct, fie indirect, prin intermediul pointerului.

Un pointer constant de obiecte se declară sub forma:

`persoana * const pp;`

cea ce sugerează că `pp` este **constant**, nu și `*pp`, conținutul lui.

Programul principal de mai jos poate fi împărțit în trei părți:

- prima, care exemplifică lucru cu pointeri constanți;
- a doua, care lucrează cu pointeri ce adreseză obiecte constante
- a treia, unde apare pointer constant de conținut constant.

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

class persoana
{
public:
    int vîrstă;
    double salariu;
    persoana(int v=1, float s=1000000.):
        vîrstă(v), salariu(s) {}

    int spune_vîrstă() const { return vîrstă; }
    double spune_salariu() { return salariu; }
    void f_m(const persoana p) {}

};

void main()
{
    persoana const p1;
    persoana p2,p3;
    p2.salariu= 3000.; p2.f_m(p1);
    // persoana *const p_cst= &p1;
    // - Eroare: un pointer constant, dar pointand continut variabil
    //           nu poate fi initializat cu adresa unui obiect constant
    persoana *const p_cst= &p2;
    // p_cst=&p3;
    // - Eroare: pointerul constant nu poate fi reincarcat
    p_cst->f_m(p1); // obiectul pointat nu e const

    persoana const *p_con_cst;
```

```
// echivalent cu: const persoana *p_con_cst;  
// *p_con_cst = (const persoana)p2;  
// Eroare: operator= returneaza obiect variabil  
// *p_con_cst=p1;  
  
persoana const * const p_cst_continut_cst = &p1;  
// echivalent cu const persoana * const p_cst_continut_cst = &p1;  
cout << "p1 are vîrstă " << p1.spune_vîrstă();  
}
```

**p2** este un obiect variabil, putând fi modificat și putându-și activa toate metodele sale, în conformitate cu drepturile de acces specifice clasei din care face parte. Se observă chiar că poate activa și funcția membră **f\_m**, care prelucră obiecte constante (p1) și în care p2 apare transmis implicit, prin **this**.

Pointerul constant **p\_cst** nu poate fi încărcat cu adresa lui p1 care este un obiect constant, deoarece nu pointează conținut constant, ci doar adresa conținută de el este constantă. **p\_cst** poate să preia adresa lui p2 și să-i activeze metodele, cum face însuși p2, dar rămâne definitiv legat de p2, nemaiputându-se reîncărca cu adresa lui p3. Faptul că pointează obiecte neconstante ne-o demonstrează și apelul **p\_cst->f\_m(p1)**, căci **f\_m** nu lucrează cu obiect constant, transmis prin **this**.

**p\_con\_cst**, pointerul de conținut (obiect) pointat constant își poate modifica valoarea, comutând de pe un obiect constant pe altul, dar zona pointată chiar când e adunată via pointer, se comportă conform unui obiect constant, adică în ansamblu ei nu e lvaloare și nici elementele ei individuale nu pot fi modificate.

**p\_cst\_continut\_cst** este un pointer constant care pointează un obiect constant; el este legat pentru totdeauna de adresa obiectului cu care a fost încărcat la declarare și în același timp își protejează acest obiect de orice tentativă de modificare a datelor sale sau de invocare a metodelor ce i-ar putea modifica aceste date.



## SUPRAÎNCĂRCAREA OPERATORILOR ȘI FUNCȚIILOR

- Supraîncărcarea funcțiilor independente și a funcțiilor membre
- Aspecte generale și restricții privind supraîncărcarea operatorilor
- Supraîncărcarea operatorilor
- Conversii între obiecte de diferite tipuri
- Aplicații

## 2.1 Supraîncărcarea funcțiilor independente și a funcțiilor membre

Supraîncarcarea (overloading) funcțiilor și operatorilor reflectă **posibilitatea de a atribui unui simbol mai multe semnificații**, ce pot fi deduse din contextul de folosire. Ea este o caracteristică de bază a limbajelor de programare, sesizabilă, spre exemplu, la operatorii aritmetici, care știu să opereze cu numere și întregi și flotante, deși acestea au reprezentări interne total diferite; aceeași expresie  $a+b$  poate ascunde operații elementare, total diferite când  $a$  și  $b$  sunt întregi, sau sunt *double*.

Limbajul C++ extinde această facilitatea, dând programatorului posibilitatea de a introduce sensuri multiple uneia și acelorași funcții sau să atribuie semnificații noi, operatorilor recunoscuți deja în limbaj pentru tipurile de bază.

Selectarea funcției, dintr-un set de funcții cu același nume, cu aceleași obiective de prelucrare, dar nuanțe de la un caz la altul, se face în principiu, pornind de la numărul și tipul parametrilor, adică de la **signatura funcției**. Putem scrie aşadar mai multe funcții *suma*, fiecare fiind specializată să adune câte ceva: numere reale, numere complexe, matrice, persoane etc. Compilatorul va ști la un moment dat pe care să o apeleze în funcție de ce trebuie să adune; o astfel de funcție care efectuează prelucrări diferite de la un context la altul se numește **funcție polimorfică**.

Dacă numărul parametrilor este același, **tipul** lor devine esențial în selectarea funcției de apelat. Pe de altă parte, când sunt apelate cu parametri de alt tip decât cel specificat în prototip, funcțiile C pot antrena și ele conversii de tip pentru adaptare la prototip. În acest caz, frecvent se intră în conflict între folosirea tipurilor din apel pentru selectarea funcției de apelat și conversiile de tip efectuate la transmiterea parametrilor, pentru adaptare la prototip. Rezolvarea acestui conflict se face **etapizând selectarea funcției de apelat**:

- ➊ în prima fază, se încearcă identificarea versiunii funcției conform tipului parametrilor din apel, fără operarea vreunor conversii;
- ➋ dacă nu există o astfel de funcție, se aplică un set de conversii numite **nedegradante** (fără pierderi de informații): *char, short* în *int*, respectiv *float* în *double* și numai dacă nu se realizează individualizarea funcției se continuă cu aplicarea altor conversii;

- ➌ dacă încă după această etapă nu a fost selectată univoc funcția, se continuă cu operarea conversiilor **degradante** (cu pierderi de informații): de la numeric la numeric, indiferent de tip; între pointeri de orice tip și *void*; de la o constantă întreagă către pointer sau de la pointer de clasă derivată la pointer de clasă de bază;
- ➍ într-o ultimă etapă, procesul de identificare a funcției de apelat continuă prin aplicarea eventualelor conversii introduse de utilizator, prin supraîncărcarea operatorului *cast*.

Căutarea se oprește în momentul în care o singură funcție răspunde criteriilor de selecție; dacă într-o etapă, după operarea conversiilor, mai multe funcții răspund criteriului de căutare, este semnalată o **eroare de ambiguitate**.

Dacă după parcurgerea tuturor etapelor nu este selectată nici o versiune, este semnalată **eroare la linkeditare** (simbol nedefinit).

Procesul selectării versiunii funcției de apelat este complicat și prin posibilitatea definirii funcțiilor cu același nume în fișiere sursă, compilabile separat, reunite abia la linkeditare. Acest lucru obligă compilatorul să atâșeze numelor funcțiilor și un cod atribuit în raport cu parametrii de apel (**decorarea numelor**).

Valoarea returnată nu intră niciodată în discuție, deoarece tipul returnat este criteriu de validare sintactică în compilare. Valoarea returnată de o funcție poate fi transferată prin apel altrei funcții (componerea funcțiilor) și deci ea trebuie cunoscută aprioric și ca tip.

**Valoarea returnată de o funcție nu este criteriu de identificare a versiunii de apelat** și pentru faptul că atunci când valoarea returnată nu este preluată nu s-ar știi despre care versiune este vorba.

Pentru înțelegerea mai exactă a celor expuse mai sus vom face referiri la funcția:

```
void f(int i, double d)
{
    cout << "\n i = " << i << " d = " << d << endl;
}
```

Conversiile operate la diferitele forme de apelare a funcției sunt sintetizate în tabelul 2.1.

După cum se vede compilatorul alege cu prioritate conversiile ce nu antrenează pierderi de informație; dacă am avea în schimb două funcții *f(int,double)* și *f(double,int)*, la un apel *f(2.5, 3.7)* compilatorul neștiind care

dintre cei doi *double* să-l convertească în *int*, nu va selecta nici una din versiunile lui *f()*, semnalându-ne ambiguitate.

Forma de apel	Conversiile aplicate	Observații
<i>f(1, 2.7)</i>	nici o conversie	—
<i>f('a', 2.7)</i>	<i>char</i> în <i>int</i> , în faza 2	fără pierdere de informație
<i>f(1.5, 2.7)</i>	primul parametru ( <i>double</i> ) e convertit în <i>int</i> , în faza 3	se pierde zecimala
<i>f(2, 3)</i>	<i>int</i> în <i>double</i> , la parametr 2, în faza 3	fără pierdere de informație

Tab. 2.1 Conversii la apel

Funcțiile membre respectă aceleași reguli de supraîncărcare, ca și funcțiile independente. Trebuie remarcat totuși că funcțiile membre poartă pe lângă nume și clasa sau obiectul cărora aparțin, element care le distinge de alte funcții care se numesc la fel, dar sunt membre în altă clasă; problema supraîncărcării funcțiilor membre apare doar când e vorba de mai multe funcții care se numesc la fel și aparțin aceleleași clase.

## 2.2 Aspecte generale și restricții privind supraîncărcarea operatorilor

Operatorii sunt asimilați unor funcții cu numele format din cuvântul cheie *operator* și simbolul grafic al unui operator din limbaj. În acest mod, o clasă poate fi înzestrată cu operații specifice ei (în cazul clasei *persoana* incrementarea vârstei și vechimii unei persoane, avansarea, retrogradarea, cumulul de funcții etc.), operații care se invocă apoi extrem de simplu cu ajutorul operatorilor.

Operatorii apar deci ca niște **funcții care au și forme simple de apel**. Spre exemplu, *a+b* trebuie interpretat ca un apel de forma *a.operator+(b)*, adică funcția membră numită *operator+( )*, aparținând obiectului *a*, este apelată având ca parametru de intrare obiectul *b*.

Ca orice funcții, operatorii pot fi supraîncărați. Există totuși și câteva restricții în supraîncărcarea operatorilor, acestea sunt:

① **Precedența și direcția de evaluare** a operatorilor nu pot fi schimbate prin supraîncărcare. Pot fi folosite însă parantezele pentru a

introduce ordinea de prioritate, după aceeași sintaxă ca la expresiile aritmetice uzuale.

② **Asociativitatea** operatorilor nu poate fi schimbată prin supraîncărcare.

③ **Cardinalitatea (pluralitatea)** operatorilor trebuie conservată prin supraîncărcare, exceptând:

- operatorul funcție (), care poate primi oricărți parametri prin supraîncărcare;
- operatorii + și – care pot fi și unari, ca operatori de semn și binari, ca operatori aritmetici;
- & și \* pot fi și binari și unari (SI pe biți și înmulțire, respectiv extragere de adresă și extragere de conținut) putând beneficia de supraîncărcări în ambele ipostaze.

④ Nu pot fi creați noi operatori în limbaj și deci **nu pot fi supraîncărați decât operatorii existenți**.

⑤ Operatorii **nu se compun automat**; spre exemplu: existența unor supraîncărcări pentru + și pentru = nu înseamnă că putem folosi deja + și pentru obiecte; acest operator poate fi el însuși supraîncărat explicit.

⑥ Sunt **exceptați de la supraîncărcare** operatorii . . \* :: ?: și *sizeoff()* care au semnificații implicate și în cazul claselor, sau au forme dificile pentru a fi supraîncărați (ca în cazul operatorului condițional, compus din trei părți separate prin două simboluri cheie);

⑦ Supraîncărcarea se poate face pentru unii operatori **numai prin funcții membre nestaticice** ale clasei sau numai prin funcții friend și deci afectează doar clasele definite de utilizator; aceasta deurge din faptul că unul din argumentele funcției operator trebuie să fie totdeauna obiect al clasei. Când supradefinierea se face prin funcție membră nestatică obiectul este recunoscut implicit, datorită pointerului *this*; în cazul funcțiilor friend obiectul trebuie să apară, după cum s-a văzut, ca parametru de intrare în funcție.

- Astfel, pentru operatorii () // -> și = cu formele sale compuse, se admite supraîncărcare doar prin **funcție membră nestatică** a clasei, nu și prin **funcție friend**.
- Pentru operatorii *new* și *delete* se acceptă supraîncărcări numai prin **funcții friend** sau prin **funcții membre statice**.

⑧ **Nu se garantează comutativitatea**, ea depinzând de modul în care se face supraîncărcarea

⑨ Tratarea post- și pre – fixării pentru operatorii care în mod uzual beneficiază de acest tratament în raport cu alte operații, impune folosirea unor prototipuri diferite pentru metodele ce supraîncarcă respectivii operatori.

## 2.3 Supraîncărcarea operatorilor

### Supraîncărcarea operatorilor unari ++ și --

Operatorii unari ++ și -- pot fi supraîncărcați atât prin funcții membre în clasă, cât și prin funcții independente. Cei doi operatori au o particularitate față de ceilalți operatori unari și anume pot fi prefixați sau postfixați, având efecte diferite. Se știe că efectul se manifestă diferit numai dacă operatorul intră în compunere cu un alt operator, cu o instrucțiune sau cu o funcție; altfel expresiile  $p_1++$  și  $++p_1$  au același efect, pe când  $p_1 = p_2++$ ; întâi copiază  $p_2$  în  $p_1$  apoi incrementează vârsta lui  $p_2$ .

Să ne imaginăm pentru simplificare clasa *persoana* cu o singură dată membră – vârsta, iar **operator++** incrementează această vârstă.

Să optăm pentru o supraîncarcare prin funcții independente; obiectul va fi deci primit ca parametru de intrare. Pentru a distinge între formele pre- și post-fixate standardul C++ prevede ca forma postfixată  $p_1++$  să genereze un apel explicit de genul *operator++(obiect, int)* la supraîncarcare prin funcție independentă, respectiv *p.operator++(int)* la supraîncarcarea prin funcție membră. Parametrul int este introdus artificial, doar pentru marcarea versiunii postfixate. În varianta prescurtată de invocare ( $p_1++$ ) întregul nu apare în apel, dar într-o eventuală variantă explicită (unde ++ ocupă același loc în numele funcției ca la prefixare) parametrul întreg trebuie dat pentru distincție: *p.operator++(1)*:

```
#include <iostream.h>
class persoana
{
public:
    int vîrstă;
    persoana(int v=0):vîrstă(v) {}
    friend const persoana & operator++(persoana &); // prefixat
    friend const persoana operator++(persoana &, int); // postfixat
};
```

```
// Prefixat returneaza valoarea incrementata
const persoana& operator++(persoana& a)
{
    a.vîrstă++;
    return a;
}

// Postfixat returneaza valoarea de dinainte de incrementare
const persoana operator++(persoana& a, int)
{
    persoana aux=a; // conservarea stării initiale
    a.vîrstă++;
    return aux;
}
```

Din textul sursă se vede că varianta postfixată conservă starea obiectului de la momentul intrării în funcție, într-o variabilă auxiliară; funcția modifică apoi obiectul primit, dar îl returnează tot pe cel vechi.

Pentru această variantă s-a ales returnarea obiectului prin valoare (copierea pe stivă a obiectului returnat), spre deosebire de varianta prefixată, care returnează referința obiectului primit și incrementat. Nu trebuie să ne facem griji asupra validității referinței, căci ea este cea primită de funcție.

Dacă și în varianta postfixată am fi returnat referința, trebuia să ne asigurăm că obiectul auxiliar mai există și după ieșirea din funcție, adică să fi declarat **static persoana aux**. Funcțiile cu variabile locale statice nu sunt deloc agreate, deoarece ridică restricții majore în programarea multithreading, când mai multe fire de execuție lucrează pe aceeași variabilă și nu mai putem controla cine și când o modifică.

Programul principal:

```
void main()
{
    persoana p1(25),p2;
    p2=++p1; cout << p2.vîrstă;
    p2=p1++; cout << p2.vîrstă;
    cout << p1.vîrstă;
}
```

afisează: 26 26 27, deoarece vârsta a fost preincrementată față de prima atribuire, în timp ce la a doua atribuire mai întâi a copiat, apoi a incrementat, deci vârsta lui p1 devine 27, adică dublu incrementată.

Evident că puteam supraîncarca ++ și printr-o funcție void, deoarece efectul propriu-zis (incrementarea vârstei) era atins, însă atunci nu se punea

de loc problema pre și post fixării, deoarece operatorul nu mai putea intra în compunere cu un altul; astfel, o expresie  $p1 = p2++$ ; ar încearca să copieze în  $p1$  void returnat la evaluarea părții din dreapta, lucru inaceptabil.

Efectul pre/post incrementării se manifestă și în raport cu operatorul adică  $cout <<(p1++) . virsta$  afișează altceva decât  $(++p1) . virsta$ ; parantezele au fost necesare pentru a mări prioritatea operatorului ++ și pentru a nu ne păcăli, incrementând doar un întreg, nu obiectul.

Ne punem întrebarea dacă efectul incrementării se poate pune în evidență și în cascadă, prin expresii de forma  $+++p1$  sau  $p1+++$ .

Modifierul *const* pentru referință, respectiv valoarea obiectului returnat de funcțiile de supraîncarcare a operatorului ++ previn aplicarea în cascadă a operatorului, adică  $p1+++$  sau  $+++p1$ . Să ne imaginăm că ridicăm acest modifier pentru ambele funcții, atât din prototipul dat în clasă, cât și din zona de implementare propriu-zisă. Aplicăm  $+++p1$  și constatăm listând  $p1.vârsta$  că se produce dublă incrementare; modificând programul, punând în loc  $p1+++$ , de asemenea acceptat, de această dată constatăm că expresia indică doar o singură incrementare. Explicația o găsim în faptul că preincrementarea a fost supraîncarcată prin funcție ce returnează referința obiectului primit; aplicată în cascadă, efectele se cumulează asupra aceluiași obiect.

Postincrementarea returnează o copie a obiectului inițial, trimisă mai departe la incrementat. Ambele variante primesc obiectul prin referință, dar primul operator ++ primește într-adevăr referința lui  $p1$ , dar a doua aplicare a lui ++ primește referința obiectului temporar returnat prin valoarea pe stivă, deci altul decât  $p1$ . A doua incrementare se aplică deci obiectului temporar !

Rezultă că probabil aplicarea modificadorului *const* ieșirilor din funcțiile de supraîncarcare este de dorit, prevenind astfel aplicarea în cascadă a unui operator, decât să ne bazăm pe rezultate greu de anticipat.

Să mentionăm, în final, că în cazul operatorilor unari nu putem opta pentru supraîncărcări simultane prin funcție independentă și prin funcție membră, deoarece apelul  $p++$  ar fi ambiguu, neștiind ce versiune de supraîncarcare să se aplice. În cazul operatorilor binari vom vedea că lucrurile stau un pic diferit.

### Supraîncărcarea operatorilor binari + de adunare și +=

Prima întrebare pe care trebuie să ne-o punem, în general în legătură cu supraîncărcarea operatorilor este legată de semnificația pe care o

atribuim operatorului. Spre exemplu, ce semnificație am putea da unei expresii cu obiecte persoane, de forma  $p1 + p2$  ?

Am putea să considerăm că expresia modelează cumulul de funcții sau însumarea salariailor a două persoane, soț-soție. În această accepție, probabil că supraîncărcarea ar putea să verifice și dacă numele celor două persoane coincid (pot distri eventual prenumele, dacă e vorba de soț-soție).

Altineva poate înțelege altceva, spre exemplu, crearea unui nou obiect persoana, care cumulează salariile celorlalte două persoane.

Important este ca sensul să fie acceptat de cei mai mulți care lucrează cu clasa respectivă și să fie ușor de reținut.

**Ce întoarce o funcție operator?** Este o altă întrebare importantă pentru supraîncărcarea operatorilor aflându-se în strânsă legătură cu răspunsul de la prima întrebare. În prima accepție, funcția returnează *double* (salarii cumulate), în cea de-a doua returnează foarte probabil un obiect de tip *persoana*.

Practic, programatorul este cel care alege tipul returnat de funcția operator, exceptând câțiva operatori, care au un tip de return impus:

- operatorul **cast** întoarce tipul la care a fost convertit obiectul curent,
- operatorul **[]** întoarce referință la un obiect pentru a-l putea localiza într-un vector de obiecte, furnizând deci o lvaloare;
- operatorul **new** returnează pointer la zona alocată pentru un obiect
- **operator=** returnează referință de obiect destinație, pentru a se putea compune în cascadă.

Tipul returnat ar putea fi și tipul *void*, dar să nu ne așteptăm la compuneri de tipul  $p1+p2+p3$ , pentru că void returnat de primul operator+ nu se mai poate compune cu nimic!

Operatori	Recomandări de supraîncărcare
operatori unari	prin funcție membră
= () [] ->	obligatoriu prin funcție membră
+ = -= /= *= ^=	prin funcție membră
& =  = %= >>= <<=	prin funcție membră
alți operatori binari	prin funcție independentă

Tab. 2.2 Recomandări de supraîncărcare a unor operatori

**Cum alegem: funcție membră sau funcție independentă?** Din păcate nici la această întrebare răspunsul nu este unic (tabelul 2.2). Putem alege funcție membră și atunci prototipul va fi:

*tip\_return operator+(persoana);*

sau funcție friend:

*friend tip\_return operator+(persoana, persoana);*

Într-o formă simplificată a clasei, dar suficientă pentru a funcționa de sine stătător exemplul de supraîncărcare a operatorului +, prin funcție membră sau prin funcții prietene poate fi evidențiată prin programul:

```
#include <iostream.h>
#include <string.h>
class persoana
{
private:
    friend double operator+( persoana &, double );
    friend double operator+( double, persoana & );
public:
    char nume[20];
    double salariu;
    persoana(char *n="Anonim ",double s=0):salariu(s)
    { strcpy(nume,n); }

    double operator+( persoana& );
    double operator+=( double );
};

double operator+( persoana &p, double spor) { return p.salariu + spor; }
double operator+( double spor, persoana &p) { return p.salariu + spor; }
double persoana::operator+(persoana &p) { return salariu + p.salariu; }
double persoana::operator+=( double spor) { return salariu += spor; }

void main()
{
    double spor = 1.;

    persoana p1("Popa Ion",85000.), p2=persoana("Popa Elena",87000.);
    persoana p3("Adamescu Virgil",75000);
    cout << "\nDupa spor " << p3.nume << " ar avea " << p3 + spor;
    p3+=spor;
    cout << "\n" << p3.nume << " chiar are acum " << p3.salariu << " lei.";
    cout << "\nFamilia " << p1.nume << " are " << p1+p2 << " lei\n";
}
```

Operatorul + a fost multiplu supraîncărcat; de două ori prin funcții friend și o dată printr-o funcție membră a clasei, care returnează suma salarilor a două persoane. Am presupus de asemenea că operator+ indiferent de forma sa de supraîncărcare nu trebuie să modifice cei doi termeni ai adunării, așa cum într-o expresie  $c = a + b$ ,  $a$  și  $b$  rămân nemodificați. În schimb, am supraîncărcat operator+= (compunerea adunării cu o atribuire) pentru cazul în care vrem ca unul din membrii adunării să fie afectat permanent.

În primele două cazuri, operatorul de adunare are semnificația de sporire a salariului cu o sumă dată. Se observă că spre deosebire de cazurile anterioare, salariul este aici de acces public, pentru a putea ușor lista rezultatele aplicării diverselor forme de supraîncărcare ale operatorului +.

Individualizarea funcției care se va apela, având și tipuri diferite de return, se face pe baza tipului și numărului de parametri de apel. Prima variantă, deși avea nevoie de datele a două persoane, primește doar referința uneia, celalaltă fiind primită implicit (prin pointerul *this*), adică este vorba de obiectul căruia aparține funcția însăși. În cea de-a doua variantă, funcția fiind nemembră, primește toate datele ca parametri de intrare, inclusiv referința persoanei ce beneficiază de sporirea salariului.

Rezultatele afișării sunt:

*Dupa spor Adamescu Virgil ar avea 75001*

*Adamescu Virgil chiar are acum 75001 lei.*

*Familia Popa Ion are 172000 lei*

După cum se poate deduce, supraîncărcarea unui operator printr-o funcție membră beneficiază de accesul funcției membru la datele clasei, nemanește transferul obiectului ca parametru în funcție și nici **garantarea drepturilor de acces**. Caracterul nestatic al funcției membru este cerut de necesitatea individualizării de către funcție a fiecărui obiect în parte (pentru a fi tratabil prin operator) și deci ea însăși trebuie să aparțină unui obiect și nu clasei în genere.

Nu trebuie scăpat din vedere, în virtutea formei simplificate în care apar operatorii supradefiniți, că ei ascund de fapt funcții. Astfel, expresia *p3+spor* este de fapt *p3.operator+(spor)*; aceeași expresie scrisă sub forma *spor+p3*, conduce la eroare, deoarece funcția operator nou introdusă aparține și tipului de bază *float*, căruia aparține variabila *spor*, iar adunarea unui flotant cu o clasă nu a fost definită explicit pentru tipurile de bază. Comutativitatea noului operator nu este deci asigurată implicit. Situația s-a rezolvat în programul de mai sus supraîncărcând adunarea persoanelor cu double prin

două funcții *friend* (*persoana+double*, respectiv *double+persoana*); vom vedea ulterior în acest capitol, că mai puteam rezolva această problemă supraîncărcând operatorul de cast, dar semnificațiile ar fi fost altele.

Să studiem acum asociativitatea operatorului. Expresia *p1+p2* ascunde în fapt *p1.operator+(p2)*; ca să afișăm suma salariilor a trei persoane, scriem *p1+p2+p3*; operator+ fiind binar și evaluându-se de la stânga la dreapta, se aplică primelor două persoane și returnează *double*; următorul + înseamnă adunare *double cu persoana* și va fi rezolvat de una din funcțiile *friend* de adunare. Se poate introduce *p4*, o altă persoană cu salariul ei, iar suma salariilor va fi returnată, de expresia *p1+p2+(p3+p4)*, echivalentă cu *p1.operator+(p2) + p3.operator+(p4)*. Sub această formă, este vizibil că + intermediar este o banală adunare de *double*, iar parantezele sunt necesare pentru a controla noi asociativitatea operatorului nou introdus; altfel, asociativitatea adunării în expresia *p1+p2+p3+p4* ar conduce la adunare de două persoane, adunare *double cu persoana* și din nou adunare *double cu persoana*. Efectul este același, dar alte funcții îl realizează.

De remarcat că adunare *double cu persoana* se poate face numai prin funcție *friend*, care nu aparține nici clasei *persoana*, nici tipului *double*. Nu poate aparține clasei *persoana* căci operatorul trebuie să aparțină tot timpul primului operand (*double*, aici); nu poate aparține nici tipului *double*, căci supraîncărcările se fac numai pentru clasele introduse de utilizator nu și pentru cele de bază.

Deoarece nu modifică datele nici unui obiect, comutativitatea este asigurată, *p1+p2* returnând aceeași valoare ca *p2+p1*; dacă însă suma cumulată s-ar depune ca salarul pentru una din persoane, ordinea de apel ar fi esențială.

Semnificația operatorilor trebuie să se păstreze, ca o condiție a polimorfismului; mai mult, efectele trebuie să fie comparabile, adică nu este normal ca *double+persoana* să aibă alte efecte decât *persoana+double*.

Supraîncărcările operator+ ar fi discutabile din acest punct de vedere, dacă o supraîncărcare ar modifica salarul, alta nu; dar cum operator+ în C/C++ cere doar evaluare, nu și modificare de operanzi, pentru varianta care modifică salarul s-a ales pentru supraîncărcarea operatorului *+=*.

### Supraîncărcări ale operatorilor >> și <<

După cum s-a văzut deja, în C++ operatorii >> și << au fost supradefiniți să efectueze operații de intrare / ieșire cu conversii implicate pentru toate tipurile de bază. La definirea unui tip de utilizator (o nouă

clasă), se poate continua supraîncărcarea acestor operatori astfel încât ei să efectueze intrări / ieșiri adecvate și acestui tip.

*cin* și *cout* sunt obiecte flux (*stream*), de tip *istream*, respectiv *ostream* ce se asociază perifericelor standard de intrare, respectiv ieșire, spre care orientăm fluxul informațional. Orientarea operatorilor (<< sau >>) indică direcția de circulație a informațiilor:

**cin >> x;** de la fișierul de intrare standard (tastatura) către variabila *x*;

**cout << x;** din variabila *x* spre fișierul standard de ieșire (monitorul).

Există o clasă de bază *ios* și clase derivate din ea (*istream*, *ostream*, *ifstream*, *ofstream* etc.) specializate în lucru cu diferite fluxuri de intrare / ieșire. Mai multe detalii despre intrări / ieșiri folosind *stream*-uri găsiți în Capitolul 4 – *Operații de intrare / ieșire orientate pe stream-uri*.

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>
class persoana
{
public:
    char nume[30]; int vîrstă;
    friend ostream & operator<< ( ostream &, persoana );
    friend istream & operator>> ( istream &, persoana & );
    friend ifstream & operator>> ( ifstream &, persoana & );
};

ostream & operator<< ( ostream & iesire, persoana p )
{
    iesire <<p.nume << " " << p.vîrstă << endl; return iesire;
}

istream & operator>> ( istream & intrare, persoana & p )
{
    cout <<"Nume: "; intrare>> p.nume;
    cout <<"Vîrstă: "; intrare>> p.vîrstă;
    return intrare;
}

ifstream & operator>> ( ifstream & intrare, persoana & p )
{
    intrare >>p.nume >>p.vîrstă; return intrare;
}

void main()
{
    persoana p1;
    cin >>p1; // incarcare obiect
    cout <<p1; // afisare obiect
    {
        // scriere obiect în fisier pe disc
    }
}
```

```

ofstream fisout("FIS.DAT");
if(!fisout) { cout << "\nEroare de alocare fisier"; exit(1); }
fisout << p1; fisout.close();
}

// citire obiect din fisier pe disc și afișare pe ecran
ifstream fisin("FIS.DAT");
if(!fisin) { cout << "\nFisier negasit!"; exit(1); }
fisin >> p1; cout << "\nS-a citit din fisier: "; cout << p1; fisin.close();
}

```

Prin programul de mai sus, operatorii `<<` și `>>` au fost supradefiniți să expliceze operațiile de intrare și ieșire pe perifericele standard (`cout` și `cin`), precum și pe disc, sub formă de fișiere, pentru noul tip introdus prin clasa `persoana`.

Se observă că fișierul pe disc a fost deschis implicit, de fiecare dată, constructorii claselor putând să preia și această sarcină. Se putea realiza același lucru și apelând explicit funcția membră `fisout.open()`.

Operatorii `>>` și `<<` pentru a realiza operații de intrare / ieșire se supraîncarcă prin funcții *friend* deoarece contextul de folosire impune acest lucru: `cout << p1`; deci, mai întâi trebuie să se primească obiectul flux (`cout`) care este de tip *ostream* și apoi obiectul ce va fi afișat (`p1`) de tip *persoana*, comutativitatea în acest caz nefiind echivalentă. Ne amintim că spre deosebire de o funcție *friend*, o metodă primește totdeauna ca parametru implicit obiectul însuși prin pointerul *this*, ca prim parametru. Din declarațiile, funcțiilor *friend* care supraîncarcă operatorii `<<` și `>>` putem observa că se returnează o referință la obiectul *stream* respectiv. Acest lucru este necesar doar când dorim ca astfel de operații să se realizeze în cascadă: `cout << p1 << p2`; unde, `p1` și `p2` presupunem că sunt două obiecte tip *persoana*.

### Supraîncărcarea operatorului `[ ]`

`operator[ ]` are deja o funcționalitate în cadrul claselor, adresând un element în cadrul unui vector de obiecte. El poate fi supraîncarcat numai prin funcție membră nestatică, rolul său de bază fiind păstrat prin supraîncărcare.

Funcția de supraîncărcare primește un `int`, folosit la reperarea elementului și returnează referință de obiect astfel încât `operator[ ]` să

poată fi folosit în ambele părți ale unei atribuiriri, cum se lucrează și în mod normal.

Prezentăm în continuare câteva motive de supraîncărcare pentru `operator[ ]`.

### 1 Supraîncărcare pentru a verifica încadrarea indicelui în domeniu

Presupunem că avem o clasă care gestionează un vector de `double`, memorat dinamic; clasa ocolește astfel restricția din limbaj, de a lucra numai cu variabile masiv cu dimensiuni constante.

Dimensiunea fiind variabilă, furnizată constructorului și păstrată ca membru în clasă, probabil că merită să verificăm la momentul localizării unui element, încadrarea indicelui în dimensiunea vectorului.

```

#include <iostream.h>
class vector
{
    double * pe;
    int dim; static double err;
public:
    vector(int n = 1) { dim=n; pe=new double[n]; }
    ~vector() { delete[] pe; }
    double & operator[](int i)
    {
        if( i >= 0 && i < dim ) return pe[i];
        else { cout << "Eroare indice: "<< i; return err; }
    }
    double vector::err=0;
};

void main()
{
    vector v(3); vector x[10];
    v[2]=10; cout << v[2];
    x[1][0]=20.; cout << endl << x[1][0];
}

```

### 2 Supraîncărcare pentru a scurta circuita un nivel de adresare

În exemplul de mai sus, trebuie să mai observăm un aspect interesant: `operator[ ]` ne permite să adresăm un element din vector, chiar dacă nu știm cum se numește membrul clasei, care ține adresa primului element; adică putem scrie `v[2]` în loc de `v.pe[2]`, sau și mai clar `*(v.pe+2)`.

Dar atenție mare; astfel supraîncărat, *operator[]* fiind funcție membră permite în plus, și accesul pe zona privată, lucru nepermis în adresările *v.pe[2]* și *\*(v.pe+2)*, care lucrează doar dacă *pe* ar fi public.

De remarcat că problema evitării cunoașterii denumirilor câmpurilor dintr-o structură se menține și dacă vectorul ar fi stocat într-o variabilă obișnuită.

### 3 Supraîncărcare pentru a ascunde prelucrări complexe

Supradefinind *operator[]*, este important să conservăm semnificația sa, aceea de a localiza o instanță și să o folosim într-o formă chiar avansată. Este interesant spre exemplu, să folosim operatorul *[ ]* pentru a localiza o persoana după marca sa, bazându-ne pe o *indexare liniară*.

Vom lucra cu două clase: *persoana* și *bdpers*; clasa *bdpers*, văzută ca o bază de date de persoane, este de tip *server* în sensul că oferă servicii de indexare clasei **client** *persoana*. În acest mod rezolvăm și o altă problemă, aceea că într-un program obiectele se nasc și mor, în funcție de blocul în care au fost definite și nu avem o listă a tuturor obiectelor active la un moment dat, indiferent cum se numește variabila ce conține obiectul.

Din punct de vedere al implementării relației între clase, se poate observa că s-a optat pentru includerea obiectului server în obiectul client.

Indexul va conține toate obiectele existente la un moment dat, chiar sortate după un câmp cheie, încât accesul să fie optimizat. Constructorul clasei *persoana* trebuie să încarce acest index, deoarece doar el știe când a fost creat un obiect și unde a fost stocat, iar deconstructorul va șterge din index referințele obiectelor distruse.

Funcțiile de întreținere index sunt cele clasice de lucru cu liste sortate, memorate ca vectori. Căutarea în index este una de tip binar, conducând la reducerea substanțială a timpului de căutare.

```
#include <iostream.h>
#include <string.h>
class bdpers;
class persoana
{
private:
    friend class bdpers;
public:
    int marca;
    char nume[20];
    persoana(bdpers, char *, int);
    ~persoana();
}
```

```
static int total_pers;
};

class bdpers
{
public:
    friend class persoana;
    static persoana *lista[20];
    static char caut_bin(int , int &);
    void insert( persoana* );
    static void del( persoana * );
    persoana * operator[ ](int);
    static void afis();
};

// inserare persoana în baza de date
void bdpers::insert( persoana* pp )
{
    int i, poz;
    if( ! bdpers::caut_bin(pp->marca, poz) )
    {
        for(i=persoana::total_pers; i>poz; i--)
            bdpers::lista[i] = bdpers::lista[i-1];
        bdpers::lista[poz]=pp;
    }
}

persoana::persoana(bdpers bd, char *n="Anonim ", int m=0) : marca(m)
{
    strcpy(nume,n); bd.insert(this) ; total_pers++;
}

int persoana::total_pers=0;
persoana * bdpers::lista[20];
persoana::~persoana(){ bdpers::del(this); total_pers--; }

// căutare persoana după marca ei
persoana * bdpers::operator[ ](int marca)
{
    int poz;
    if(caut_bin( marca, poz ) ) return lista[poz];
    else return NULL;
}

// căutare binara în vector
char bdpers::caut_bin(int marca, int &poz)
{
    int inc=0, sf= persoana::total_pers, m;
    if(persoana::total_pers==0) { poz = 0; return 0; }
    while( inc <= sf)
```

```

{
    m =(inc+sf)/2;
    if(bdpers::lista[m]->marca == marca) { poz = m; return 1; }
    if(bdpers::lista[m]->marca < marca ) inc= m+1; else sf = m-1;
}
poz = inc;
return 0;
}

// stergere persoana din baza de date
void bdpers::del( persoana *pp)
{
    int poz;
    if( bdpers::caut_bin(pp->marca, poz) )
        for( poz< pp->total_pers-1; poz++)
            bdpers::lista[poz] = bdpers::lista[poz+1];
}

//afisarea persoanelor din baza de date
void bdpers::afis()
{
    int i; cout << endl << endl;
    for(i=0; i<persoana::total_pers; i++)
        cout << "\n " << bdpers::lista[i]->marca << " " << bdpers::lista[i]->nume;
}

void main()
{
    bdpers bd;
    persoana p1(bd, "Popa Ion ",25);
    persoana p2(bd, "cinci", 5), p3(bd, "trei",3);
    {
        persoana p4(bd, "patru", 4);
        bdpers::afis();
        cout << "\n" << bd[5]->nume; cout << "\n" << bd[3]->nume;
    }
    bdpers::afis();
}

```

### Supraîncărcarea operatorilor *new* și *delete*

Operatorii *new* și *delete*, introduși în C++, realizează gestiunea memoriei dinamice într-o manieră specifică programării pe obiecte. Operatorul *new* a fost proiectat să fie aplicabil unui tip individual sau masiv

(formele *ptr\_tip=new tip*, respectiv *ptr\_tip=new tip[n]*), să facă eventual inițializare prin apelul explicit al unui constructor (forma *ptr\_tip=new tip(val\_init)*) și în plus, să returneze pointer la tipul alocat, sau NULL, fără a mai necesita conversie de pointer cu cast.

În cazul claselor introduse de utilizator, *new* și *delete* își păstrează funcțiile lor, putând în plus să fie eventual supraîncărați, prin funcții membre ale clasei sau prin funcții independente.

Funcțiile membre ce supradefinesc acești operatori, fiind de interes general pentru clasa respectivă, sunt automat statice, chiar dacă nu s-a precizat explicit acest lucru. Oricum operatorul *new* nu ar putea fi supraîncărat prin funcție nestatică, deoarece el nu ar putea apartine unui obiect pe care tot el să-l și aloce.

Chiar și supraîncărat, operatorul *new* va conlucra cu constructorul clasei în alocarea și eventual inițializarea unui obiect dinamic. La supradefinire, funcția operator va primi de asemenea dimensiunea zonei de alocat și va returna adresa memoriei alocate:

```
void *operator new( unsigned dim );
```

Practic, la supraîncarcare se crează dubluri ale definițiilor operatorului, fiind recunoscută în paralel și definiția inițială (când operatorul este precedat de rezoluția contextului global `::new`).

În cazul unei **supraîncărcări prin funcție independentă declarată global**, definiția inițială a operatorului *new* nu mai este recunoscută pentru nici un tip, de bază sau clasă, alocarea și gestiunea memoriei dinamice revenindu-i în exclusivitate programatorului.

Operatorul *delete* apelează repetat destrutorul clasei, pentru fiecare membru al masivului.

```
#include <iostream.h>
#include <string.h>
class persoana
{
private:
    int marca;    char nume[20];
public:
    persoana(char n[20]="Anonim ", int m=0)
        { cout << "\nHei, rup ! "; strcpy(nume,n); marca=m; }
    persoana( persoana &p ) { p.marca=0; }
    void *operator new (unsigned nr_pers)
        { return new char[nr_pers * sizeof(persoana)]; }
    void operator delete ( void *p ) { delete p; }
```

```

    char *spune_nume( );
};

char * persoana::spune_nume( ) { return nume; }

void main()
{
    persoana *pp; pp = new persoana[3];
    cout << "\n" << (pp+2)->spune_nume( );
    delete pp;
}

afișează:

```

```

Hei, rup !
Hei, rup !
Hei, rup !
Anonim

```

Dacă pointerul citat în *delete* este nul, operatorul nu face nimic; după o ștergere cu succes, *delete* nu pune pointerul pe nul, lăsând posibilitatea recunoașterii erorilor de încercare de ștergere multiplă a aceleiași zone.

Operatorul *delete* nu acceptă în intrare pointeri de *void*; când un astfel de pointer adresează zone alocate corect, se recomandă conversia prin cast a pointerului, înainte de ștergere.

Folosirea operatorilor *new* și *delete* obligă la o bună cunoaștere a lucrului cu pointeri. Trebuie reamintit că un pointer de obiect poate ține adresa unui obiect sau a mai multor obiecte; invocarea lui *delete* trebuie să fie adecvată situației (*delete px*; pentru cazul în care ștergem un obiect, sau forma *delete [ ]px*; când *px* gestionează mai multe obiecte). În varianta vectorială se recomandă să nu menționăm numărul obiectelor dezalocate, pentru a nu crea dependențe inutile ale programului de diverse valori ale parametrilor; alocatorul de memorie oricum cunoaște dimensiunea zonei de memorie gestionată printr-un pointer.

Supraîncărcările pentru *new* și *delete* funcționează doar pentru obiecte individuale. La alocarea masivelor de obiecte este selectată implicit definiția initială a operatorului, nu eventualele supraîncărcări ale operatorilor simpli. Ea apelează automat constructorul de clasă fără parametri, pentru fiecare element al vectorului; din acest motiv, trebuie să existe totdeauna un constructor fără parametri de apel, sau cu parametri implicați.

Practic, putem supraîncărca *new* și *new[ ]*, respectiv *delete* și *delete [ ]*. Comentând pe rând cele două linii sursă din *main*, putem sesiza cînd se apelează fiecare din versiunile de supraîncărcare, simple sau vectoriale.

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

```

#include <string.h>
class persoana
{
public:
    double salariu;
    persoana(double s=0):salariu(s) { }
    void * operator new( unsigned n )
        { cout << "\n new simplu"; return ::new char[n]; }
    void operator delete(void* p)
        { cout << "\n delete simplu"; ::delete [ ]p; }
    void * operator new[ ]( unsigned n )
        { cout << "\n new vectorial"; return ::new char[n]; }
    void operator delete[ ](void* p)
        { cout << "\n delete vectorial"; ::delete [ ]p; }
};

void main()
{
    persoana *p1=new persoana;     delete p1;
    persoana *p3=new persoana[3];  delete []p3;
}

```

Nu este permisă mixarea celor două mecanisme de alocare / eliberare memorie (cu funcții *malloc()* și *free()*, respectiv cu operatorii *new* și *delete*), adică alocare cu *malloc()* și dezalocare cu *delete*, sau alocare cu *new* și dezalocare cu *free()*.

Pentru zonele alocate pe toată durata programului se recomandă folosirea pointerilor constanti, pentru a preveni modificarea adresei conținută de pointer și pierderea accidentală a legăturii cu zonele de memorie alocate dinamic.

Pentru zonele alocate disparat, dar prelucrate unitar, reamintim utilitatea vectorilor de pointeri la obiecte; putem spre exemplu, nota într-un astfel de vector, adresele tuturor obiectelor existente la un moment dat, în variabile sau în memorie dinamică și avem posibilitatea să le prelucrăm într-un *for*.

### Supraîncărcarea operatorului *cast*

O particularitate aparte o manifestă operatorul *cast* de conversie, a căruia supraîncărcare trebuie să asigure programatorului **posibilitatea conversiilor între obiecte de clase diferite, sau între obiecte și tipurile fundamentale**.

Funcția de supraîncărcare trebuie să fie totdeauna membră a clasei convertite (clasa sursă a conversiei).

Supraîncărcarea operatorului **cast este ușor de recunoscut**, deoarece prototipul funcției menționează **tipul rezultat alături de semnul grafic** al operatorului și **nu înaintea numelui funcției**, ca la toate celelalte funcții.

În exemplul nostru se face o *conversie forțată* a tipului *persoana* în tipul *double*, prin extragerea salariului.

```
#include <iostream.h>
#include <string.h>
class persoana
{
public:
    char nume[20];
    double salariu;
    persoana(char *n="Anonim ",double s=0.) : salariu(s)
        { strcpy(nume,n); }
    persoana( persoana &p ) { p.salariu=0.;}
    operator double( ) { return salariu; }
    double cumul( double s) { return s + salariu ;}
};

void main()
{
    double s=1000000.;
    persoana p1("Popa Ion ",85000.);
    persoana p2("Popa Elena",87000.);
    persoana p3("Adamescu Virgil",75000);
// 1
    cout << "\n" << p3.nume << " are " << (double)p3 << " lei.";
// 2
    cout << "\nFamilia " << p1.nume << " are " << p1.cumul(p2) << " lei";
// 3
    s = p1 + p2;
    cout << "\nFamilia " << p1.nume << " are " << s << " lei";
// 4
    cout << "\nTrei persoane " << " au " << p1+p2+p3 << " lei";
// 5
    cout << "\nCele trei persoane " << " au impreuna "
        << p1.cumul(p2) + p3 << " lei";
// 6
    cout << "\nLe lipsesc 53000. " << " pentru a avea impreuna "
        << p1+p2+p3+53000 << " lei" << endl;
}
```

**Primul afișaj** apelează la o conversie explicită a obiectului *p3* în *double*, judecând într-un fel un om după banii lui.

**Al doilea** aspect ilustrează un cast implicit, deoarece funcția *cumul()* a fost declarată ca primind parametru de tip *double*, iar la apel î se furnizează o persoană! Conversia făcându-se înainte de apelul propriu-zis, constructorul de clasă nu este implicat în copierea vreunui obiect, în vederea transferului, ceea ce se transferă fiind un *double*.

Lucrurile ar sta similar și dacă funcția ar fi una independentă, fără a avea vreo legătură cu clasa persoana, cum are funcția *cumulat()*.

**Al treilea caz** surprinde o conversie tot implicită, impusă pe de o parte de inexistența unei supradefiniri a operatorului **+**, care să știe să adune două persoane, iar pe de altă parte de aducerea la tipul membrului stâng al unei atribuiri.

Chiar și în cazul unei simple expresii *p1+p2+p3*, conversia tot se face, în absența supraîncărcării adunării, lucru care se poate constata din prima formă de afișare a salariului celor trei persoane (**cazul 4**).

**Forma 5**, tot de afișare a salariului celor trei persoane, beneficiază de conversia implicită *persoana* în *double*, impusă prin tipul *double* returnat de funcția *cumulat()*; nexistând adunare *double* cu *persoana*, în ultima etapa de identificare a funcției de apelat se recurge la conversiile indicate de programator prin supraîncărcări ale operatorului *cast*.

Rezultatele rulării sunt următoarele:

```
Adamescu Virgil are 75000 lei.
Familia Popa Ion are 172000 lei
Familia Popa Ion are 172000 lei
Trei persoane au impreuna 247000 lei
Cele trei persoane au impreuna 247000 lei
Le lipsesc 53000. pentru a avea impreuna 300000 lei
```

Puteam pune în clasa *persoana* și un membru *int* virsta, care ne-ar permite încă o supraîncărcare a operatorului *cast*, de genul:

```
operator int( ) { return vîrstă; }
```

Care atunci când se căută un *int* în loc de *persoana*, direcționează conversia spre *int*. Ce se va întâmpla atunci la evaluări de tipul *p1+p2*? Neexistând supraîncărcări pentru adunare de persoane, se vor căuta cast-urile date de programator; din nefericire, se vor găsi două (pentru *double* și pentru *int*) și se va intra în ambiguitate. Putem rezolva ambiguitatea indicând noi explicit compilatorului ce cast să opereze: *s=(double)p1+(int)p2*, ceea ce ne-ar scoate din ambiguitate, dar nu rezolvă logic problema, căci adună salariu cu

vârsta! Acum înțelegem ușor de ce compilatorul nu-și asumă el această sarcină.

Chiar cu două supraîncărcări de cast, apelul *cumul(p)* nu generează ambiguitate, deoarece prototipul lui *cumul* este scris cu *double*, ceea ce direcționează automat compilatorul către cast-ul spre *double*.

Operatorii supraîncărcați prin **funcție nemembră** trebuie să fie declarați ca funcție *friend* în clasă, pentru a avea drepturi de acces direct pe zonele *private* și *protected* ale clasei; altfel ei pot fi implementați apelând la funcții de acces ale clasei pentru a manipula membrei *private* și *protected*, conducând la ineficiență codului executabil, prin apeluri multiple de funcții.

Supraîncărcarea prin funcție *friend* devine obligatorie dacă în stânga operatorului este un tip predefinit, deoarece tipurile predefinite nu permit supraîncărcări în clasa lor.

Dacă se dorește ca un obiect să apară doar în stânga operatorului, supraîncărcarea se poate face prin funcție membră a clasei; dacă obiectul apare și în partea dreaptă a operatorului, atunci poate fi supraîncărat prin funcție membră a clasei aflată în stânga operatorului, sau prin funcție independentă, *friend*. Se justifică această afirmație prin faptul că o funcție membru primește totdeauna obiectul care a declanșat apelul pe prima poziție, prin pointerul *this*. Prin comparație, supraîncărcând operatorul printr-o funcție *friend*, independentă, nu mai este valabilă această constrângere ci programatorul poate să stabilească locul obiectului în lista de parametri după cum dorește.

Supraîncărcările prin funcții *friend* sunt mult mai flexibile; ele lasă compilatorului posibilitatea de a opera mai multe tipuri de conversii, uneori conversii în lanț, pentru a se ajunge la un prototip existent de funcție. Spre exemplu, supraîncarcarea lui *+* prin funcție *friend*, într-o expresie *I + p*, face posibilă atât conversia lui *I* în *persoana*, cât și conversia lui *p* într-un întreg, dar nu ambele simultan. Cu alte cuvinte, o supraîncărcare a operației *+* prin funcție *friend* lasă programatorului posibilitatea ulterioară de a supraîncărca fie operatorul *cast*, fie constructorul, pentru a deservi conversiile.

### Supraîncărcarea operatorului virgulă

Permite evaluarea unei liste de obiecte, returnând referința ultimului obiect din listă. Programul de mai jos marchează trecerea prin *operator ,* ;

parantezele în afișarea din *main* sunt necesare pentru controlul priorității, altfel s-ar evalua flux *ostream*.

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

class persoana
{
    char nume[30]; int vîrstă;
    friend ostream & operator<< ( ostream &ies, persoana p )
        {   ies << p.nume << endl; return ies; }

public:
    const persoana & operator,(const persoana& p) const
        {   cout << "\n\operator,\n"; return p; }
    persoana(char *n) { strcpy(nume,n); }

};

void main()
{
    persoana p1("p1"), p2("p2"),p3("p3");
    cout << (p1,p2, p3);
}
```

După evaluarea listei, programul afișează obiectul *p3*.

### Supraîncărcarea operatorului funcție

**Operatorului funcție** se supraîncarcă prin funcție membră și este singurul operator care poate avea orice număr de parametri.

El oferă o modalitate elegantă de transfer al unei funcții ca parametru într-o altă funcție, simplificând sintaxa; în loc să transferăm pointer de funcție se transferă un obiect, care la momentul folosirii se transformă în funcție.

```
#include <iostream.h>
class less
{
public:
    bool operator() (int a, int b) {return a<b; }

};

bool f(int a, int b) {return a>b ;}

typedef bool FB(int, int);
typedef FB *PFB ;
```

```

class greater
{
    public:
        PFB operator() () { return f; }
        operator PFB () { return f; }
};

bool compara ( less mm, int a, int b) { return mm(a,b); }
bool compara ( bool (*pf)(int,int), int a, int b) { return (*pf)(a,b); }

void main()
{
    int a=1,b=2;
    less mai_mic; greater mai_mare;
    cout << mai_mic(a, b)<< endl;
    cout << compara(mai_mic, a,b)<< endl<< endl;
    cout << compara(mai_mare(), a,b)<< endl;
    cout << compara(mai_mare, a,b)<< endl;
}

```

Programul de mai sus definește două obiecte care la nevoie se transformă în funcție. Obiectul de tip *less*, având supraîncărcat *operator()*, poate fi folosit pe post de funcție, sub forma *mai\_mic(a, b)* sau transferat ca parametru într-o funcție este definită să primească astfel de parametri (prima versiune a funcției *compara*).

Pentru a înțelege cum se comportă obiectele de tip *greater* reamintim rolul pointerilor de funcții la transmiterea unei funcții ca parametru într-o altă funcție. Pentru simplificarea descrierilor s-au introdus progresiv tipurile FB - funcție ce primește doi întregi și returnează *bool* și apoi PFB - pointer la o astfel de funcție:

```

typedef bool FB(int, int);
typedef FB *PFB ;

```

Apoi obiectele de tip *greater* au fost înzestrate cu două supraîncărcări:

- ① una de *operator()*, care le permit să se transforme, la cerere, în pointer de funcție ce primește *int* și *int* și returnează *bool*; astfel obiectul devine transportor de funcție (*f*) către o altă funcție (*compara*):

```
compara( mai_mare( ), a,b);
```

parantezele interioare devin obligatorii, pentru a semnala invocarea operatorului funcție, fără parametri de intrare;

- ② alta, pentru operator *cast*, care înstruiește obiectul să se transforme la **nevoie** în pointer de funcție ce primește *int* și *int* și returnează *bool*; apelul:

```
compara( mai_mare, a, b );
```

impune această necesitate, deoarece funcția nu e definită să recunoască în intrare și obiecte *greater*, așa că pentru adaptare la prototip obiectele devin pointeri de funcții. Exemplu ne ajută să facem distincția între *operator()* și *cast*.

### Supraîncărcarea operatorului ->

Folosește la implementarea unor mecanisme de adresare cu pointeri a unor zone de memorie necontigui, scurtcircuitând nivele de adresare. Ne putem imagina o populație de *total\_pers* persoane, gestionată prin vector de pointeri ; obiectele în sine rezidă împrăștiate la diverse adrese de memorie, în funcție de disponibilitățile existente la un moment dat. Vectorul de adrese se comportă ca un container, dar adresarea unui obiect presupune localizarea adresei în container, apoi încă un nivel de adresare pentru a ajunge la nivel de obiect ; în plus, pe nivelul de mijloc trebuie să realizăm și incrementarea pointerului, pentru a ne deplasa în vector.

Putem construi un obiect numit **iterator**, care ține adresa containerului sau este declarat obiect membru al unui container, având supraîncărcări *operator->* și *operator++* pentru a realiza referirea unui element din container și respectiv, deplasarea în cadrul containerului.

```

#include <iostream.h>
#include <string.h>
#include <stdio.h>
class Persoana
{
public:
    Persoana(char *nm="Anonymus") { strcpy(ume,nm); }
    char ume[50];
};

class Container
{
    static Persoana * vpp[100];
    static int total_pers;
public:

```

```

const static int dim;
Container()
{
    total_pers = 0;
    memset(vpp,NULL,dim * sizeof(Persoana*));
}
static void add()
{
    if(total_pers >= dim) return;
    char aux[50];
    sprintf(aux,"Pers_%3d ", total_pers);
    vpp[total_pers++] = new Persoana( aux);
}
friend class Iterator;
};

const int Container::dim = 100;
int Container::total_pers= 0;
Persoana * Container::vpp[100];

class Iterator
{
    Container* grup ;
    int index;
public:
    Iterator(Container* pop)
    {   index = 0;     grup = pop; }

    int operator++()
    {
        if((grup->vpp[+index] == NULL)|| (index >=  grup->dim) )
            return 0;
        else return 1;
    }
    Persoana* operator->() const
    {
        if(grup->vpp[index])      return grup->vpp[index];
        static Persoana nil;       return &nil;
    }
};

void main()
{
    const int nrp = 10;
    Container pop;
    for(int i =0;i < nrp;i++)
        pop.add();
}

```

```

Iterator sp(&pop);
do { cout << sp->nume << endl; } while(++sp);
}

```

Programul de mai sus crează containerul *grup*, îl populează pe măsura generării obiectelor prin memorie și-i asociază iteratorul *sp* « smart pointer ». Într-o instrucțiune repetitivă, cu ajutorul iteratorului se parcurg elementele containerului într-o formă simplă, lizibilă, dar care se bazează pe supraîncărcări dificil de înțeles și de realizat de către începători.

## 2.4 Conversii între obiecte de diferite tipuri

Un aspect aparte îl reprezintă **conversia unui obiect în alt obiect**, deoarece ca presupune folosirea unuia dintre constructorii claselor. Practic, se pot alege ca variante de lucru:

- ❶ **supraîncărcarea constructorului clasei rezultate**, pentru a onora conversiile implicate sau explicate;
- ❷ **supradefinirea operatorului cast al clasei sursă** printr-o funcție care să returneze un obiect de tipul clasei destinație.

Ambele modalități dovedesc strânsa corelație care există între conversii și supraîncărcarea operatorilor și funcțiilor, inclusiv a constructorilor de clasă.

Vom ilustra cele menționate prin conversii de la clasa *profesor* la clasa *persoana* folosind operatorul cast supraîncarcat în clasa *profesor*.

```

#include <iostream.h>
#include <string.h>
class persoana
{
private:
    int virsta;
public:
    char nume[20];      float salariu;
    persoana(char *n="Anonim ", int v=0, float s=0) : virsta(v),salariu(s)
    { strcpy(nume,n); }
    persoana( persoana &p ) : virsta(p.virsta), salariu(p.salariu)
    { strcpy(nume,p.nume); }
    char *spune_nume( );
};

```

```

char * persoana::spune_nume() { return nume; }

class profesor
{
private:
    char functie[10];
public:
    char nume[20];
    float salariu;
    profesor(char *n=" ", char *f="", float s=1) : salariu(s)
        { strcpy(nume,n); strcpy(functie,f); }
    operator persoana()
    {
        persoana p;
        strcpy(p.nume,nume); p.salariu =salariu;
        return p;
    }
};

void main()
{
    profesor pr1("Vasilescu Gh.", "profesor", 150000.);
    persoana p1;
    cout << "\nLa inceput " << p1.spune_nume()
        << " are " << p1.salariu << " lei!";
    cout << "\nProf. " << pr1.nume << " are " << pr1.salariu << " lei !";
    p1=pr1;
    cout << "\nPersoana " << p1.spune_nume()
        << " are " << p1.salariu << " lei!";
}

```

Programul afișează:

```

La inceput Anonim are 0 lei!
Prof. Vasilescu Gh. are 150000 lei !
Persoana Vasilescu Gh. are 150000 lei!

```

Esențială este prezența constructorului de copiere al clasei persoana (destinația conversiei), deoarece el este implicat în obținerea formei finale a obiectului rezultat din conversie.

Rezultate identice prin rularea aceleiasi funcții *main()* se obțin și după o supraîncărcare a constructorului persoana, pentru a recunoaște în intrare și tipul *profesor*. De reținut condițiile care apar în ordinea de definire a celor două clase și care ne obligă la explicitarea constructorului în afara clasei, după cunoașterea definiției complete a clasei *profesor*:

```

class profesor;
class persoana
{
private:
    int vîrstă;
public:
    char nume[20];      float salariu;
    persoana(char *n="Anonim ", int v=0, float s=0) : vîrstă(v), salariu(s)
        { strcpy(nume,n); }
    persoana( profesor &prof );
    char *spune_nume();
};

class profesor
{
private:
    char functie[10];
public:
    char nume[20];
    float salariu;
    profesor(char *n=" ", char f[10]="", float s=1) : salariu(s)
        { strcpy(nume,n); strcpy(functie,f); }
};

persoana::persoana( profesor &prof ):salariu(prof.salariu)
    { strcpy(nume,prof.nume); }

char * persoana::spune_nume() { return nume; }

Cele două variante prezentate se exclud reciproc, prezența lor simultană generând ambiguitate sub multe versiuni de compilatoare.

```

Pentru a lămuri mai bine acest tip de ambiguitate și diferențele moduri de rezolvare a ei, să urmărим împreună programul următor.

```

#include <iostream.h>
class X;
class Y
{
    int y;
public:
    Y(int n=0):y(n) { }
    explicit Y(X);
};

class X
{
public:

```

```

int x;
operator Y() { cout << "\nY cast"; return Y(x); }

Y::Y(X ox) : y(ox.x) { cout << "\nY cons"; }
void f(Y oy) { }
void main()
{
    X ox; Y oy;
    oy = ox;
    f( Y(ox) );
    f( (Y)ox );
}

```

Ambiguitatea se produce deoarece există două căi de obținere a unui obiect Y dintr-un obiect X, prin constructor (care este calea uzuală, de obținere a oricărui obiect), dar și prin cast supraîncărcat în clasa X, ca în figura 2.1.

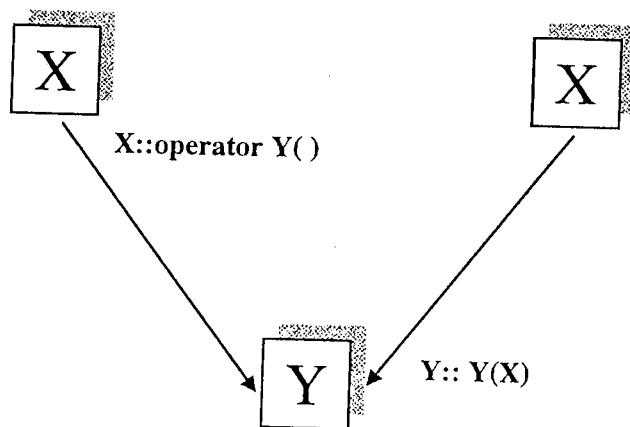


Fig. 2.1 Conversii între obiecte de clase diferite

Dacă în *main* scriem *f( ox )* în loc de *f( Y(ox) )* se intră pe *cast*, deoarece **explicit** interzice conversiile implicate prin constructor (chiar prioritare). Puteți încerca diferite combinații, comentând pe rând operatorul *cast* sau constructorul *Y( X )* și observați ce se apelează pentru diverse forme de apel *f( ox )*, *f( Y(ox) )* sau *f( (Y)ox )*. Încercați același lucru eliminând cuvântul cheie *explicit*, din declarația constructorului *Y(X)*.

În unele implementări (Visual C++ 6.0), ieșirea din ambiguitate se face apelând cu prioritate constructorul, iar dacă acesta nu există se caută o conversie prin *cast*.

Așadar **constructorul** rezintă primul nivel de conversie între obiecte; el acționează drept **convertor** doar dacă are un singur parametru de intrare; el nu apare drept convertor, pentru că obiectul din mai multe componente de intrare.

În calitate de convertor, constructorul poate fi invocat pentru conversii implicate, fără să sesizăm ușor acest lucru. Pentru a inhiba aceste conversii, uneori neavizate, constructorul poate fi declarat folosind modificatorul *explicit*:

#### **explicit persoana ( profesor & )**

Declarația trebuie interpretată astfel: nu se operează conversii din *profesor* în persoană, pentru adaptare la prototip, decât dacă se solicită expres acest lucru: *f( (persoana) pr1 )*, prin program.

Conversiile bazate pe constructor nu operează către tipurile de bază, ci doar dinspre tipurile de bază către cele de utilizator. Explicația este simplă: tipurile predefinite au comportament de uz general, ce nu poate fi modificat, căci ar crea confuzii. Pentru conversiile către tipurile predefinite singura modalitate rămasă este cea bazată pe *cast*.

Se poate observa de asemenea, că supraîncărcările operatorilor au prioritate în raport cu conversiile de tip; la acestea se apelează doar *în extremis*, ca o soluție de compromis; adică se caută prototipuri existente care să corespundă unui apel și numai dacă nu se găsește unul convenabil, se operează conversii de adaptare la prototipurile cunoscute.

Programul de mai jos evidențiază alt tip de ambiguitate apărută în legătură cu conversia de obiecte. Clasa X dispune de *cast* către alte două tipuri: Y și Z; mai există și două funcții *f()*, care lucrează pe tipurile Y și Z. Totul este în ordine până în momentul în care *f()* se apelează cu obiecte de tip X, lucru ce antrenează conversii prin *cast*, neexistând constructori de tip convertor. Compilatorul semnalează eroare de ambiguitate, neputând alege între cele două forme ale lui *f()*, susținute prin două *cast*-uri echivalentiale.

```

#include <iostream.h>
class Y
{
public:
    int y;
    Y(int n = 0): y(n) {}
};

```

```

class Z
{
    int z;
public:
    Z(int n=0):z(n) {}
};

class X
{
public:
    int x;
operator Y() { cout << "\nY cast"; return Y(x); }
operator Z() { cout << "\nZ cast"; return Z(x); }
};

void f(Y oy) {}
void f(Z oz) {}

void main()
{
    X ox;
//    f( ox );
}

```

Eliminarea ambiguității se face cerând explicit un cast, sub forma `f( ( Y )ox )` sau `f( ( Z )ox ).`

O generalizare deosebit de interesantă se obține prin extinderea conversiilor în cazul folosirii unui operator supradefinit într-o clasă, pentru a opera asupra altrei clase. În acest sens, vă prezentăm o conlucrare a două clase, *fisier* și *pozitie* în fișier, care asigură adresarea în fișier după modelul lucrului cu vectori. Astfel, o expresie de forma `f[i] = c` trebuie interpretată ca o scriere a unui caracter în fișier pe poziția *i*, iar o expresie `f[i] = sir`, ca memorare a unui sir în fișier, începând cu poziția *i*. Reciproc, `c=f[i]` va semnifica preluarea unui caracter din fișier, de la poziția *i*.

Clasa *fisier* conține de fapt pointerul la structura de tip *FILE* prin care se asigură gestiunea unui fișier, iar la generarea unui obiect de acest tip se face și deschiderea fișierului deja existent. Obiectul *fisier* este instruit să se autopozitioneze atunci când este asociat cu un *operator[ ]* sub forma `f[i]`, furnizând în același timp în ieșire un obiect *pozitie*. Obiectele de tip *pozitie* continuă munca, ele știind să convertească o poziție într-un caracter, datorită suprascrierii operatorului cast: *operator const char( )*.

Tot în clasa *pozitie* a fost suprascris în mai multe variante *operator=*, astfel încât un caracter sau un sir de caractere să poată fi atribuit unei poziții, lucru ce se traduce printr-o scriere la o poziție dată.

```

#include <iostream.h>
#include <stdio.h>
#include <process.h>

class fisier;
class pozitie
{
private: fisier *pof; long poz;
public:
    friend fisier;
    pozitie(fisier &f, long p): pof(&f), poz(p) {}
    void operator=(char c);
    void operator=(char *);
    operator const char();
};

class fisier
{
private: FILE *pf;
public:
    friend pozitie;
    fisier( const char *nume)
    {
        pf=fopen(nume,"w+");
        if(!pf) { cout << "\nEsuare deschidere fisier"; exit(1); }
    }
    ~fisier() { fclose(pf); }
    pozitie operator[](long p)
    { fseek(pf,p,SEEK_SET); return pozitie(*this,p); }
};

void pozitie::operator=(char c) { if( pof->pf ) putc(c,pof->pf); }
void pozitie::operator=(char *) { if( pof->pf ) fputs(s,pof->pf); }
pozitie::operator const char()
{ if( pof->pf ) return getc(pof->pf); return EOF; }

void main()
{
    fisier f("fisvect.dat");
    int i; char c;
    for(i=0;i<5; i++) f[i]=i+'0';
    f[5]='A'; f[6]="BBBBBBBBBB";
    cout << "\nPrimii 16B sunt: ";
    for(i=0;i<16; i++) { c=f[i]; cout << c; }
}

```

## 2.5 Aplicații

### 1 Un exemplu simplu: clasa complex !

Paradoxal, poate semn al evoluției gradului de percepție a structurilor abstracte, clasa complex nu e complexă, ci simplă. Cel puțin în majoritatea implementărilor, obiectele complex nu au membri stocați ca extensii în memoria dinamica, iar operatorii au acceptiuni larg recunoscute. Am ales ca exemplificare această clasă tocmai pentru a valorifica lucrurile deja cunoscute despre această tipologie, în ideea învățării de lucruri noi, aplicabile și altor clase.

```
#include <iostream.h>
#include<math.h>

class complex
{
private:
    double Re,Im;
public:
    complex(double r=0., double i=0.) : Re(r), Im(i){ }
        // constructor cu valori implicate
    friend ostream & operator<<( ostream &, complex );
        // afisare numar complex
    complex operator+(complex z)
    {
        return complex(Re + z.Re, Im + z.Im);
        // adunare folosind apel de constructor
    }
    complex operator-(complex z)
    {
        complex t; t.Re=Re-z.Re; t.Im=Im-z.Im; return t;
        // scadere folosind un obiect temporar
    }
    complex operator-()
    {
        return complex(-Re, -Im);
        // schimbare de semn
    }
    complex operator~() { return complex(Re,-Im); }
        // conjugare numar complex
    complex operator*(complex z)
    {
        return complex ( Re*z.Re-Im*z.Im, Re*z.Im+Im*z.Re );
        // produs de numere complexe
    }
    double operator()( ) { return sqrt(Re*Re+Im*Im); }
}
```

```
// modulul numarului complex ca operator unar
operator double(){ return !(*this); }
    // cast spre double, prin preluarea modulului
complex operator/(complex z)
{
    return complex( (Re*z.Re+Im*z.Im)/(z.Re*z.Re+z.Im*z.Im),
                    (-Re*z.Im+Im*z.Re)/(z.Re*z.Re+z.Im*z.Im) );
}
complex operator*(double n) // inmultire cu un scalar
{
    return complex ( Re*n, Im*n );
}
complex operator/(double n) // impartire cu un scalar
{
    return complex ( Re/n, Im/n );
}
// se poate și return complex(*this * ~z / (z * !z));
    // dar trece prin constructor de copiere !
complex operator+=(complex z)
{
    return *this = *this + z;
    // atribuire compusa, definită prin operațiile simple
}
friend bool operator==(complex z1, complex z2)
{
    return (z1.Im-z2.Im < 0.00001) &&
           (z1.Re-z2.Re <0.00001) ? 1:0;
}
friend bool operator!=(complex z1,complex z2)
{
    return !(z1==z2);
}
ostream & operator<<( ostream &out, complex z )
{
    out << z.Re << (z.Im<0 ? "-" : "+") << z.Im << "i"; return out;
}
```

Pentru supraîncarcarea operatorilor uzuali, s-au folosit atât forme în care constructorul e invocat pe return (vezi operator+), cât și forme clasice, în care se alocă mai întâi un obiect temporar; se calculează apoi elementele lui, după care obiectul este returnat (vezi operator-).

De remarcat că **operator-** apare în două ipostaze: ca operator **binar** în operația de scădere și ca operator de semn, deci **unar**. Ambele versiuni fiind supraîncărcări prin funcții membre, doar unul din parametri se transferă explicit pentru operatorul binar, în timp ce pentru operatorul unar, singurul operand se transferă implicit.

Pentru resolosirea codului scris operator/( ) poate fi introdus și ca înmulțire cu conjugatul, corectat cu pătratul modulului. De asemenea, uzând de supraîncarcare, a fost definită și împărțirea cu un scalar; analog putea fi definită și înmulțirea cu scalari.

Dintre operatorii unari au fost supraîncărcați **operator~()** cu semnificația de conjugare, respectiv **operator!()** pentru calculul modulului numărului complex.

De remarcat că nu puteam alege **|** pentru a simboliza modulul deoarece acesta este operator binar, iar la supraîncărcare trebuie respectată cardinalitatea operatorului.

Pe baza modulului, s-ar putea introduce și operatorul de cast în **double**, acceptând că atunci când un număr complex apare pe o poziție în care se cere un număr real, să i se aplique o conversie în real, prin modulul său.

Operatorii de atribuire compuse se pot introduce pe baza compunerii dintre o atribuire simplă și operația cu care se compune (vezi operator **+=( )**, introdus sub forma **\*this = \*this + z;** care modifică conținutul obiectului curent).

O problemă care s-ar putea pune aici este cea referitoare la aplicarea în cascadă a atribuirilor sau a atribuirilor compuse, sub forma **z = z1 = z2;** ușor se poate constata că definirile introduse de noi, compuse cu un operator **=()** implicit (copiere bait cu bait a numărului complex, la o altă locație) răspund acestor cerințe.

În ce privesc comparațiile între numere complexe, trebuie avute în vedere cel puțin două aspecte:

- mulțimea numerelor complexe nu este total ordonată;
- reprezentarea pe **double** a celor două componente ale numărului complex ar putea dифeri ușor la ultimele zecimale atunci când se pornește de la forma trigonometrică față de cazul când se pornește de la forma algebrică.

În virtutea celor de mai sus s-a optat aşadar pentru supraîncărcarea operatorilor de **==** și **!=**, iar maniera în care s-a făcut se bazează pe acceptarea ca egale a două numere care au primele cinci zecimale egale, pentru părțile lor reale, respectiv imaginare.

În foarte multe cazuri se lucrează cu forma trigonometrică a numărului complex **z = r(cos t + i sin t)**, unde **r** este modulul, iar **t** argumentul numărului complex.

Ne-ar fi util un constructor bazat pe modul și unghi, dar acestea fiind tot de tip **double**, ar intra în ambiguitate cu constructorul **complex(double, double)** deja existent. Nu e recomandat nici un constructor care să primească unul din parametri (unghiul) ca **int**, deoarece în unele situații dăm constantele **double** fără . dacă nu au parte zecimală; acest lucru ar putea conduce la

confuzii (considerându-se că numărul complex e dat trigonometric, deoarece al doilea double e dat ca **int**) și la obiecte incorect inițializate.

Pentru a individualiza noul constructor s-a preferat „marcarea” lui, printr-un parametru de tip introdus de utilizator: **unghi**. În esență el este tot un **double**, dat de singurul lui membru. Pentru a evita cunoașterea numelui datei membre (**n**) și pentru a scurta circuita acest nivel introdus superficial, noul tip a fost înzestrat și cu operatori de cast spre **double**, invocat automat de cele mai multe ori, care doar furnizează în afară conținutul, ca la o funcție de acces.

```
#define PI 3.141592
class unghi
{
public:
    double u;
    unghi(int v):u(PI*v/180.) {}
    operator double() { return u; }
};
```

S-a profitat de introducerea noului tip scriind și un constructor de **unghi** care primește unghiu în grade sexagesimale, deși în esență conținutul este stocat în radiani așa cum este cerut de funcțiile trigonometrice din **<math.h>**.

În clasa **complex** putem introduce acum și un constructor pornind de la forma trigonometrică a numărului complex:

```
complex(double r, unghi t) : Re(r*sin(t)), Im(r*cos(t)) {}
    // constructor pornind de la forma trigonometrică
```

Un posibil program principal de testare ar arăta astfel:

```
void main()
{
    complex z,z1(1., -31.), z2(1., -1.), v, k;
    double m; unghi u=30;
    z1=complex(1,(unghi)30 );
    z2=complex(0.5, sqrt(3.)/2 );
    //cout << (z1==z2 ? "egale" : "Diferite");
    //cout << (double)z1;
    cout << (z += z1 += z2);
    cout << endl;
}
```

## ② O clasă prezintă aproape peste tot: clasa String

Clasa String este ideală pentru a ilustra lucru cu obiecte cu extensie în memoria dinamică; întreg conținutul șirului este stocat în memorie dinamică, obiectul ținând doar **pointerul** la conținut și **lungimea** șirului. Ca urmare, vom regăsi obligatoriu cele patru elemente specifice obiectelor cu membri pointeri: constructori și destructor expliciti, constructor de copiere și operator=.

Constructorul fără parametri crează un sir vid, ce conține doar terminatorul \0; se putea lucra și cu pointer nul dacă șirul era vid, dar complica inutil algoritmi de lucru cu String, deoarece funcțiile din <string.h> nu acceptă în prelucrare, pointer nul de sir.

Majoritatea operațiilor cu String sunt redirectate către funcțiile specifice tipului *char\**, dar String permite în plus operații precum: atribuiri cu =, redimensionări automate, concatenări și comparații folosind operatori, test de sir vid etc., păstrând totodată facilitatea de localizare a unui caracter prin poziția sa (operator[ ]).

Funcția de acces **setString** este folosită și de constructor pentru a inițializa un sir pornind de la un vector de char. Tot o funcție de acces este și **strlen** (prin analogie cu **strlen**) care furnizează în afară valoarea membrului privat **lg** (lungimea șirului).

```
#include <iostream.h>
#include <process.h>
#include <iomanip.h>
#include <string.h>

class String
{
private:
    int lg;           // lungime
    char *ps;         // pointer la primul caracter
    void setString( const char * ); // functie de acces
    friend ostream &operator<<( ostream &out, const String &s )
        { out << s.ps; return out; }
    friend istream &operator>>( istream &inp, String &s )
    {
        char temp[ 100 ];
        inp >> setw( 100 ) >> temp;
        s = temp; return inp;
    }
}
```

public:

```
String( const char * = "" ); // constructor de sir (eventual vid)
String( const String & );      // copy constructor
~String();                      // destructor

char &operator[]( int );          // extragere caracter
const char &operator[]( int ) const; // incadrare în dimensiune
String operator()( int, int );    // extragere subsir
int strlun() const;              // lungime

const String &operator=( const String & ); // atribuire
const String &operator+=( const String & ); // concatenare

bool operator!() const;          // test de sir vid
bool operator==( const String & ) const;
bool operator!=( const String & s2 ) const
    { return !( *this == s2 ); }

bool operator<( const String & ) const;
bool operator>( const String &s2 ) const
    { return s2 < *this; }

bool operator<= ( const String &s2 ) const
    { return !( s2 < *this ); }

bool operator>= ( const String &s2 ) const
    { return !( *this < s2 ); }

};

String::String( const char *s ) { lg= strlen( s ); setString( s ); }

String::String ( const String &sursa ) : lg( sursa.lg )
    { setString( sursa.ps ); }

const String &String::operator=( const String &s2 )
{
    if ( &s2 != this )
        { delete [ ] ps; lg = s2.lg; setString( s2.ps ); }
    return *this;
}

String::~String() { delete [ ] ps; }

int String::strlun() const { return lg; }

void String::setString( const char *string2 )
{
    ps = new char[ lg + 1 ];
    if( ps ) strcpy( ps, string2 );
}
```

```

const String &String::operator+=( const String &s2 )
{
    char *aux = ps; lg += s2.lg;
    ps = new char[ lg + 1 ];
    if(ps==NULL) { cout << "\nEsuare alocare memorie "; exit(1); }
    strcpy( ps, aux ); strcat( ps, s2.ps );
    delete [] aux; return *this;
}

bool String::operator!() const { return lg == 0; }

bool String::operator==( const String &s2 ) const
{ return strcmp( ps, s2.ps ) == 0; }

bool String::operator<( const String &s2 ) const
{ return strcmp( ps, s2.ps ) < 0; }

char &String::operator[]( int poz )
{
    if( poz >= 0 && poz < lg ) return ps[ poz ];
    else {cout << "\n Indice in afara domeniului \n"; return ps[0]; }
}

const char &String::operator[]( int poz ) const
{
    if( poz >= 0 && poz < lg ) return ps[ poz ];
    else { cout << "\n Indice in afara domeniului \n"; return ps[0]; }
}

String String::operator()( int index1, int index2 )
{
    if( index1 < 0 ) index1=0; // index negativ
    if( index2 >= lg ) index2=lg-1; // depasire index
    int lun=index2-index1+1;
    if(lun<0) // indexi inversati
        { lun=-lun;int aux =index2; index2 = index1; index1=aux; }

    char *aux = new char[ lun + 1 ];
    if( aux )
    {
        strncpy( aux, &ps[ index1 ], lun ); aux[ lun ] = '\0';
        String temp ( aux ); delete [] aux;
        return temp;
    }
    else {cout << "\nEsuare alocare memorie "; exit(2); }
}

```

```

void main()
{
    String s1( "programarea orientata obiect" ), s2( " in c++ " ), s3;
    // test operatori relationali
    cout << "s1 = " << s1 << " s2 = " << s2 << " s3 = \" " << s3 << "\" "
         << "\ns2 == s1 intoarce "
         << ( s2 == s1 ? "true" : "false" )
         << "\ns2 != s1 intoarce "
         << ( s2 != s1 ? "true" : "false" )
         << "\ns2 > s1 intoarce "
         << ( s2 > s1 ? "true" : "false" )
         << "\ns2 < s1 intoarce "
         << ( s2 < s1 ? "true" : "false" )
         << "\ns2 >= s1 intoarce "
         << ( s2 >= s1 ? "true" : "false" )
         << "\ns2 <= s1 intoarce "
         << ( s2 <= s1 ? "true" : "false" );

    // test de sir vid cu operator }()
    if( !s3 ) cout << "\ns3 vid: \" " << s3 << "\" ";
        // test suprîncarcare operator=
    s3 = s1;
    if( !s3 ) cout << "\ns3 nevid: " << s3 << endl;

    // test concatenare cu operator+=
    s1 += s2; cout << "\nConcatenare: << s1;

    // test conversie prin constructor
    s1 += " Editia 2003";
    cout << "\nConcatenare cu apel constructor:\n s1= " << s1 << "\n";

    // test suprîncarcare operator() pentru extragere subsir
    cout << "\n s1(2, 4), este: " << s1( 2, 4 ) << "\n";

    // test extragere subsir cu indici inversati sau in afara domeniului
    cout << "\nSubsir s1(13, 0): " << s1( 13, 0 ) << "\n";

    // test constructor de copiere
    String *psir = new String( s1 );
    cout << "*psir = " << *psir << "\n\n";

    // test lucru cu pointeri de sir si operator=() pe auto-asignare
    cout << "asignare *psir la *psir\n";
    *psir = *psir;
    cout << "*psir = " << *psir << "\n";

    // test destructor
    delete psir;
}

```

```

// test operator[ ]
s1[ 0 ] = 'P'; s1[ 32 ] = 'C';
cout << "\ns1 dupa atribuire la nivel de caracter \n" << s1 << "\n\n";
// test incadrare in indice
cout << "Incercare de atribuire s1[100]= 'x' intoarce: ";
s1[ 100 ] = 'x'; // ERROR: poz out of range
}

```

**operator( )** putând primi oricâți parametri, a fost supraîncărcat să extragă un subșir indicat prin două poziții (date în orice ordine), cu verificarea încadrării în dimensiuni.

**operator[ ]** este supraîncărcat în două variante, cu const și fără const, compilatorul putând face distincția între cele două prototipuri; utilizatorul clasei poate folosi oricare dintre cele două versiuni, deci își poate proteja anumite locații de memorie împotriva suprascrierii, lucrând cu șiruri constante.

Metodele specifice tipului **String** din biblioteca standard C++ sunt mult mai numeroase, dar ne-am mărginit la cele mai importante, pentru a dezvăluui principiile de bază după care au fost create aceste metode. Afisarea rezultatelor de mai jos atestă funcționarea corectă a metodelor clasei **String**.

```

s1 = programarea orientata obiect s2 = in c++ s3 = ""
s2 == s1 intoarce false
s2 != s1 intoarce true
s2 > s1 intoarce false
s2 < s1 intoarce true
s2 >= s1 intoarce false
s2 <= s1 intoarce true
s3 vid: ""
s3 nevid: programarea orientata obiect

```

Concatenare: programarea orientata obiect in c++

Concatenare cu apel constructor:

s1= programarea orientata obiect in c++ Editia 2003

s1(2, 4), este:ogr

Subșir s1(13, 0): programarea

\*psir = programarea orientata obiect in c++ Editia 2003

asignare \*psir la \*psir

\*psir = programarea orientata obiect in c++ Editia 2003

s1 după atribuire la nivel de caracter

Programarea orientata obiect in C++ Editia 2003

Incercare de atribuire s1[100]= 'x' intoarce:  
Indice în afara domeniului

### ③ Clasa fractie pentru operațiile uzuale cu fracții

Clasa introduce obiectul **fractie** ca fiind format din **doi întregi** cu semn; având în vedere că semnul nu poate fi decât + sau -, din considerente de tipărire, îl stochează doar la numărător. Constructorul cu valori implicate crează fracția 0 / 1; când constructorul primește un număr oarecare, îi pune numitor 1, după ce în prealabil îi îndepărtează partea zecimală ! Împărțirea la zero se tratează prin constructorul simulând infinitul prin **MAX\_INT**.

Două funcții utilizare specifice lucrului cu întregi, determină **cel mai mare divizor comun** după algoritmul lui Euclid și **cel mai mic multiplu comun**, ca raport între produsul numerelor și cel mai mare divizor comun. Cele două funcții folosesc la aducerea la același numitor a două fracții.

**Amplificarea și simplificarea** sunt alte două operații uzuale, cărora le corespunde câte o funcție membră. Simplificarea se face cu cel mai mare divizor comun, în timp ce amplificarea primește factorul cu care operează.

Operațiile de bază reproduc varianta ușuală de lucru cu fracții.

Pentru a reduce efortul de programare, a fost introdusă supraîncărcarea **operatorului unar de semn (-)**, iar scăderea a devenit astfel adunare cu opusul fracției. Din aceleași considerente, împărțirea este descrisă ca înmulțire cu inversul fracției.

Pentru operatorii relaționali, se preferă aducerea fracțiilor la double, pentru comparații simple între valorile lor.

```

#include<iostream.h>
#include<math.h>
#define MAX_INT 0x7FFFFFFF

int cmMdc(int a, int b)
{
    if(a<b) cmMdc(b, a);
    if(a%b) return cmMdc(b,a%b); else return b;
}

int cmmmc(int a, int b)
{
    return a*b / cmMdc(a, b);
}

class fractie
{
    int nr,nm;

```

```

public:
    explicit fractie(int n=0,int m=1)
    { if(m<0) m=-m, n=-n; nr = m?n:MAX_INT; nm= (m? m:1); }
    friend ostream & operator<< (ostream & ies, fractie& f)
    { ies << " " << f.nr << "/" << f.nm; return ies; }

    void simplif()
    {
        int factor=cmMdc(abs(nr),abs(nm));
        nr/=factor, nm/=factor;
    }

    void amplif( int factor)
    { nr *= factor ; nm *= factor; }

// adunarea a doua fractii
    fractie operator+ (fractie f2)
    {
        fractie f1= *this,rez; f1.simplif(); f2.simplif();
        int mc=cmmmc(f1.nm,f2.nm);
        f1.amplif( mc/f1.nm); f2.amplif(mc/f2.nm);
        rez.nr=f1.nr+f2.nr; rez.nm=mc; rez.simplif();
        return rez;
    }

// schimbare de semn
    fractie operator-() { return fractie(-nr, nm); }

// scaderea ca adunare cu -f2
    fractie operator- (fractie f2) { return fractie(*this + (-f2));}

// inmultirea a doua fractii
    fractie operator* (fractie f2)
    {
        fractie temp;
        temp.nr=nr*f2.nr;temp.nm=nm*f2.nm;
        temp.simplif();
        return temp;
    }

// impartire prin inmultire cu inversa
    fractie operator/ (fractie f2) { return *this * fractie(f2.nm,f2.nr);}

    bool operator<= (fractie f2)
    { return ((double)nr/nm) <= ((double)f2.nr/f2.nm) ? 1:0; }

    operator double() { return (double)nr/nm; }
};

```

Metodele definite până acum sunt suficiente pentru cele mai uzuale operații; chiar și adunarea unei fracții cu un număr este posibilă, când constructorul nu poartă atributul explicit, pentru că se va intra automat prin constructor, cu transformarea numărului în fracție.

Nu se pot efectua însă operații între un număr și o fracție; pentru acoperirea acestor situații a fost introdus un operator cast spre double, care transformă astfel de operații în simple operații pe *double*. Pentru a nu intra în ambiguitate la efectuarea unor conversii ( prin cast și prin constructor), constructorul a fost declarat explicit, conversia prin constructor făcându-se doar când programatorul cere explicit acest lucru.

Un posibil program de testare a clasei **fractie** este următorul:

```

void main(int argc, char* argv[])
{
    fractie f1(3,12), f2(5,14);

    cout<<endl << f1 << " + " << f2 << endl ;
    cout<<endl << f1 << " - " << f2 << endl ;
    cout<<endl << f1 << " * " << f2 << endl ;
    cout<<endl << f1 << " / " << f2 << endl ;
    cout<<endl << 2 << " + " << f1 << endl ;
    cout<<endl << f1 << " + " << 2 << endl ;
    cout<<endl << f1 << " + " << (fractie)2 << endl ;
                                            << f1+(fractie)2 << endl ;

    f1.simplif();f2.simplif();
}

```

Rezultatele afișate de programul de mai sus:

$$\begin{aligned}
 \frac{3}{12} + \frac{5}{14} &= \frac{17}{28} \\
 \frac{3}{12} - \frac{5}{14} &= -\frac{3}{28} \\
 \frac{3}{12} * \frac{5}{14} &= \frac{5}{56} \\
 \frac{3}{12} / \frac{5}{14} &= \frac{7}{10} \\
 2 + \frac{3}{12} &= 2.25 \\
 \frac{3}{12} + 2 &= 2.25 \\
 \frac{3}{12} + \frac{2}{1} &= \frac{9}{4}
 \end{aligned}$$

confirmă conversiile explicite prin constructor și cele implicate prin cast.

#### ④ Clasa BigInt pentru operații cu întregi mari

După cum se știe, întregii de tip **long** pot ajunge până la valoarea +2147483647, respectiv 4294967295 ( 0xFFFFFFFF ) fără semn. Tipul **double** poate ajunge până la numere cu 15 – 16 poziții în sistemul zecimal.

Pentru a lucra cu valori mai mari, programatorul trebuie să preia sarcinile unei aritmetici extinse; noi o vom face definind clasa **BigInt** care stochează numere întregi pozitive, de dimensiuni mari, sub formă de sir de cifre. Cifrele vor fi ținute într-un vector de char, **cifra[NR\_CIF]**, dar în reprezentare internă, nu în ASCII; în acest fel, la nivel de cifră, calculele se pot face direct, fără nici o conversie prealabilă. Ordinea de stocare este de la stânga la dreapta, adică cifrelor de rang mare le corespund elementele de început ale vectorului. Numărul cifrelor semnificative ale unui obiect se determină cu un apel **nr\_cifre()**.

```
#include<iostream.h>
#include<iomanip.h>

#include<string.h>
#define NR_CIF 100

class BigInt
{
    signed char cifra[NR_CIF];

public:
    BigInt(long val=0);
    BigInt (char *);

    BigInt operator+ (BigInt &);

    BigInt operator+ (int);

    BigInt operator+ (char *);

    BigInt prod_1( signed char );

    void deplas(int);

    int nr_cifre();

    BigInt operator* (BigInt &);

    friend ostream &operator<<(ostream&,BigInt&);

};

BigInt::BigInt(long val)
{
    register i;
    for(i=0;i<NR_CIF;i++) cifra[i]=0;
    for(i=NR_CIF-1;val && i>=0;i--)
        val=cifra[i]+val*10;
}
```

```

        cifra[i]=val%10, val/=10;
    }

    BigInt::BigInt(char * sir_cif)
    {
        register i,j;
        for(i=0;i<NR_CIF;i++) cifra[i]=0;// cifrele in exces sunt 0
        for(i=NR_CIF-strlen(sir_cif),j=0; i<NR_CIF ;i++,j++)
            cifra[i] = sir_cif[j]-'0'; // cifrele normale se tin in cod intern
    }

    BigInt BigInt::operator+ (BigInt &bi2)
    {
        BigInt temp,int carry=0,i;
        for(i=NR_CIF-1;i>=0;i--)
        {
            temp.cifra[i]=cifra[i]+bi2.cifra[i]+carry;
            if(temp.cifra[i]>9)
                { temp.cifra[i]%=10; carry=1; }
            else
                carry=0;
        }
        return temp;
    }

    BigInt BigInt::operator+ (int n) { return *this + BigInt(n); }

    BigInt BigInt::operator+ (char* sir_cif)
    { return *this + BigInt(sir_cif); }

    BigInt BigInt::prod_1( signed char cif)
    {
        BigInt aux = *this; char carry=0, i;
        for(i=NR_CIF-1;i>=0;i--)
        {
            aux.cifra[i]=aux.cifra[i]*cif + carry;
            if(aux.cifra[i]>9)
                { carry=aux.cifra[i]/10;aux.cifra[i]%=10; }
            else
                carry=0;
        }
        return aux;
    }

    void BigInt::deplas(int n)
    {
        int i, j;
```

```

if(n > 0 && n<NR_CIF )
{
    for( i=0, j=n; j< NR_CIF ; i++,j++)
        cifra[i]=cifra[j];
    for( ; i< NR_CIF ; i++)cifra[i]=0;
}

int BigInt::nr_cifre()
{
    for(int i=0;cifra[i]==0; i++);
    return NR_CIF-i;
}

BigInt BigInt::operator* (BigInt & b2)
{
    int nr_cif, nr_poz, i;BigInt rez;
    nr_cif=b2.nr_cifre();
    for(i=NR_CIF - 1,nr_poz=0 ; i>=NR_CIF -nr_cif;i--)
    {
        BigInt aux = prod_1(b2.cifra[i]);aux.deplas(nr_poz);
        cout<<"\nxx"<<setw(50-aux.nr_cifre() )<<" "<< aux;
        nr_poz++;
        rez=rez+aux;
    }
    return rez;
}

ostream &operator<<(ostream& out,BigInt& bi)
{
    register i;
    for (i=0; (bi.cifra[i]==0 )&&i<NR_CIF;i++);
    if(i==NR_CIF)
        out<<0;
    else
        for(;i<NR_CIF;i++) out<< (char)(bi.cifra[i] +'0');
    return out;
}

void main()
{
    BigInt b1("12345678901234567890"),b2(9999);
    cout << "\nb1 = " << b1 << endl;
    cout << "\n" << b1 << " + " << b2 << " = " << b1+b2 << endl;
    b1="123459789";
}

```

```

b2="1999999";
cout << "\n" << b1 << " * " << b2 << " = " << b1*b2 << endl ;
}

```

Constructorii de clasă generează obiecte pornind de la întregi obișnuite (implicit 0) sau de la numere date ca text, cu maxim NR\_CIF cifre.

Calculele reproduc varianta manuală de lucru, adică operând cifră de cifră și ținând minte cifra de transport. Adunarea introdusă prin **operator+** recunoaște în intrare doi întregi mari, un întreg mare și unul mic, sau un întreg mare și altul dat ca text. Din supraîncărcări se observă că toate adunările sunt până la urmă redirectate cu ajutorul constructorilor către adunarea de două obiecte BigInt. O parte din redirectări se făceau și implicit deoarece constructorii cu un parametru sunt folosiți de compilator drept convertori, pentru a încerca identificarea unui prototip pentru **operator+**.

Operația de înmulțire se realizează în două etape; **prod\_1** face înmulțirea unui obiect BigInt cu o cifră și obține tot un obiect BigInt; operația se repetă pentru fiecare cifră a înmulțitorului, obiectele auxiliare rezultate fiind deplasate corespunzător spre stânga (funcția **deplas()**) și adunate la rezultatul final. Pentru facilitarea înțelegерii în supraîncărcarea **operator\***, a fost pusă o afișare intermedieră.

*b1 = 12345678901234567890*

*12345678901234567890 + 9999 = 12345678901234577889*

<i>xx</i>	<i>1111138101</i>
<i>xx</i>	<i>11111381010</i>
<i>xx</i>	<i>111113810100</i>
<i>xx</i>	<i>1111138101000</i>
<i>xx</i>	<i>11111381010000</i>
<i>xx</i>	<i>111113810100000</i>
<i>xx</i>	<i>123459789000000</i>
<i>123459789 * 1999999 =</i>	<i>246919454540211</i>

## 8 Clasa Date pentru manipularea datei calendaristice

În general, bibliotecile de clase conțin și clase care permit gestiunea timpului și a datei calendaristice; aceste definiții cooperează cu elemente ale sistemului de operare care posedă și întrețin informații despre timp.

Exemplul de mai jos își propune să ilustreze cum se poate derula astfel de cooperare între funcțiile sistem și metodele clasei *Data*.

Conținutul informațional de bază al clasei se reduce la întregii zile luna, anul, dar există multe metode specifice.

Constructorul fără parametri cere data sistemului de operare, după care își extrage numai informația cu care inițializează obiectul *Data*. Formatul *time\_t*, sub care obține informația despre timp (funcția *time*) este un echivalent al tipului *long int*, adică numărul secundelor scurse de la 1 ianuarie 1970; constructorul apelează apoi o altă funcție sistem (*localtime*) pentru a structura această informație astfel încât să poată extrage numai elementele dorite. În același timp, se fac și mici corecții, în sensul că luna se aduce la numerotarea 1-12 (nu 0-11, cum o ține sistemul), anul se stochează complet (nu exces 1900).

Pentru constructorul cu parametri, o funcție auxiliară **setData** face minime validări ale întregilor primiți pentru zi, lună, an.

Afișarea în clar a datei (ziua din săptămână, ziua din lună, numele lunii, anul) presupune calcule mai sofisticate. Pentru simplificare, s-a apelat tot la funcții sistem; astfel o dată calendaristică este furnizată funcției *mktime()*, pentru a fi adusă în formatul *time\_t*. Aceeași dată este readusă în format structurat (apelul *gmtime*), pentru că trecând-o prin acest proces, primim de la sistem informația completă (inclusiv ziua din săptămână ce corespunde datei noastre). Afișarea propriu-zisă mai are nevoie acum doar de identificarea unor nume în niște variabile statice cu denumiri de zile și luni.

```
#include <iostream.h>
#include <time.h>

class Data
{
private:
    int luna;
    int ziua;
    int anul;
    static const int Nr_zile[];
    void avans();
    friend ostream &operator<<( ostream &, const Data & );
public:
    Data( int, int, int );
    Data( );
    void setData( int, int, int );
    Data &operator++();
}
```

```
Data operator++( int );
const Data &operator+=( int );
bool AnBisect( int ) const;
bool Sf_de_luna( int ) const;
static bool isSarbatoare(Data);
static const struct sarbatori {public: int z,l;} s[];;
};

const int Data::Nr_zile[ ] =
{ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

const struct Data::sarbatori Data::s[ ]=
{ {1,1},{2,1},{1,5},{1,12},{25,12},{26,12}};

bool Data::isSarbatoare(Data d)
{
    for(int i=0; //sarbatoare din lista ?
        i<sizeof(Data::s)/sizeof(d.s[0]); i++)
        if(d.ziua==s[i].z && d.luna==s[i].l) return 1;

    struct tm cindva, *ptm; time_t altfel;// test de week end
    cindva.tm_year=d.anul-1900; cindva.tm_mon=d.luna-1;
    cindva.tm_mday=d.ziua;
    cindva.tm_hour=cindva.tm_min=cindva.tm_sec=20;
    cindva.tm_isdst=0;
    altfel = mktime(&cindva); ptm=gmtime(&altfel);
    if(ptm->tm_wday==0 || ptm->tm_wday==6) return 1;
    return 0;
}

Data::Data( int zi, int lun, int an ) { setData( zi, lun, an ); }

Data::Data()
{
    time_t data_ora; time( & data_ora );
    struct tm *d_o = localtime(&data_ora);
    ziua=d_o->tm_mday; luna=d_o->tm_mon+1;
    anul=d_o->tm_year+1900;
}

void Data::setData( int zi, int lun, int an )
{
    luna = ( lun >= 1 && lun <= 12 ) ? lun : 1;
    anul = ( an >= 1900 && an <= 2100 ) ? an : 1900;
    if ( luna == 2 && AnBisect( anul ) )
        ziua = ( zi >= 1 && zi <= 29 ) ? zi : 1;
    else
```

```

        ziua = ( zi >= 1 && zi <= Nr_zile[ luna-1 ] ) ? zi : 1;
    }

Data &Data::operator++()
{
    avans(); return *this;
}

Data Data::operator++( int )
{
    Data temp = *this;
    avans(); return temp;
}

const Data &Data::operator+=( int delta )
{
    for ( int i = 0; i < delta; i++ ) avans();
    return *this;
}

bool Data::AnBisect( int an ) const
{
    return
        ( an % 400 == 0 )||( an % 100 != 0 && an % 4 == 0 )? 1: 0;
}

bool Data::Sf_de_luna( int d ) const
{
    if ( luna == 2 && AnBisect( anul ) )
        return d == 29; // februarie în an bisect
    else
        return d == Nr_zile[ luna -1];
}

void Data::avans()
{
    if ( Sf_de_luna( ziua ) && luna == 12 )
        { ziua = 1; luna = 1; ++anul; } // zi sfîrșit de an
    else if ( Sf_de_luna( ziua ) )
        { ziua = 1; ++luna; } // zi sfîrșit de luna
    else
        ++ziua; // zi obisnuită
}

ostream &operator<<( ostream &output, const Data &d )
{
    static char *Nume_luna[] =
    {
        "ianuarie ", "februarie ", "martie ", "aprilie ",
        "mai ", "iunie ", "iulie ", "august ",
        "septembrie", "octombrie ", "noiembrie ", "decembrie ";
}

```

```

static char *Nume_zi[] =
{
    "duminica", "luni ", "marti ", "miercuri", "joi ",
    "vineri ", "sambata ";
}

struct tm cindva, *ptm; time_t altfel;
cindva.tm_year=d.anul-1900;
cindva.tm_mon=d.luna-1; cindva.tm_mday=d.ziua;
cindva.tm_hour=cindva.tm_min=cindva.tm_sec=20;
cindva.tm_isdst=0;
altfel = mkttime(&cindva); ptm=gmtime(&altfel);
output << Nume_zi[ptm->tm_wday]<< ' '<< d.ziua << '
                << Nume_luna[ d.luna-1 ]<< ' '<< d.anul;
return output;
}

void main()
{
    Data d1(28,2,1976),d2; d1++;
    cout<< d1<< endl;
    cout<< d2<< endl;
    d2=Data(1,12,2002);
    for(int i=1;i<=31;i++)
    {
        if(Data::isSarbatoare(d2))
            cout << "\n Sarbatoare: "<< d2;
        d2++;
    }
}

```

Funcția **isSarbatoare** consideră sărbători legale zilele de sămbătă și duminică, precum și cele înscrise într-o listă de sărbători.

Clasa Data mai cuprinde înapoi, supraîncărcări pentru operatorii **++** și **+=**. Toate se bazează pe efectul funcției **avans**, care avansează câte o zi, sesizând sfârșitul de lună și sfârșitul de an.

duminica 29 februarie 1976

duminica 11 august 2002

Sarbatoare: duminica 1 decembrie 2002

Sarbatoare: sambata 7 decembrie 2002

Sarbatoare: duminica 8 decembrie 2002

Sarbatoare: sambata 14 decembrie 2002

Sarbatoare: duminica 15 decembrie 2002

Sarbatoare: sambata 21 decembrie 2002

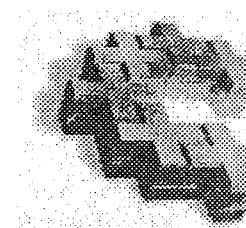
Sarbatoare: duminica 22 decembrie 2002

Sarbatoare: miercuri 25 decembrie 2002

Sarbatoare: joi 26 decembrie 2002

Sarbatoare: sâmbătă 28 decembrie 2002

Sarbatoare: duminică 29 decembrie 2002



## **apitolul III**

### **CLASE DERIVATE. MOȘTENIRI. FUNCTII VIRTUALE**

- Derivarea claselor. Moștenirea unor caracteristici
- Funcții virtuale
- Moșteniri multiple

### 3.1 Derivarea claselor. Moștenirea unor caracteristici

Mecanismul derivării permite crearea facilă de noi clase, care preiau caracteristicile unor clase de bază, deja definite. Derivarea are ca obiectiv reutilizarea soft-ului, prin folosirea unor funcții deja scrise pentru clasele existente și eliminarea redundanței descrierilor, în cazul claselor care au elemente comune, funcții sau date.

Declarația clasei deriveate (D) anunță clasa de bază (B) din care provine, precum și tipul accesului pe care îl asigură pentru partea informațională moștenită (*public*, *private* sau *protected*) sub forma:

```
class D : public B
{
    // date și funcții specifice clasei deriveate;
};
```

Indiferent de tipul derivării, funcțiile membre ale clasei deriveate nu au acces pe zona *private* a clasei de bază; problema nuanțării accesului la derivare se pune așadar numai pentru zonele *public* și *protected*.

**Prin derivare public**, membrii publici ai clasei de bază își păstrează caracterul public; clasa derivată are de asemenea, prin definiție, drepturi de acces asupra membrilor *protected* ai clasei de bază. Practic, derivarea publică extinde pe *public* și *protected* tipul accesului din bază și pentru membrii derivatei.

**Prin derivare private**, membrii *public* și *protected* ai clasei de bază devin membri priavați în clasa derivată. Se observă că prin derivare, membrii *private* din clasa de bază își păstrează totdeauna caracterul privat, în toate clasele deriveate din aceasta și pe toate nivelurile de derivare, când sunt mai multe niveluri de derivare ierarhică. Practic, prin derivare privată se oprește posibilitatea de transmitere către urmăși a dreptului de acces la clasa de bază, deoarece pentru primul nivel de derivare zonele *public* și *protected* sunt totuși accesibile.

**Prin derivare protected**, membrii *public* și *protected* ai clasei de bază devin *protected* pentru clasa derivată, putând fi transmis dreptul de acces și altor clase, prin derivare succesivă.

Tipul accesului moștenit prin derivare, dacă nu se specifică, este implicit privat. În cazul structurilor, în versiunile care acceptă derivarea structurilor, aceasta este implicit publică. După cum se observă din figura 3.1, prin toate tipurile de derivare, membrii priavați sunt inaccesibili pentru clasele derivate.

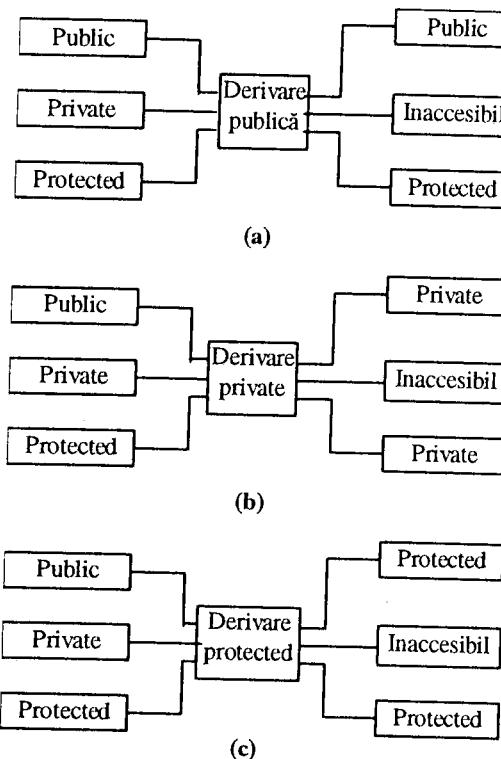


Fig. 3.1 Reguli de derivare

Prin derivare privată, membrii publici ai clasei de bază devin priavați; dacă programatorul dorește ca o parte dintre aceștia să rămână publici ca și în bază, adică derivarea privată să nu se exercite asupra întregii zone moștenite, poate apela la aşa numita *publicizare*, adică citarea pe zona public a clasei deriveate, a membrilor moșteniți.

```
#include <iostream.h>
class B
{
    int x;
public:
```

```

int y;
void f( ) { }
B(int i=0, int j=1):x(i),y(j){}
};

class D : B
{
public:
    B::y;
    B::f;
};

void main() { B b; D d; cout << b.y<<" "<< d.y; d.f(); }

```

După cum se observă, pentru a distinge între noile declarații de membri care se numesc la fel ca în bază, la *publicizare* nu se mai dă tipul, respectiv prototipul membrilor și în plus se indică și rezoluția de clasă de bază. *Publicizarea* apare deci ca o relaxare a restricțiilor introduse prin derivare și nu ridică restricții impuse de creatorul clasei de bază.

Noua clasă D își are propriii ei constructori, impliciti și / sau expliciti, care îndeplinind parțial același obiectiv cu constructorul clasei B, solicită explicit (prin forma cu *:nume\_constructor*) sau implicit, serviciile acestuia. Constructorul clasei D este responsabil cu inițializarea corectă și a datelor moștenite; în mod normal el apelează la constructorul B pentru inițializarea datelor moștenite, chiar dacă noi nu cerem expres acest lucru, după care completează el însuși clasa cu datele și funcțiile specifice clasei derivate.

Vom exemplifica printr-o nouă clasă, *student*, care poate fi introdusă ca o particularizare a clasei *persoana*. Pentru simplitate am considerat că noua clasă are în plus o singură informație, cea privind numărul matricol.

Pentru a sesiza conlucrarea între constructorii celor două clase, să urmărim programul următor:

```

#include <iostream.h>
#include <string.h>
class persoana
{
private:
    int vîrstă;
public:
    char nume[20];
    persoana(char *n="Anonim ", int v=0 ):vîrstă(v)
        { strcpy(nume,n); }
    int spune_vîrstă( ) { return vîrstă; }
};

```

```

class student : public persoana
{
    int matricol;
public:
    student(char *nm = "FICTIV ", int vs=0, int m=0)
        : persoana(nm,vs), matricol(m) {}
};

void main( )
{
    persoana p1;
    student s1, s2("Stan Emil", 25, 110);
    cout << "\n" << s1.nume << p1.nume << s2.nume;
}

```

Acst program produce ieșirea :

*FICTIV Anonim Stan Emil*

Obiectul p1, fiind de tip *persoana* a fost inițializat implicit cu numele *Anonim*, în timp ce pentru studentul s1, numele implicit, deși pus de constructorul B, îi este transferat de constructorul D care știe să denumească implicit prin *FICTIV*. Chiar acolo unde valorile sunt date explicit, ca în cazul lui s2, munca este tot divizată între constructori.

În exemplul care urmează, clasa D nu are un constructor explicit, astfel încât valorile asumate sunt cele puse de constructorul B. Se observă de asemenea, că există și constructor de copiere implicit, el stând la baza conversiilor隐式的 ale unui student într-o persoană simplă. Conversia inversă *B => D* nu este implicită, datorită sensului unic de moștenire. Pentru această conversie este necesară supraîncarcarea operatorului de atribuire (=) al clasei destinație, conversia fiind acceptată doar pentru clasele derivate public.

Conversia de la derivat către baza (*D => B*) se efectuează implicit și asupra pointerilor și referințelor de obiecte, ceea ce face posibil și un apel al constructorului B, cu referire directă asupra unui obiect de tip D, sub forma *D:: D(D& d) : B(d)*.

```

#include <iostream.h>
#include <string.h>
class persoana
{
private:    int vîrstă;
public:    char nume[20];
    persoana(char *n="Anonim ", int v=0 ):vîrstă(v)
        { strcpy(nume,n); }
    int spune_vîrstă( ) { return vîrstă; }
};

```

```

int spune_virsta( ) { return virsta; }

};

class student: public persoana
{
    public: int matricol;
};

void main()
{
    persoana p1; student s1, s2;
    cout << "\n" << s1.nume << p1.nume << s2.nume;
    s1.matricol=101; strcpy(s1.nume,"Transferat");
    cout << "\n" << s1.matricol << s1.nume << s1.spune_virsta();
    p1=s1; cout << "\n" << p1.nume << p1.spune_virsta();
}

```

Dacă D nu are un constructor de copiere, el va fi creat automat de compilator pentru a suplini sarcinile operatorului de copiere obiecte, necesar în cazul unei atribuirii sau transferului de obiecte în/din funcții. Evident și constructorul de copiere apelează automat la constructorul de copiere al lui B.

Dacă se definește explicit și un constructor de copiere pentru D, el va fi cel apelat și se va ține seama de referirile expuse făcute de acesta către constructorul lui B. Astfel, dacă constructorul D nu prevede explicit un transfer de sarcini către constructorul B ( forma D( D& d ), adică fără referire explicită la constructorul lui B), atunci acesta din urmă este apelat cu lista de parametri vidă. Dacă un constructor cu acest prototip nu există se va semnală eroare.

Pentru forma D( D& d ) :B ( lista parametri ), constructorul B este căutat și apelat după prototipul indicat; însă, dacă acesta nu există, se va semnală, de asemenea, eroare.

În cazul destructorului, lucrurile stau similar, numai că ele se derulează în ordine inversă. La crearea unui obiect derivat, ordinea de execuție este: constructor de bază, constructor derivat, iar la dezalocare: destructor derivat, destructor de bază.

Așa cum spuneam deja, clasele derivate nu au drepturi de acces speciale asupra membrilor clasei de bază, exceptând domeniul protected. Astfel o clasă derivată public din alta, nu are acces asupra domeniului private din clasa de bază; altfel s-ar putea eluda drepturile de acces prin derivarea unor clase fictive, numai în scopul de a câștiga drepturi de acces !

Clasa D moștenește atât datele, cât și metodele clasei B, (chiar dacă le poate sau nu accesa !) fiind recomandat ca pentru aceeași funcționalitate să nu se recurgă la crearea de funcții noi. În felul acesta, crește claritatea programului. Când totuși D introduce funcții noi, purtând același nume cu cele

din B, selectarea se face cu prioritate din D. Când se dorește apelul funcției cu același nume, prin intermediul unui obiect derivat, dar versiunea moștenită din B, este necesară calificarea completă, folosind și rezoluția de clasă ( B::f( ); ). Vom exemplifica acest lucru prin programul de mai jos, din care se deduce că obiectul derivat are două versiuni ale funcției *spune\_nume()*, una proprie și una moștenită:

```

#include <iostream.h>
#include <string.h>
class persoana
{
private: int virsta;
public: char nume[20];
persoana(char n[20] = "Anonim ", int v=0): virsta(v)
{
    strcpy(nume,n);
}
int spune_virsta( ) { return virsta; }
char *spune_nume( ) { return nume; }
};

class student: public persoana
{
public: int matricol;
char * spune_nume( ) { return nume + 1; }
};

void main( )
{
    persoana p1; student s1, s2; s1.matricol=101;
    cout << "\n" << s1.matricol << s1.nume << s1.spune_virsta();
    strcpy(s1.nume,"Nou_numit");
    cout << "\n" << s1.persoana::spune_nume( ) << " diferit de "
        << s1.spune_nume( );
}

```

Funcția din D returnează numele fracționat (pointerul indică începutul pe a doua literă din nume), astfel încât se afișează:

```

101Anonim 0
Nou_numit diferit de *ou_numit

```

În privința pointerilor la funcțiile membre din clasă de bază sau din clase derivate se pot face următoarele precizări. Deoarece pointerii de funcții membre poartă și rezoluția clasei căreia aparține funcția membru, nu se crează confuzii privind funcția punctată. Datorită moștenirii ierarhice, este admisă doar conversia implicită a unui pointer de funcție membră în clasa de bază în pointer de funcție membră în clasa derivată, cu condiția să aibă același

prototip, exceptând doar clasa de apartenență. Se observă că sensul conversiei este opus celui de la obiecte sau pointeri de obiecte, unde conversiile se făceau pe direcția obiect derivat - obiect de bază; acest lucru se explică prin faptul că pointerii de membri în clasă nu conțin adrese absolute, ci adrese relative în cadrul clasei; cum offset-ul este mai mare în clasa derivată prin adăugarea membrilor specifici, rezultă că doar un pointer de membru în clasa de bază are un echivalent în clasa derivată, nu și invers.

Supradefinirea operatorilor în clasele derivate este posibilă, când nu se realizează acest lucru, se moștenește efectul supradefinițiilor din clasa de bază. Operatorul = are o moștenire particulară. Când el este supradefinit în clasa derivată, aceasta este versiunea care se va aplica întotdeauna. Dacă în clasa derivată el nu este supradefinit, se aplică operatorul din clasa de bază pentru partea comună celor două clase, iar pentru datele suplimentare din clasa derivată se face o copiere membru cu membru.

Funcțiile *friend* ale clasei de bază, rămân *friend* și pentru clasele derivate. Calitatea friend a unei clase nu se moștenește.

### Funcții care nu se moștenesc integral

Există membri, funcții și operatori, foarte strâns legați de o clasă ce nu pot fi moșteniți ad literam. Chiar când sunt puși automat de către compilator ei sunt sintetizați din cei din clasa de bază și din operații specifice datelor din clasa derivată. **Constructorii și destructorul** reprezintă cel mai bun exemplu; ei nu au cum să fie moșteniți exact ca în clasa de bază, deoarece cei din clasa derivată au sarcini sporite, legate de partea specifică clasei derivate.

**operator=** pus automat de compilator apelează pe cel din bază, dar face în plus, copierea părții specifice clasei derivate.

**Operatorul cast** este de asemenea moștenit într-o formă sintetizată, în sensul că dacă există suficiente indicii pentru realizarea unei conversii, compilatorul sintetizează un cast, capabil să convertească obiecte derivate.

Programul următor ilustrează aceste lucruri, trasând trecerile prin funcțiile clasei de bază, deși nu lucrează cu nici un obiect de bază.

```
#include <iostream.h>
class Persoana
{
public:
    char nume[50];
```

```
char *spune_nume() { return nume; }
};

class Vehicul
{
public:
    Vehicul() { cout << "Vehicul()\n"; }
    Vehicul(Vehicul&) { cout << " Vehicul(Vehicul & )\n" ; }
    Vehicul(int) { cout << " Vehicul(int)\n" ; }
    Vehicul& operator=(const Vehicul&)
    {
        cout << "Vehicul::operator=(+)\n";
        return *this;
    }
    Persoana proprietar;
    operator Persoana() const
    {
        cout << "Vehicul::operator cast spre Persoana()\n";
        return proprietar;//Persoana();
    }
    ~Vehicul() { cout << "~Vehicul()\n"; }
};

class Autoturism : public Vehicul { };
void impoziteaza(Persoana i) /* impoziteaza cladiri, auto ... */
void main()
{
    Autoturism d1; // Apeleaza implicit si constructor din Vehicul
    Autoturism d2 = d1; // Apeleaza implicit copy-constructor din Vehicul
    d1 = d2; // Operator= implicit pus de compilator, nu moștenit
              // El apeleaza pe cel din baza !
    impoziteaza(d1); // cast catre Persoana, moștenit prin derivare
    ( (Persoana) d1 ).spune_nume();
}
```

Prin rularea programului se afișează:

```
Vehicul()
Vehicul(Vehicul & )
Vehicul::operator=()
Vehicul::operator cast spre Persoana()
Vehicul::operator cast spre Persoana()
~Vehicul()
~Vehicul()
```

Clasa **Autoturism**, derivată din **Vehicul**, nu dispune de constructor și destrutor expliciti, dar cei puși de compilator îl invocă pe cei din clasa de bază, după cum se observă din afişare.

La atribuirea dintre două obiecte deriveate se observă de asemenea trecerea prin atribuirea definită în clasa de bază.

Lucru interesant este și că un obiect **Autoturism** nu dispune de convertor prin cast spre persoana, dar cu ajutorul compilatorului este înzestrat cu un astfel de operator, dovdă că poate satisface solicitarea de impozitare, utilizând cast-ul moștenit.

Probabil, în cele mai multe din cazuri, operatorul moștenit trebuie redefinit în clasa derivată.

### **Moștenire versus includere de clase**

Derivarea apare foarte apropiată de includerea unei clase în alta; în fond ambele conduc la o clasă mai mare, care conține toți membrii unei alte clase mai mici; ambele urmăresc și considerente de reutilizare a codului sursă. Din modul în care se comportă membrii celor două clase ne dăm seama că deosebirile sunt esențiale. Legăturile existente în lumea reală între clase sunt cele care dictează ce cale de combinare între clase trebuie să alegem.

**Derivarea** se folosește pentru a exprima o relație de forma **is a** și se stabilește între clase de același fel.

**Includerea** (sau compunerea) se folosește pentru a exprima o relație de formă **has a** și se stabilește între clase diferite. Putem spune aşadar "automobil **is a** vehicul", "automobil **has a** roata" pentru a surprinde relații diferite între clase. Funcțiile moștenite prin derivare se apelează ca funcții proprii, pe când funcțiile unei clase incluse se apelează ca niște servicii ale unui server inclus.

## **3.2 Funcții virtuale**

Am observat că un obiect derivat poate dispune de două versiuni ale aceleiași funcții, ambele cu același prototip, dar una fiind moștenită. În mod normal, versiunea funcției care se apelează la un moment dat, se stabilește încă de la compilare - legătura statică (timpurie) - (early binding), ceea ce

repräsentă un inconvenient major, neputându-se decide la momentul execuției, în raport de context, asupra unei versiuni de funcții sau alta.

Nu ne este de vreun ajutor nici utilizarea pointerilor, deoarece în ciuda conversiilor obiect derivat în obiect de bază, valabilă și pentru pointeri, variabila pointer nu ține seama de tipul obiectului adresat, iar funcția sistematic selectată va fi cea din clasa de bază. Conversia unui student într-o simplă persoană, în exemplul nostru ( $b = d$ ; obiect derivat în obiect de bază, în general), își găsește rațiunea că fiecare student este în același timp și o persoana ("d is a b"), dar nu și invers.

Pentru identificarea la momentul execuției a versiunii adecvate a funcției de executat, limbajul C++ oferă conceptul de funcție virtuală, bazat pe legătura dinamică (întârziată) (late binding).

Ideea de legătură dinamică este preluată în C++ din limbajul C standard, unde există posibilitatea declarării unui vector de pointeri la funcții, pentru un set de prelucrări. În timpul execuției, conținutul elementelor vectorului se schimbă, astfel încât setul de prelucrări ce trebuiau executate se stabilă dinamic, în funcție de context. Mecanismul de apel al funcțiilor virtuale se bazează tocmai pe această facilitate de limbaj.

Pentru fiecare clasă, compilatorul construiește o tabelă de pointeri la funcțiile virtuale. Fiecare obiect al clasei primește la naștere un pointer la această tabelă a clasei. La execuție se preia din obiect adresa tabelei de funcții virtuale specifice clasei, se citește din această tabelă adresa actualizată a funcției de executat și abia apoi se execută funcția. Apelul de funcție virtuală devine aşadar implicit mediat de pointeri, iar pointerii pot fi actualizați în raport cu contextul. Faptul că punctul de pornire îl reprezintă tot un obiect concret explică și de ce o funcție virtuală nu poate fi statică, pentru că trebuie legată de un obiect, nu de clasa în sine.

O funcție virtuală anunță de fapt că în fiecare din deriveatele clasei va avea versiuni proprii, iar distincția variantei apelate se va face abia la momentul execuției, nemaifiind necesară recomplarea claselor.

Declarația de virtual se poate da oriunde în ierarhia derivării, adresa funcției virtuale înscriindu-se atât în tabela clasei respective, cât și a descendenților ei.

Am reținut aşadar ca polimorfismul este realizat, în esență, prin două mecanisme distincte:

- supraîncarcarea unor funcții din cadrul unei clase;
- redefinirea conținutului unor funcții virtuale, în clasele deriveate.

În primul caz, identificarea corectă a funcției se face pe baza numărului și tipului parametrilor de apel. În mecanismul funcțiilor virtuale, toate funcțiile au același prototip (tip valoare returnată, nume funcție, număr

și tip parametri de apel). Când apelul se face sub forma b.f() sau d.f(), adică pornind de la un obiect de bază sau derivat, lucrurile sunt simple pentru că obiectul identifică versiunea corectă a lui f().

Ce se întâmplă însă când apelul se face pornind de la un pointer la obiect al clasei de bază (pb) și care conform moștenirii poate conține atât adrese de obiecte de bază, cât și adrese de obiecte derive (d1 sau d2). În acest caz, nu prototipul funcției este criteriul de discriminare ci tipul obiectului pointat, chiar dacă adresa lui este stocată într-un pointer spre obiect de bază (pb=&d1; pb->f()); pentru versiunea funcției din prima clasă derivată dintr-o clasă de bază; sau pb=&d2; pb->f() pentru versiunea din cea de-a doua clasă derivată din clasa de bază; sau: pb=&b; pb->f(); pentru versiunea funcției din clasa de bază).

Programul următor, exemplifică acest lucru pentru două clase, *muncitor* și *inginer*, derive din aceeași clasă de bază, *persoana*. Vom presupune că fiecare din clasele derive are o metodă specifică de calcul al salariului și care diferă și de cea din clasa de bază. Pentru simplificare nu am procedat la scrierea algoritmului efectiv de calcul al salariului, ci ne-am mărginit la a marca doar prin mesaje trecerea prin funcția adecvată. Vom vedea cum decurge selectarea funcției de apelat atunci când folosim obiectele însele ca punct de pornire, sau pointeri la obiectele de bază sau derive.

```
#include <iostream.h>
class persoana
{
private: float salariu;
public: char nume[20];
        virtual float calc_sal() { cout << "\n Salariu persoanei"; }
};
class inginer : public persoana
{
public: float calc_sal() { cout << "\n Salariu în regie"; }
};
class muncitor : public persoana
{
public: float calc_sal() { cout << "\n Salariu în acord"; }
};
void main()
{
persoana p, *pp; inginer i,*pi; muncitor m, *pm;
pp = &p; pi=&i; pm=&m;
cout << "\nFolosind obiecte și pointeri la obiecte: ";
p.calc_sal(); pp->calc_sal();
}
```

```
i.calc_sal(); pi->calc_sal();
m.calc_sal(); pm->calc_sal();
cout << "\nFolosind conversia în pointer la obiect de bază: ";
pp=pi; pp->calc_sal();
pp=pm; pp->calc_sal();
cout << "\nFolosind conversia în obiect de bază: ";
p=i; p.calc_sal();
p=m; p.calc_sal();
}
```

cu afișarea:

*Folosind obiecte și pointeri la obiecte:*

*Salariu persoanei*

*Salariu persoanei*

*Salariu în regie*

*Salariu în regie*

*Salariu în acord*

*Salariu în acord*

*Folosind conversia în pointer la obiect de bază:*

*Salariu în regie*

*Salariu în acord*

*Folosind conversia în obiect de bază:*

*Salariu persoanei*

*Salariu persoanei*

Când derivarea se face pe mai multe nivele, tot tipul obiectului pointat, și anume cărei clase din ierarhia derivează aparțin obiectul a cărei adresă este punctul de plecare în calificare, este criteriul de selecție a funcției virtuale corecte.

Am văzut aşadar, că apelarea unei funcții pornind de la un pointer de obiect de bază permite scrierea același cod sursă, deși se traduce prin execuția unui cod diferit, după cum pointerul conține o adresă de obiect de bază sau de obiect derivat, convertită implicit în adresă de obiect de bază.

Deși obiectele derive sunt și ele convertite implicit în obiecte de bază, apelurile pornind de la obiectele de bază nu țin seama de faptul că obiectul provine dintr-unul derivat. Putem scrie noi un cod sursă astfel încât să obligăm trecerea obiectului pe care se bazează apelul prin faza de adresă sau referință; astfel putem beneficia de virtualizare și când pornim de la obiect, nu numai de la pointer de obiect.

În programul care urmează, funcția de citire a fost declarată virtuală la nivelul clasei de bază, acest caracter fiind moștenit de clasele derive din ea, chiar dacă acestea poartă sau nu, explicit, atributul de virtual. Exemplul

este unul dintre cele mai frecvente, deoarece încărcarea prin citire trebuie făcută adecvat pentru fiecare dintre clase, ele diferind ca volum de informații.

O funcție externă de tip *friend*, *apel()*, acționează ca selector, trece obiectul prin faza de referință distingând astfel după tipul parametrului actual primit, când să apeleze una sau cealaltă dintre funcțiile de încărcare a unui obiect, prin citire.

```
#include <iostream.h>
class persoana
{
private: float salariu;
protected: int vîrstă;
public:
    char nume[20];
    virtual void citire()
    {
        cout << "\n Nume: "; cin >> nume;
        cout << "\n Vîrstă: "; cin >> vîrstă;
        cout << "\n Salariu: "; cin >> salariu;
        cin.ignore();
    }
    friend void apel( persoana & );
};

class student : public persoana
{
private: int matricol;
public:
    void citire()
    {
        cout << "\n Nume : "; cin >> nume;
        cout << "\n Vîrstă : "; cin >> vîrstă;
        cout << "\n Matricol: "; cin >> matricol;
        cin.ignore();
    }
};

void apel( persoana &p) { p.citire(); }

void main()
{
    persoana p; student s;
    apel(s); apel( p);
}
```

Mesajul care premerge introducerea efectivă a datelor și numărul de date solicitate, ne permite să verificăm ușor că fiecare apel din programul nostru se adresează unor funcții *citire()*, diferite! Deși funcția *apel()* are ca parametru de intrare o referință de persoană, ea acceptă și referințe de student. Referința de obiect derivat este convertită implicit în referință de obiect de bază, dar la adresa respectivă se găsește în fapt tot un obiect derivat și el conține tabela de funcții virtuale actualizată. În acest mod, în raport de tipul referinței primite, se direcționează acțiunea către funcția de citire adecvată.

Spre deosebire de supraîncărcarea funcțiilor și operatorilor (overloading), funcțiile virtuale folosesc conceptul de redefinire (overriding), nuanțând astfel mecanismul de realizare a polimorfismului. Dacă o funcție virtuală apare declarată cu mai multe prototipuri (diferă fie numărul, fie tipul unei variabile formale), ea își pierde natura virtuală fiind interpretată ca o simplă supraîncărcare.

Merită de asemenea observat că mecanismul supraîncărcării este mai general, el aplicându-se și funcțiilor nemembre ale unei clase, în timp ce redefinirea funcțiilor virtuale vizează doar funcțiile membre ale unor clase de bază sau derive.

În sfârșit, trebuie menționat că funcțiile virtuale respectă ierarhia moștenirii. Când într-o clasă derivată nu apare redefinită o funcție virtuală, la apelul ei va răspunde funcția virtuală din clasa ierarhic superioară sau ultima definiție din ierarhie. Cu alte cuvinte, natura virtuală a unei funcții se moștenește (de altfel nici nu mai este necesar declaratorul "virtual" în clasele derive, el fiind implicit). Funcția de pe ultimul nivel unde a fost redefinită răspunde și pentru subierarhia ei, unde nu a mai fost redefinită.

O categorie aparte de funcții virtuale sunt funcțiile virtuale pure. Definiția unei funcții virtuale pure în clasa de bază nu cuprinde nimic de executat ci doar prototipul ei = 0, sub forma:

**virtual tip nume ( parametri ) = 0;**

Initializarea cu zero care pare ceva lipsit de sens în acest context, are menirea să anunțe tipul "pure" al funcției virtuale forțând clasele derive să facă redefiniri ale acesteia; altfel compilatorul va semnala eroare chiar din faza de compilare. Definiția din clasa de bază se pune doar pentru ca un pointer spre obiectul de tip bază să poată vedea și el funcția. Altfel, funcția ar fi văzută doar prin obiecte de tip derivat sau prin pointeri spre aceștia, căci doar la nivelul derivat s-a dat definiția funcției. Declarată în clasa de bază, funcția se moștenește și nu apare doar ca un element specific nivelului de derivare și

deci se va putea folosi mecanismul virtualizării pentru a selecta versiunea adecvată pentru fiecare dintre obiectele deriveate.

Dacă inițializarea cu zero lipsește, nu i se atribuie caracterul pur funcției virtuale iar la linkeditare va fi căutată versiunea concretă a funcției pe acest nivel pentru rezolvarea referințelor în apel. Cum conținutul specific acestui nivel nu va fi găsit, evident se va semnala ca eroare și nu se va depăși această fază.

O clasă care conține cel puțin o funcție virtuală pură, se numește **clasa abstractă**, deoarece nu poate avea obiecte concrete din moment ce definiția clasei este incompletă. Prin urmare, clasele abstracte folosesc doar ca suport pentru derivare, din ele moștenindu-se aspectele generale, comune mai multor subtipologii.

Să revizităm un exemplu anterior, modificat în sensul că nu există salariu pentru o persoană în general ci doar pentru *derivate* ale acesteia care lucrează efectiv, iar fiecare profesie își are metoda sa specifică de calcul al salariului.

Se observă că nu ni s-a mai permis definirea de obiecte de tip persoana în general, ci doar de muncitori sau ingineri; în schimb, am putut declara pointeri la obiecte abstracte care vor fi concretizați abia la momentul execuției, evident doar în adrese de obiecte deriveate. De asemenea, tipul unei clase abstracte se poate aplica și unei referințe.

```
#include <iostream.h>
class persoana
{
private: float salariu;
public:
    char nume[20];
    virtual float calc_sal( )=0;
};
class inginer : public persoana
{
public:
    float calc_sal( )
        { cout << "\n Salariu in regie"; return 1.; }
};
class muncitor : public persoana
{
public:
    float calc_sal( )
        { cout << "\n Salariu in acord"; return 1.; }
};
```

```
void main( )
{
    persoana *pp;
    inginer i,*pi;
    muncitor m, *pm;
    pi=&i; pm=&m;
    cout << "\nFolosind obiecte si pointeri la obiecte: ";
    i.calc_sal( ); pi->calc_sal( );
    m.calc_sal( ); pm->calc_sal( );
    cout << "\nFolosind conversia in pointer la obiect de baza: ";
    pp=pi; pp->calc_sal( );
    pp=pm; pp->calc_sal( );
}
```

Rezultatul rulării este:

*Folosind obiecte și pointeri la obiecte:*

*Salariu în regie*

*Salariu în regie*

*Salariu în acord*

*Salariu în acord*

*Folosind conversia în pointer la obiect de baza:*

*Salariu în regie*

*Salariu în acord*

Este posibil ca și într-o clasă derivată, dar care mai are descendenți prin derivare, o funcție virtuală pură să fie declarată tot pură, conferind caracterul abstract și acestei clase și urmând ca ulterior să fie redefinită în descendenții săi pentru o folosire efectivă.

Prin mecanismul virtualizării, funcțiile deja compilate sunt capabile să se adapteze contextului unei noi clase fără a necesita modificarea și recompilarea codului.

Din punct de vedere tehnic, funcțiile virtuale fiind implementate prin mecanismul *legării întârziate*, apelurile sunt mai lente, necesită memorie suplimentară pentru o tabelă de adrese de funcții virtuale dar oferă o flexibilitate enormă, permitând ca la un mesaj să existe mai multe metode de tratare, denumite la fel dar specifice fiecărui obiect care-l receptionează. Manipularea obiectelor prin intermediul pointerilor, în fond suportul mecanismului de virtualizare, generează însă și multe capcane.

### 3.3 Moșteniri multiple

Este posibilă derivarea unei clase pornind de la două sau mai multe clase de bază; se realizează astfel definirea unor structuri mai complexe de clase, de **tip rețea**, spre deosebire de ierarhiile de clase, obținute prin derivarea simplă.

Moștenirea multiplă este **contestată** de mulți programatori, datorită ambiguităților pe care le poate genera; spre exemplu biblioteca de clase Microsoft Foundation Class folosește doar ierarhii de clase.

Folositoră doar când reflectă o realitate, moștenirea multiplă are un avantaj major: moștenesci ușor **attribute și comportamente** variate, provenind de la clase ce pot差别 substanțial, permitând **dezvoltarea incrementală a unor comportamente complexe**.

Toate caracteristicile moștenite trebuie să se regăsească cel puțin într-o formă proprie și la obiectul derivat; de cele mai multe ori unele caracteristici nu au sens și pentru obiectul derivat, în general optându-se pentru un compromis între avantajele moștenirii multiple și **forțarea sensului** unor caracteristici moștenite.

Există și situații când într-o derivare multiplă, folosim și o clasă cu date puține, sau fără date, ea aducând doar metode generale, cum ar fi setarea unor parametri de afișare; acest tip de moștenire multiplă se mai numește **mixin**.

Un model general pentru derivare multiplă ar putea fi:

```
class B1 { ... };
class B2 { ... };
class B3 { ... };

class D : public B1, private B2, protected B3
{ /* membrii specifici ai clasei D */ };
```

În fața fiecărei clase poate fi pus tipul derivării: **public**, **private** sau **protected**, ceea ce înseamnă că **moștenirile pot fi controlate pentru fiecare clasă de bază în parte**. Când tipul moștenirii lipsește se asumă cel **private**. Reiese că și prin moștenire multiplă pot fi introduse noi restricții de acces, dar nu pot fi ridicate cele prevăzute în clasele de bază.

Clasa derivată (D) moștenește toate datele membre ale unor clase de bază (B1, B2, ...), dar are acces numai la cele publice sau protected și

poate apela doar funcțiile **public** sau **protected** moștenite. Care este sensul moștenirii unor date și funcții inaccesibile? Răspunsul este simplu: nu are acces direct la aceste informații, dar indirect, prin funcțiile de acces moștenite, poate accesa și funcții și date private, moștenite.

Constructorul lui D va preciza explicit transferul de sarcini către toți constructorii moșteniți prin derivare:

```
D::D(...) : B1(...), B2(...), ..., BN(...){ }
```

Fiecare constructor e responsabil de zona lui; nici nu se poate altfel, deoarece zonele private moștenite, chiar public, nici nu sunt accesibile dintr-o funcție a clasei derivate.

Când constructorul lui D lipsește, este pus de compilator un **constructor sintetizat** din constructorii de bază, în sensul că fiecare constructor își inițializează zona lui (transfer implicit de sarcini către constructorii de bază moșteniți), după care se inițializează zona specifică doar obiectului derivat.

La moștenirea multiplă, **ordinea** în care sunt apelați constructorii clasei de bază este de obicei cea **în care clasele de bază sunt citate în lista de derivare**; nu contează ordinea în care sunt citați constructorii în lista de inițializatori D::D(...) : B1(...), B2( ), ..., BN( ) {}

Ordinea în care sunt apelați destructorii clasei de bază este de obicei inversă celei **în care clasele de bază sunt citate în lista de derivare**.

Similar stau lucrurile și cu copy-constructorul; când programatorul scrie un **copy-constructor** explicit, trebuie să procedeze la fel; dacă va lăsa zone neinitializate, ele vor fi **automat inițializate** de compilator folosind **constructorul de clasă** implicit, după care vor fi aplicate copierile indicate prin constructorul de copiere (**constructor de copiere corectat**).

La derivarea pe mai multe niveluri, constructorii nu sunt invocați din aproape în aproape, ci direct; spre exemplu, dacă pentru clasele B1, B2 există o clasă de bază unică, BB (bază a bazelor) constructorul BB va fi invocat direct din D, la creare de obiecte d; constructorii B1 și B2 vor invoca și ei constructorul BB, dar numai pentru crearea de obiecte b1, b2, ignorându-l în rest.

**Ambiguități la adresarea membrilor moșteniți, care se numesc la fel în două dintre clasele de bază**

Spuneam că derivarea din mai multe clase de bază introduce o serie de ambiguități. O primă categorie o reprezintă ambiguitățile apărute când

funcția sau data moștenită se numește la fel în două sau mai multe dintre clasele de bază.

În general, rezolvarea se face prin folosirea **rezoluției de clasă**, la apelul sau adresarea membrului moștenit specificându-se explicit filiera de moștenire a fiecărui:

d.B1:: f( );	sau d.B2:: f( );	- pentru funcții membre
d.B1:: x;	sau d.B2:: x;	- pentru date membre

Pentru funcțiile moștenite, rezolvarea se poate face și prin **redeclararea funcției în clasa derivată**, într-o formă proprie, sau optând pentru una din moșteniri și ascunderea celorlalte forme moștenite:

```
int D::f() { return B1::f(); }
```

O problemă interesantă apare în legătură cu **upcasting-ul**: clasa derivată mai este un tip (respectarea relației *is a* presupusă la conversia clasei derivate în clasă de bază) de clasă de bază? De care clasă de bază, căci acum sunt mai multe? Dacă alegerea moștenirii multiple s-a făcut corect, ne putem explica de ce sunt acceptate implicit conversiile în sus (upcasting), de la clasa derivată spre oricare dintre clasele de bază. Mai mult, aceste conversii au și un sens. La fel stau lucrurile cu pointerii de obiecte. În consecință, putem manipula un obiect derivat, cu un pointer de oricare obiect din tipurile de bază ce asigură moștenirea multiplă:

```
B1 *pb1; B2 *pb2; D d;
pb1=&d; pb2=&d;
```

#### Moșteniri multiple din clase cu o bază comună. Ambiguități la moștenirile din clase de bază cu o bază comună

Un alt inconvenient al moștenirii multiple este și **includerea multiplă a membrilor moșteniți dintr-o clasă de bază comună** pe două sau mai multe filiere distincte de moștenire. Mai precis, fie derivarea din figura 3.2 căreia ar putea să-i corespundă o descriere de genul:

```
class BB { public: int x; };
class B1 : public BB { public: B1() { x = 1; } };
class B2 : public BB { public: B2() { x = 2; } };
class D : public B1, public B2 { };
```

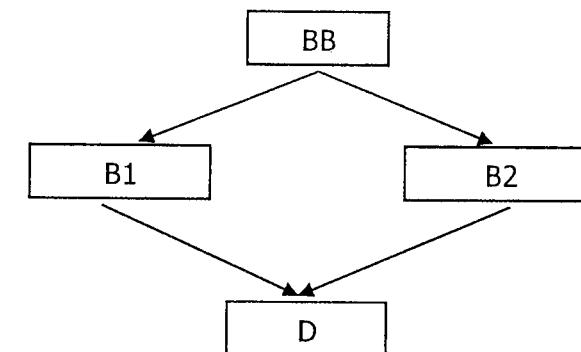


Fig. 3.2 Rețea de clase

Făță de ambiguitățile deja menționate, la derivarea multiplă din mai multe clase care au la rândul lor o bază comună, apar alte ambiguități noi. Nu este nevoie ca ambele clase B1 și B2 să aibă membri care se numesc la fel; ambiguitatea e propagată de undeva mai de sus, pe direcția de derivare.

Se observă că D va moșteni membrul x din BB în duplicat datorită celor două filiere de moștenire, via B1 și via B2. Dacă acest lucru a dorit și programatorul, totul este perfect și obiectul derivat dispune de posibilități de identificare univocă a fiecărei variante moștenite, folosind rezoluția de clasă: *d.B1::x*, respectiv *d.B2::x*.

```
void main()
{
    D d;
    cout << "\n x moștenit via B1: " << d.B1::x;
    cout << "\n x moștenit via B2: " << d.B2::x;
}
```

Programul de mai sus ne confirmă acest lucru afișând valorile 1 și 2, puse de fiecare constructor pentru membru moștenit pe fiecare din cele două filiere. De obicei, fiecare constructor de bază își actualizează propriul exemplar, chiar dacă toate exemplarele conțin sau nu aceeași valoare. Menținerea dublurilor ridică însă două probleme:

- cea a **actualizărilor concomitente** a tuturor exemplarelor, când ele au același conținut;
- cea a **consumului inutil de memorie**, deloc neglijabil, când se lucrează cu multe obiecte.

Similar stau lucrurile și în cazul funcțiilor moștenite din clasa BB. În cazul funcțiilor putem rezolva ambiguitatea și prin redefinirea funcției în D, chiar printr-o simplă redirectare, către una din versiunile moștenite:

```
int D::f( ) { return B1::f( ); }
```

Se observă că rezolvările sunt identice cu cele de la ambiguitățile simple, chiar dacă de data aceasta le-a generat o clasă de bază comună claselor de bază; explicația constă în faptul că și de data aceasta este suficient pentru precizarea filierei de moștenire, menționarea uneia din clasele B1 sau B2, prin care s-a făcut moștenirea.

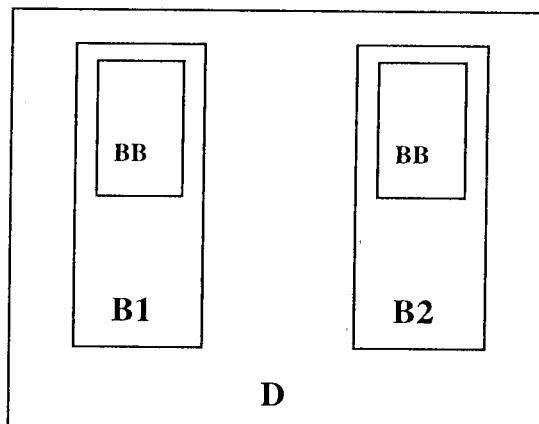


Fig. 3.3 Moșteniri multiple din clase cu o bază comună

O ambiguitate nouă care apare în legătură cu moștenirile din clase care au la rândul lor o bază comună este cea legată de upcasting. Cum este și firesc, putem continua conversiile de pointeri pe relația *is a*, până ajungem la baza comună; aşadar vom putea manipula obiecte D și prin pointeri de clasă BB, numai că o atribuire directă:

```
pb1 = &d;
```

produce ambiguitate, chiar din compilare. Dacă privim cum stau lucrurile structural (figura 3.3), ne explicăm ușor această ambiguitate: un obiect de tip D conține două subobiecte de tip BB, unul înglobat într-un obiect de tip B1 și unul înglobat într-un obiect de tip B2. Conversia adresei *&d* ar urma să extragă adresa unuia dintre subobiecte, dar care? Nu mai putem folosi rezoluția de clasă căci nu mai avem cu cine să asociem (d nu face parte din B1 și nici din B2), aşa încât singura rezolvare constă în etapizarea conversiilor:

```
BB * pbb; B1 *pb1; pb1=&d; pbb=pb1;
```

```
BB * pbb; B2 *pb2; pb2=&d; pbb=pb2;
```

După cum este ușor de dedus, conversiile de pointeri fac și modificări de adresă, nu numai schimbarea semantică pointerului; evident, cele două subobiecte BB au adrese diferite, iar conversiile în etape nu fac decât să identifice univoc unul dintre ele.

### Derivare virtuală

Dacă nu este dorită moștenirea în duplicat a membrilor unei clase comune BB, prin intermediul a două sau mai multe clase B1, B2 ..., alese ca bază a derivării, putem folosi conceptul de *clasă virtuală*, solicitând compilatorului la derivare să includă un singur exemplar:

Deși atributul de virtual se aplică lui B1 și B2, efectul se exercită asupra descendenților acestora. Clasele astfel deriveate se numesc *clase virtuale*; a nu se confunda cu clasele care conțin funcții virtuale pure, care se numesc abstracte, neputând avea obiecte concrete.

Exemplarul din BB se va moșteni pe așa numita *filieră dominantă*, adică dată de prima dintre clasele virtuale menționată în lista de derivare.

Constructorii claselor virtualizate se apelează totdeauna primii, indiferent de ordinea din lista de inițializare. În mod normal, constructorul clasei D trebuie să precizeze în clar transferurile de informații către constructorii lui B1 și B2:

```
D::D( ... ): B1( ... ), B2( ... ), BB( ... ) { }
```

Dacă nu o face, ultimul constructor implicit de clasă virtuală, care a lucrat pe *x*, lasă valoarea finală, regăsită ulterior în obiectul derivat.

```
class BB
{
    public:
        int x;           BB() { x=0; }
        virtual ~BB() {}
};

class B1 : virtual public BB
{
    public:          B1() { x=1; }
};

class B2 : virtual public BB
{
    public:          B2() { x=2; }
};
```

```
class D : public B1, public B2 {      };
```

Majoritatea compilatoarelor acceptă și acum adresările `d.B1::x` și `d.B2::x`, dar identifică același exemplar, unic moștenit din BB; oricum se trece prin toți constructorii, în ordinea de la derivare; acest lucru e important, când constructorii modifică un membru moștenit, astfel încât ultimul constructor apelat stabilește și inițializarea finală pentru acest membru.

```
void main() { BB *pbb; D d; cout << d.B1::x; cout << d.B2::x; }
```

Programul de mai sus afișează 2, în ambele cazuri, confirmând că `x` există într-un singur exemplar, a cărei valoare a fost stabilită de ultimul constructor care a lucrat pe el (B2, conform listei de derivare). Schimbând ordinea în lista de derivare sub forma:

```
class D : public B2, public B1 {      };
```

programul va afișa 1, în ambele cazuri, constructorul B1 fiind ultimul care a lucrat pe `x`.

O clasă de bază virtualizată nu se poate inițializa prin lista de inițializatori.

Nu se poate converti prin cast un pointer de clasă de bază în pointer de nici o clasă derivată din ea.

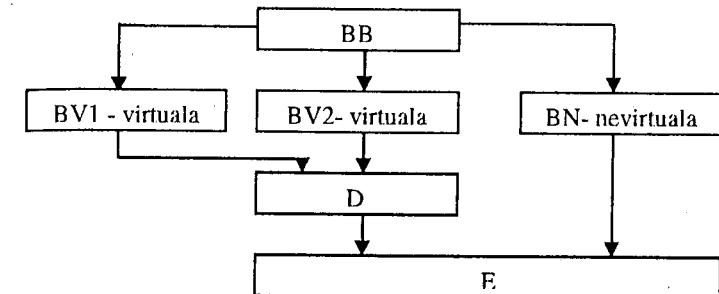
În schimb, **upcasting**-ul funcționează acum fără ambiguitate și peste mai multe niveluri; adică un pointer de clasă BB poate fi încărcat direct cu o adresă de obiect D, putând și el să implementeze polimorfismul prin virtualizarea funcțiilor, nu numai pointerii de clasă B1 sau B2:

```
void main()
{
    BB *pbb;
    pbb = new D; cout << pbb.x;
    delete pbb;
}
```

Acest lucru e posibil deoarece, derivând virtual pe **ambele filiere**, un obiect derivat conține un singur obiect BB, iar `pbb` va ști pe care să pointeze !

### Componente virtuale și nevirtuale în aceeași clasă

De remarcat că **toate filierele de moștenire** dintr-o bază comună trebuie filtrate prin **virtualizare**, altfel o filieră nevirtuală transferă totul prin moștenire.



*Fig. 3.4 Rețea de clase virtuale și nevirtuale*

Spre exemplu, clasa E din figura 3.4, va conține două exemplare din clasa de bază comună BB, unul moștenit virtual prin clasele virtuale BV1 și BV2 și unul moștenit nevirtual, prin intermediul clasei de bază nevirtualizată, BN.

### Ambiguități la funcții virtuale moștenite din clase derivate virtuale

Cea mai mare parte a ambiguităților ce țin de moștenirile din clase de bază ce au la rândul lor o bază comună, se rezolvă prin derivare virtuală, pe nivelul B1 - B2; dar ce se întâmplă cu moștenirile de funcții virtuale?

```

class BB
{
public:
    int x;
    virtual char* f() const = 0;
    virtual ~BB() {}
};

class B1 : virtual public BB
{
public:
    char* f() const { return "B1"; }
};

class B2 : virtual public BB
{
public:
    char* f() const { return "B2"; }
};
  
```

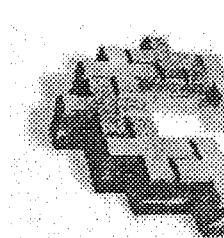
```
class D : public B1, public B2
{
    public:      char* f() const { return B1::f(); }
};
```

Deliberat, exemplul anterior a fost complicat, introducându-se și o funcție virtuală, pentru a remarcă acestă problemă apărută în cazul moștenirilor multiple. Funcția f() declarată **virtual pură** în BB, este explicitată în clasele B1 și B2; în acest fel, cele două forme apar pe zona specifică nivelului 2 (B1, B2), neputându-se prelua varianta din BB, virtual pură;

Dacă pe o ierarhie de derivare nu se dă o implementare pentru o funcție virtuală, cea mai recentă variantă dată, va fi cea moștenită implicit pe nivelurile inferioare. Deci dacă nu dăm o versiune de implementare în D, ne-am aștepta să existe firesc o versiune moștenită; din păcate există două versiuni moștenite (cea din B1 și cea din B2) aflate pe același nivel de derivare față de D, iar compilatorul nu poate alege în locul nostru. Ieșirea din ambiguitate se poate face dând o versiune proprie în D, sau redirectând în D funcția f(), către una din versiunile moștenite, dar acest lucru cade în sarcina programatorului, nefiind implicit.

```
void main()
{
    BB *pbb;
    pbb = new D;
    cout << pbb->f();
    delete pbb;
}
```

Programul de mai sus afișează *B1*, confirmându-ne că obiectul derivat deține o formă a funcției virtuale f() și anume cea aleasă prin redirectare. În general, compilatoarele pot da un **avertisment** la adresarea *d.f()*, semnalând că e vorba de funcția moștenită pe filiera dominantă; altele sancționează cu **eroare** o adresare de forma *d.B2::f()*, care forțează filiera B2, când filiera dominantă este B1 !



## apitolul IV

### OPERAȚII DE INTRARE / IEȘIRE ORIENTATE PE STREAM-URI

- Lucru cu obiectele *cin* și *cout*
- Intrări / ieșiri cu formatarea datelor
- Detectarea erorilor apărute în operațiile de intrare / ieșire
- Intrări / ieșiri pe fișiere nestandard
- Formatarea datelor în memoria internă

## 4.1 Lucru cu obiectele *cin* și *cout*

În viziunea orientată obiect, limbajul C++ pune la dispoziția programatorului două obiecte, predefinide în fișierul header *iostream.h*, unul cu rolul de afișare pe display (*cout*) și altul cu rolul de citire de la consolă (*cin*) a valorilor unor variabilelor.

Viziunea obiectuală asupra operațiilor de intrare / ieșire are avantajul securității manipulării informațiilor și fișierelor. Constructorii fac automat toate inițializările, iar destrutorul închide fișierele, scutind programatorul de sarcini ce pot fi automatizate.

Un flux (*stream*) de intrare / ieșire este un obiect capabil să stocheze și / sau să formateze baiți de informație. Pentru aceasta dispune de un membru buffer (de clasă *streambuf*) și de metode de formatare. Operatorii << și >> au fost supraîncărcați pentru a declanșa operații de intrare / ieșire. Sensul săgeților indică direcția de "curgere" a informațiilor. Când fluxul curge de la tastatură, disc sau rețea ce corespund unor obiecte stream, către variabile în memorie se cheamă că extragem informație din stream, iar operatorul folosit (>>) se numește *extractor* și realizează o operație de intrare. În sens invers, avem de-a face cu o inserare de informație din memorie în obiectul stream, iar operatorul implicat (<<) se numește *inserter*, căci introduce o informație în obiect.

Obiectul *cout* este de clasă *ostream* și încapsulează elementele necesare afișării de date de tipuri fundamentale. Obiectul *cin* este de clasă *istream* și încapsulează elementele necesare citirii de la consolă a valorilor variabilelor de tipuri fundamentale.

În manieră procedurală, aceste operații erau efectuate cu ajutorul funcțiilor *printf* și *scanf*. Programul:

```
#include <stdio.h>
void main()
{
    int a,b;
    printf("\n Introduceti doi intregi:"); scanf("%d%d",&a,&b);
    printf("Daca se aduna %d cu %d se obtine %d",a,b,a+b);
}
```

este echivalent din punct de vedere funcțional cu programul:

```
#include <iostream.h>
void main()
{
    int a,b;
    cout<<"\n Introduceti doi intregi:"; cin>>a>>b;
    cout<<"Daca se aduna <<a<< cu <<b<< se obtine <<a+b;
}
```

Analizând programul care folosește obiecte pentru realizarea operațiilor de intrare / ieșire se poate observa că:

- trebuie să se includă fișierul *iostream.h*, pentru că acolo sunt definite obiectele *cout* și *cin*;
- se folosește operatorul << în combinație cu *cout* pentru afișare pe ecran;
- se folosește operatorul >> în combinație cu *cin* pentru citire de la tastatură;
- pentru citirea, respectiv afișarea mai multor variabile se înlănuie componentele cu ajutorul operatorului >> sau <<, invocând o singură dată obiectul, de exemplu *cin>>a>>b*;

Privind construcția *cin>>a>>b*; mai tehnic, din punct de vedere al limbajului C++, ea este o expresie unde *cin*, *a* și *b* sunt operanzi de tip flux de intrare respectiv, *int* și *int*, iar >> este operatorul de intrare. Așa cum expresia  $5+2+3$  se evaluează de la stânga la dreapta și mai întâi se adună 5 cu 2, rezultă 7 (un întreg) care se adună cu 3 și rezultatul final este 10, expresia *cin>>a>>b*; se evaluează tot de la stânga la dreapta și mai întâi se efectuează *cin>>a* ceea ce determină citirea valorii variabilei *a*; pentru a realiza și citirea lui *b*, din evaluarea parțială (*cin>>a*) va rezulta tot un *cin*.

Modul de descriere a acestor operații sugerează un flux de operații de intrare sau ieșire, de aceea în literatura de specialitate se folosește conceptul de *stream* sau flux de intrare / ieșire.

Operațiile de intrare / ieșire pot fi realizate și prin apelul altor metode specifice, nu numai prin construirea unor expresii folosind operatorii de deplasare pe biți. Astfel, ne putem aminti că preferam folosirea funcției *gets()* pentru citirea sirurilor de caractere în locul funcției *scanf()*, deoarece funcția *gets()* avea ca terminator doar codul tastei ENTER ('\n'). De exemplu, dacă se folosește secvența:

```
char np[50];
cin>>np;
cout<<np;
```

pentru a introduce numele unei persoane și se tastează *Popescu Adrian*, atunci se va afișa doar *Popescu*, deci *cin* pentru siruri se comportă similar cu funcția

`scanf()` cu descriptorul `%s`. Pentru a evita acest neajuns se va folosi metoda `getline()` a clasei `istream`, în maniera:

```
char np[50];
cin.getline(np,50);
cout<<np;
```

În variabila `np` va fi încărcat întreg sirul introdus, după cum se va observa și la afișare.

Metoda `getline()` are ca parametri: adresa unde se vor memora caracterele tastate și numărul maxim de caractere care se vor stoca la acea adresă. Ea mai poate primi un ultim parametru, ce are valoarea implicită `'\n'`, care precizează terminatorul operației de intrare. Dacă se dorește schimbarea valorii lui, el se va explicita în apelul metodei, ca în exemplul:

```
char sir[30];
cin.getline(sir,30,'#');
```

Caracterul `#` devine în acest caz terminator al operației de citire.

Pentru afișarea unui sir cu lungime cunoscută se poate folosi metoda `write()` care primește ca parametri sirul de afișat și lungimea lui.

Secvența:

```
char sir1[10];
cout<<"\n Numele si prenumele:";
cin.getline(sir1,9);
cout.write(sir1,strlen(sir1));
```

realizează citirea și afișarea unui sir de caractere. Reamintim că funcția `strlen` determină lungimea unui sir de caractere și are prototipul în fișierul header `string.h`.

Există metode specializate pentru citirea, respectiv scrierea unui caracter. Acestea sunt `get()` și `put()`. Secvența:

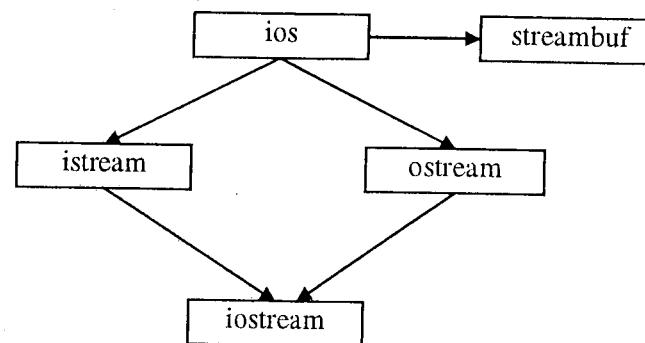
```
char x;
cout<<"\n Dati un caracter:";
x=cin.get();
cout.put(x);
```

introduce un caracter și apoi îl afișează. Se observă că metoda `get()` returnează codul caracterului citit.

Din prezentarea făcută ați putut afla că implementarea obiectuală a operațiilor de intrare / ieșire presupune existența mai multor clase. Acestea nu sunt clase independente ci formează o ierarhie complexă descrisă schematic în figura 4.1. De bază sunt două clase:

- `streambuf` – care gestionează memoria pentru bufferele de intrare ieșire;
- `ios` – care modelează lucru cu un flux în general sub aspectul: formatării datelor, determinării stării fluxului, tratării erorilor apărute în lucru cu fluxul etc. și face legătura cu clasa `streambuf` prin intermediul unui pointer la un obiect `streambuf` care este membru al clasei `ios`;

Din clasa `ios` derivă două clase specializate pentru a realiza operații de intrare / ieșire pe dispozitive standard (`istream` și `ostream`). Clasa `iostream` se află în relație de moștenire multiplă cu clasele `istream` și `ostream`.



*Fig. 4.1 Clase implicate în implementarea operațiilor de intrare / ieșire pe dispozitive standard*

În mediul de programare Microsoft Visual C++ sunt date și definiții noi ale acestor clase, mai precis sunt şabloane de clase, iar pentru a le putea folosi se va include fișierul `iostream` (fără nici o extensie).

## 4.2 Intrări / ieșiri cu formatarea datelor

Din exemplele prezentate, în care se efectuau operații de intrare / ieșire, s-a putut observa că lipsesc specificatorii de format. După cum ne amintim, funcțiile `printf` și `scanf` erau de neutilizat fără acești specificatori, chiar dacă de cele mai multe ori se folosea forma minimală (doar descriptorii de format: `%d`, `%s`, `%lf` etc). Modalitatea obiectuală de efectuare a operațiilor de intrare / ieșire pare mai puțin pretențioasă folosind conversii

deduse din tipul variabilelor din lista de intrare / ieșire, însă în aplicații este câteodată necesar a se forma explicit datele, mai ales când acestea trebuie prezentate într-o formă tabelară.

Pentru a avea o vizuire de ansamblu asupra formatării datelor vom prezenta modul cum sunt definite informațiile necesare formatării precum și felul în care ele pot fi modificate.

#### A. Informație necesară formatării stocată în variabile fundamentale

##### 1. caracterul de umplere (*fill*)

*int ios::fill( )* – returnează codul caracterului folosit la umplerea spațiilor libere din zona de afișare (implicit este blanc);

*int ios::fill(int)* – stabilește caracterul folosit la umplerea spațiilor libere din zona de afișare; returnează caracterul folosit anterior acestui apel.

##### 2. lățimea zonei de afișare (*width*)

*int ios::width( )* – returnează valoarea curentă a lățimii zonei de afișare (0, dacă n-a fost stabilită explicit; valoarea zero indică dimensionarea la minim a zonei de afișare necesară reprezentării fără trunchieri a valorii de afișat).

*int ios::width(int)* – stabilește lățimea zonei de afișare; o valoare pozitivă indică dimensiunea minimă a zonei de afișare; dacă din conversie au rezultat mai multe caractere, toate vor fi afișate, depășind valoarea existentă în *width*; dacă din conversie au rezultat mai puține caractere, după cadrul, spațiile libere se umplu cu caracterul din câmpul *fill*.

Deoarece este doar orientativă, putând să fie modificată de valorile afișate ce necesită o lățime mai mare, câmpul *width* este repus automat pe zero, după fiecare inserție sau extracție.

##### 3. precizia afișării (*precision*)

*int ios::precision( )* – returnează precizia folosită la afișarea valorilor în virgulă mobilă; implicit este 6;

*int ios::precision(int)* – fixează precizia folosită la afișarea valorilor în virgulă mobilă; returnează valoarea existentă anterior acestui apel. Interpretarea preciziei depinde de formatul de afișare (științific, fix sau automatic)

#### B. Indicatoare (flag-uri) pentru controlul formatării, cu informație stocată la nivel de bit

##### 1. Indicatoare de tip *on / off* (cu două stări)

Acești indicatori sunt: *ios::skipws*, *ios::showbase*, *ios::showpoint*, *ios::showpos*, *ios::uppercase*, *ios::unithbuf*, *ios::stdio* efectul lor este prezentat în tabelul 4.2.

#### 2. Indicatoare organizate pe câmpuri

Aceste indicatoare sunt grupate după rolul lor, atât câmpul în ansamblu, cât și fiecare indicator în parte este identificat printr-o constantă enumerativă. La un moment dat doar un indicator din grup are sens să fie activat, iar ceilalți trebuie dezactivați (excludere reciprocă); din păcate setarea unuia nu-i dezactivează automat pe ceilalți, astfel încât a fost nevoie de o supraîncărcare a funcției *setf()*, care să primească atât codul întregului câmp (pentru a-l dezactiva), cât și codul indicatorului din acest grup (pentru a-l activa). Neinspirat, ordinea parametrilor este inversă succesiunii de derulare a acțiunilor: mai întâi sunt dezactivați toți biții din câmp, după care este setat bitul care interesează. Se poate ghici deci ușor, că toate constantele enumerative corespunzătoare indicatoarelor sunt puteri ale lui 2, individualizând câte un bit.

Spre exemplu, apelul:

```
cout.setf(ios::hex, ios::basefield);
```

dezactivează mai întâi toți biții din grupul *basefield*, ce precizează baza de conversie (*dec*, *oct*, *hex*), după care activează bitul corepunzător bazei 16.

Câmpurile și valorile ce le pot lua sunt prezentate în tabelul 4.3.

Formatarea în operațiile de intrare / ieșire se face în mai multe moduri.

##### 1 O primă modalitate o constituie *apelul unor metode prin intermediul obiectelor cin și cout*.

De exemplu, dacă se dorește afișarea unui număr întreg într-un spațiu de 10 caractere, cu umplerea spațiilor neocupate cu caracterul '\*', se va scrie secvența:

```
int a=2371;
cout.fill('*'); cout.width(10);
cout<<a<<"\n";
```

care va afișa:

```
*****2371
```

Se observă că:

- metoda *width()* stabilește lungimea în caractere a zonei în care se va afișa o anumită valoare;
- metoda *fill()* precizează caracterul de umplere a spațiului neocupat, fixat prin apelul metodei *width()*, dacă pentru afișarea unei valori e necesar un număr mai mic de caractere.

Apelul metodelor de formatare în operațiile de intrare / ieșire în forma prezentată are ca dezavantaj modul prea laborios de scriere a secvenței de program.

- ② O altă modalitate de formatare o constituie *folosirea manipulatorilor*. În esență, manipulatorii sunt funcții speciale care se plasează în lanțul de operatori << sau >>. Se poate ușor deduce că ele întorc tot o referință de *stream*, ce va prelua mai departe sarcinile de intrare / ieșire. Pentru funcțiile care necesită argumente programul va include și fișierul header *iomanip.h*. Secvența de program anterioară se poate scrie, folosind manipulatori, astfel:

```
int a=2371;
cout<<setfill('*')<<setw(10)<<a<<endl;
```

Manipulatorii și efectul pe care ei îl realizează sunt prezentati în tabelul 4.1.

Manipulator	Efect
dec	convertește întregii în baza 10; echivalent cu %d
hex	convertește întregii în baza 16; echivalent cu %x
oct	convertește întregii în baza 8; echivalent cu %o
setprecision(int)	precizează numărul de cifre zecimale ale unui număr real
endl	trimită o linie nouă fluxului de ieșire și descarcă buffer-ul
ends	insereză caracterul nul (\0) în flux
ws	elimină spațiile libere din fluxul de intrare
flush	descarcă buffer-ul fluxului de ieșire
setbase(int)	setează baza de numerație pentru un întreg (poate fi: 8, 10, 16)
setfill(int)	stabilește caracterul folosit la umplerea spațiilor libere din zona de afișare
setw(int)	stabilește lățimea zonei de afișare
setiosflags(long)	stabilește valoarea bițiilor de formatare specificați în argument
resetiosflags(long)	reinițializează valoarea bițiilor de formatare specificați în argument

*Tabelul 4.1 Manipulatori*

Folosirea manipulatorilor care nu au argumente nu necesită includerea fișierului *iomanip.h*.

- Exemplu de folosire a unor manipulatori:

```
#include <iostream.h>
#include <iomanip.h>
```

void main()

```
{
    int a,b=100;
    cout<<"\n Numarul 100 in octal este:" <<oct<<b<<endl;
    cout<<" Introduceti un nr. in baza 16:"; cin>>hex>>a;
    cout<<"\n Nr. introdus este: "<<hex<<a<<" in baza 10: "<<dec<<a;
}
```

Programatorul poate construi proprii manipulatori, sub forma unor funcții ce primesc un flux de ieșire, îl modifică după cum doresc, apoi îl returnează mai departe, spre folosire în cascadă. Să presupunem că vrem ca sumele în valută să fie prefixate cu semnul valutei și să aibă format fix cu 5 zecimale exacte. Manipulatorul de mai jos introduce *money\_format* folosit mai apoi în *main* :

```
#include <iostream.h>
#include <iomanip.h>

ostream & money_format( ostream &s)
{
    s<< setiosflags( ios::fixed);
    s<< setprecision(5) << setfill('$') << setw(15);
    s<< setiosflags( ios::showpoint);
    return s;
}

void main(int argc, char* argv[])
{
    double a=12345.6789;
    cout << money_format <<a<< setprecision(2);
    cout << "\n" <<a;
}
```

- ③ Formatarea datelor se realizează și prin *modificarea unor indicatori* (biți) dintr-o variabilă de tip *long*, care este membră a clasei *ios*.

Constanțele ce se vor folosi pentru a seta anumiți biți din variabila *long* sunt definite sub forma unui tip enumerativ, membre în clasa *ios*. Metoda care setează anumiți biți din variabila *long* (*flag-uri*) este *setiosflags()* și primește ca parametru o dată de tip *long* sub forma unei constante enumerative definită în clasa *ios*, cu rol bine definit în formatarea datelor. Prototipul metodei este:

```
long ios::setiosflags( long );
```

Metoda *resetiosflags()* primește ca parametru tot un *long*, sub forma unei constante, la fel ca la metoda *setiosflags()* și realizează reinițializarea

bițiilor respectivi, adică aducerea pe valorile implicate ale câmpului referit prin oricare din membrii sai.

Constantă	Efect
ios::left	cadrare la stânga
ios::right	cadrare la dreapta
ios::internal	fixează caracterul de umplere zona
ios::dec	conversia unui întreg în baza 10
ios::hex	conversia unui întreg în baza 16
ios::oct	conversia unui întreg în baza 8
ios::showpos	afişarea semnului chiar dacă numărul e pozitiv
ios::showbase	afisează prefixul bazei de numerație (0x pentru hexazecimal și 0 pentru octal)
ios::scientific	reprezentarea unui real în format științific
ios::fixed	reprezentarea unui real în format fix
ios::showpoint	afişarea punctului zecimal chiar dacă numărul nu are cifre semnificative la partea zecimală
ios::skipws	skip white space – salt peste spații albe (blanc, tab, enter...); este implicit pentru operațiile de intrare;
ios::stdio	sincronizarea operațiilor de I/E, când se folosesc amestecat funcții aparținând ambelor mecanisme, stream-uri declarate în <i>iostream.h</i> , respectiv funcții independente declarate în <i>stdio.h</i> . Altfel, ordinea derulării operațiilor nu mai corespunde succesiunii în care apar în program (afisează cu <i>printf</i> un mesaj de cerere date, după ce deja le citise cu <i>cin&gt;&gt;</i> )
ios::uppercase	folosirea majusculelor A-F la afişarea cifrelor hexa, pentru întregi, respectiv E la afişarea valorilor în virgulă mobilă, în notația exponentială
ios::unitbuf	buffer-izare unitară; la fiecare operație de scriere, buffer-ul este golit pe loc, fără a se aștepta mai întâi umplerea lui completă; în acest fel funcționarea devine unitară (similară operației de intrare)

Tabelul 4.2 Constante pentru stabilirea indicatorilor de formatare a datelor

Ca exemplu, se va afișa o situație scolară privind rezultatele la o sesiune în care s-au susținut două examene. Situația va fi afișată sub forma unui raport care conține numele studentului, notele la cele două examene și media obținută (cu două zecimale).

```
#include <iostream.h>
#include <iomanip.h>
struct student
{
```

```
    char np[30];
```

```
    int note[2];
};

void main()
{
    student s[] = {
        {"Ionescu P.", {10,8}},
        {"Vladimir T.", {9,8}}
    };
    int i,n = sizeof(s)/sizeof(student);
    cout<<setw(50)<<setiosflags(ios::right)<<"Situatie scolara"<<endl<<endl;
    cout<<"      Nume si prenume      " << " Matematica"
        << " Informatica" << " Media" << endl<<endl;
    cout<<setw(67)<<setfill('=')<<" " << endl<<setfill(' ');
    for(i=0;i<n;i++)
        cout<<setw(32)<<setiosflags(ios::left)<<s[i].np<<setw(10)
            <<resetiosflags(ios::left)<<s[i].note[0]
            <<setw(12)<<s[i].note[1]
            <<setw(7)<<setiosflags(ios::fixed)<<setprecision(2)<<
                (s[i].note[0]+s[i].note[1])/2.<<endl;
}
```

După cum se poate observa din programul anterior, metodele *setiosflags()* și *resetiosflags()* constituie manipulatori pentru că întorc referință de stream și se pot apela în fluxul operațiilor de intrare sau ieșire.

Biți cu rol în formatarea operațiilor de intrare sau ieșire pot fi modificați și prin apelul unor metode obișnuite, deci nu ca manipulatori.

Metoda *setf()* primește doi parametri și are prototipul:

```
long ios::setf( long val, long indic );
```

- *val* – specifică valoarea indicatorului;
- *indic* – precizează indicatorul ce urmează a fi modificat.

Indicatorii ce sunt definiți în clasa *ios* și constantele acceptate sunt prezentate în tabelul 4.3.

Indicatorii	Constance acceptate	Efect
basefield	ios::dec, ios::hex, ios::oct	modifică baza de numerație
floatfield	ios::fixed, ios::scientific	specifică modul de reprezentare a numerelor reale
adjustfield	ios::left, ios::right, ios::internal	stabilizează cadrul datei

Tabelul 4.3 Indicatorii și constantele asociate cu rol în formatarea datelor

Ca exemplu, se va descrie secvența pentru afișarea unei variabile de tip *double* în format științific, într-un spațiu de 20 caractere, cadrată la stânga.

```
double a=11263.89876;
cout.setf(ios::scientific,ios::floatfield);
cout.setf(ios::left,ios::adjustfield);
cout<<setw(20)<<a;
```

Resetarea valorii indicatorului se face prin apelul metodei *unsetf* ce are prototipul:

```
long ios::unsetf( long );
```

Spre exemplu, apelul:

```
cout.unsetf(ios::showpos);
```

Inhibă afișarea semnului + pentru numerele pozitive.

### 4.3 Detectarea erorilor apărute în operațiile de intrare / ieșire

Este foarte important ca în cadul unui program să se obțină informații cu privire la corectitudinea derulării unei operații, mai ales când aceasta implică un periferic.

Erorile sunt semnalate prin setarea unor indicatori (biți) de stare ai fluxurilor. Acești biți sunt setați prin intermediul unor constante, definite într-un tip enumerativ în clasa *ios*. Biți desemnați prin intermediul constantelor poartă denumiri semnificative:

- *failbit* – setat cu valoarea 1 dacă o operație de intrare a detectat un caracter neconcordant cu tipul variabilei sau dacă o operație de ieșire nu s-a executat corect;
- *badbit* – setat cu valoarea 1, dacă fluxul este eronat (eroare la citire). Există metode ce nu primesc parametri, dar returnează printr-un întreg cu semnificația de adevărat sau fals modul cum a decurs o operație de intrare / ieșire. Dintre cele mai folosite sunt:
  - *good()* – ce returnează un întreg nenul (adevărat) dacă nici un bit de eroare nu este setat;

- *fail()* – returnează un întreg nenul (adevărat) dacă bitul *failbit* sau *badbit* este setat;
- *bad()* – returnează un întreg nenul (adevărat) dacă bitul *badbit* este setat.

Un exemplu de apariție a unei erori constă în faptul că se dorește citirea unui întreg și se tastează caractere nenumerice:

```
int a;
cout<<"\n Dat un nr. intreg:";
cin>>a;
if(cin.fail()) cerr<<"Eroare de citire a variabilei a !";
```

*cerr* este un flux de ieșire conectat la dispozitivul standard de eroare (monitorul) și spre deosebire de *cout* nu folosește nici un buffer, deci va afișa imediat mesajul, fără a aștepta umplerea buffer-ului.

Metoda *clear()* resetează flag-urile de eroare pentru a putea fi ulterior sesizate alte erori detectate în operațiile de intrare / ieșire și totodată pentru a debloca lucru cu fluxul respectiv. Metoda *clear()* are un parametru cu valoarea implicită, 0 ce semnifică faptul că se resetează toți indicatorii de eroare ai fluxului. Este posibil să și poziționăm un anumit indicator de stare a fluxului, ceilalți rămânând nemodificați printr-un apel de genul:

```
cin.clear( cin.rdstate() | flag );
```

unde:

- *cin* – precizează tipul fluxului (de intrare);
- *rdstate()* – este o metodă a clasei *ios* care returnează starea fluxului sub formă unui întreg în care sunt împachetați indicatorii de stare;
- *flag* – este indicatorul de stare ce va fi setat; el se va indica printr-o constantă de genul: *failbit*, *badbit* etc.

O altă modalitate de semnalare a erorilor o constituie utilizarea expresiei de citire sau scriere pe post de condiție în instrucțiuni. Ca exemplu, să presupunem că dorim citirea elementelor unui vector până la tastarea lui CTRL/Z. Secvența de program este:

```
int v[100],i=0;
while( cout<<"\n Dat elementul ["<<i<<"]=", cin>>v[i] ) i++;
```

În acest caz se folosește conversia implicită a lui *istream* la *void \**, adică va returna NULL dacă s-a produs o eroare la operația de citire; eroarea este semnalată la tastarea lui CTRL/Z. Dacă în continuarea secvenței de program să ar mai dori citirea unei alte variabile, atunci ar trebui resetați indicatorii de eroare. De exemplu, presupunem că mai citim un număr real:

```
double t;
cin.clear();
cout<<"\n Dati un nr. real:";
cin>t;
```

O altă cale pentru depistarea erorilor constă în aplicarea operatorului ! stream-ului. Acest operator este supraîncărcat cu sensul că va returna diferit de 0 dacă a apărut o eroare la operația de intrare / ieșire; 0 dacă ea a decurs corect:

```
int a;
cout<<"\n Dati un nr. intreg."; cin>a;
if(!cin) cerr<<"EROARE!!!";
```

Numai detectarea erorilor nu este suficientă pentru a realiza corecția operațiilor de intrare / ieșire. Astfel, este necesar să efectuăm și golirea buffer-ului aferent tastaturii pentru ca o variabilă ce trebuie încărcată prin citire să nu preia caractere rămase în buffer de la o introducere anterioară.

De exemplu, dacă se citește un întreg și apoi un sir de caractere, secvența:

```
int a;
char x[100];
cout<<"\n Dati un nr. intreg."; cin>a;
cout<<" Dati sirul: "; cin>x;
cout<<"\n a=<<a<<" sirul:"<<x;
```

are dezavantajul că dacă se introduc de la tastatură caractere numerice și nenumericе, din buffer se vor prelua doar primele caractere numerice consecutive și se vor asigna variabilei *a*, cele nenumericе vor fi lăsate în buffer. Când urmează să se citească sirul de caractere, programul nu va mai aștepta tastarea de noi caractere, ci vor fi preluate cele rămase în buffer, cum se poate vedea din rezultatul secvenței anterioare:

```
Dati un nr. intreg:453dsf
Dati sirul:
a=453 sirul:dsf
```

În cazul în care se dorește citirea distinctă a celor două variabile va trebui să se golească buffer-ul înainte de citirea sirului de caractere.

◆ Un mod de a efectua golirea buffer-ului constă în extragerea explicită a fiecărui caracter din buffer, cât timp caracterul extras e diferit de ENTER ('\n'), secvența este:

```
while(cin.get()!='\n');
```

- ◆ Un alt mod de golire a bufferului presupune apelul metodei *ignore* a clasei *istream* care are prototipul:

```
istream& istream::ignore(int=1, int _delim=EOF);
```

unde, primul parametru specifică numărul maxim de caractere ce vor fi scoase din buffer, cel de-al doilea specifică terminatorul, adică ultimul caracter care va fi scos din buffer; în caz că acesta nu există în buffer se vor scoate caractere în numărul specificat de primul parametru.

Secvența de citire a variabilelor, prezentată anterior, se va completa cu cod pentru detectarea erorilor și pentru golirea buffer-ului astfel:

```
int a;
char x[100];
cout<<"\n Dati un nr. intreg.";
cin>a;
if(!cin) cerr<<"EROARE!!!";
cin.clear();
cin.ignore(256,'\n');
cout<<" Dati sirul: ";
cin>x;
cout<<"\n a=<<a<<" sirul:"<<x;
```

Se observă că metoda *ignore* a fost apelată astfel încât să scoată din buffer maxim 256 caractere sau până întâlnește caracterul cu codul '\n', adică ENTER.

#### 4.4 Intrări / ieșiri pe fișiere nestandard

Folosirea stream-urilor pentru a realiza operații de intrare / ieșire a fost extinsă și pentru fișiere nestandard. Biblioteca C++ pune la dispoziția programatorului clasele:

- *ifstream* – pentru lucru cu fișiere în intrare (existente), în vederea consultării;
- *ofstream* – pentru lucru cu fișiere în ieșire, în scopul scrierii de informații în ele;
- *fstream* – pentru lucru cu fișiere în intrare / ieșire, adică este permisă atât consultarea, cât și scrierea de informații în ele.

Lucrul cu obiecte de aceste tipuri impune includerea fișierului header *fstream.h*. Aceste clase încapsulează metodele uzuale de lucru cu fișiere,

cum ar fi: deschiderea, închiderea, poziționarea, citirea / scrierea, cu sau fără format.

Dacă în programarea structurată, identificatorul intern al fișierului era un *FILE \** sau un întreg numit *handler*, folosind stream-uri, unui fișier i se va asocia un obiect de tipurile precizate anterior.

Asocierea dintre un fișier și obiectul care va fi identificatorul lui intern se poate realiza în două moduri:

- prin apelul constructorului care are ca parametru explicit numele fișierului:

```
ifstream fs("fis.dat");
```

- prin apelul metodei *open*, care primește ca parametru explicit numele fișierului de deschis:

```
ofstream fd;
fd.open("persoane.dat");
```

Pe lângă numele fișierului se mai poate specifica, explicit, modul în care se va deschide fișierul. Acest lucru se precisează cu ajutorul unor constante definite cu ajutorul tipului enumerativ, în clasa *ios*. Ele se pot combina folosindu-se operatorul SAU pe biți (|). Aceste constante sunt prezentate în tabelul 4.4.

Constantă	Efect
<i>ios::in</i>	fișier deschis în input
<i>ios::out</i>	fișier deschis în output
<i>ios::ate</i>	după deschidere poziția curentă va fi la sfârșitul fișierului
<i>ios::app</i>	deschidere pentru scriere la sfârșitul fișierului
<i>ios::trunc</i>	sterge conținutul fișierului dacă el există
<i>ios::nocreate</i>	dacă fișierul nu există atunci el nu va fi creat
<i>ios::noreplace</i>	nu deschide fișierul dacă el există
<i>ios::binary</i>	deschide fișierul în mod binar

Tabelul 4.4 Constante ce definesc modul de deschidere a unui fișier

Exemple:

Dacă se dorește deschiderea fișierului *fis1.dat* ca fiind binar, în ieșire, pentru a adăuga informații, se poate apela constructorul în forma:

```
ofstream fi("fis1.dat", ios::applios::binary);
```

Să se deschidă fișierul *fis2.dat* ca fiind binar în intrare și ieșire și să nu fie suprascris dacă el există; se va folosi apelul metodei open în forma:

```
fstream fie;
fie.open("fis2.dat", ios::inios::outios::noreplacelios::binary);
```

Operatorii *>>* și *<<* se aplică acestor obiecte pentru a realiza operații de citire, respectiv scriere din / în fișiere cu formatarea datelor.

Ca exemplu, să se creeze fișierul *mat.dat*, listabil ASCII, deci în format extern, care să conțină informații referitoare la materiale, după cum urmează:

- cod material de tip întreg;
- denumire material, sir de maxim 20 caractere;
- unitate de măsură, sir de maxim 4 caractere;
- cantitate, de tip întreg;
- preț unitar de tip double, șase cifre la partea întreagă și două la cea zecimală.

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
void main()
{
    int cm, cant;
    char dm[20], um[4];
    double pu;
    ofstream fiso("mat.dat");
    if(!fiso)
    {
        cerr<<"Fisierul nu se deschide!!!!"; return;
    }
    while(cout<<"\nCod material:", cin>>cm)
    {
        cin.ignore();
        cout<<"Denumire material:"; cin.getline(dm,19);
        cout<<"Unitate de masura:"; cin.getline(um,4);
        cout<<"Cantitate:"; cin>>cant;
        cout<<"Pret unitar:"; cin>>pu;
        fiso<<setw(5)<<cm<<setw(22)<<setiosflags(ios::left)<<dm
            <<setw(5)<<um<<setw(7)<<resetiosflags(ios::left)<<cant
            <<setw(9)<<setiosflags(ios::fixed)<<setprecision(2)<<pu<<endl;
    }
    fiso.close();
```

A se observă:

- modul de formatare a datelor ce urmează a se scrie în fișier, care este similar cu cel folosit pentru afișarea datelor pe dispozitivul standard de ieșire (monitorul);
- modul de depistare a erorilor apărute în lucru cu stream-uri pe fișiere nestandard, care este similar cu cel descris în subcapitolul anterior;
- modul de control al întreruperii citirii (tastarea CTRL/Z pentru cod material).

Pentru testarea rezultatului programului anterior, să se construiască un alt program care să citească informații din fișierul creat anterior (*mat.dat*) și să afișeze o situație conținând: codul materialului, denumirea lui, unitatea de măsură, cantitatea, prețul și valoarea, afișată cu două zecimale.

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>

void main()
{
    int cm, cant;
    char dm[20], um[4];
    double pu;
    ifstream fisi("mat.dat");
    if(!fisi)
    {
        cerr<<"Fisierul nu se deschide!!!!";
        return;
    }
    while(fisi>>cm>>dm>>um>>cant>>pu)
        cout<<setw(5)<<cm<<setw(22)<<setiosflags(ios::left)<<dm
            <<setw(5)<<um<<setw(7)<<resetiosflags(ios::left)<<cant
            <<setw(9)<<setiosflags(ios::fixed)<<setprecision(2)<<pu
            <<setw(16)<<cant*pu<<endl;
    fisi.close();
}
```

Pentru tratarea fișierului la nivel de octet, stream-urile pun la dispoziția programatorului metodele *get* și *put*. Se va exemplifica folosirea lor printr-un program care va copia un fișier (sursă) în altul (destinație). Numele celor două fișiere vor fi indicate prin intermediul argumentelor funcției *main()*.

```
#include <fstream.h>
#include <ctype.h>
```

```
void main(int argc, char *argv[])
{
    if(argc!=3)
    {
        cerr<<"!!! EROARE numar necorespunzator de argumente!!!!";
        return;
    }
    ifstream fs(argv[1], ios::binary | ios::nocreate);
    if(!fs) { cout<<"\n !!! Fisierul "<<argv[1]<<" nu exista!!!!"; return; }
    ofstream fd(argv[2], ios::binary | ios::noreplace);
    if(!fd)
    {
        char rasp;
        do
        {
            cout<<"\n !!! Fisierul "<<argv[2]<<" deja exista!!!!"<<endl;
            cout<<"Suprascrieti fisierul (D/N) ???";
            cin>>rasp;    cin.ignore(256,'n');
        }
        while(toupper(rasp)!='D' && toupper(rasp)!='N');
        if(toupper(rasp)=='N') return;
    }
    fd.clear();
    fd.open(argv[2], ios::binary);
    int c;
    while( ( c=fs.get() ) != EOF ) fd.put ( (char)c );
    fs.close();           fd.close();
    cout<<"\n Copiere terminata !!";
}
```

A se observă modul de depistare a existenței sau inexistenței fișierului. Astfel, fișierul sursă a fost deschis, astfel încât să se semnaleze dacă el există, setând indicatorul *ios::nocreate*. Fără poziționarea acestui indicator fișierul sursă ar fi fost creat și nu se semnala o eroare pe flux. În cazul fișierului destinație, se observă că a fost deschis astfel încât să nu fie rescris decât în caz că utilizatorul dorește acest lucru. După semnalarea erorii, pentru refolosirea fluxului s-au sters indicatorii de eroare cu metoda *clear()*.

Sfârșitul de fișier a fost sesizat prin faptul că apelul metodei *get()* a returnat EOF, adică -1 ca și în cazul folosirii funcției *getc()* din *stdio.h*.

Metodele *read* și *write* au o mai mare generalitate și efectuează citirea / scrierea din / în fișier a unui număr specificat de baiți la / de la o adresă dată ca parametru a metodelor, fără să efectueze formatarea datelor.

Metoda *read* are prototipul:

```
istream& istream::read(char *, int);
```

unde, primul parametru reprezintă adresa unde se va scrie, în memoria internă, ceea ce se va citi din flux; al doilea parametru precizează câți bajți se vor citi din flux și se vor scrie la acea adresă.

Metoda *write* are același prototip doar că semnificațiile sunt diferite, primul parametru este adresa de memorie de unde se vor scrie în flux un număr de bajți precizat prin intermediul celui de-al doilea parametru. Metoda se va apela pornindu-se de la un obiect *ostream*.

Ca exemplu, să se construiască un fișier cu numele *mat.dat* care să conțină informații referitoare la materiale, după cum urmează:

- *cm*, cod material de tip întreg;
- *denm*, denumire material de tip sir de maxim 30 caractere;
- *um*, unitate de masură de tip sir de maxim 4 caractere;
- *cant*, de tip întreg;
- *pu*, preț unitar de tip *double*.

```
#include <iostream.h>
#include <fstream.h>
struct material
{
    int cm;
    char denm[30], um[4];
    int cant;
    double pu;
};
void main()
{
    material m;
    ofstream fmat("mat.dat",ios::binary);
    while(cout<<"\nCod material:",cin>>m.cm)
    {
        cin.ignore(256,'\n');
        cout<<"Denumire material:"; cin.getline(m.denm,29);
        cout<<"Unitate de masura:"; cin.getline(m.um,4);
        cout<<"Cantitate:"; cin>>m.cant;
        cout<<"Pret unitar:"; cin>>m.pu;
        fmat.write((char*)&m,sizeof(material));
    }
    fmat.close();
}
```

S-au citit, de la tastatură, informații despre materiale și au fost scrise în fișier, cât timp nu s-a tastat CTRL/Z la introducerea codului materialului.

Să se citească fișierul *mat.dat* creat prin programul anterior și să se afișeze pe monitor informațiile stocate.

```
#include <iostream.h>
#include <fstream.h>
struct material
{
    int cm;
    char denm[30], um[4];
    int cant;
    double pu;
};

void main()
{
    material m;
    ifstream fmat("mat.dat",ios::binary | ios::nocreate);
    if( !fmat.is_open() ) { cerr<<"\nFisier inexistent!!!!"; return; }
    while( fmat.read((char*)&m,sizeof(material)) )
    {
        cout<<"\nCod material:"<<m.cm;
        cout<<"\nDenumire material:"<<m.denm;
        cout<<"\nUnitate de masura:"<<m.um;
        cout<<"\nCantitate:"<<m.cant;
        cout<<"\nPret unitar:"<<m.pu;
    }
    fmat.close();
}
```

După cum ați observat, este foarte important să știm dacă un fișier a fost sau nu deschis. În exemplele anterioare acest lucru era realizat prin testarea apariției unei erori în lucru cu fluxul. Implementări mai noi ale limbajului C++ pun la dispoziția programatorului o metodă specializată în acest sens care este *is\_open()*, ea returnează adeverat dacă fișierul a fost deschis, fals altfel.

Sfărșitul de fișier a fost sesizat ca și în cazul citirii de la tastatură, cu *cin*, adică prin apelul metodei care supraîncarcă operatorul cast (de conversie) la *void \** care returnează NULL în caz de eroare în lucru cu fluxul; se putea explicita testul și sub forma:

```
while( fmat.read( (char*)&m,sizeof(material) ) != NULL )
```

S-a putut observa că testul de sfârșit de fișier la toate exemplele prezentate s-a făcut implicit la momentul citirii informațiilor din fișier, deoarece când s-a ajuns la sfârșit de fișier se semnalează o eroare în lucru cu fluxul respectiv. Există și posibilitatea de a testa explicit dacă s-a ajuns la sfârșitul fișierului, deoarece există un indicator (un bit) care se setează cu valoarea 1 dacă s-a ajuns la sfârșitul fișierului. Acest bit este setat prin intermediul unei constante definite într-un tip enumerativ în clasa *ios* și se numește *eofbit*. Tot în acel tip *enum* mai sunt definite și constantele *failbit* și *badbit* care au fost prezentate în subcapitolul privind *Detectarea erorilor apărute în operații de intrare / ieșire*. Metoda *eof()* testează valoarea acestui bit și returnează adevărat dacă s-a ajuns la sfârșitul fișierului și fals altfel.

Secvența de program care citește articole din fișierul *mat.dat* se poate scrie aşa încât să testeze explicit ajungerea la sfârșitul fișierului astfel:

```
while( fmat.read((char*)&m,sizeof(material)), !fmat.eof() )
```

Bitul de sfârșit de fișier va fi și el resetat (pus pe 0) prin apelul metodei *clear()*.

Operația de poziționare într-un fișier face posibilă implementarea fișierelor în acces direct sau indexat, adică exploatarea lor altfel decât secvențial. Sunt definite două metode de poziționare, câte una pentru fiecare din fluxurile ce se pot defini (intrare / ieșire adică *get* și *put*). Astfel, dacă se deschide un fișier în intrare, pointerul de citire se mută cu metoda *seekg()*. Metoda *seekp()* mută pointerul de scriere, dacă s-a definit un flux în ieșire. O astfel de metodă are două prototipuri:

```
istream& istream::seekg(streamoff, ios::seekdir);
```

unde:

- *streamoff* – reprezintă poziția în baiți la care se va face poziționarea față de un reper;

- *ios::seekdir* – definește reperul care poate fi:
  - . începutul fișierului    - *ios::beg*
  - . poziția curentă       - *ios::cur*
  - . sfârșitul fișierului   - *ios::end*

alt prototip este:

```
istream& istream::seekg(streampos);
```

unde, *streampos* reprezintă poziția în baiți față de începutul fișierului.

Ca observație, *streamoff* și *streampos* sunt tipuri de date sinonime cu tipul *long*.

Exemple:

- *seekg(10)*; mută pointerul de citire la 10 baiți distanță față de începutul fișierului;
- *seekp(4, ios::cur)*; mută pointerul de scriere cu 4 baiți, după poziția curentă a acestuia;
- *seekg(-2, ios::end)*; mută pointerul de citire cu 2 baiți înaintea sfârșitului de fișier;
- *seekg(0)*; mută pointerul de citire la începutul fișierului, apelul fiind echivalent cu folosirea funcției *rewind()* din *stdio.h*.

Determinarea poziției curente în fișier a pointerului de scriere se face apelând metoda *tellp()*, care returnează numărul de octeți de la începutul fișierului, până la poziția curentă a pointerului. Dacă fișierul este deschis în intrare, atunci același lucru se face prin apelul metodei *tellg()*, a clasei *istream*, evident cu referire la pointerul de citire din fișier.

În situația în care se deschide un fișier în intrare / ieșire, adică se utilizează un obiect *fstream*, cele două metode (*seekg* și *seekp*) pot fi utilizate, dar ele vor opera asupra unuia și același pointer de citire / scriere în / din fișier. Pentru exemplificare se consideră creat un fișier binar, care conține 100 de întregi (de la 0 la 99). Secvența folosită pentru construirea fișierului este:

```
ofstream fint("int.dat",ios::binary);
for(int i=0;i<100;i++) fint.write((char*)&i,sizeof(int));
fint.close();
```

Se va construi un program care deschide în intrare / ieșire acest fișier, va muta pointerul de scriere la sfârșitul fișierului, iar pointerul de citire la întregul cu valoarea 50, după care se vor efectua operațiile:

- se va citi întregul și se va afișa valoarea lui;
- se va scrie întregul cu valoarea 100;

```
#include <iostream.h>
#include <fstream.h>
void main()
{
    fstream fint("int.dat",ios::in | ios::out | ios::binary |
                ios::nocreate | ios::noreplace);
    int k,t;
    if(!fint.is_open())
    {
        cerr<<"\n Fisier inexistent!!"; return;
    }
```

```

// mutare pointeri
fint.seekp(0,ios::end);
fint.seekg(50*sizeof(int));
// citire intreg
fint.read((char*)&k,sizeof(int));
cout<<"\n Nr. citit este:"<<k<<endl;
// scriere intreg
t=100;
fint.write((char*)&t,sizeof(int));
// revenire la inceput si afisare continut fisier
fint.seekg(0);
for(; fint.read((char*)&k,sizeof(int)),!fint.eof(); cout<<k<<" ");
fint.close();
}

```

Se constată, la afişarea întregului fișier, că valoarea 100 nu a fost scrisă la sfârșitul fișierului, aşa cum s-a dorit, ci a fost suprascrisă valoarea 51 din fișier cu valoarea 100. Cu alte cuvinte, se observă clar că pointerul de citire / scriere este unul singur indiferent de metoda prin intermediul căreia îl reposiționăm.

Pentru a putea lucra independent cu două poziții în același fișier (o poziție pentru citire și alta pentru scriere), trebuie să se creeze două fluxuri: unul de intrare și altul de ieșire pe respectivul fișier. Acest lucru se realizează prin faptul că fișierul se va deschide de două ori (odată în intrare și apoi în ieșire). La exploatarea unui fișier existent, în acces direct, cu doi pointeri independenți (de citire și de scriere) apare o problemă când acesta va fi deschis în ieșire (scriere) deoarece dacă el există va fi suprascris, deci se va pierde informația existentă în el. Pentru a soluționa această problemă, la modul de deschidere a fișierului în ieșire se va folosi atributul *ios::ate* care va muta pointerul la sfârșitul fișierului, deci nu-l va suprascrise (vezi tabelul 4.4). Folosirea atributului *ios::app* realizează de asemenea mutarea pointerului de scriere la sfârșitul fișierului, dar indiferent unde va fi acesta reposiționat, scrierea efectivă se va face doar la sfârșitul fișierului.

Ca exemplu, în acest sens, se va prezenta programul care modifică informații în fișierul *mat.dat*, care a fost creat cu un program prezentat anterior; vom respecta aceeași structură de articol.

```

#include <iostream.h>
#include <fstream.h>
struct material
{
    int cm;

```

```

char denm[30], um[4];
int cant;
double pu;
};

void main()
{
    material m;
    // constituirea a două fluxuri
    ifstream fmati("mat.dat", ios::in | ios::binary | ios::nocreate);
    ofstream fmate("mat.dat", ios::out | ios::ate | ios::binary );
    if(!fmati.is_open()) { cerr<<"\n Fisier inexistent!!!!"; return; }
    // determinarea numărului de articole din fisier
    int i,nr_art=fmate.tellp()/sizeof(material);
    // citire articol, modificare cimpuri dorite,
    // rescrierea articoului în aceeași pozitie
    for(fmate.seekp(0),i=0; i<nr_art ; i++ )
    {
        fmati.read((char*)&m,sizeof(material));
        cout<<"\n Codul=<<m.cm<<" Noul cod:";
        cin>>m.cm; cin.ignore(256, '\n');
        cout<<" Denumire material:<<m.denm<<" Noua denumire:";
        cin.getline(m.denm,29);
        cout<<" Unitate de masura:<<m.um<<" Noua unitate de masura:";
        cin.getline(m.um,4);
        cout<<" Cantitate:<<m.cant<<" Noua cantitate:"; cin>>m.cant;
        cout<<" Pret unitar:<<m.pu<<" Noul pret:"; cin>>m.pu;
        fmate.write((char*)&m,sizeof(material));
    }
    // inchidere fluxuri
    fmati.close(); fmate.close();
}

```

Din program se observă:

- constituirea a două fluxuri, unul de intrare și altul de ieșire, pentru același fișier (*mat.dat*);
- s-a calculat numărul de articole prin împărțirea numărului total de baiți ai fișierului la numărul de baiți ocupat de un articol;
- parcurgerea fișierului s-a făcut pe baza numărului de articole;
- citirea unui articol și rescrierea lui nu a mai necesitat reposiționări, fiecare pointer avansând corespunzător, pe măsura citirii / scrierii.

Dacă nu se construiau două fluxuri (de intrare și ieșire) pentru fișier, operația de citire muta pointerul după înregistrarea ce trebuia modificată, iar

scrierea ei necesita readucerea pointerului pe începutul înregistrării curente, utilizând un apel de tipul:

```
fmate.seekp(-sizeof(material), ios::cur);
```

## 4.5 Formatarea datelor în memoria internă

Din cele prezentate până acum s-a observat că formatarea datelor s-a făcut în legătură cu operațiile de intrare / ieșire pe dispozitive standard (monitor / tastatură) sau în memoria externă (fișiere nestandard).

Formatarea datelor în memoria internă presupune citirea / scrierea valorilor variabilelor din / în șiruri de caractere. Intuitiv, ne putem aduce aminte că funcțiile `sscanf()` și `sprintf()`, din `stdio.h`, realizau același lucru dar în manieră procedurală.

În biblioteca de bază C++ pentru formatarea datelor în memoria internă există clasele:

- `ostrstream` – folosită pentru a stoca în flux datele rezultate prin formatare;
- `istrstream` – folosită pentru a formata date din flux.

Pentru exemplificare, se va scrie programul care va trimite în fluxul de ieșire două variabile de tip `int` și `double` împreună cu text explicativ. Adresa șirului astfel constituit se poate obține prin intermediul metodei `str` ce are prototipul:

```
char *ostream::str();
#include <iostream.h>
#include <strstrea.h>
void main()
{
    ostrstream sir;
    int a=7;
    double x=567.99;
    sir<<"\n Numarul a este "<<a<< " iar x este "<<x<<ends;
    cout<<sir.str();
}
```

Se observă că:

- scrierea în flux s-a terminat cu `ends` deoarece în acest fel este adăugat terminatorul (0) la șirul care tocmai s-a constituit;

- șirul rezultat a fost afișat pe monitor cu `cout`.

Pentru a exemplifica citirea datelor dintr-un flux de intrare s-a construit programul:

```
#include <iostream.h>
#include <strstrea.h>
void main()
{
    istrstream sir("123 789.78 @");
    int a;
    char c;
    double x;
    sir>>a>>x>>c;
    cout<<"\n Intregul:"<<a<<" Caracterul:"<<c<<" Nr. real:"<<x;
}
```

Se observă că s-a creat un flux de intrare (`sir`) dintr-un sir de caractere furnizat ca parametru constructorului. Din acest șir au fost încărcate trei variabile de tipuri diferite (`int`, `char` și `double`). Pentru verificarea corectitudinii operației, ele au fost afișate pe display, cu `cout`.

Dacă se dorește ca șirul din care se preiau informațiile, pentru a fi formatate, să se introducă dinamic, la momentul execuției, se va utiliza pentru a face conversii în memoria internă clasa `strstream`. Aceasta permite atât lucru în intrare (preluarea unui șir) cât și în ieșire (convertirea șirului). Sunt posibile aceste operații, deoarece se moștenesc ambii operatori supraîncărcați (`<<` și `>>`). Exemplul următor preia un șir de caractere (`s`) de la tastatură, îl încarcă în obiectul `sir` după care îl extrage convertindu-l într-un întreg (`a`):

```
#include <iostream.h>
#include <strstrea.h>
void main()
{
    strstream sir;
    int a;
    char s[50];
    cout<<"\n Dati un intreg:";
    cin.getline(s,49);
    sir<<s;
    sir>>a;
    cout<<"\n Intregul introdus:"<<a;
}
```

Utilitatea formatării datelor în memoria internă este legată de faptul că sunt funcții de afișare text, îndeosebi în modurile grafice de lucru, care permit afișarea doar la nivel de sir de caractere. De aceea când vrem să afișăm valorile unor variabile de tipuri fundamentale trebuie să le formatăm într-un sir de caractere, care apoi va fi afișat (de exemplu, funcțiile *TextOut()* sau *DrawText()* care permit afișarea de texte în ferestre sub Windows).

Pentru exemplificarea procesului invers, adică citirea valorilor variabilelor de tipuri fundamentale din siruri de caractere, putem să ne amintim că argumentele funcției *main()* sunt numai de tip sir de caractere, interpretarea unora dintre ele sub diferite tipuri de date presupune efectuarea în prealabil a unui proces de formatare.



## **IMPLEMENTAREA OBIECTUALĂ A STRUCTURILOR DE DATE DINAMICE**

- Structuri de date dinamice liniare
- Structuri arborescente

Structurile de date dinamice de tip liniar (lista, stiva, coada) și arborescent sunt frecvent folosite în aplicații informatiche. Producătorii de medii de programare au avut în vedere implementarea unora dintre aceste structuri de date, constituind chiar biblioteci specializate, după cum vei vedea în capitolul 8.

Folosirea corectă a unor astfel de structuri de date, deja implementate presupune înțelegerea construirii lor în manieră obiectuală. În acest capitol vom folosi concepte ce țin de programarea orientată obiect, cum ar fi încapsularea, supraîncărcarea operatorilor, derivarea etc., pentru implementarea structurilor de date dinamice.

## 5.1 Structuri de date dinamice liniare

### Listă simplu înlățuită

Construirea obiectuală a acestei structuri de date se va concretiza în definirea unui tip nou de dată, numit *lista*. După cum ne amintim din programarea structurată, structura de tip listă era alcătuită din noduri care contineau informație utilă și o legătură spre un alt nod de același tip. Prin definiri de funcții, programatorul implementa operații de lucru cu lista (inserări de noduri, traversarea structurii, sortarea ei etc) și gestiona întreaga structură prin intermediul unui pointer la primul nod al listei, numit și cap de listă.

Din punct de vedere obiectual, speculând proprietatea de încapsulare, structura de listă poate fi modelată prin intermediul a două clase:

- una care corespunde nodului listei ;
- alta care modeleză structura de listă, în ansamblul ei.

În figura 5.1 se observă că legătura dintre listă și nod se face printr-un membru din clasa lista care este de tip pointer la nod și referă totdeauna primul nod al listei (capul de listă).

Clasa *nod* pentru lista simplu înlățuită are următoarea definiție:

```
#include <iostream.h>
#define Tip_elem int
class nod
{
// clase prietene nodului
```

```
friend class lista;
friend class stiva;
friend class coada;
Tip_elem info;
nod *next;

public:
nod( Tip_elem k, nod *urm=NULL ) : info(k), next(urm) { }
nod *get_next() { return next; }
friend ostream& operator<<( ostream& os, nod* n )
{ os<<n->info; return os; }

};
```

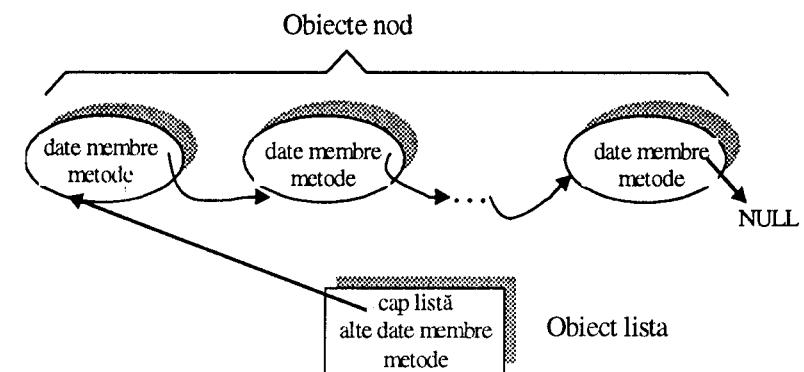


Fig 5.1 Legătura dintre listă și noduri

Pentru generalitate, s-a definit tipul informației utile printr-o constantă simbolică: `#define Tip_elem int`

Ca date membru un nod conține:

- informația utilă (*info*);
- legătura cu nodul următor (*next*),

iar ca metode:

- constructorul, cu rol de inițializare a informației utile și a legăturii, cu valoare implicită NULL, în ipoteza că nodul va fi inserat la sfârșitul listei ;
- funcția de acces *get\_next()* pentru obținerea adresei următorului nod ;
- *operator<<()* supraîncărcat pentru a afișa informația utilă a unui nod.

Gestionarea nodurilor care alcătuiesc lista se face prin intermediul clasei *lista* care are ca date membre:

- un pointer la *nod*, numit *cap*, care pointează totdeauna primul nod din listă;
- un întreg *n* ce va memora numărul de noduri din listă.

```

class lista
{
    bool identic(nod *, nod *);
    void sterge_el(nod *&, Tip_elem);
    void sterge_tot(nod *);

protected:
    nod *cap;
    int n;
    bool este_vida() { return cap==NULL; }

public:
    lista():cap(NULL),n(0) { }
    void ins_incep(Tip_elem );
    void ins_sf(Tip_elem );
    int count() { return n; }
    void sort();
    lista& operator+=(lista );
    lista& operator+=(Tip_elem );
    lista& operator=(Tip_elem );
    Tip_elem& operator[](int );
    int operator==(Tip_elem );
    bool operator==(lista& l) { return identic(cap,l.cap); }
    lista& operator=(lista& );
    friend ostream& operator<<(ostream& , lista& );
    ~lista() { sterge_tot(cap); }
};

```

Funcțiile membre ale clasei *lista* sunt:

- constructorul implicit, definit *inline*, cu rol de construire a unei liste vide (*cap=NULL* și *n=0*); se știe că inițial se pornește de la o listă vidă;
- funcții de inserare a unui element la începutul listei (*ins\_incep()*) și la sfârșitul ei (*ins\_sf()*);

Metoda de *inserare a unui element la începutul listei* are definiția:

```

void lista::ins_incep(Tip_elem k)
    { nod *t=new nod(k,cap); cap=t; n++; }

```

ea alocă mai întâi memorie pentru un nou nod, îl încarcă legătura și informația utilă, după care actualizează capul listei și incrementează variabila *n*, care memorează numărul de noduri din listă;

Metoda de *inserare a unui element la sfârșitul listei* are definiția:

```

void lista::ins_sf(Tip_elem k)
{
    nod *t = new nod(k), *cp = cap;
    n++;
    if( ! este_vida() )
    {
        while(cp->next) cp = cp->next;
        cp->next = t;
    }
    else cap = t;
}

```

Ea alocă memorie pentru un nou nod și incrementează *n* (numărul de noduri); în caz că lista e vidă, noul nod devine capul listei, altfel se parcurge lista pentru a leagă noul nod de ultimul nod al listei. Metoda *este\_vida()*, definită *inline*, returnează adevărat dacă lista este vidă, fals altfel.

- *count()* definită *inline*, este o funcție de acces și returnează **numărul de noduri** din listă;
- *sort()* pentru *sortarea* crescătoare a unei liste, după informația utilă:

```

void lista::sort()
{
    int i,j;
    nod *cp,*tmp;
    for(i=1;i<n;i++)
        for(cp=cap,j=0;j<n-i;j++)
            if(j)
            {
                if(cp->next->info > cp->next->next->info)
                {
                    tmp = cp->next;
                    cp->next = tmp->next;
                    tmp->next = cp->next->next;
                    cp->next->next = tmp;
                }
                cp = cp->next;
            }
            else
            {
                if(cp->info > cp->next->info)
                {

```

```

        tmp = cp->next;
        cp->next = tmp->next;
        tmp->next = cp;
        cap = cp = tmp;
    }
}

```

Sortare se face prin interschimbarea legăturilor dintre noduri, astfel încât capul listei să refere nodul cu cea mai mică informație utilă (sortare crescătoare, după informația utilă). Numărul iterării necesare efectuării sortării, se controlează prin numărul de noduri ale listei.

Noul tip de date (*lista*), beneficiază de operații care să poată fi invocate prin operatori supraîncărcați în clasa *lista*:

- ① operatorul `+=` a fost supraîncărcat încât să poată fi aplicat între două liste cu semnificația de **concatenare** a două liste, respectiv între o listă și un element de tipul informației utile din nod, cu semnificația de inserare a unui nod la sfârșitul listei.

Supraîncărcarea operatorului `+=` care lucrează cu două liste a fost definită astfel încât să insereze la sfârșitul listei inițiale, nodurile listei cu care se va concatena:

```

lista& lista::operator +=(lista l)
{
    while( !l.este_vida() )
        { ins_sf(l.cap->info); l.cap=l.cap->next; }
    return *this;
}

```

Operatorul `+=` între o listă și un element de tipul informației utile din nod s-a definit foarte simplu, în sensul că s-a apelat metoda care inserează un nod la sfârșitul listei:

```

lista& lista::operator +=(Tip_elem t)
{ ins_sf(t); return *this; }

```

- ② operatorul `--` a fost supraîncărcat pentru a fi aplicat între o listă și un element și *sterge* toate nodurile listei care au informația utilă egală cu elementul:

```

lista& lista::operator -(Tip_elem t)
{ sterge_el(cap, t); return *this; }

```

Să observăm din definire, că eliminarea propriu-zisă a nodurilor din listă o va realiza metoda privată *sterge\_el()* care este definită, în manieră recursivă, astfel:

```

void lista::sterge_el(nod *&cp, Tip_elem k)
{
    if(cp)
        if(cp->info==k)
            {
                nod *a=cp;
                cp=a->next;
                delete a; n--;
                sterge_el(cp,k);
            }
        else sterge_el(cp->next,k);
}

```

- ③ operatorul `[]` a fost supraîncărcat pentru a **accesa un element din listă dând poziția lui**; primul element, am considerat că se află pe poziția 0, pentru a respecta convenția limbajului C, de dispunere a elementelor într-un masiv:

```

Tip_elem& lista::operator[](int i)
{
    static Tip_elem nil;
    if(i>=0 && i<n)
    {
        nod*cp=cap;
        for(int k=0;k<i;k++) cp=cp->next;
        return cp->info;
    }
    else
    {
        cout<<"\n !! Indice eronat!!!"<<endl;
        return nil;
    }
}

```

Să observăm că:

- metoda returnează o referință la informația utilă din nod, lucru care permite atât obținerea valorii elementului, cât și modificarea ei;
- metoda testează indicele primit, în caz că acesta nu e valid se returnează referința unui element *static*, definit în metodă.

- ④ operatorul == a fost supraîncărcat încât să poată fi aplicat între două liste cu semnificația de test **dacă două liste sunt identice**, respectiv între o listă și un element, cu sensul de căutare a unui element în listă.

Metoda care supraîncarcă operatorul == pentru a testa dacă două liste sunt identice este definită *inline* și apelează metoda privată, recursivă *identic()* care face propriu-zis verificarea.

```
bool lista::identic(nod *cp1, nod *cp2)
{
    if( cp1==NULL && cp2==NULL ) return true;
    else
    {
        if(cp1==NULL || cp2==NULL) return false;
        if(cp1->info == cp2->info) return identic(cp1->next, cp2->next);
        else return false;
    }
}
```

Metoda care supraîncarcă operatorul == pentru a fi aplicat între o listă și un element are ca scop **căutarea elementului în listă**; metoda returnează poziția nodului în listă, dacă el conține elementul ca informație utilă, sau -1 în caz că nu există.

```
int lista::operator==(Tip_elem t)
{
    nod*cp=cap;
    for(int k=0;k<n;k++)
        if(cp->info==t) return k;
        else cp=cp->next;
    return -1;
}
```

- ⑤ operatorul = a fost supraîncărcat pentru a putea face **atribuirea între două liste**:

```
lista& lista::operator=(lista& srs)
{
    if(!este_vida()) sterge_tot(cap);
    while( ! srs.esteva() )
        { ins_sf(srs.cap->info); srs.cap= srs.cap->next; }
    return *this;
}
```

Se observă din definirea metodei, că se șterge întreaga listă destinație, dacă ea nu e vidă, după care la ea se inserează elementele listei ce se atribuie. La o primă analiză am putea trage concluzia că metoda este total ineficientă, copiind element cu element în lista destinație, când ar fi mai simplu să copiem numai partea vizibilă a listei (capul și numărul de noduri), lucru pe care-l face și versiunea pusă automat de compilator pentru operator =. În realitate, pot apărea extrem de multe erori, dacă cele două liste (sursă și destinație) partajează aceeași memorie dinamică, ce conține nodurile. Metoda care șterge întreaga listă este *sterge\_tot()* și are ca scop dezalocarea tuturor zonelor de memorie ce aparțin listei, după care o marchează ca vidă (*cap=NULL* și *n=0*):

```
void lista::sterge_tot(nod *cp)
{
    if(cp) { sterge_tot(cp->next); delete cp; }
    cap=NULL; n=0;
}
```

- ⑥ operatorul << a fost supraîncărcat, printr-o funcție *friend*, pentru a **afișa elementele unei liste** sub forma:

```
(element1, element2, ..., elementn)  
ostream& operator<<(ostream& os, lista& l)  
{  
    nod *cp=l.cap;  
    os<<"(";  
    while(cp)  
    {  
        os<<cp; if(cp->get_next())os<<", "; cp=cp->get_next();  
    }  
    os<<")";  
    return os;  
}
```

În final se observă că s-a definit *inline* destrctorul clasei *lista*, care apelează metoda *sterge\_tot()* în scopul dezalocării tuturor zonelor de memorie ce aparțin listei.

Pentru a testa noul tip introdus și anume lista simplu înlățuită, vom folosi programul:

```
void main()
```

```
int el;  
lista l1,l2,l3;
```

```

// inserari in liste
    l1.ins_incep(10); l1.ins_sf(56); l1.ins_incep(90);
    l2.ins_sf(1); l2.ins_sf(2);
// afisari de liste
    cout<<"Lista 1:"<<l1<<endl;
    cout<<"Lista 2:"<<l2<<endl;
// inserarea unui element la o lista si apoi concatenarea a doua liste
    l1+=l2+=101;
    cout<<"Lista 2 + 101 adaugat la l1:"<<l1<<endl;
// afisarea numarului de noduri dintr-o lista
    cout<<"Lista 2 are "<<l2.count()<<" noduri!"<<endl;
// folosirea operatorului [] pentru a adresa indexat elemente din lista
    cout<<"El al treilea :"<<l1[2]<<endl;
    l1[0]=199;
// folosirea operatorului == pentru cautarea unui element in lista
    cout<<"Lista 1:"<<l1<<endl;
    el=l1==101;
    if(el!=-1)cout<<"El 101 pe poz "<<el<<endl;
    else cout<<"El inexistent "<<endl;
    l3+=1; l3+=2; l3+=101;
// testarea daca doua liste sunt identice
    if(l2==l3) cout<<"Liste identice!!!"<<endl;
    else cout<<"Listele NU sunt identice!!!"<<endl;
// atribuirea a doua liste
    l3=l1; cout<<"Lista 3 ca lista 1:"<<l3<<endl;
// stergerea unui element (nod) dintr-o lista
    l3-=56; cout<<"Lista 3 mai putin 56 :"<<l3<<endl;
// sortarea unei liste
    l3.sort(); cout<<"Lista 3 sortata:"<<l3<<endl;
}

```

Putem să punem în evidență câteva aspecte legate de implementarea obiectuală a structurii de date lista:

- implementarea unor operații s-a făcut prin apelul unor metode private, recursive, similare funcțiilor ce implementau operațiile cu liste în mod procedural; se optează pentru această variantă pentru că se știe că recursivitatea se controlează de cele mai multe ori printr-un parametru de apel (capul listei) și care nu trebuie dat ca parametru într-o metodă ce aparține părții publice a unei clase (interfeței obiectului); de exemplu în partea de interfață a clasei *lista* avem metoda care supraîncarcă operatorul == pentru a elimina din listă toate nodurile ce au o anumită informație utilă. Ea apelează metoda privată, recursivă, *sterge\_el()* care primește o referință la capul listei

- pentru a controla procesul recursiv (și totodată pentru a-l modifica dacă este cazul) și elementul de șters;
- un avantaj ce decurge din implementarea obiectuală se referă la faptul că se poate exploata structura de listă atât în maniera clasică cât și într-o manieră proprie lucrului cu masive datorită faptului că s-a supraîncărcat operatorul []; modul de accesare a unui element dând poziția lui presupune parcurgerea listei până la acel element, ceea ce face ca această adresare să fie ineficientă în comparație cu accesul direct pe care-l realizează în cazul unui masiv de date stocat într-un spațiu contigu de memorie;
- unele operații cum ar fi: numărarea nodurilor, construirea listei vide, eliberarea zonelor de memorie alocate la momentul ieșirii din blocul în care a fost definit obiectul lista se fac fie implicit, fie cu un efort minim;
- implementarea structurii de date folosind două clase (una pentru nod și alta pentru listă) are avantajul că în nod se pot defini metode specifice de prelucrare a informației utile; dacă avem în vedere că, spre exemplu, informația utilă din nod ar fi de tip *double* și am dori afișarea ei în format științific atunci nu am modifica decât metoda de afișare din clasa *nod*, nimic în clasa *lista*.

Pentru a justifica generalitatea acestei implementări, ne propunem să lucrăm cu o listă care are ca informație utilă un obiect de tip *persoana*. Acest tip îl introducem prin descrierea clasei *persoana*:

```

class persoana
{
    char np[30];
    int sal;
public:
    persoana(char *nm="Persoana", int s=0):sal(s) { strcpy(np,nm); }
    bool operator>(persoana& p) { return sal>p.sal; }
    bool operator==(persoana &p) { return strcmp(np,p,np) == 0; }
    friend ostream& operator<<(ostream& os, persoana& p)
    {
        os<<p.np<<" "<<p.sal; return os;    }
};

```

Ea conține :

- ca date membre, numele și prenumele persoanei și salariul ei ;
- un constructor cu rol de inițializare a datelor membre din date elementare (joacă și rol de constructor implicit când toți parametrii au valori implicite) ;

- operatorul `>`, supraîncărcat pentru a testa dacă o persoană are salariul mai mare decât alta;
- operatorul `==`, supraîncărcat pentru a testa dacă numele a două persoane sunt identice;
- operatorul `<<`, supraîncărcat în vederea afișării unei persoane sub forma `nume_prenume și salariul ei`.

Având definite cele două clase (*lista* și *persoana*), putem construi și exploata o listă de persoane dacă definim tipul elementului sub forma:

`#define Tip_elem persoana`

Programul următor prezintă modul de lucru cu o listă de persoane fără a modifica ceva în clasa *lista*:

```
void main()
{
    lista l;
    // inserari de persoane in lista
    l.ins_sf(persoana("Vasile",456666));
    l.ins_incep(persoana());
    l+=persoana("Ion",432229);
    // afisarea listei
    cout<<l<<endl;
    // sortarea liste dupa salariul persoanelor
    l.sort(); cout<<l<<endl;
    // afisarea celei de-a doua persoane din lista
    cout<<l[1]<<endl;
    // eliminarea lui Ion din lista
    l-=persoana("Ion",0); cout<<l;
}
```

În concluzie, putem menționa că este obligatoriu ca orice tip ce va juca rol de informație utilă pentru nodurile listei să aibă definiți operatorii relaționali `==`, `>` și `<<` pentru afișare.

### Stiva

Structura de stivă este similară structurii de listă, numai că disciplina de exploatare este proprie (LIFO – Last Input First Output); ea este deci o listă cu metode proprii de acces.

Deoarece structura de stivă are la bază o listă, atunci se va defini clasa *stiva* ca fiind derivată din clasa *lista*. Derivarea se va face în mod privat pentru că

metodele clasei *lista* să nu poată fi accesibile din exterior, datorită faptului că structura de stivă are propria interfață.

```
class stiva : private lista
{
public:
    bool este_vida() { return lista::este_vida(); }
    void push(Tip_elem k) { ins_incep(k); }
    Tip_elem pop()
    {
        nod *t=cap;
        Tip_elem z;
        if(cap)
        {
            z=t->info;
            cap=cap->next; delete t; n--;
        }
        else cout<<"\n STIVA vida !!"<<endl;
        return z;
    }
    stiva& operator<<(Tip_elem k) { push(k); return *this; }
    stiva& operator>>(Tip_elem& k) { k=pop(); return *this; }
};
```

Specifice pentru stivă sunt operațiile:

- testare dacă stiva este sau nu vidă, metoda `este_vida()` care implementează această operație este definită *inline* și apelează metoda cu același nume din clasa *lista*; pentru a evita autoapelul, este obligatorie folosirea operatorului de rezoluție (*lista::*);
- inserarea unui element în stivă se face fie apelând metoda `push()`, fie folosind operatorul `<<` care s-a supraîncărcat în clasa *stiva*; deoarece metoda realizează practic o inserare în capul listei, s-a apelat metoda `ins_incep()` din clasa *lista*;
- extragerea unui element din stivă se face fie apelând metoda `pop()`, fie folosind operatorul `>>` care s-a supraîncărcat în clasa *stiva*; această operație are ca scop obținerea informației utile ce aparține nodului din capul listei și ștergerea respectivului nod.

Folosirea clasei *stiva* se va exemplifica prin programul:

```
void main()
{
    int el;
    stiva s;
```

```
// inserare elemente in stiva
s<<100<<103; s.push(99);
// extragere elemente din stiva
if(!s.estе_vida()) { s>>el; cout<<el<<" "; }
while(!s.estе_vida()) cout<<s.pop()<<" ";
```

### Coada

Structura de coadă are la bază tot o listă, numai că disciplina de lucru este FIFO (First Input First Output), adică inserările se fac numai în coada listei, iar extragerile se fac numai din capul listei.

Ca și în cazul stivei, clasa *coada* este derivată privat din clasa *listă*; interfața ei conține doar operațiile:

- test de vidă; metoda *este\_vida()* care implementează această operație este definită *inline* și apelează metoda cu același nume din clasa *listă*;
- inserarea unui element în coadă se face fie apelând metoda *put()*, fie folosind operatorul << care s-a supraîncărcat în clasa *coada*; metoda realizează practic o inserare la sfârșitul listei;
- extragerea unui element din stivă se face fie apelând metoda *get()*, fie folosind operatorul >> care s-a supraîncărcat în clasa *coada*; această operație presupune obținerea informației utile ce aparțin nodului din capul listei și ștergerea respectivului nod.

```
class coada : private lista
{
    nod *sf;
public:
    coada():sf(NULL) { }
    bool este_vida() { return lista::este_vida(); }
    void put(Tip_elem k);
    Tip_elem get();
    coada& operator<<(Tip_elem k) { put(k); return *this; }
    coada& operator>>(Tip_elem& k) { k=get(); return *this; }
};
```

Pentru a eficientiza procesul de inserare a unui element în coadă (la sfârșitul listei), structura de coadă are în plus față de listă un pointer (*sf*) care referă ultimul nod al listei, după cum se poate observa în figura 5.2. Acest lucru se

justifică pentru a nu traversa lista de fiecare dată când se inserează un element în coadă.

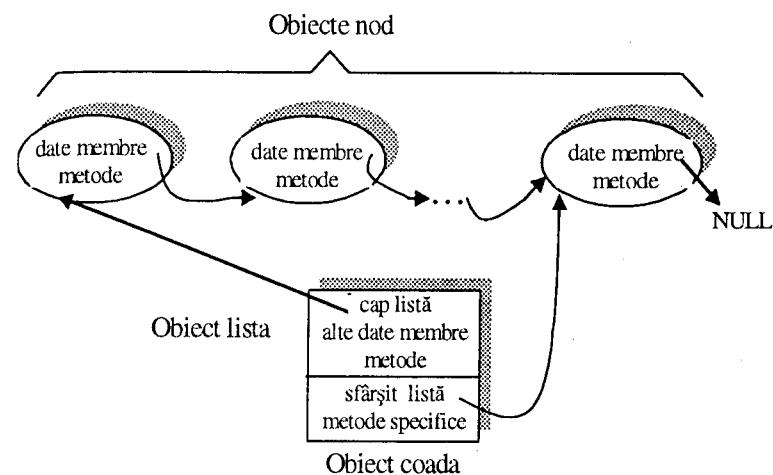


Fig 5.2 Legătura dintre coadă și noduri

Metoda de inserare are definiția:

```
void coada::put(Tip_elem k)
{
    nod *t=new nod(k); n++;
    if(!cap) cap=sf=t;
    else { sf->next=t; sf=t; }
```

iar cea de extragere este:

```
Tip_elem coada::get()
{
    Tip_elem z;
    if(cap)
    {
        nod *t=cap;
        z=cap->info; cap=cap->next;
        if(!cap) sf=cap;
        delete t; n--;
    }
    else cout<<"\n COADA vida !!"<<endl;
    return z;
}
```

Se observă că ambele metode gestionează ambi pointeri: *cap* – ce referă nodul din capul listei și *sf* – care referă nodul de la sfârșitul listei.

Programul următor va exemplifica utilizarea structurii de coadă:

```
void main()
{
    int el;
    coada q;
    // inserare de elemente in coada
    q.put(123); q.put(55); q.put(324); q.put(500);
    // extragere de elemente din coada
    if(!q.estE_vida()) { q>>el; cout<<el<<" "; }
    while(!q.estE_vida()) cout<<q.get()<<" ";
}
```

## 5.2 Structuri arborescente

Structura arborescentă implementată obiectuală în acest subcapitol este arborele binar de căutare. Ca și în cazul structurilor dinamice liniare și arborele binar de căutare va fi descris prin intermediul a două clase:

- una ce corespunde nodului arborelui binar (*nod*);
- alta care modeleză arborele binar de căutare propriu-zis (*arbbin*).

Clasa *nod*, va conține informația utilă (*info*) și doi pointeri către același tip de nod pentru a adresa subarborele stâng (*ss*), respectiv drept (*sd*):

```
#include <iostream.h>
#define Tip_elem int
class nod
{
    friend class arbbin;
    Tip_elem info;
    nod *ss,*sd;
public:
    nod(Tip_elem k, nod *s=NULL, nod *d=NULL):info(k),ss(s),sd(d) {}
    friend ostream& operator<<(ostream& os, nod* n)
        { os<<n->info; return os; }
};
```

Constructorul clasei nod are ca scop încărcarea informației utile și a legăturilor din parametrii de apel; legăturile au și valori implicate (*NULL*) ceea ce înseamnă că dacă nu se furnizează explicit alte valori, nodul va fi frunză (fără nici un descendenter). În plus, nodul mai are supraîncărcat operatorul << în vederea afișării informației lui (utile).

Pentru o maximă generalitate tipul informației utile a fost introdus tot prin intermediul unei constante simbolice:

```
#define Tip_elem int
```

Între arborele binar de căutare și noduri este o relație tot de tip colecție, în sensul că un arbore binar de căutare conține mai multe noduri.

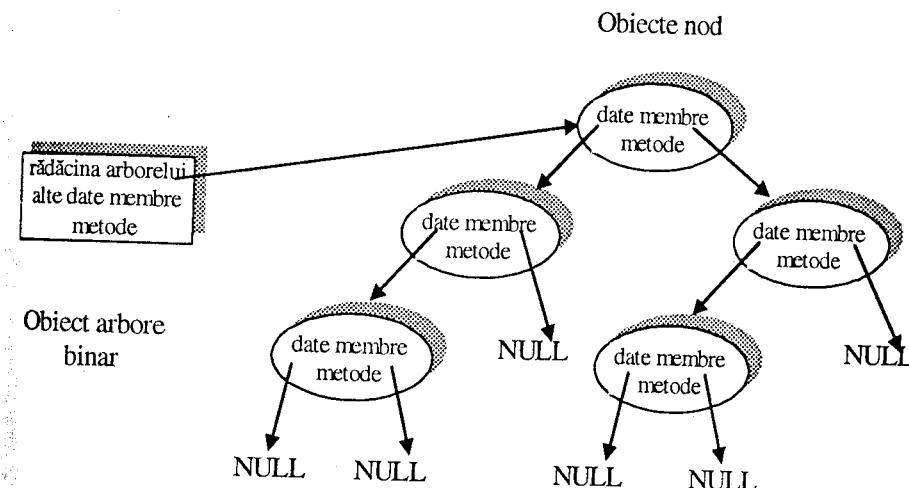


Fig 5.3 Legătura dintre arbore și noduri

Nodurile sunt dispuse în arbore respectând următoarea disciplină: pentru orice nod curent al arborelui, subarborele din stânga, dacă există, conține noduri ce au informația utilă mai mică decât informația utilă a nodului curent, iar subarborele din dreapta, dacă există, conține noduri ce au informația utilă mai mare decât informația utilă a nodului curent. De reținut că această disciplină dacă e valabilă pentru toate nodurile unui arbore binar atunci el este unul de căutare.

Legătura dintre arborele binar și nodurile sale se face prin intermediul unui membru din clasa arbore binar (*arbbin*), de tip pointer la nod, numit *rad* și care referă nodul rădăcină a arborelui, ca în figura 5.3.

```

class arbbin
{
    nod *rad;
    int nrn;
    nod *ins(nod*, Tip_elem);
    void sterge_arb(nod*);
    void preord(nod*);
    void inord(nod*);
    nod *copiere(nod *);
    void sterge_nod(nod *&, Tip_elem);
    Tip_elem sterg(nod*& );
    bool egal( nod *, nod *);
    int inalt(nod*);
    bool cauta(nod*, Tip_elem);
    bool frunza(nod *r) { return r->ss==NULL && r->sd==NULL; }
    friend int max(int a, int b) { return a>b ? a: b; }

public:
    arbbin():rad(NULL),nrn(0) {}
    void afis_RSD() { preord(rad); }
    void afis_SRD() { inord(rad); }
    int count() { return nrn; }
    bool este_vid() { return rad==NULL; }
    int inaltimea() { return inalt(rad); }
    arbbin& operator<<(Tip_elem k){ rad=ins(rad,k); return *this; }
    arbbin& operator=(arbbin& );
    bool operator==(arbbin& a) { return egal(rad,a.rad); }
    bool operator==(Tip_elem k) { return cauta(rad, k); }
    arbbin& operator-=(Tip_elem k) { sterge_nod(rad,k); return *this; }
    friend ostream& operator<<(ostream &os, arbbin& a)
        { a.afis_SRD(); return os; }
    ~arbbin() { sterge_arb(rad); }
};

```

Pe lângă rădăcina arborelui, clasa *arbbin* mai conține ca variabilă membru și numărul de noduri din arbore (*nrn*).

Funcțiile membre ale clasei *arbbin* implementează diverse operații care se efectuează pe arbori binari de căutare; unele operații se invocă folosind și operatori supraîncărcăți în clasa *arbbin*:

- constructorul clasei are ca scop construirea unui arbore binar vid (*rad=NULL* și *nrn=0*); un astfel de arbore este unul de căutare;
- *count()* este o funcție de acces și are ca scop obținerea **numărului de noduri** ale arborelui;

- *este\_vid()* este o metodă care testează dacă arborele binar este sau nu vid;
- *inaltimea()* este o metodă care determină **înălțimea unui arbore** binar, adică numărul de niveluri pe care el îl are; este definită *inline* și apelează metoda privată recursivă *inalt()* care determină efectiv înălțimea arborelui:

```

int arbbin::inalt(nod *r)
{
    if(r) return 1+max(inalt(r->ss),inalt(r->sd));
    else return 0;
}

```

*max()* este o funcție friend care determină maximul dintre doi întregi și este definită în interiorul clasei;

- pentru *afisare* s-au definit două metode care să afișeze atât informațiile utile din noduri, cât și legăturile dintre noduri; pentru surprinderea legăturilor s-a ales afișarea cu paranteze în formele:

- *Rădăcină ( Stânga, Dreapta )* realizată de metoda *afis\_RSD()* care este definită *inline* și apelează la rândul ei metoda privată *preord()*, care are la bază traversarea în preordine a arborelui:

```

void arbbin::preord(nod *r)
{
    if(r)
    {
        cout<<r;
        if( !frunza(r) )
        {
            cout<<"(";
            preord(r->ss); cout<<", "; preord(r->sd);
            cout<<")";
        }
        else cout<<".";
    }
}

```

Metoda *frunza()* este privată, definită *inline* și testează dacă un nod este sau nu nod frunză.

- *( Stânga ) Rădăcină ( Dreapta )* realizată de metoda *afis\_SRD()* care este definită *inline* și ea apelează la rândul ei metoda privată *inord()*, care are la bază traversarea în inordine a arborelui:

```
void arbbin::inord(nod *r)
{
    if(r)
    {
        if(!frunza(r)) { cout<<"("; inord(r->ss); cout<<")"; }
        cout<<r;
        if(!frunza(r)) { cout<<"("; inord(r->sd); cout<<")"; }
    }
    else cout<<"";
}
```

În clasa *arbbin* s-au supraîncărcat operatorii:

- ① << pentru a **insera un nou element** (nod) în arbore, astfel încât acesta să rămână tot arbore de căutare; metoda care inserează elementul propriu-zis este *ins()* și are forma:

```
nod *arbbin::ins(nod *r, Tip_elem k)
{
    if(r)
    {
        if(k==r->info);
        else if(k>r->info) r->sd=ins(r->sd,k);
        else r->ss=ins(r->ss,k);
        return r;
    }
    else return nrn++, new nod(k);
}
```

Se observă că metoda este recursivă și actualizează și numărul de noduri;

- ② = pentru a **atribui un arbore altui arbore**:

```
arbbin& arbbin::operator=(arbbin& srs)
{
    if(rad) sterge_arb(rad);
    rad=NULL; nrn=srs.nrn;
    rad=copiere(srs.rad);
    return *this;
}
```

Se observă că mai întâi se sterge conținutul arborelui destinație, prin apelul metodei private, recursive *sterge\_arb()*:

```
void arbbin::sterge_arb(nod *r)
{ if(r) { sterge_arb(r->ss); sterge_arb(r->sd); delete r; } }
```

după care se copiază arborele sursă în destinație apelând metoda privată, recursivă *copiere()*:

```
nod *arbbin::copiere(nod *r)
{
    if(r) return new nod(r->info,copiere(r->ss),copiere(r->sd));
    else return NULL;
}
```

- ③ == a fost definit în două variante:

- una care să poată fi aplicată între doi arbori pentru a **testa dacă doi arbori sunt identici**; metoda este definită *inline* și apelează metoda privată recursivă *egal()* care face proprietatea verificarea:

```
bool arbbin::egal( nod *r1, nod *r2)
{
    if(!r1) return r2==NULL;
    else if(!r2) return false;
    else return r1->info==r2->info && egal(r1->ss,r2->ss)
        && egal(r1->sd,r2->sd);
}
```

- cea de-a doua permite ca operatorul == să fie aplicat între un arbore și un element cu scopul de **căutare a unui element în arbore**; metoda este definită *inline*, căutarea propriu-zisă este făcută de metoda privată, recursivă *cauta()*:

```
bool arbbin::cauta(nod* r, Tip_elem k)
{
    if(!r) return false;
    else if(k==r->info) return true;
    else if(k>r->info) return cauta(r->sd, k);
    else return cauta(r->ss, k);
}
```

Căutarea se face tinând cont de faptul că arborele este binar de căutare.

- ④ -= pentru a **sterge un nod din arbore** astfel încât acesta să rămână tot de căutare; metoda este definită *inline* și apelează metoda privată, recursivă *sterge\_nod()*, care realizează proprietatea stergerea:

```
void arbbin::sterge_nod(nod *&r, Tip_elem k)
{
    nod *aux;
    //se cauta nodul de sters
    if(!r) cout<<"\n Nodul inexistent!!!"<<endl;
```

```

else if(k==r->info)
    {
        // nodul s-a gasit
        aux=r;
        // se testeaza daca are un descendant vid
        if(!aux->sd) { r=aux->ss; delete aux; }
        else if(!aux->ss) { r=aux->sd; delete aux; }
        else // nu are nici un descendant vid
            r->info=sterge(r->sd);
    }
    else if(k>r->info) sterge_nod(r->sd,k); // se cauta in stinga
    else sterge_nod(r->ss,k); // se cauta in dreapta
}

```

De observat construcția *nod \*k*, care înseamnă că se transferă referința unui pointer în scopul modificării adresei în pointerul original (nu copia pointerului).

Această metodă verifică dacă există nodul de șters în arbore; în caz afirmativ, realizează ștergerea lui efectivă doar dacă are cel puțin un descendant vid. Dacă ambii subarbore există, atunci ștergerea propriu-zisă este efectuată de către metoda privată *sterge()*:

```

Tip_elem arbbin::sterge(nod*& pa)
{
    if(pa->ss)
        return sterge(pa->ss); // se merge catre cel mai din stinga nod
    else
    {
        nod *a=pa;    Tip_elem k;
        k=a->info;   pa=pa->sd;
        delete a;     return k;
    }
}

```

În acest caz, ștergerea se face înlocuind informația utilă a nodului care trebuie șters cu informația utilă a celui mai din stânga nod al subarborelui din dreapta, care apoi va fi șters fizic; procedând aşa, arborele rămâne tot de căutare.

- 5 << a fost supraîncărcat, printr-o funcție *friend*, pentru a afișa un arbore în forma (*Stânga*) *Rădăcină* (*Dreapta*); funcția este definită în cadrul clasei și apelează metoda *afis\_SRD()*; cu alte cuvinte constituie o alternativă de afișare a arborelui în inordine.

În final s-a definit un destructor, care are ca scop dezalocarea tuturor zonelor de memorie aferente arborelui respectiv, apelând metoda *sterge\_arb()*.

Programul următor are ca scop testarea clasei arbore binar de căutare:

```

void main()
{
    arbbin a,b;
    // inserari de noduri
    a<<56<<12<<34<<90<<67<<80<<110;
    // afisarea arborelui
    a.afis_RSD();
    // obtinerea numarului de noduri
    cout<<"\n Arborele are "<<a.count()<<" noduri!"<<endl;
    // afisare arbore folosind operatorul <<
    cout<<a<<endl;
    // atribuirea dintre doi arbori
    b=a;
    b.afis_SRD();
    // testare daca doi arbori sunt identici
    if(a==b) cout<<"\n Arboare identici!!!"<<endl;
    else cout<<"\n Arboare diferiti!!!"<<endl;
    // stergerea unui nod din arbore
    b-=67; b.afis_RSD();
    // cautarea unui nod in arbore
    if(b==67) cout<<"\n Nodul exista!"<<endl;
    else cout<<"\n Nod inexistent!!!"<<endl;
    // determinarea inalțimii unui arbore
    cout<<"\n Inaltimea arborelui este de :"<<a.inaltimea()<<" nivele";
}

```

Având tipul *persoana* definit (identic cu cel folosit la structura de listă), pentru a lucra cu un arbore binar de căutare care să aibă ca informație utilă un obiect de tip *persoana*, se va defini constanta simbolică *Tip\_elem* în forma:

```

#define Tip_elem persoana
și fără a modifica ceva în clasa arbbin, programul următor va funcționa corect:
void main()
{
    arbbin a;

```

```
// inserare de obiecte persoana in arbore  
a<<persoana("p1",56)<<persoana()<<persoana("Vasi",90)  
    <<persoana("Gabi",66);  
// afisare arbore  
cout<<a<<endl;  
// eliminarea persoanei p1 din arbore  
a=persoana("p1",0);  
a.afis_RSD();  
}
```



## **ŞABLOANE DE CLASE** **(Clase *template*)**

- Funcții și clase *template*
- Instantierea şabloanelor. Constante în clasele *template*
- Specializări
- Relații între şabloane

Se spune despre *derivarea* claselor că asigură reutilizarea, atât codului sursă, cât și a celui obiect, adică o dată scrise funcții, ele pot fi moștenite de clasele derivate.

Într-o oarecare măsură și prin *componere* se reutilizează codul obiect, deoarece clasele ce includ alte clase, permit reutilizarea codului obiectului inclus, invocându-le ca pe serviciile unui server.

Facilitățiile de *template* furnizează o altă metodă de a reutiliza cod, de data aceasta cod sursă, bazat pe un şablon. Ca și macrodefinițiile cu parametri, clasele și funcțiile *template* (şablon) oferă o cale de a obține cod sursă C++, după un model de expandare.

Spre deosebire de macrodefiniții, care permit parametrizarea textului sursă pentru orice entitate care diferă de la o *producere* la alta (spre exemplu, părți din instrucțiuni, tipuri de date, nume de variabile, grupuri de instrucțiuni etc., organizate ca argumente ale macrodefiniției), şabloanele de clase și funcții asigură doar parametrizarea tipului de date și/sau a valorii unor constante. Ele apar aşadar ca niște specializări ale *macrodefinițiilor* cu parametri și nu strică să ne imaginăm că în spatele facilității de template stau niște macrodefiniții care au ca parametri tipuri de date.

## 6.1 Funcții și clase *template*

### Funcții *template*

De ce este nevoie de funcții *template*?

Să luam un exemplu banal: o funcție care adună două numere:

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

Dacă vrem să adunăm două variabile de tip float sau double funcția nu este bună, căci aplică conversii implice pentru adaptarea la prototip, pierzând zecimalele ce asigură precizia numărului; astfel, apelată sub forma:

```
cout << suma(12.5, 2.5);
```

valoarea afișată este 14, nu 15 cum ne-am fi așteptat.

O soluție, nu tocmai eficientă, este să scriem doar funcția ce lucrează pe tipul cel mai puternic (*double*); este de altfel și soluția pentru care s-a optat pentru majoritatea funcțiilor din biblioteca matematică.

Este posibil ca timpul necesar conversiilor, durata mare a calculelor pe *double*, comparativ cu aritmetica *nativă* pentru calculator, a întregilor, să facă total ineficientă această soluție. Este de ajuns să estimăm diferențele de performanță între calculul expresiei  $x^y$  pe *int*, respectiv pe *double*.

Alteori scrierea acelorași funcții lucrând pe tipuri diferite de date devine obligatorie. Avem la îndemână un exemplu la fel de simplu:

```
void swap(int &, int &);
```

pentru interschimbul a două numere. Suntem obligați să facem transferul prin referință sau prin adresă pentru ca interschimbul să fie efectiv, altfel am interschimba doar copiile parametrilor de intrare. Constatăm imediat că pentru referințe și adrese nu mai sunt operate conversii implice și deci funcția pe *int&* nu poate fi folosită pentru interschimb de *double*.

În concluzie, există extrem de multe situații în care trebuie să scriem variante ale aceleleași funcții, dar care lucrează pe tipuri diferite de date. Facilitatea template permite să dăm un model după care sunt scrise automat mai multe funcții.

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

template <class TIP>
void swap(TIP &a, TIP &b) { TIP aux; aux=a, a=b, b=aux; }

void main( )
{
    double x=1.1,y=2.2;
    swap( x,y); cout <<x<<y<<endl;
    int a=1,b=2;
    swap( a,b); cout <<a<<b<<endl;
}
```

Deși funcția în sine nu are legătură cu vreo clasă, se observă că tipul (*TIP*) a fost introdus ca o clasă, iar *template< class TIP>* este un lait-motiv care anunță că ce urmează nu e o funcție propriu-zisă, ci un model (şablon) după care va fi generat cod sursă pentru mai multe funcții similare.

În versiunile mai noi de compilatoare se poate folosi și sintaxa: *template <typename TIP>*. În loc de cuvântul *class* se poate folosi *typename* tocmai pentru a nu se confunda cu tipul *class* din C++.

Pentru câte funcții și când se va genera efectiv codul? Sunt întrebări firești din partea celor care au lucrat cu macrodefiniții și știu că programatorul era cel care decidea când invoca macrodefiniția și pentru care anume parametri. Important era ca la compilare să fie regăsit acel cod sursă.

Aici lucrurile au fost oarecum simplificate, deoarece toți parametrii se referă la tipuri de bază sau de utilizator, dar care sunt cunoscute de compilator la un moment dat. Ca urmare, compilatorul poate *instantia* singur modelul dat de programator, fără să-i cerem noi explicit acest lucru.

Implicit noi sugerăm toate tipurile pentru care dorim instanțieri, prin faptul că apelăm funcția cu parametri de diverse tipuri. Așadar, când compilatorul întâlnește un apel al funcției pe un nou tip de date, nu se gândește ce conversii pentru adaptare la prototip să facă, ci se apucă să scrie el însuși funcția conformă cu prototipul cerut, după modelul dat de noi.

Ce nume dă funcțiilor generate de el? Evident numele indicat de programator în şablon, dar să ne amintim că beneficiind de supraîncărcare putem avea funcții care se numesc la fel, dar care au signature diferite.

Am văzut cum se construiesc funcțiile *template* care lucrează pe tipurile de bază. Ce presupune ca şablonul să poată fi invocat și pentru construirea unei funcții care lucrează cu tipuri de utilizatori? Nimic altceva decât să dăm clasa care descrie noul tip introdus de utilizator:

```
class persoana
{
    public:
        persoana( char *n="Noname" ) { strcpy( nume,n); }
        char nume[50];
        friend ostream& operator<<(ostream &out, persoana p)
            { out << p.nume; return out; }
};
```

Putem adăuga acum în *main* fără probleme, apelul pentru a testa cum lucrează funcția *template* pe noul tip de dată:

```
persoana p1("UNU"), p2("DOI");
swap( p1,p2 );
cout <<p1<<p2<<endl;
```

Clasa *persoana* folosită de funcția *template* nu este o clasă *template*, ci o clasă obișnuită, deoarece în interiorul ei toți membrii au tipul cunoscut. Vom vedea în cele ce urmează că o funcție *template* poate folosi la rândul ei tipuri generice, descrise prin clase *template*.

### Clase *template*

O facilitate deosebită introdusă în versiunile mai noi ale compilatorului de C++ se referă la clase *template*, numite și şabloane de

clase. Clasele template sunt descrieri parametrizate de clasă, ce vor fi adaptate ulterior diferitelor tipuri de date recunoscute în limbaj; cu alte cuvinte, ele sunt clase generice, care permit obținerea unor clase, particularizând tipul membrilor cu care lucrează.

Sintaxa generală pentru a defini un şablon de clasă este:

```
template <class T1, ..., class Ti, ..., class Tn, tip1 c1, ..., tipj cj, ..., tipm cm>
class nume_c
```

```
{
```

```
.....
```

```
};
```

unde:

- *nume\_c* – reprezintă numele clasei şablon;
- *T<sub>i</sub>* – tip generic de dată;
- *tip<sub>j</sub>* – tip concret de dată;
- *c<sub>j</sub>* – constantă de tip *tip<sub>j</sub>*.

Din aceasta sintaxă se deduce că parametrizarea clasei se face nu numai la nivel de tip de dată, ci și în sensul parametrizării unor valori (constante) utilizate în clasă.

Spre exemplu, putem declara o listă ca structură generală, cu funcțiile specifice de prelucrare, pe care s-o putem folosi indiferent de tipul datelor stocate în noduri (*int*, *float* sau un tip introdus de utilizator). Mecanismul prin care aceste clase generice sunt particularizate obținându-se clase operaționale, efective, este similar apelului macrodefinițiilor cu parametri.

Într-un şablon de clasa, metodele pot fi definite atât în interiorul clasei (*inline*) și nu necesită o sintaxă specială, cât și în exteriorul ei, caz în care se folosește sintaxa :

```
template <class T1, ..., class Ti, ..., class Tn, tip1 c1, ..., tipj cj, ..., tipm cm>
tip_r nume_c < T1, ..., Ti, ..., Tn, c1, ..., cj, ..., cm >::nume_funct( lista_p_f )
```

```
{
```

```
.....
```

```
};
```

unde:

- *nume\_c* – reprezintă numele clasei şablon;
- *T<sub>i</sub>* – tip generic de dată;
- *tip<sub>j</sub>* – tip concret de dată;
- *c<sub>j</sub>* – constantă concordantă tipului *tip<sub>j</sub>*;
- *tip\_r* – tipul funcției membru;

- *nume\_funct* – nume funcție membră;
- *lista\_p\_f* – listă parametri formali.

Să urmărim împreună un exemplu în care se definește o clasa *template* pentru un obiect *vector* în memorie dinamică, conținând adresa de început și dimensiunea lui, cerută la alocare. Inițial, tipul elementelor stocate în vector este dat generic *T*, iar în *main()* se solicită particularizări ale clasei *template* pe tipul *int* și *float*.

```
#include <iostream.h>
template <class T> class vector
{
    T * pe;
    int dim;
public:
    vector(int);
    ~vector() { delete[ ] pe; }
    T& operator[] (int i) { return pe[i]; }
    void afis(); void sort();
};

template <class T> vector<T>::vector(int n):dim(n)
{
    pe = new T[n];
    for (int i = 0; i < dim; i++)
        { cout << "\n elem_ " << i << " "; cin >> pe[i]; }
}

template <class T> void vector<T>::afis()
{
    cout << endl;
    for (int i = 0; i < dim; i++) cout << pe[i] << " ";
}

template <class T> void vector<T>::sort()
{
    int i,j; T aux;
    for (i = 0; i < dim-1; i++)
        for ( j = i+1; j < dim; j++)
            if(pe[i] > pe[j])
                { aux= pe[i]; pe[i]=pe[j]; pe[j]=aux; }
}

void main()
{
    vector<int> vi(3);
```

```
vi.afis(); vi.sort(); vi.afis();
vector<float> vf(3);
vf.afis(); vf.sort(); vf.afis();
cout << "\n elementul minim este:" << vf[0];
}
```

Se observă că:

- *vi* este un vector cu trei elemente de tip întreg;
- *vf* este un vector cu trei elemente de tip *float*;
- metodele se apelează în mod obișnuit, fără a necesita o sintaxă specială și se adaptează automat la tipurile particulare, fără nici o modificare.

Ce va trebui să adăugăm, de această dată, programului pentru a-l face să lucreze și pe un tip nou, introdus de utilizator? Evident, va trebui să dăm în primul rând caracteristicile nouului tip: datele membre, metodele și operatorii recunoscuți. Ne vom menține tot la tipul *persoana*, pentru care avem descrieri, construind un vector de persoane și ne propunem ca funcția de sortare să însemne în acest caz sortare alfabetică. Nu vom modifica nimic în clasa *template*, ci vom adăuga exterior clasei, informațiile despre noul tip; ne mărginim la a declara supraîncărcări doar pentru operatorii <<, >> implicați în intrări / ieșiri cu obiecte și *operator>* care apare în sortare. Dăm în continuare numai textul prin care programul diferă de cel anterior.

```
class persoana
{
public:
    char nume[40];
    persoana( char *n="Noname") { strcpy(nume,n); }
    friend ostream & operator<< ( ostream &, persoana );
    friend istream & operator>> ( istream &, persoana & );
    int operator> ( persoana p ) { return strcmp(p.nume,nume)>0 ? 1 :0; }

ostream & operator<< ( ostream & iesire, persoana p )
    { iesire << p.nume << " " << endl; return iesire; }

istream & operator>> ( istream & intrare, persoana & p )
    { cout << "Nume: " ; intrare>> p.nume; return intrare; }

void main()
{
    vector<int> vi(3);
    vi.afis(); vi.sort(); vi.afis();
    vector<float> vf(3);
    vf.afis(); vf.sort(); vf.afis();
}
```

```

vector<persoana> vp(2);
vp.afis(); vp.sort(); vp.afis();
}

```

## 6.2 Instanțierea şablonelor. Constante în clasele template

Un şablon de clasă poate fi instantiat devenind o clasă concretă prin substituirea tipurilor generice cu tipuri concrete (figura 6.1). În terminologia folosită s-a specificat că un obiect este la rândul lui o instanță a unei clase.

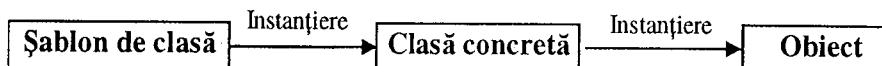


Fig. 6.1 Definirea entităților prin instanțieri

La definirea şablonelor de clasă atât datele, cât și funcțiile membre care lucrează cu ele se definesc în legătură cu unul sau mai multe tipuri generice de date. Evident, funcțiile membre pot beneficia de supraîncărcare și deci pot avea același nume. Nu același lucru se poate spune însă despre clase. După cum știm nu putem avea două clase care să poarte același nume, deoarece o clasă corespunde în fapt unui tip de date, *user build-in* și nu putem avea două tipuri de date care să se numească la fel; (putem avea în schimb, un singur tip care să se numească în mai multe feluri - facilitatea *typedef*)

Așadar, compilatorul e forțat să dea și nume claselor construite după modelul dat de programator. Totul este în ordine, numai că aceste nume de clase trebuie cunoscute și de programator pentru a se folosi de ele, ca să creeze obiecte din clasele respective. Prin convenție, programatorul poate referi clasele construite din template după un nume compus din numele dat de el în şablon (*vector*), urmat de diverse nume de tip pentru care se cere *instanțierea* modelului.

Pot avea în consecință clasele *vector<int>*, *vector<double>*, *vector<persoana>* etc., care sunt particularizări ale modelului de clasă *vector*, pentru tipurile *int*, *double*, respectiv *persoana*.

Din exemplele prezentate se observă că instanțierea şablonului clasei *vector* s-a făcut în momentul definirii unui obiect:

```
vector<int> vi(3);
```

O astfel de instanțiere este denumită *instanțiere implicită*.

Numele de instanță a clasei *template* poate fi folosit nu numai la generarea de obiecte ale clasei, ci și pentru a descrie funcții care nu sunt template, dar care lucrează cu obiecte provenind din clase template. Putem scrie spre exemplu, funcția care măsoară doi vectori, sub aspectul numărului de componente, și returnează -1, 0, +1, după cum primul vector este mai scurt, egal sau mai lung decât cel de-al doilea.

```

int masoara( vector<int>& v1, vector<double>& v2)
{
    int dim1=v1.spune_dim(), dim2=v2.spune_dim();
    if(dim1==dim2) return 0;
    else if(dim1<dim2) return -1;
    else return +1;
}

```

După cum se vede, funcția lucrează cu referințe de tipuri provenind din template, dar ea în sine nu e template; dovedă că ea nu are în față antetul *template<classT>* și nici nu face în interior referiri la vreun tip *T*, generic, ci doar la tipuri deja instantiate: *vector<int>* și *vector<double>*. Ce se întâmplă dacă vrem să comparăm acum doi vectori, de *char* și de *float*? Evident trebuie să mai scriem o variantă a funcției masoară ():

```
int masoară (vector<char>&, vector<float>&);
```

O soluție mult mai elegantă este să dăm și pentru această funcție un şablon. Problema nu e banală, deoarece trebuie remarcat că, spre deosebire de funcțiile template obișnuite, modelul pentru funcția *masoara* ar urma să fie instantiat după tipuri, care la rândul lor sunt obținute din alte *template*, de data aceasta şablonane de clase. Într-adevăr, compilatorul de C++ permite acest lucru sub forma:

```

#include <iostream.h>
template <class T>
class vector
{
    T * pe; int dim;
public:
    vector(int);
    ~vector( ) { delete[ ] pe; }
    int spune_dim(){return dim; }
};

```

```

template <class T>
vector<T>::vector(int n) : dim(n)
{
    pe = new T[n];
    for (int i = 0; i < dim; i++)
        { cout << "\n elem_" << i << " "; cin >> pe[i]; }
}

template <class TT1, class TT2>
int masoara( vector<TT1>& v1, vector<TT2>& v2)
{
    int dim1=v1.spune_dim(), dim2=v2.spune_dim();
    if(dim1==dim2) return 0;
    else
        if(dim1<dim2) return -1;
        else return +1;
}

```

folosite ca în programul:

```

void main()
{
    vector<int> vi(2);
    vector<double> vd(2);
    cout << masoara(vi,vd);
}

```

În primul rând, am parametrizat funcția după două tipuri, semn că funcțiile create pe baza acestui model intenționează să compare și vectori de tipuri diferite de elemente.

În al doilea rând, am denumit intenționat tipurile cu *TT* sugerând cititorului ca sunt *Tipuri* provenind din *Template*, date în altă parte, la *templatizarea* clasei *vector*.

Dacă am fi avut de comparat totdeauna diferite tipuri de vectori, doar cu un vector de *int*, am fi putut defini modelul:

```

template<class TT>
int măsoară (vector<TT> &, vector<int> &)

```

Se vede clar aici diferența dintre *<class TT>* care e generică și *vector<int>*, care provine din același template, dar este deja instanțiat. Este ca și cum am spune că reperul de comparare este *fix*, pe când elementele comparate variază.

Din exemplele anterioare pot fi sesizate două momente distincte în lucru cu clase template:

- momentul definirii şablonului
- momentul instanțierii (folosirii) şablonului pentru obținerea de clase efective.

În general, acest ultim moment este legat de lucru concret cu o clasă.

Când definirea template-ului se face în același program cu utilizarea claselor bazate pe el, decalajul dintre cele două momente este mic. Să ne amintim însă că de cele mai multe ori construim biblioteci de clase, care pot fi folosite mai târziu, decalajul în timp între definire și utilizare devenind semnificativ. Ne punem problema dacă nu am putea amâna unele decizii privind structura clasei pentru cea de-a două fază. Spre exemplu, dacă vectorul din exemplul anterior nu ar fi fost stocat prin alocare dinamică, la o dimensiune decisă la execuție, stocarea lui într-o variabilă ar fi ridicat problema dimensionării lui printr-o constantă, dată în clasa generică. Ar fi fost foarte util dacă această constantă ar putea fi dată nu ca o constantă definitivă în şablon, ci declarată drept constantă, dar fixată efectiv doar la momentul instanțierii şablonului. Deosebirea este esențială: una este să stabilim constantă definitiv în şablon, ceea ce ar însemna producerea de clase vector cu același număr de componente, indiferent de tipul de date din vector și de variabila vector, și alta să putem preciza constantă pentru fiecare instanțiere în parte, producând clase de vectori de dimensiuni diferite, pentru fiecare caz în parte.

Descrierea şablonului va fi atunci :

```

#include <iostream.h>
template <class T, int n>
class vector
{
    T v[n];           // stocare vector în variabilă
public:
    int spune_dim() { return n; }
};

template < class TT1, class TT2 >
int masoara( TT1 & v1, TT2 & v2)
{
    int dim1 = v1.spune_dim(), dim2=v2.spune_dim();
    if(dim1 == dim2) return 0;
    else
        if(dim1<dim2) return -1;
        else return +1;
}

```

```
void main()
{
    vector<int,3> vi;
    vector<double, 2> vd;
    int v= masoara(vi, vd);
    cout << v;
}
```

Vectorul *vi* dimensionează masivul din obiect ca având trei elemente în timp ce vectorul *vd* îl dimensionează la două elemente.

Obținem astfel, constante în raport cu o clasă efectivă, dar nu în raport cu şablonul, unde dimensiunea este dată generic prin *n*.

Atât pentru tipurile generice de dată cât și pentru constante se pot stabili și *valori implicite*. Astfel, vom descrie şablonul clasei *vector* pentru a lucra implicit cu un masiv de 50 elemente de tip *char*.

```
#include <iostream.h>

template <class T=char, int n = 50>
class vector
{
    T v[n];
public:
    int spune_dim() { return n; }
};
```

Programul :

```
void main()
{
    vector<> v1;
    vector<double> v2;
    vector<int, 4> v3;
    cout << v1.spune_dim() << endl;
    cout << v2.spune_dim() << endl;
    cout << v3.spune_dim();
}
```

declară trei obiecte de tip *vector* :

- *v1*, este masiv de tip *char* cu 50 de elemente; a se observa prezența parantezelor unghiulare *<>* la definirea obiectului;
- *v2*, este un masiv de tip *double* cu 50 elemente;
- *v3*, este un masiv de tip *int* cu 4 elemente.

Dacă biblioteca de clase furnizată unui beneficiar este de tip sursă totul este în ordine, căci atât şabloanele cât și instanțierile lor intră împreună în compilare.

Ce se întâmplă dacă vrem să dăm clasa ca bibliotecă obiect (*LIB*) ? Compilatorul nu are cum să mai vadă ce instanțieri viitoare se vor face din acel şablon. Standardul limbajului a prevăzut în acest caz posibilitatea forțării unor instanțieri, anunțând viitoarele clase; declarația:

```
template class vector<int,10>;
```

cere instanțierea clasei *vector* pentru 10 întregi și e comparabilă cu invocarea macrodefiniției care face definirea clasei pe un caz particular. O astfel de instanțiere este denumită *instanțiere explicită*.

Din nefericire, funcțiile membre nu sunt *instantiate* odată cu clasa, ci doar dacă sunt invocate; logica este simplă: instanțierea şablonului însemnă generarea de cod pentru recunoașterea descrierii clasei, dar nu rezervare de memorie pentru membri. Funcțiile membre respectă aşadar convenția funcțiilor *template*: nu este generat codul funcției, decât pentru tipurile pentru care este folosită funcția (dacă apare un apel al funcției pe tipul respectiv). Deci, dintre funcțiile membre se generează cod doar pentru acelea apelate efectiv. Compilatorul permite și în acest caz o cerere explicită de instanțiere:

```
template class vector<int,10>:: vector(int);
```

cere instanțiere constructorului clasei instanțiată și ea mai sus, pe când:

```
template int masoara <int, double> (int,double);
```

cere instanțierea unui exemplar al funcției *masoara*, ce lucrează pe *int* și *double*.

### 6.3 Specializări

Prin specializări se înțelege definirea de funcții, metode sau clase pentru unul sau mai multe tipuri concrete de date, care vor fi utilizate în locul şablonului general. Acest lucru este cerut de comportamentul ușor diferit al clasei pe un tip dat, față de şablonul general al clasei.

În acest capitol s-a definit şablonul clasei *vector* având tipul generic *T*. El a fost folosit pentru lucru cu vectori de diferite tipuri (*int*, *float*) adaptându-*si* automat metodele la tipul concret. În cazul în care dorim ca vectorul să lucreze cu şiruri de caractere, tipul concret trebuie să fie *char\**. O declaraţie de tipul:

```
vector<char*> vs(3);
```

ar fi suficientă pentru a lucra cu un vector de trei şiruri de caractere ?

Pentru a da răspunsul trebuie să facem o analiză a şablonului:

```
template <class T> class vector
{
    T * pe;
    int dim;
public:
    vector(int);
    ~vector() { delete[ ] pe; }
    T& operator[] (int i) { return pe[i]; }
    void afis(); void sort();
};
```

- *pe* devine membru de tip *char \*\**;
- constructorul ar aloca un spaţiu de memorie pentru un vector capabil să stocheze trei pointeri spre tipul *char*, dar citirea şirurilor nu se va putea realiza decât după alocarea spaţiului de memorie necesar stocării lor;
- sortarea vectorului de şiruri presupune ordonarea alfabetică a şirurilor; ştim că pentru a compara două şiruri identificate prin adrese de caracter trebuie să folosim funcţia *strcmp* şi nu operatorul *>* care aici ar realiza compararea a două adrese şi nu a două şiruri.

Din această analiză deducem, pe de o parte că simpla utilizare a şablonului *vector* pentru tipul *char\** NU rezolvă problema lucrului cu un vector de şiruri de caractere; pe de altă parte, se sugerează şi ceea ce trebuie rescris în clasa *vector*: constructorul şi metoda de sortare.

Sintaxa generală în cazul specializării unei metode este:

```
tip_r nume_c <tipc>::nume_m( lista_p_f )
{
    .....
}
```

unde:

- *tip\_r* – tipul returnat de metodă;

- *nume\_c* – reprezintă numele clasei şablon;
- *tic* – un tip concret de date;
- *nume\_m* – numele metodei;
- *lista\_p\_f* – listă parametri formali.

În continuare vom specializa constructorul clasei *vector* şi metoda de sortare pentru a putea lucra cu tipul *char\**:

```
#include <string.h>
#define L_SIR 100
vector<char*>::vector(int n):dim(n)
{
    pe = new char*[dim];
    cin.ignore(L_SIR, '\n');
    for (int i = 0; i < dim; i++)
    {
        cout << "\n elem_ " << i << " ";
        pe[i]=new char[L_SIR];
        cin.getline(pe[i],L_SIR);
    }
}
void vector<char*>::sort()
{
    int i,j; char *aux;
    for (i=0; i < dim-1; i++)
        for ( j = i+1; j < dim; j++)
            if(strcmp(pe[i],pe[j])>0)
                { aux= pe[i]; pe[i]=pe[j]; pe[j]=aux; }
}
```

Se observă că :

- *L\_SIR* – este o constantă simbolică ce indică lungimea maximă a unui şir de caractere ;
- constructorul alocă memorie pentru fiecare şir de caractere, apoi se face citirea ;
- sortarea foloseşte funcţia de bibliotecă *strcmp* pentru compararea a două şiruri de caractere, dar se interschimbă adresele şi nu şirurile.

Datorită faptului că se alocă de către constructor memorie pentru vector, cât şi pentru fiecare şir în parte, se impune să se specializeze şi deconstructorul în vederea dezalocării întregii memorii alocate de constructor:

```
vector<char*>::~vector()
{
    for (int i = 0; i < dim; i++) delete [L_SIR] pe[i];
}
```

```
    delete [dim]pe;
}
```

Având şablonul clasei *vector* definit și metodele specializate pentru tipul *char\** programul:

```
void main()
{
    vector<int> vi(3);
    vi.afis(); vi.sort(); vi.afis();
    vector<float> vf(3);
    vf.afis(); vf.sort(); vf.afis();
    vector<char*> vs(3);
    vs.afis(); vs.sort(); vs.afis();
    cout<<"\n Primul sir:"<<vs[0];
}
```

funcționează corect atât cu tipurile numerice (*int, float*), cât și cu siruri de caractere.

O altă observație importantă constă în faptul că s-a afișat vectorul de siruri apelând metoda *afis* și s-a obținut primul sir din vector folosindu-se operatorul [] supraîncărcat în clasa *vector*, fără ca aceste metode să fie specializate pentru tipul *char\**. Acest lucru a fost posibil deoarece codul ramâne identic chiar în condițiile în care tipul concret este *char\**; deci nu trebuie specializate decât acele metode care trebuie să conțină un cod diferit de cel din şablonul standard, pentru un anumit tip de date (în cazul nostru *char\**).

Ca regulă, trebuie reținut că specializările au prioritate față de instantierea şablonului pentru un tip concret de date. Când compilatorul găsește o formă particulară de implementare a unei funcții *template* o folosește pe aceasta, nu mai generează alta pe baza şablonului general.

Din cele prezentate s-a observat că specializarea clasei *vector*, pentru a permite lucru cu siruri de caractere, s-a făcut la nivelul unor metode. Există posibilitatea de a specializa întreaga clasă *vector* pentru a putea lucra într-un mod special cu un tip concret de date. Specializarea întregii clase este utilă atunci când se dorește ca pentru un anumit tip de date să se adauge noi metode sau date membre. Dezavantajul este dat de faptul că trebuie rescrise toate metodele clasei specializate, chiar când codul din şablonul general ar fi valabil și pentru tipul respectiv de date.

Definirea specializării unui şablon de clasă se face folosind sintaxa:

```
template <> class nume_c <tipc>
{ ..... }
```

unde:

- *nume\_c* – reprezintă numele clasei şablon pentru care se definește specializarea;
- *tipc* – tipul concret de date pentru care se face specializarea clasei.

Pentru exemplificarea acestei facilități vom specializa întregul şablon al clasei *vector* pentru a permite lucru cu siruri de caractere (*char\**). În plus, se va mai defini, pentru această specializare, o metodă care să determine lungimea efectivă a unui anumit sir din vectorul de siruri; altfel spus o metodă de determinare a lungimii unui element din vector. O astfel de metodă nu se justifică a fi definită în şablonul general pentru că pe tipuri numerice de date lungimea elementelor este aceeași și coincide cu lungimea tipului specificat.

```
#include <string.h>
#define L_SIR 100
template <> class vector<char*>
{
    char **pe;
    int dim;
public:
    vector(int);
    char*& operator[ ] (int i) { return pe[i]; }
    void afis(); void sort();
    int lungime_element(int);
    ~vector( );
};

vector<char*>::vector(int n):dim(n)
{
    pe = new char*[dim];
    cin.ignore(L_SIR, '\n');
    for (int i = 0; i < dim; i++)
    {
        cout << "\n elem_ " << i << " ";
        pe[i]=new char[L_SIR];
        cin.getline(pe[i],L_SIR);
    }
}

void vector<char*>::sort()
{
    int i,j; char *aux;
    for (i=0; i < dim-1; i++)
    {
        for (j=i+1; j < dim; j++)
        {
            if (strcmp(pe[i],pe[j]) > 0)
            {
                aux = pe[i];
                pe[i] = pe[j];
                pe[j] = aux;
            }
        }
    }
}
```

```

        for ( j = i+1; j < dim; j++)
            if(strcmp(pe[i],pe[j])>0)
                { aux= pe[i]; pe[i]=pe[j]; pe[j]=aux; }

    void vector<char*>::afis()
    {
        cout << endl;
        for (int i = 0; i < dim; i++) cout << pe[i] << " ";
    }

    vector<char*>::~vector()
    {
        for (int i = 0; i < dim; i++) delete [L_SIR] pe[i];
        delete [dim]pe;
    }

    int vector<char*>::lungime_element(int i)
    {
        return strlen(pe[i]);
    }

```

Având definit şablonul clasei *vector* și specializarea pentru tipul *char\** în forma prezentată, programul următor prezintă modul de folosire a şablonului pentru diferite tipuri de date.

```

void main()
{
    vector<int> vi(3);
    vi.afis(); vi.sort(); vi.afis();
    vector<float> vf(3);
    vf.afis(); vf.sort(); vf.afis();
    vector<char*> vs(3);
    vs.afis(); vs.sort(); vs.afis();
    cout << "\n Primul sir:" << vs[0] << " are lungimea:" << vs.lungime_element(0);
}

```

Există posibilitatea de a realiza și *specializări parțiale* pentru şabioane. Această specializare are două forme de manifestare:

- definirea şablonului când acesta substituie tipul generic cu pointer la tip, ca în exemplul următor:

```

template<class T>
class clss
{
    // sablon de clasa
};

```

```

template <class T>
class clss<T*>
{
    // specializare pentru pointer la tip
};

```

Având cele două definiții, o instanță de tipul: *clss<int> a*; va folosi şablonul general al clasei, pe când instanța: *clss<double\*> b*; va utiliza specializarea pentru pointer la tip (*double\**).

- definirea şablonului când unele tipuri se păstrează generice, în timp ce altele devin concrete; să presupunem definirea următorului şablon având două tipuri generice:

```

template<class T1,class T2>
class cls
{
    // sablon general
};

```

și a unei specializări parțiale care va păstra primul tip (T1) generic, iar pe cel de-al doilea îl va concretiza și va fi *int*:

```

template<class T1>
class cls<int>
{
    // specializare parțială
};

```

Având cele două definiții, o instanță de tipul: *cls<int, char> a*; va folosi şablonul general al clasei, pe când instanța: *clss<double, int> b*; va utiliza specializarea parțială, având al doilea tip *int*.

De remarcat că o parte din specializările şabloanelor nu sunt acceptate sub compilatoarele Visual C++ 5, 6 și nici chiar în 7 (Visual Studio.Net).

## 6.4 Relații între şabioane

În diferite capituloare anterioare am pus în evidență diferite tipuri de relații între clase. Unele relații sunt intrinsec legate de conceptul de

programare orientată obiect, altele sunt definite și întreținute de către programator în scopul dezvoltării de aplicații.

### Derivarea claselor *template*

O relație foarte importantă care definește și o proprietate a programării orientată obiect este cea de ***derivare***. Această relație este valabilă și pentru sabloanele de clasă, mai precis se va furniza un şablon de clasă, derivat dintr-un alt şablon de clasă (de bază).

Să presupunem că dorim să derivăm din şablonul clasei *vector*, şablonul clasei *vect\_sort*, care ține totdeauna elementele sortate. Clasa de bază dispune de metoda de sortare, dar nu o invocă decât la cerere. Noua clasă realizează *autosortarea*, adică la orice schimbare în vector se invocă automat sortarea.

Prima sortare a vectorului se va face evident prin constructor; în acest sens constructorul lui *vect\_sort* va apela mai întâi constructorul clasei de bază, după care va invoca sortarea.

Să presupunem ca supraîncărçăm în clasa *vector* operatorul ***+=*** pentru concatenarea a doi vectori; *vect\_sort* va moșteni metoda, dar o va supraîncărca, astfel încât după concatenare să invoke automat și sortarea, deoarece concatenarea produce modificări structurale.

```
#include <iostream.h>
#include <string.h>
template <class T> class vector
{
    T * pe;
    int dim;
public:
    vector(int);
    vector & operator+=(vector &);
    T& operator[ ](int i) { return pe[i]; }
    void afis();
    void sort();
    ~vector() { delete[ ] pe; }
};
template <class T> vector<T>::vector(int n):dim(n)
{
    pe = new T[n];
    for (int i = 0; i < dim; i++)

```

```
{ cout << "\n elem_ " << i << " "; cin >> pe[i]; }

template <class T>
vector<T> & vector<T>::operator+=( vector<T>& v2)
{
    T *nou = new T[dim + v2.dim];
    for (int i = 0; i < dim; i++) nou[i]=pe[i];
    for ( i = dim; i < dim+v2.dim; i++) nou[i]=v2.pe[i-dim];
    delete [] pe; pe=nou; dim+=v2.dim;
    return *this;
}

template <class T> void vector<T>::afis()
{
    cout << endl;
    for (int i = 0; i < dim; i++) cout << pe[i] << " ";
}

template <class T> void vector<T>::sort()
{
    int i,j; T aux;
    for (i = 0; i < dim-1; i++)
        for ( j = i+1; j < dim; j++)
            if(pe[i] > pe[j])
                { aux= pe[i]; pe[i]=pe[j]; pe[j]=aux; }
}

template <class T>
class vect_sort:public vector<T>
{
public:
    vect_sort(int );
    vect_sort & operator+=(vect_sort<T> & v2);
};

template <class T>
vect_sort<T>::vect_sort(int n) : vector<T>(n) { sort(); }

template <class T>
vect_sort<T>& vect_sort<T>::operator+=(vect_sort<T> & v2)
{
    *this.vector<T>:: +=(v2); sort(); return *this;
}

void main()
{
    //vector<int> vi(3), vi2(2);
}
```

```

//vi+=vi2; vi.afis(); vi.sort(); vi.afis();
// vector<float> vf(3);
// vf.afis(); vf.sort(); vf.afis();
vect_sort<int> vis1(3), vis2(2);
vis1+=vis2; vis1.afis();
}

```

Ca observație, se poate specifica faptul că metoda de adăugare a unui element în vectorul sortat se poate face prin inserție directă; pentru simplificarea înțelegerei am adoptat totuși metoda de adăugare a elementului în vector după care s-a apelat metoda de sortare.

Exemplul anterior ilustrează derivarea unei clase *template* pornind de la o clasă de bază care este tot *template*; în realitate sunt posibile următoarele combinații pentru relația de moștenire:

- clasă *template* derivată din clasă *template*;
- clasă *template* derivată din clasă non *template*;
- clasă non *template* derivată din clasă *template*.

Programul următor definește o clasă *template b*, din care se derivează clasa non *template d*; deoarece clasa *d* nu are parametrizat nici un tip, pentru clasa de bază trebuie folosite doar instanțieri concrete ale clasei *template* (*d* este derivat din *b<int>* nu din *b<T>*).

```

template <typename T>
class b
{
public:
    b() { }
    virtual ~b() { }
};

class d : public b<int>
{
public:
    d() { }
    virtual ~d() { }
};

void main()
{
    b<float> a;
    d c;
}

```

Pentru a ilustra derivarea unei clase *template* dintr-o clasă non *template* modificăm programul de mai sus sub forma:

```

class b
{
public:
    b() { }
    virtual ~b() { }

};

template <typename T>
class d : public b
{
public:
    T x;
    d() { }
    virtual ~d() { }

};

void main()
{
    b a;
    d<float> c;
}

```

Relația de *moștenire multiplă* prezintă anumite particularități când se aplică pentru şablonane de clase.

Şablonanele claselor de bază fiind parametrizate în funcție de unele tipuri de date, şablonul clasei derivate trebuie să fie parametrizat în funcție de tipurile generice ale claselor de bază și eventual poate avea și propriile tipuri generice. Moștenirea tipurilor generice de la şablonanele claselor de bază se justifică în momentul în care tipurile claselor de bază pot fi diferite când se instanțiază şablonul clasei derivate.

Spre exemplu, avem două şablonane de clase de bază definite astfel:

```

#include<iostream.h>

template<class T1>
class b1
{
protected:
    T1 a;
public:
    b1(T1 x):a(x) { }
    void afis()
    {
        cout<<"Membru din b1:"<<a<<endl;
    }
};

```

```

template<class T2>
class b2
{
protected:
    T2 c;
public:
    b2(T2 x):c(x) {}
    void afis()
    {
        cout<<"Membru din b2:"<<c<<endl;
    }
};

```

Se va defini şablonul clasei derivate (*d*) care va moşteni cele două şabioane, având două tipuri generice, unul aferent şablonului clasei *b1* şi al doilea aferent şablonului clasei *b2*.

```

template<class T1, class T2>
class d : public b1<T1>, public b2<T2>
{
public:
    d(T1 y, T2 z):b1<T1>(y),b2<T2>(z) {}
    void afis() { cout<<"Valorile :"<<a<<" si "<<c<<endl; }
};

```

În programul următor se prezintă modul de utilizare a şablonului clasei derivate (*d*) care va fi instantiat utilizându-se două tipuri concrete diferite:

```

void main()
{
    d<int,float> obd(45,90.67);
    obd.afis();
}

```

Dacă instanțierea şablonului clasei derivate presupune inducerea aceluiăș tip de dată şabioanelor claselor de bază, atunci şablonul clasei derivate se poate defini în funcție de un singur tip generic de dată:

```

template<class T>
class d : public b1<T>, public b2<T>
{
public:
    d(T y, T z):b1<T>(y),b2<T>(z) {}
    void afis() { cout<<"Valorile :"<<a<<" si "<<c<<endl; }
};

void main()
{
    d<int> obd(45,55);
}

```

```

    obd.afis();
}

```

A se observă în *main* că şablonul clasei derivate a fost instantiat cu un singur tip concret de date.

### Componerea claselor *template* prin includere

Relația de includere a claselor, adică de definire a unei clase în altă clasă se poate extinde și la nivelul şabioanelor de clasă.

Pentru a exemplifica această relație se va defini şablonul clasei lista simplu înlănțuită, care va include definirea şablonului nod; tipul informației utile din nod va fi unul generic.

```

template<class T>
class lista
{
    class nod
    {
        friend lista;
        T info;
        nod *next;
    public:
        nod(T k, nod *urm=NULL):info(k),next(urm) {}
        nod *get_next() { return next; }
        friend ostream& operator<<(ostream& os, nod* n)
        { os<<n->info; return os; }

        nod *cap;
        int n;
        void sterge_tot(nod *);
    public:
        lista():cap(NULL),n(0) {}
        void ins_incep(T );
        int count() { return n; }
        friend ostream& operator<<(ostream& , lista& );
        ~lista() { sterge_tot(cap); }
    };

```

Se observă că declarația de şablon *template<class T>* s-a făcut numai pentru clasa *lista* nu și pentru clasa inclusă *nod* cu toate că și nodul este un şablon și folosește același tip generic (*T*). Cu alte cuvinte, dacă o clasă este inclusă într-un şablon, devine automat un şablon.

Din acest exemplu se mai poate surprinde și un alt tip de relație și anume relația de tip *colecție* în sensul că lista este *privită* ca fiind o colecție de noduri. Legătura dintre cele două şabloane s-a făcut explicit prin pointerul la nod *cap*.

```
template<class T>
void lista<T>::sterge_tot(nod *cp)
{
    if(cp) { sterge_tot(cp->next); delete cp; }
    cap=NULL; n=0;
}

template<class T>
void lista<T>::ins_incep(T k) { nod *t=new nod(k,cap); cap=t; n++; }

template<class T>
ostream& operator<<(ostream& os, lista<T>& l)
{
    lista<T>::nod *cp=l.cap;
    os<<"(";
    while(cp) { os<<cp; if(cp->get_next())os<<" , "; cp=cp->get_next(); }
    os<<")";
    return os;
}
```

La definirea şablonului funcției *friend* care supraîncarcă operatorul << pentru afișarea elementelor listei, se observă modul în care se definește un pointer la un nod al listei: *lista<T>::nod \*cp*; aceasta deoarece nodul a fost definit în interiorul şablonului clasei *lista*. Tipul generic (*T*) se specifică doar pentru şablonul clasei *lista*, deoarece *lista* a fost declarată explicit ca fiind un şablon, nodul fiind inclus în lista devine implicit şablon de clasă. Dacă tipul ar fi, de exemplu, *int*, declararea pointerului la nod va fi: *lista<int>::nod \*pni*.

În programul următor se va exemplifica utilizarea şablonului *lista* pentru două tipuri de date: *int* și *double*.

```
void main()
{
    lista<int> li;
    li.ins_incep(10); li.ins_incep(56);
    cout<<"Lista de intregi:"<<li<<" are "<<li.count()<<" elemente"<<endl;
    lista<double> lr;
    lr.ins_incep(140.44); lr.ins_incep(54.67); lr.ins_incep(901);
    cout<<"Lista de nr reale:"<<lr<<endl;
}
```

### Componerea claselor *template* prin parametrizare cu altă clasă *template*

Un şablon de clasă poate primi când se instanțiază un tip dat la rândul lui un alt şablon de clasă. Până acum am instanțiat şabloane de clasă substituind tipul generic cu un tip concret de date, de exemplu:

*lista<int> li;*

Să considerăm un exemplu în care instanța unui şablon substituie tipul generic cu un tip dat prin alt şablon de clasă.

Vom construi şablonul clasei *persoana* care va avea tip generic de date pentru salariajii persoanei. Justificăm acest lucru prin faptul că exprimat în lei, salariajul va fi de tip *int*; dacă avem în vedere că poate primi salariajul în Euro atunci tipul va fi *float* pentru că diviziunile sunt importante.

```
template<class T>
class persoana
{
    char nume[50];
    T sal;
public:
    persoana(char *np, T s):sal(s) { strcpy(nume, np); }
    friend ostream& operator<<(ostream& os, persoana &p)
    {
        os<<p.nume<<" "<<p.sal;
        return os;
    }
};
```

În plus, vom mai defini şablonul unei clase colecție numită *salariati*, care va opera cu persoanele ce sunt salariajii unei unități economice. Pentru memorarea salariajilor se va folosi şablonul clasei *lista*, definit anterior, cu tipul *persoana*. Se știe că *persoana* a fost definită tot ca un şablon de clasă, deci un obiect tip *lista* se va defini sub forma :

*lista< persoana<T> > ls;*

Ca observație sintactică, se poate preciza că între cele două paranteze unghiulare consecutive se va insera un spațiu pentru a nu se confunda cu operatorul de shiftare la dreapta (>>).

Şablonul clasei *salariati* se va defini astfel:

```
template<class T>
class salariati
{
    lista< persoana<T> > ls;
public:
    void add(persoana<T> prs) { ls.ins_incep(prs); }
    friend ostream& operator<<(ostream& os, salariati& l)
    {
        os<<l.ls;
        return os;
    }
};
```

și are ca metode:

- *add* – pentru a adăuga o persoană la colecția de salariați ;
- operatorul *<<* supraîncărcat pentru afișarea statului de plată sub forma: nume persoană și salariu.

În programul următor se va exemplifica utilizarea celor două sabloane pentru construirea și afișarea statului de plată:

```
void main()
{
    salariati<int> stat;
    stat.add(persoana<int>("Valentina",96785));
    stat.add(persoana<int>("Daniel",90333));
    stat.add(persoana<int>("Liviu",4585));
    cout<<stat;
}
```



## **DOMENII DE NUME - NAMESPACE**

- Definirea și actualizarea domeniilor de nume
- Utilizarea domeniilor de nume

## 7.1 Definirea și actualizarea domeniilor de nume

Variabilele globale și numele de clase sunt vizibile la nivel global, adică la nivelul unic al programului. Putem divide acest domeniu în subdomenii, numite și *spații de nume* folosind facilitatea de *namespace* și putem atribui fiecare dintre identificatorii globali căte unui subdomeniu. Acest lucru ne permite să lucrăm cu identificatori care se numesc la fel, dar care aparțin unor domenii diferite. În continuare prezentăm câteva din cele mai importante operații specifice lucrului cu domeniile de nume:

- *definirea* unui domeniu și includerea unor identificatori:

```
namespace MySpace
{
    extern int x;
    void f();
}
```

- *adăugarea* de noi identificatori unui domeniu existent, fără a se considera redefinire de domeniu (se poate face și în fișiere diferite de cel care a introdus domeniul):

```
namespace MySpace
{
    int y;
    void g();
    class cls { } z;
}
```

- crearea unui *alias*, pentru un domeniu existent:

```
namespace Meu = MySpace;
```

- crearea / adăugarea de elemente la un *domeniu fără nume*, garantează că fiecare unitate de compilare dispune de cel puțin un domeniu unic de nume pentru identificatorii ei:

```
namespace
{
    double t;
}
```

calificarea membrilor se face fără a adăuga ceva în fața numelui.

- funcțiile *friend* ale unei clase inclusă într-un domeniu, devin automat membre ale domeniului respectiv (funcțiile *friend* ale unei clase preiau contextul clasei):

```
namespace Meu
{
    class cls_meia
    {
        //...
        friend void f_straina();
    }
    void Meu::f_straina() { };
```

## 7.2 Utilizarea domeniilor de nume

- 1 Utilizarea unui domeniu de nume folosind *operatorul de rezoluție*:

```
void Meu:: f() { }           // definirea propriu-zisă a lui f
void f() { }                 // alta funcție, numita tot f, dar în domeniu
                            // fără
                            // nume
Meu::cls::cls() { cout << "Constructor cls"; }
void main()
{
    MySpace::y = 1; cout << Meu::y;
    // y = 2; eroare: y nedefinit în afara domeniului Meu::
    // MySpace::x = 1; eroare: x definit extern și neidentificat la
    //                  linkeditare
}
```

După cum se observă, un spațiu de nume se asemănă cu o structură sau clasă, dar nu poate avea instanțe; el este doar o grupare de identificatori globali pentru a rezolva eventualele conflicte de nume. S-au folosit ambele alternative (*MySpace::* și *Meu::*) pentru a desemna același domeniu de nume.

Includerea într-un spațiu de nume este o simplă declarație; nu ține loc de definire, nici pentru variabile declarate extern, nici pentru funcții sau clase date doar ca prototip. Spre exemplu, nu mai este nevoie de o definire a lui *y* sau *z*, care sunt definite complet în domeniul, dar trebuie citat la linkeditare un fișier care să facă definirea lui *x*, declarat doar în domeniu, nu și definit.

Această includere este însă suficientă pentru a discerne între *y* și *Meu::y*, sau între *f()* și *Meu::f()*; dovedă că *y* fără rezoluția de scop nu este recunoscut, iar pentru *f()* putem avea două exemplare cu același prototip, căci spațiu de nume particularizează prototipurile.

Funcțiile doar citate ca prototip în domeniu nu sunt căutate la linkeditare decât dacă sunt și apelate efectiv. În prototipul funcțiilor, domeniu se asociază numelui funcției și apare după specificarea tipului returnat de funcție.

În cazul claselor, operatorul de rezoluție urmează celui de domeniu de nume *Meu::cls::*.

## ② Utilizarea unui domeniu de nume folosind directiva *using namespace*

Pentru a evita tastarea întregului nume de calificare a unui identificator, este posibilă folosirea unei directive *using namespace* care anunță că toți identificatorii dintr-o funcție care n-au fost definiți în contextul global al programului, aparțin domeniului citat prin directivă.

Spre exemplu, putem scrie o nouă funcție care folosește cea mai mare parte dintre identificatori ca aparținând domeniului *Meu::*, fără să le dăm de fiecare dată numele complet:

```
void f_noua()
{
    using namespace Meu;
    cls c1; int yy=7; cout <<y;
}
```

Astfel, declarația *cls c;* presupune că s-a descris clasa *cls* în domeniul *Meu::*, altfel, în afara funcției ce anunță *using namespace Meu;* o declarație *cls c2;* ar da eroare de compilare, trebuind să fie scrisă:

*Meu::cls c2;*

Variabila *y* este cea definită în același domeniu și nu mai necesită altă definire, în timp ce *yy* este o variabilă aparținând domeniului local, fără nume, al funcției. Într-o funcție care anunță *using namespace Meu,* totul se petrece ca și cum orice referire *xx* nerezervată se va căuta de forma *Meu::xx.*

Este posibil să anunțăm folosirea a două sau mai multe domenii de nume:

```
using namespace Meu;
using namespace Tau;
```

și se vor căuta rezolvări în toate, dar pot apărea și conflicte de nume, dacă avem identificatori ce se numesc la fel în două domenii. Din fericire, conflictul este semnalat numai la folosirea efectivă a numelor; adică putem utiliza două domenii implicate de nume care au membrii ce se numesc la fel, dar să nu lucrăm cu acci identificatori ce pot crea ambiguități.

## ③ utilizarea unui domeniu de nume folosind declarația de *using*

Pentru domeniile ce au membri ce se numesc la fel, putem preciza punctual care versiune folosim pentru fiecare membru ce crează ambiguități; spre exemplu, cele două domenii *Meu* și *Tau* au câte un membru *y* și *f();* funcția *f\_noua()* anunță că *y* și *f()* folosiți de ea sunt cei din domeniul *Tau:*

```
namespace Tau
{
    int y=2222; void f() {} }

void f_noua()
{
    using namespace Meu;
    using namespace Tau;
    using Tau::y; using Tau::f;
    f(); cout <<y;
}
```

Evident, pentru un identificator (ce poate sau nu crea ambiguități de calificare) putem da o singură declarație de *using*.

O declarație de *using* nu precizează și tipul identificatorului; acest lucru poate crea probleme când în domeniu avem o funcție *f()* supraîncărcată (nu în cazul identificatorilor de variabile, unde nu pot avea două variabile ce se numesc la fel, chiar dacă sunt de tipuri diferite); compilatorul distinge între supraîncărcări datorită signaturilor diferite, dar citând *using Tau::f;* (pentru funcții nu se dă prototipul, pentru că el există în domeniul respectiv de nume) nu putem individualiza despre care supraîncărcare este vorba, astfel încât toate versiunile de supraîncărcare ale lui *f()* folosite sunt numai cele din domeniul citat.

Avem în schimb și un avantaj legat de supraîncărcare: în domenii diferite putem folosi funcții cu prototipuri identice, dacă la folosire precizăm printr-o declarație de *using*, versiunea din care domeniu o vom folosi;

acest lucru ne apare ca și cum putem supraîncărca o funcție prin versiuni cu același prototip.

O declarație de *using* poate fi dată și în conținutul unui domeniu de nume; acel domeniu apare deci ca o regrupare de identificatori aparținând altor domenii:

```
namespace Unu
{
    void f () { cout << "\n f din Unu";}
    void g () { cout << "\n g din Unu";}
    // ...
}

namespace Doi
{
    void f () { cout << "\n f din Doi";}
    void g () { cout << "\n g din Doi";}
    // ...
}

namespace Mixt
{
    using Unu::f;
    using Doi::g;
    // ...
}

void main()
{
    using namespace Mixt;
    f();      // Apel Unu::f();
    g();      // Apel Doi::g();
}
```



## BIBLIOTECA DE ȘABLOANE STANDARD C++ (*Standard Template Library - STL*)

- Structura de ansamblu a bibliotecii
- Containere secvențiale
- Containere asociative
- Containere adaptive
- Iteratori
- Algoritmi
- Aplicații

## 8.1 Structura de ansamblu a bibliotecii

Două sunt elementele de dificultate legate de folosirea şabloanelor standard; *primul* ţine de înțelegerea corectă a facilității de template, folosită majestuos în realizarea STL (Standard Template Library); capitolul 6 oferă un sprijin esențial în depășirea acestei probleme, iar consultarea surselor ce definesc principalele clase template (fișierele antet ce se includ pentru recunoașterea unei structuri parametrizate) ne permite construirea unor funcții template de utilizator, care să conlucreze cu cele din STL.

*Cel de-al doilea* element de dificultate ţine de alegerea corectă a structurii adecvate problemei de rezolvat; capitolul de față își propune și o fundamentare logică a acestei alegeri, pornind de la particularitățile de implementare a claselor din STL. Vom începe cu o privire de ansamblu asupra acestei biblioteci.

STL are trei componente majore:

1. **Containerele** – care implementează şabloanele principalelor structuri de date; gruparea lor se prezintă în tabelul 8.1.
2. **Iteratorii** – generalizează principalele modalități de a accesa un element dintr-un container
3. **Algoritmii** – implementează principalele operații, într-o manieră independentă de container

*Containerele sau colecțiile* se definesc ca fiind acele clase capabile să stocheze mai multe elemente de același fel.

*Containerele sevențiale* sunt structuri liniare de date, ce permit un acces bazat pe ordinea elementului în container; această ordine este deci cea care contează în regăsire și ea este controlată de programator prin locul inserărilor și ștergerilor de elemente; la alte tipuri de containere, ordinea este controlată de sistem (spre exemplu, păstrarea sortată a cheilor, pentru regăsire rapidă).

*Containerele asociative* păstrează fie valori ce pot fi considerate chei, fie perechi (*asocieri*) chei – valoare, permitând astfel accesul direct la datele stocate, prin mecanismul cheilor de regăsire; datele, pot fi regăsite direct deoarece cheile sunt păstrate totdeauna sortate.

Tip container	Clasa	Observații
Secvențiale	vector	implementează un vector stocat în memorie dinamică, într-un bloc unic de memorie; spre deosebire de vectorul din C standard, acesta poate fi redimensionat, atribuit altui vector, permite adresarea unui element cu verificarea încadrării în domeniul indexului
	list	implementează lista dublu-înlănțuită, cu stocarea elementelor disparat în memorie, dar legate prin pointeri
	deque	este implementată similar vectorului, dar pe mai multe blocuri mari, gestionate la rândul lor prin vectori de pointeri
Asociative	set	definește o mulțime de elemente, sortate și unice
	multiset	definește o mulțime de elemente, sortate
	map	definește o mulțime de elemente sortate și unice, elementul fiind o pereche compusă din cheie și valoarea asociată
	multimap	definește o mulțime de elemente sortate, elementul fiind o pereche compusă din cheie și valoarea asociată
Adaptive	stack	implementează stiva (conține metode specializate de lucru cu stiva)
	queue	implementează structura de coadă cu metodele specifice de exploatare
	priority_queue	definește o coadă cu priorități (cel mai mare element va sta în capul cozii)

Tabelul 8.1 Clasele din STL

*Containerele adaptor* nu sunt containere în adevăratul sens al cuvântului, pentru că nu dispun de implementări și alocatori proprii, ci adaptează alte tipuri de container și nu suportă nici iteratori. Avantajul lor constă în faptul că adaugă funcționalități containerelor existente, iar programatorul poate alege containerul de bază pe care să-l adapteze; altfel spus, containerele adaptor pot fi implementate doar folosind anumite containere secvențiale:

- stack poate adapta *vector*, *list*, *deque* (implicit).
- queue poate adapta doar *list* și *deque* (implicit).
- priority\_queue poate fi implementată ca *vector* (implicit) și *deque*.

*Iteratorii* sunt un fel de pointeri către un element din containerele propriu-zise (secvențiale și asociative) care păstrează informația de stare a

containerului, printre care și poziția curentă; sunt deci implementați diferit în funcție de containerul pointat, dar au un comportament general, permitând pre/post incrementare și decrementare, comparații, salt peste un număr de elemente etc.

**Algoritmii.** Unele funcții primare, specifice fiecărui container, au fost incluse în clasa respectivului container, dar altele au putut fi descrise independent de container. Algoritmii STL se referă la cca 70 de operații standard ce se pot efectua pe elementele unui container (insert, erase, find, sort, size, swap etc.) și care pot fi descrise unitar, independent de containerul la care se referă; independența față de container (generalizarea) s-a obținut prin accesarea indirectă a unui element, folosind iteratorii. Regăsim aici ideea că pointerii în C (aici iteratorii) sunt elementele esențiale în realizarea abstractizărilor și generalizărilor.

Deși multe structuri implementează aceleași operații, *nu o fac la fel de eficient*; alegerea structurii adecvate aplicației ține de abilitatea programatorului și de înțelegerea corectă a implementării și funcționalității unei structuri.

Containerele pot stoca orice tip de date (inclusiv create de utilizator), dar programatorul trebuie să se asigure că elementele stocate suportă setul de funcții specifice tipului containerului ales. Unele operații se bazează pe *operatorii definiți la nivel de element* (de obicei comparații, intrări / ieșiri), astfel încât programatorul trebuie să se asigure de existența supraincărcărilor respective. Inserarea într-un container presupune copierea unui element, *constructorul de copiere* fiind indispensabil.

## 8.2 Containere secvențiale

### Vectorul ca şablon

Cea mai cunoscută și utilizată structură de date, masivul unidimensional, este regăsită în STL sub numele de template *vector*.

Se consideră un exemplu banal de construire a unui vector, cu elemente de tip întreg, care ulterior vor fi afișate.

```
#include <iostream.h>
#include <vector>
```

```
using namespace std;
void main()
{
    vector<int> v;
    v.push_back(10); v.push_back(15); v.push_back(98);
    cout<<"\n Afisare elemente!"<<endl;
    for(int i=0;i<v.size();i++) cout<<v[i]<<endl;
}
```

Se observă că:

- s-a inclus fișierul *vector* fără nici o extensie deoarece biblioteca standard de şabloane este descrisă în fișiere header fără extensie;
- s-a folosit construcția *using namespace std*; pentru a preciza că se lucrează cu şabloane din domeniul *std*;
- metoda *push\_back* adaugă un element la vector;
- metoda *size* returnează numărul efectiv de elemente ale vectorului;
- accesarea unui anumit element din vector s-a făcut prin utilizarea operatorului de indexare *[]*, supraîncărcat în clasa *vector*.

În general, operațiile definite pe obiectele de tip colecție (container) presupun traversarea adică accesarea fiecărui element ce compune colecția. Pentru vector acest lucru se face simplu, utilizându-se operatorul de indexare *[]*, specializat în acest sens. Tot pentru accesarea unui anumit element din vector, clasa furnizează metoda *at*, care face în plus și validarea încadrării în domeniu:

```
v.at(0)=87;
cout<<"\n Primul element este:"<<v[0];
```

O modalitate unitară de traversare a elementelor aparținând unui container, definit în STL, o constituie folosirea *iteratorilor*. Pentru exemplificare se va scrie secvența de program care afișează elementele unui vector de tip întreg, folosind un iterator.

```
vector<int> v;
v.push_back(10); v.push_back(15); v.push_back(98);
vector<int>::iterator it;
for(it=v.begin(); it!=v.end(); it++) cout<<*it<<endl;
```

Se observă că:

- s-a definit un obiect iterator inclus într-o clasă *vector* cu elemente de tip întreg:

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

- metoda *begin* a clasei *vector* returnează un iterator care referă primul element din colecție;
- metoda *end* indică locația ulterioară ultimului element din colecție (similar cu EOF pentru fișiere);
- operatorul *++*, supraîncărcat în clasa *iterator*, determină accesarea următorului element din colecție;
- operatorul *\**, supraîncărcat în clasa *iterator*, s-a folosit pentru obținerea elementului curent din colecție, adică cel referit de iterator.

Cu alte cuvinte, s-au accesat rând pe rând elementele colecției, de la început (*begin*) până la sfârșit (*end*).

Vectorul folosește spațiu contiguu de memorie dinamică, astfel încât permite adresarea unui element prin operatorul [ ], calculând deplasarea față de început; la epuizarea spațiului alocat, vectorul se reallocă automat într-un alt bloc mai mare, unic, contiguu, eliberând zona veche. Acest lucru presupune: invocarea alocatorului specific clasei *vector*; apelul repetat al constructorului de copiere pentru mutarea fiecărui obiect în noul bloc de memorie; apelul destructorului pentru fiecare obiect din blocul vechi și eliberarea memoriei containerului.

Ideal este ca un vector să fie alocat o singură dată, atunci când putem estimă capacitatea maximă la care va fi încărcat.

Puteam controla dimensiunea alocării la declararea vectorului, prin constructorul de clasă sau ulterior, prin funcția membră *reserve*.

Din modul de stocare se deduce ușor că structura *vector* este eficientă la inserările / extragerile cu *push\_back()*, respectiv *pop\_back()*. La astfel de operații nu e nevoie de deplasarea elementelor pentru a face loc obiectului inserat, respectiv pentru ocuparea locului eliberat prin extragerea unui element. Este de asemenea eficientă adresarea directă a unui obiect din container, deoarece ea presupune incrementarea adresei de început a blocului cu indexul (poziția) elementului în vector.

Sunt ineficiente inserările la început sau în interiorul vectorului, pentru că presupun deplasări ale elementelor, pentru a face loc nouului element.

Ineficiente sunt din același motiv legat de alocare, stergerile la începutul sau în interiorul vectorului. Secvența următoare realizează stergerea elementului din mijlocul vectorului:

```
it = v.begin() + v.size() / 2;
v.erase(it);
```

Metoda *erase* șterge elementul referit de iteratorul *it*.

Modul contiguu de stocare ne mai conduce la o concluzie destul de importantă: existența iteratorilor invalizi; prin reallocarea vectorului, fie forțată prin program (*v.resize(v.capacity() + 1)*; unde metoda *capacity* returnează capacitatea vectorului în elemente), fie implicit, cerută de inserări repetate ce conduc la depășirii ale capacitatii curente, iteratorii de container nu mai pot fi corect folosiți; unii vor pointa în urma reallocării alte obiecte, alții vor deveni chiar invalizi, pointând zone ce nu mai aparțin acum programului. Acest lucru ne obligă ca înainte de refolosire, iteratorii să fie totdeauna reîncărați.

Tot invalid devine un iterator și în urma stergerii obiectului pointat (*v.erase(it);*).

### Şablon pentru containerul secvențial *list*

Clasa template *list* este implementată ca *listă dublu înlățuită*, permisând prin pointerii *previous* și *next* traversarea în ambele sensuri. Ocupă memorie necontiguă, sarcina localizării unui element fiind preluată de iteratori, care folosesc cele două tipuri de înlățuiriri dintre elemente.

Modul de implementare ne sugerează că *list* se folosește pentru stocarea de obiecte mari, cu inserări și stergeri multiple, mai ales în interiorul listei, pentru că ele presupun doar refacerea de legături, nu și mutarea obiectelor în memorie. Chiar și funcțiile de *sort* și *reverse*, membre ale clasei *list*, lucrează tot cu pointeri, nu cu obiectele stocate; se recomandă folosirea lor și nu a alternativelor lor generice, definite unitar pentru toate containerele. Modul de stocare necontiguu, care nu obligă la reallocări ale obiectelor, ne conduce la concluzia că un iterator odată încărcat cu adresa unui obiect din container, nu poate deveni invalid, decât dacă i-a fost sters nodul pe care pointa.

Traversarea se face mai lent decât la *vector*, deoarece presupune deplasarea prin pointeri, din aproape în aproape, nu calcul de adresă; *list* ocupă și memorie suplimentară, pentru gestiunea legăturilor înainte și înapoi.

Pentru a exemplifica mai bine modul unitar de exploatare a structurilor de date prin intermediul iteratorilor în lucru cu obiecte de tip colecție din STL, se va construi o listă cu elemente de tip *double* care apoi va fi traversată, în vederea afișării elementelor ei. Pentru a lucra cu şablonul listei trebuie să se includă fișierul *list*.

Inserarea în listă se poate face atât în capul listei, cât și în coada ei. Pentru inserarea unui element la începutul listei se poate utiliza metoda *push\_front*, iar pentru adăugarea unui element la sfârșitul listei se utilizează metoda *push\_back*.

```
list<double> l;
l.push_back(45.67);
l.push_front(154.9);
l.push_back(5.1);
```

În vederea inserării de elemente în listă există și metoda *insert*, care poate fi folosită atât pentru a insera elemente în capul listei, cât și în coada ei:

```
l.insert(l.begin(), 78.3);
l.insert(l.end(), 178.3);
```

Se observă că indicarea locului unde se va realiza inserarea s-a făcut prin intermediul iteratorilor; apelul *l.begin()* returnează un iterator care indică poziția de început a listei, iar *l.end()* returnează un iterator care marchează sfârșitul colecției. Metoda *insert* este definită în mai multe forme, astfel încât poate insera chiar mai multe elemente la un singur apel:

```
double vr[] = {56.7, 12.3};
l.insert(l.begin(), vr, vr+2);
```

Secvența determină inserarea în capul listei a două elemente de tip *double*, preluate dintr-un vector clasic, de la adresa dată de *vr*, până la *vr+2*.

Pentru a traversa lista nu se mai poate folosi operatorul de indexare *[]*, deoarece acesta nu este definit în clasa *list*. Deci, iteratorul devine indispensabil în vederea traversării listei. Secvența următoare realizează afișarea elementelor listei:

```
list<double>::iterator it;
for(it=l.begin(); it!=l.end(); it++) cout << *it << endl;
```

### Sablonul *deque*

Sablonul *deque*, **double ended queue – lista cu două capete**, combină facilitățile de la vector și listă într-o singură clasă. Ocupă memorie necontiguă, împărțită pe blocuri mari contigui, alocate pe măsura extinderii containerului și gestionate ușor prin vectori de pointeri. Această implementare permite toate operațiile de bază de la vector și în plus, *push\_front* și *pop\_front*.

Structura *deque* este eficientă pentru inserări la început și sfârșit, în timp ce listă este eficientă și pentru inserări în interiorul listei. Implementarea permite întreținerea facilă a unei discipline FIFO, precum și adresarea unui element prin index, ca la vector. Adresarea vectorială presupune trecerea printr-o fază preliminară, de identificare a blocului ce conține obiectul căutat; în rest, lucrurile stau ca la vector.

Structura *deque* nu este recomandată pentru inserarea și stergerea în / din interiorul containerului; aceste operații sunt optimizate doar în ideea minimizării numărului de elemente copiate pentru a păstra iluzia că elementele sunt stocate contiguu.

Programul de mai jos umple o structură *deque* cu primele zece numere Fibonacci, folosind un algoritm de generare ce primește tipul inserter-ului, numărul de elemente de generat și funcția generator. Se alocă apoi un vector de dimensiune adevarată contextului și se copiază elementele din *deque*, folosind adresare vectorială pentru ambele structuri.

```
#include <iostream>
#include <deque>
#include <vector>
#include <algorithm>
using namespace std;
int Fibonacci(void)
{
    static int r, f1 = 0, f2 = 1;
    r = f1 + f2; f1 = f2; f2 = r;
    return f1;
}
void main()
{
    int n = 10;
    deque<int> d; generate_n( back_inserter(d), n, Fibonacci );
    vector<int> v; v.reserve(d.size());
    deque<int>::iterator pd;
    for(pd = d.begin(); pd != d.end(); pd++) cout << *pd << " ";
    cout << endl;
    for(int i=0; i<n; i++) { v[i]=d[i]; cout << v[i] << " "; }
}
```

Structura *deque* nu poate conține iteratori invalizi, pentru că nu renunță la memoria odată alocată; chiar la inserări în interiorul unui bloc ocupat în întregime, se alocă un bloc suplimentar, dar toți iteratorii deja încărcați pointează pe zone de memorie ce aparțin încă programului.

Structura *deque* este de preferat structurii vector, atunci când nu se poate estimă aprioric numărul obiectelor stocate, sau el variază foarte mult de la o rulare la alta. Pentru programe de înaltă performanță, cu volum mare de date și cu prelucrări multiple, se recomandă chiar folosirea structurii *deque* în faza de construire a containerului, după care containerul este copiat în altul, de tip vector, pentru facilitarea prelucrărilor.

### 8.3 Containerele asociative

#### Containerele asociative *set*, *multiset*

Se suprapun peste conceptul de mulțime, permitând stocarea și regăsirea rapidă a unor elemente, ce pot fi considerate în același timp și chei. Ordinea elementelor este controlată de un obiect specializat, care lucrează ca un comparator și care are în acest sens supraincărcat operatorul funcție, *operator()*. Acest *obiect funcție* se poate transmite ca parametru constructorilor de containere *template*, care au nevoie de implementarea unor discipline de stocare; odată ajuns într-o funcție, un obiect funcție poate fi folosit pe post de funcție, pentru că el se transformă automat în funcție.

Containerele asociative sunt implementate ușual prin arbori binari de căutare, timpul mediu de căutare fiind minim pentru arborii echilibrați. Singura deosebire între *set* și *multiset* este că *multiset* permite și valori duplicate.

Desigur elementele stocate într-o mulțime pot fi de tip fundamental, pointeri, tipuri create de programator sau pointeri către aceste tipuri. Pentru aprofundarea containerului *set* vă propunem următorul program:

```
#pragma warning(disable:4786)
// dezactiveaza avertismentul de prescurtare identificatori
#include <set>
#include <iostream>
using namespace std;
class persoana
{
public:
    int virsta;
    char nume[50];
```

```
persoana(int v=0, char * n="Anonim") : virsta(v) { strcpy(nume,n); }
bool operator<(const persoana p2)
    const { return virsta < p2.virsta ? 1:0; }
};

typedef set<persoana > MULTIME_PERS;

void main()
{
    persoana p[10]={
        persoana(45,"Mateescu Dan"),
        persoana(25,"Ionescu Tudor")
    };
    MULTIME_PERS echipa_1;
    set<persoana, greater<persoana>, allocator<persoana> > echipa_2;
    echipa_1.insert(p[1]);    echipa_1.insert(p[6]);
    cout << "\n" << p[0].nume << " este " <<
        (echipa_1.key_comp()( p[0], p[1]) ?
         " mai tinar(a) ":" mai batrin(a) ")
        << " decat " << p[1].nume << endl;

    cout << "\n" << p[0].nume << " este " <<
        (echipa_1.value_comp()( p[0], p[1]) ?
         " mai tinar(a) ":" mai batrin(a) ")
        << " decat " << p[1].nume << endl;

    MULTIME_PERS::iterator iter_pers;
    p[5].virsta=25;
    iter_pers = echipa_1.find(p[5]);
    //    iter_pers = echipa_1.find(p[0]);
    iter_pers!= echipa_1.end() ?
        cout << "\n Gasit " << iter_pers->.nume : cout << "\n Negasit ";
}
```

Definirea containerului *echipa\_1*, de tip *MULTIME\_PERS*, adică *set<persoana >* dacă urmărim *typedef*-ul, trebuie citită în clar:

```
set<persoana, less<persoana>, allocator<persoana> > echipa_1;
adică o mulțime de persoane, stocate într-un arbore binar de căutare, cu regăsire rapidă bazată pe comparații de tip <, fiecare nod având ca informație utilă un obiect persoana. Așadar containerele ce țin și regăsesc elemente sortate, înglobează în structură atât obiectul alocator specific elementelor stocate, cât și un obiect comparator, care prin supraincărcarea operatorului funcție propriu, redirecțează comparația către operatori relationali specifici obiectului stocat.
```

Ultimii doi parametri sunt și implicați, dar din valorile implicate care le iau, deducem că tipul *persoana* trebuie să fie înzestrat cu *operator<()*, care să introducă o relație de ordine totală pe mulțimea persoanelor; se observă că supraîncărcarea operatorului lucrează cu ambele obiecte *const*, aşa cum cere definirea obiectului funcție *less<T>*. Am introdus astăzi un comparator bazat pe vârstă persoanei.

Accesul la obiectul comparator se asigură prin două funcții membre *key\_comp()*(*p[0], p[1]*) și *value\_comp()*(*p[0], p[1]*), cu același efect în cazul şablonului *set*, confirmându-ne că acesta este un container asociativ, valoarea și cheia fiind identice.

Rezultatul afișat prin program ne confirmă că *Mateescu Dan* este mai bătrân(a) decât *Ionescu Tudor*. Compararea se face implicit pe datorită obiectului funcție *less<T>*, considerat implicit la definirea containerului *echipa\_1*. Puteam cita și alt obiect comparator, spre exemplu *greater<T>*, cu condiția să punem în clasa *persoana* *operator>()*; definirea mulțimii ar fi fost atunci de forma:

```
set< persoana, greater<persoana>, allocator<persoana> > echipa_2;
```

În ce privește regăsirea, observăm că un container asociativ dispune de o funcție membră de căutare *find()*, care primește un obiect *gol*, care conține doar cheia (vârstă în cazul nostru) și încarcă un iterator cu adresa obiectului găsit în container sau cu *echipa\_1.end()*, obiectul sfârșit de container, când n-a găsit obiectul cu cheia căutată. În acest sens, observăm că deși obiectul *p[5]* nu se află în container, modificându-i vârstă să coincida cu a lui *Ionescu*, iteratorul localizează corect persoana cu numele *Ionescu*. La o căutare de tipul:

```
iter_pers = echipa_1.find( p[0] );
```

ni se va răspunde cu *Negasit*, deoarece *p[0]* nu-a fost încă inserat în container. Atunci cum am putut compara în prima parte a programului cele două obiecte *p[0]* și *p[1]*, când unul nu se află în container? Secretul îl reprezintă faptul că noi am extras un pointer la funcția comparator a containerului, pe care o putem apoi folosi la orice comparații de obiecte *persoana*, inserate sau nu în container.

La o căutare de tipul *iter\_pers = echipa\_1.find(p[7])*; ni se răspunde cu *Gasit Anonim*, deși noi nu am inserat *p[7]*; răspunsul e simplu: am inserat *p[6]*, care prin constructorul cu valori implicate al clasei *persoana*, a primit aceeași caracteristici ca *p[7]*, deci și aceeași vîrstă. Rezolvarea problemei ține de posibilitatea gestionării obiectelor cu chei neunice, adică

legerea şablonului *multiset* în loc de *set*, deoarece probabilitatea ca două persoane să aibă aceeași vârstă este foarte mare.

Vom modifica programul anterior, schimbând *set* cu *multiset*; persoanele cu aceeași vârstă vor fi stocate una după alta în arbore; în multe aplicații apare problema localizării rapide a tuturor obiectelor cu aceeași cheie. Programul de mai jos face acest lucru în două moduri.

**0** O modalitate folosește clasa *pair<T,T>* din domeniul *std*, ce conține doi iteratori de același tip *T* de container; perechea este încărcată printr-un apel *equal\_range(p[1])*, primul iterator referind primul obiect ce satisfacă condiția de căutare (vârstă egală cu a lui *p[1]*), iar al doilea pe primul obiect, în ordine, ce nu mai satisfac această condiție.

```
#pragma warning(disable:4786)
```

```
#include <set>
#include <iostream>
using namespace std ;
class persoana
{
public:
    int vîrstă;
    char nume[50];
    persoana(int v=0, char * n="Anonim") : vîrstă(v) { strcpy(nume,n); }
    bool operator<(const persoana p2)
        const { return vîrstă < p2.vîrstă ? 1:0; }
};

typedef multiset<persoana > MULTIME_PERS;

void main()
{
    persoana p[10]={ persoana(45,"Mateescu Dan"),
                    persoana(25,"Ionescu Tudor"),
                    persoana(15,"Adam Doru"), persoana(25,"Brad Alina") };
    p[8]=persoana(25, "Oprescu Dana");
    MULTIME_PERS echipa_1;
    for(int i=0; i<10; i++) echipa_1.insert(p[i]);
    MULTIME_PERS::iterator it;
    std::pair<MULTIME_PERS::iterator,MULTIME_PERS::iterator> leat;
    cout << "\n Persoane de aceeași vîrstă cu p[1]: ";
    leat = echipa_1.equal_range(p[1]);
    for( it = leat.first; it != leat.second; it++)
        cout << "\n " << it->nume << "\t" << it->vîrstă;
    cout << endl << echipa_1.lower_bound(p[1]) -> nume ;
```

```
// primul obiect ce satisfac criteriul de căutare
cout << endl << echipa_1.upper_bound(p[1]) -> nume ;
// primul obiect ce nu mai satisfac criteriul de căutare
}
```

Secvența:

```
for( it=leat.first; it != leat.second; it++)
    cout << "\n" << it->nume << "\t" << it->vîrstă;
```

cu *egal* pentru primul iterator și *not egal* pentru cel de-al doilea, confirmă prin rezultatele ei selectarea corectă a persoanelor de 25 ani:

*Persoane de aceeași vîrstă cu p[1]:*

```
Ionescu Tudor 25
Brad Alina 25
Oprescu Dana 25
```

② Același lucru îl putem realiza cu două apeluri: *echipa\_1.lower\_bound(p[1])* și *echipa\_1.upper\_bound(p[1])*, pentru a obține cei doi iteratori care încadrează persoanele căutate; cele două margini, inferioară și superioară, afișate vor fi:

```
Ionescu Tudor
Mateescu Dan
```

Deoarece în lucru cu un container *set* nu se fac reallocări de noduri, un iterator nu poate deveni invalid decât în urma unei operațiuni de ștergere nod, *erase()*.

### Containerele asociative *map*, *multimap*

Sunt implementate ușual tot prin arbori binari de căutare, un nod conținând perechea cheie-valoare. *multimap* permite și regăsirea după chei duplicate, implementând o *relație unu la mulți*, spre deosebire de *map* care gestionează relații *unu la uno*.

*map* se mai numește și *vector asociativ*, deoarece poate folosi cheia ca pe un index de poziție în vector, sub forma *v[cheie]=valoare* sau *x=v[cheie];* ca și la vector, *operator[]* a fost supraîncărcat să returneze referință de valoare asociată; în schimb, când cheia furnizată ca parametru în funcția *operator[]* nu există în container, se consideră că se dorește inserarea unei noi asocieri cheie-valoare.

Un astfel de container este definit având două tipuri generice, unul pentru cheie și altul aferent valorii asociate. De exemplu, dacă presupunem că vrem să memorăm o agendă de telefon construind perechi de valori: numele și prenumele persoanei și numărul de telefon, declarația unui obiect container tip *map* va fi:

```
map<string, int> agenda;
```

unde, *string* este o clasă definită în biblioteca standard C++ și modelează lucru cu siruri de caractere.

Pentru a lucra unitar cu un element ce aparține unui container tip *map* sau *multimap*, acesta se va defini prin intermediul unei clase pereche (*pair*) în forma:

```
pair<string, int> element;
```

Accesarea cheii din element se face în forma *element.first*, iar a valorii asociate *element.second*, unde *first* și *second* sunt membri în clasa *pair*.

Lucru cu aceste containere implică includerea fișierului *map*.

Programul următor va construi o agendă de telefon după care va căuta numărul de telefon aferent unei persoane; în final se va afișa conținutul întregii agende.

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
typedef pair<string, int> element;
void main()
{
    string nume; char np[50];
    map<string, int> agenda;
    map<string, int>::iterator it;
    // construire agenda
    agenda.insert(element("Furtuna F", 8989890));
    agenda.insert(element("Tibrea I", 8989691));
    agenda.insert(element("Cristescu A", 6192513));
    // căutare în agenda
    while(cout << "\n Nume sau CTRL/Z:", cin.getline(np,50) )
    {
        nume=np;
        it=agenda.find(nume);
        if(it!=agenda.end())
            cout << "\n Persoana " << nume << " are nr. de telefon:" << (*it).second;
        else
            cout << "\n Nu există persoană cu numele " << nume;
    }
}
```

```

        cout<<"\n Persoana "<<nume<<" nu exista in agenda!!!";
    }
// afisare continut agenda
for(it=agenda.begin(); it!=agenda.end(); it++)
    cout<<"\n Persoana "<<(*it).first<<" are telefonul:"<<(*it).second;
}

```

Căutarea în containerul asociativ se poate face atât după valoarea cheii (nume), cât și după valoarea asociată (număr telefon), dar numai după cîmpul cheie căutarea este rapidă.

## 8.4 Containere adaptive

Așa cum menționam, containerele adaptoare nu sunt containere *de prima clasă*, ci le adaptează pe cele existente; ele se bazează deci pe alocatorii de memorie ai acestora, implementând doar diferite discipline de lucru cu containerele pe care se bazează.

### Adaptorul stack

Stiva implementează disciplina LIFO. Din programul de mai jos se poate observa ușor că modalitatea prin care se realizează tehnic acest lucru este o *componere de template*, adică *stack* este un *template* care are la rîndul lui în structură un *template* (*deque*, *vector* sau *list*).

```

#include <iostream>
#include <stack>
#include <vector>
#include <list>

using std::cout;

// declaratie functie de utilizator
template <class T> void scoate(T &stiva);

void main()
{
    std::stack<int> stiva_deq; // stiva ca adaptor implicit de deque
    std::stack<int, std::vector<int> > stiva_vect; // stiva ca adaptor de vector
    std::stack<int, std::list <int> > stiva_list; // stiva ca adaptor de list
}

```

```

for( int i=0; i< 10; i++)
{
    stiva_deq.push(3*i );
    stiva_vect.push(3*i+1);
    stiva_list.push(3*i+2);
}
scoate(stiva_deq); cout <<"\n";
scoate(stiva_vect); cout <<"\n";
scoate(stiva_list); cout <<"\n";
}

template <class T> void scoate(T & stiva)
{
    while(!stiva.empty()) { cout <<stiva.top()<< " "; stiva.pop(); }
}

```

Programul de mai sus lucrează cu trei stive bazate pe *deque*, *vector*, respectiv *list*, care stochează primele 30 de numere naturale, în funcție de congruența lor față de 3.

Lucrând cu mai multe tipuri de şabloane, a fost necesară includerea mai multor fișiere header standard. S-a preferat lucru cu funcțiile din *std::* și pentru intrări / ieșiri, pentru a rămâne în domeniu; altfel ar trebui inclus *<iostream.h>* și eliminată declarația *using std::cout*.

Elementul cheie al programului este declararea celor trei tipuri de stivă. Spre exemplu, *std::stack< int, std::list <int> > stiva\_vect*; definește variabila *stiva\_vect* care este o stivă bazată pe listă de întregi; se observă cum tipul *stack* este instanțiat, prin cel de-al doilea parametru, pentru un alocator de tip *list*, care este la rîndul lui instanțiat pentru tipul *int*.

Deși funcțiile de lucru cu stiva sunt supraîncărcate pentru toate cele trei tipuri de containere pe care le adaptează, pentru funcția *pop()* s-a preferat includerea ei într-o funcție *scoate()*, care face și afișarea elementelor, pe măsura scoaterii lor din stivă. Se observă astfel și cum se definesc de către utilizator funcții care lucrează cu şablonul de stivă.

### Adaptorul queue

Adaptorul *queue* introduce disciplina FIFO pe *list* sau *deque*, restricționând accesul astfel încât inserările să se facă doar la coada listei (funcția *push()* de la *queue* se expandează *inline* într-un apel *push\_back* al containerului bază), iar extragerile să se facă doar pe la capul listei (funcția

*pop()* de la *queue* se expandează *inline* într-un apel *pop\_front()* al containerului bază).

```
#include <iostream>
#include <list>
#include <queue>
using std::cout;
void main()
{
    std::queue<int, std::list<int>> coada_list;
    // coada ca adaptor de list
    std::queue<float, std::deque<float>> coada_deque;
    // coada ca adaptor pentru deque
    // echivalent cu std::queue<float> coada_deque;
    coada_list.push(10);
    coada_deque.push(1.2);
    coada_deque.push(3.4);
    coada_deque.push(5.6);
    while(!coada_deque.empty())
    {
        cout << coada_deque.front() << " "; coada_deque.pop();
    }
}
```

*std::queue<int, std::list<int>> coada\_list;* declară o coadă implementată prin listă dublu înlănțuită, iar

*std::queue<float, std::deque<float>> coada\_deque;* echivalent cu *std::queue<float> coada\_deque;* o coadă bazată explicit, respectiv implicit, pe listă simplu înlănțuită, cu două capete.

### Adaptorul *priority\_queue*

Coada cu priorități se implementează prin *vector* (implicit) sau prin *deque*. Constructorul are în descriere tipul stocat, containerul de implementare și un obiect funcție, care lucrează drept comparator; ultimii doi parametri au și valori implicate, *std::vector<int>* pentru containerul adaptat, respectiv *std::less<int>* pentru comparator.

Obiectele funcție sunt structuri parametrizate prin *template*, derivate din clasa *binary\_function*, ce conțin o supraîncărcare a operatorului funcție, încât să redirecțeze comparația spre operatorii relationali definiți în clasa tipului cu care lucrează. Spre exemplu, definirea comparatorului *std::less<T>* în fișierul header *<functional>* este următoarea:

```
template<class _Ty>
struct less : binary_function<_Ty, _Ty, bool>
{
    bool operator()(const _Ty& _X, const _Ty& _Y) const
    {return (_X < _Y); }
```

Programul de mai jos definește cinci cozi cu priorități:

- *coada\_pri\_vect*, folosind alocatorul de container implicit *vector* și comparatorul implicit *less*;
- *pri\_vect\_crescator* și *pri\_vect\_descrescator* – cu alocatori explicit de tip *vector*, dar cu comparatori diferenți (priorități inverse);
- *pri\_vect\_descrescator* și *pri\_deque\_crescator* – cu alocatori explicit de tip *deque* și cu comparatori *less*, respectiv *greater*.

```
#include <iostream>
#include <functional>
#include <list>
#include <queue>

using std::cout;
void main()
{
    std::priority_queue<float> coada_pri_vect;
    // priority_queue ca adaptor implicit de vect
    std::priority_queue< int, std::vector<int>, std::greater<int> >
        pri_vect_crescator;
    std::priority_queue<int, std::vector<int>, std::less<int> >
        pri_vect_descrescator;
    std::priority_queue< double, std::deque<double>, std::greater<double> >
        pri_deque_crescator;
    std::priority_queue<int, std::deque<int>, std::less<int> >
        pri_deque_descrescator;
    cout << "\n Vector: \n";
    pri_vect_crescator.push(10);
    pri_vect_crescator.push(30);
    pri_vect_crescator.push(20);
    while(!pri_vect_crescator.empty())
    {
        cout << pri_vect_crescator.top() << " "; pri_vect_crescator.pop();
    }
    pri_vect_descrescator.push(10);
    pri_vect_descrescator.push(30);
    pri_vect_descrescator.push(20);
    while(!pri_vect_descrescator.empty())
    {
        cout << pri_vect_descrescator.top() << " "; pri_vect_descrescator.pop();
    }
}
```

```

cout << "\n Deque: \n";
pri_deque_crescator.push(10);
pri_deque_crescator.push(30);
pri_deque_crescator.push(20);
cout << "\n coada cu prioritati are lungimea " <<
      pri_deque_crescator.size() << "\n";
while(!pri_deque_crescator.empty())
{ cout << pri_deque_crescator.top() << " "; pri_deque_crescator.pop(); }
pri_deque_descrescator.push(10);
pri_deque_descrescator.push(30);
pri_deque_descrescator.push(20);
while(!pri_deque_descrescator.empty())
{
    cout << pri_deque_descrescator.top() << " ";
    pri_deque_descrescator.pop();
}

```

Indiferent de ordinea inserărilor, coada cu priorități păstrează elementele sortate conform comparatorului asociat, lucru constatat la afișarea rezultatelor:

*Vector:*

10 20 30 30 20 10

*Deque:*

coada cu prioritati are lungimea 3

10 20 30 30 20 10

Toate cele trei adaptoare de container au funcțiile membre de bază definite *inline*, astfel încât să se evite apelul în cascadă a două funcții, cea din adaptor, care la rândul ei să apeleze pe cea din containerul de bază; primul apel va fi aşadar totdeauna înlocuit cu codul generat de apel.

## 8.5 Iteratori

Avantajele utilizării iteratorilor pot fi sintetizate astfel:

- introduc un nivel de indirectare, permitând generalizări și abstractizări;
- permit manipularea unitară a unui grup de obiecte dintr-un container;

- facilitează scrierea de cod independent de structura container aleasă, astfel încât schimbarea tipului containerului să nu oblige la rescrierea programului.

Pentru o mai bună înțelegere a conceptului de *iterator*, vom construi un exemplu de şablon de clasă container, pentru care vom defini și un iterator. În acest sens, se va defini şablonul clasei fişier, tipul articolului fiind generic, iar accesul secvențial. Fişierul este *văzut* ca un container de articole. S-a observat din exemplele anterioare, când am folosit şabloanele claselor listă și vector că iteratorii se definesc în concordanță cu clasa container folosită (*vector<int>::iterator it*; sau *list<double>::iterator it*) și au fost utilizati la traversarea stării de date.

Această modalitate unitară de parcurs a elementelor unui container ține de conceptul de programare generică. Ne putem aminti că în biblioteca standard C, conceptul de programare generică era implementat cu ajutorul pointerilor generici (*void \**) și prin furnizarea unor secvențe de cod la momentul apelului cu ajutorul pointerilor la funcții. De exemplu, funcția de sortare a elementelor unui vector, numită *qsort*, are prototipul:

*void qsort(void \*, size\_t, size\_t, int (\_cdecl \*)(const void \*, const void \*));*

Funcția primește ca parametri:

- vectorul de sortat, pentru a nu depinde de tipul elementelor, s-a transmis prin intermediul unui pointer la void;
- numărul de elemente ale vectorului;
- marimea unui element în baiți;
- un pointer la o funcție care precizează sensul sortării (ascendent / descendent).

Revenind la problema noastră, putem observa că obiectul iterator joacă rol de pointer generic, deoarece prin intermediul lui se referă un element din colecție; se justifică această afirmație și prin accea că în clasa iterator se suprăîncarcă operatorii specifici aritmeticiei de pointeri cum ar fi *\** pentru obținerea elementului referit de iterator și *++* pentru incrementare (cu sensul de trecere la următorul element din colecție). Se poate deduce de aici necesitatea definirii clasei iterator în funcție de structura de date pentru care se aplică; trecerea la următorul element se realizează în mod diferit, la vector față de listă.

Pentru a construi şablonul clasei fişier căt mai fidel implementării structurilor de date în STL, se vor defini metodele:

- *push\_back* pentru a adăuga un articol în fișier;

- *begin* pentru a returna un iterator care să refere primul articol din fișier;
- *end* pentru returnarea unui iterator care să localizeze sfârșitul de fișier;

Tot în şablonul clasei *fișier* se va defini şablonul clasei iterator (clasa inclusă), care va conține metode pentru supraîncarcarea operatorilor:

- \* pentru returnarea articolului referit de iterator;
- ++ pentru trecerea la următorul articol;
- != pentru compararea a doi iteratori, folosit în construcția *cât timp nu e sfârșit de fișier*.

Şablonul clasei *fișier*, care include și şablonul clasei iterator, este:

```
#include <stdio.h>
#include <iostream.h>
#include <process.h>

#define SCRIERE 0
#define CITIRE 1

template <typename T>
class fisier
{
    FILE *pf;
    int md;
public:
    class iterator
    {
        friend fisier;
        FILE *p_f;
        int m,eof;
        T art;
    public:
        iterator(FILE *p_fis,int k):p_f(p_fis),m(k) { }
        iterator(int p=0):eof(p) { }
        T operator*() { return art; }
        void operator++(int) { eof=fread(&art,sizeof(T),1,p_f); }
        int operator!=(iterator& i) { return eof!=i.eof; }
    };
    fisier(char *,int=SCRIERE);
    void push_back(T a) { if(md==SCRIERE) fwrite(&a,sizeof(a),1,pf); }
    iterator begin();
    iterator end() { return iterator(); }
    ~fisier() { fclose(pf); }
};
```

```
template <typename T>
fisier<T>::fisier(char *s,int mod):md(mod)
{
    switch(md)
    {
        case SCRIERE: pf=fopen(s,"wb"); break;
        case CITIRE: pf=fopen(s,"rb"); break;
        default: cout<<"\n Mod NECUNOSCUT de deschidere !!!!";
    }
    if(!pf) { cout<<"\n Fisierul nu s-a deschis!!!!"; exit(1); }
}
```

```
template <typename T>
fisier<T>::iterator fisier<T>::begin()
{
    rewind(pf);
    iterator tmp(pf,md);
    tmp.eof=fread(&tmp.art,sizeof(T),1,pf);
    return tmp;
}
```

Pentru a înțelege mai bine codul sursă vom face urmatoarele precizări:

- constructorul clasei *fișier* are ca scop deschiderea fișierului în modul indicat;
- destructorul realizează închiderea fișierului;
- clasa *iterator* este inclusă în clasa *fisier* (iterator de *fisier*) și are doi constructori:
  - unul pentru un iterator operațional, care face legătura cu clasa *fișier* pentru care a fost definit;
  - altul pentru un iterator deocamdată invalid, inițializat aici cu sfârșitul de fișier;

Într-un program principal s-a folosit acest şablon pentru a construi un fișier ce conține articole de tip întreg (*int*), iar apoi articolele au fost afișate folosindu-se un iterator.

```
void main()
{
    fisier<int> fi("fis_i.dat");
    fi.push_back(10); fi.push_back(50); fi.push_back(31);
}

fisier<int> fi("fis_i.dat",CITIRE);
```

```

fisier<int>::iterator it;
for(it=fi.begin(); it!=fi.end(); it++) cout<<*it<<endl;
}

```

Se poate observa maniera de exploatare a fișierului care este similară cu maniera de folosire a vectorului sau listei din STL.

Pentru clasele definite în STL ce modeleză structuri de date, iteratorii au capabilități diferite în funcție de operațiile permise pe respectiva structură de date. În continuare prezentăm câteva caracteristici ale iteratorilor, precum și modalitățile de utilizare.

În diferiți algoritmi este necesară parcurgerea elementelor colecției atât într-un sens cât și în celălalt, adică iteratorul să fie *bidirectional*. Acest lucru este posibil prin supraîncărcarea și a operatorului de decrementare (--) în clasa *iterator*.

Ca exemplu, vom construi o listă (*l2*) având elementele listei *l1*, dar în ordine inversă, după care vom afișa elementele listei *l2*.

```

list<double> l1, l2(30);
l1.push_back(45.67); l1.push_back(105.19);
l1.push_back(83.32); l1.push_back(67.67);
list<double>::iterator itl1,itl2;
for(itl1=l1.begin(),itl2=l2.end(),itl2--; itl1!=l1.end(); ) *itl2-- = *itl1++;
for(itl2++; itl2!=l2.end(); itl2++) cout<<*itl2<<" ";

```

Pentru accesarea unui anumit element din colecție, la un iterator se poate aduna sau scădea un întreg, rezultând tot un iterator, astfel:

*it+n* → va indica al *n*-lea element după cel referit de *it*;

*it-n* → va indica al *n*-lea element înaintea celui referit prin *it*;

unde:

*it* – reprezintă un iterator;

*n* – reprezintă un întreg.

Ca exemplu, se vor afișa doar primele *m* elemente ale unui vector:

```

int m;
vector<double> c;
vector<double>::iterator iti, itf;
c.push_back(45.67); c.push_back(105.19);
c.push_back(83.32); c.push_back(67.67);
cout<<"\n Cite elemente afișați???">>m;
if(m<=c.size())
    for(iti=c.begin(),itf=iti+m; iti!=itf; iti++) cout<<*iti<<" ";
else    cout<<"\n Vectorul are mai putine elemente!!!";

```

### Iteratori predefiniți

STL furnizează și iteratori predefiniți. Pentru folosirea lor, trebuie să se includă fișierul header *iterator*.

- ❶ *ostream\_iterator* – este un iterator de ieșire și conectează un flux de ieșire la un iterator; un iterator este numit de ieșire dacă elementul din colecție referit de el se poate doar consulta. Un astfel de obiect iterator nu se mai definește în concordanță cu un şablon container. La definirea unui obiect de tip *ostream\_iterator* trebuie să se precizeze, pentru şablon, tipul elementului din colecția de date, iar ca parametru al constructorului se va furniza fluxul de ieșire.

De exemplu, în declarația:

```
ostream_iterator<int> os_it(cout, " ");
```

se observă că:

- tipul elementelor din colecție este *int*;
- *cout* – definește ca flux de ieșire monitorul;
- " " – este o constantă sir de caractere care se va trimite în flux după afișarea unui element.

O expresie de genul:

```
*os_it=173;
```

determină afișarea pe monitor a valorii 173 urmată de un spațiu, echivalentul construcției:

```
cout<<173<<" ";
```

Ca exemplu, se va scrie un program care construiește un vector cu elemente de tip *float*, afișate apoi utilizându-se un iterator de tip *ostream\_iterator*:

```

#include <iostream>
#include <vector>
#include <iterator>
using namespace std;
void main()
{
    vector<float> v;
    v.push_back(10.9); v.push_back(15.34); v.push_back(11.98);
    ostream_iterator<float> os_it(cout, " ");

```

```

vector<float>::iterator it;
for(it=v.begin();it!=v.end();it++) *os_it=*it;
}

```

Construcția `vector<float>::iterator it;` realizează declararea unui iterator de elemente de vector de `float`; el se utilizează pentru a referi un element al vectorului, acesta urmând a fi afișat prin intermediul iteratorului `os_it` folosind expresia: `*os_it=*it;`

- ② *istream\_iterator* – este un şablon similar cu *ostream\_iterator*, deosebirea esențială fiind că acesta se conectează la un flux de intrare. Este un iterator de intrare, în sensul că permite modificarea elementului referit de el. La definirea unui obiect de tip *istream\_iterator* trebuie să se precizeze, pentru şablon, tipul elementului din colecția de date, iar ca parametru al constructorului se va furniza fluxul de intrare.

Din declarația:

```
istream_iterator<int> is_it(cin);
```

se observă că:

- tipul elementelor din colecție este `int`;
- `cin` – definește ca flux de intrare tastatura;

Apelul acestui constructor determină citirea din flux a unui element, în cazul nostru de la tastatură. Obținerea elementului se face cu ajutorul operatorului `*` care este supraîncărcat în clasa *istream\_iterator* și care returnează elementul citit:

```

int i=*is_it;
cout<<"\n i="<<i;

```

Pentru a mai citi din flux și alte valori, se utilizează operatorul `++`, deja supraîncărcat și el în clasa *istream\_iterator*.

Se va exemplifica utilizarea clasei *istream\_iterator* pentru a citi de la tastatură elementele care se vor insera într-o coadă, apoi acestea se vor afișa folosindu-se un iterator de tipul *ostream\_iterator*.

```

#include <iostream>
#include <queue>
#include <iterator>
using namespace std;
void main()
{
    queue<int> q;
    istream_iterator<int> is_it(cin);
}

```

```

while(*is_it) { q.push(*is_it); is_it++; }
ostream_iterator<float> os_it(cout, "\n");
while(!q.empty()) { *os_it=q.front(); q.pop(); }
}

```

Observații:

- pentru lucru cu coada s-a folosit şablonul clasei *queue*;
- inserarea unui element în coadă s-a făcut apelând metoda *push()*;
- se introduc elemente în coadă cât timp întregul citit de la tastatură este diferit de 0;
- extragerea unui element din coadă s-a făcut în pași:
  - localizarea elementului apelând metoda *front()*;
  - extragerea lui din coadă cu metoda *pop()*;
- metoda *empty* s-a folosit pentru a testa dacă structura de coadă este vidă.

- ③ *reverse\_iterator* – este iteratorul invers, folosit pentru a parcurge elementele unei colecții în ordine inversă, după o sintaxă similară traversării în ordine normală. Acest iterator se definește în legătură cu un şablon de clasă *container*. Pentru a-l folosi trebuie cunoscut că:

- metoda *rbegin()* aplicată unui obiect *container* returnează un iterator invers, care referă locația ulterioară ultimului element din colecție, așa cum se obține apelând metoda *end()*;
- metoda *rend()* returnează un iterator invers care referă locația primului element din colecție, similară din acest punct de vedere cu metoda *begin()*;
- operatorul `++` aplicat unui iterator invers determină în fapt decrementarea iteratorului, adică referirea unui element anterior celui curent.

Ca exemplu, se vor afișa, în ordine inversă, elementele unui vector folosind un iterator invers.

```

#include <iostream>
#include <vector>
#include <iterator>
using namespace std;
void main()
{
    vector<int> v;
    v.push_back(34); v.push_back(67);
    v.push_back(914); v.push_back(13);
}

```

```
vector<int>::reverse_iterator r_it;
for(r_it=v.rbegin(); r_it!=v.rend(); r_it++) cout<<*r_it<< " ";
}
```

- ④ *insert\_iterator* – definește un iterator specializat pentru a realiza inserări de elemente în colecții de date. Acest iterator are două forme specializate:
- *back\_insert\_iterator* – pentru inserare de elemente la sfârșitul colecțiilor de date;
  - *front\_insert\_iterator* – pentru inserare de elemente la începutul colecțiilor de date;

Ca parametru pentru aceste şabloane se va furniza tipul containerului în care se vor inseră elemente. Constructorul clasei *back\_insert\_iterator* va primi ca parametru obiectul container în care se vor inseră elementele:

```
vector<int> v;
back_insert_iterator< vector<int> > ins_it(v);
```

Pentru inserarea propriu-zisă de elemente se va folosi expresia: `*ins_it=45`, dacă se dorește adăugarea elementului 45 la vectorul *v* prin intermediul iteratorului *ins\_it*.

Iteratorul *front\_insert\_iterator* se utilizează în mod similar doar că elementul se va inseră la începutul colecției.

Acești iteratori nu pot fi folosiți pentru orice clasă container ci doar, în concordanță cu containerele care permit realizarea facilă a unor asemenea operații. Spre exemplu, într-o listă inserarea la început sau la sfârșit este o operație uzuale, însă pentru un vector, operația de inserare la început are o complexitate mai mare.

Forma *insert\_iterator* are un caracter mai general și poate fi folosită în orice condiții, pentru orice container. Acest şablon are ca parametru clasa container în care se vor inseră elementele, iar constructorul primește două argumente: obiectul container în care se vor inseră elementele și un iterator al containerului, care indică locul unde se vor inseră elementele:

```
list<int> l;
insert_iterator< list<int> > ins_it(l, l.end());
```

Ca exemplu, se va scrie programul care construiește o listă prin inserare de elemente la sfârșitul ei, utilizând un iterator de inserare, apoi elementele vor fi afișate:

```
#include <iostream>
#include <list>
#include <iterator>
```

```
using namespace std;
void main()
{
    list<int> l;
    insert_iterator< list<int> > ins_it(l,l.end());
    *ins_it=45; *ins_it=745; *ins_it=13; *ins_it=59;
    list<int>::iterator it;
    cout<<"\n Elementele listei sunt:";
    for(it=l.begin(); it!=l.end(); it++) cout<<*it<< " ";
}
```

## 8.6 Algoritmi

La începutul acestui capitol s-a precizat faptul că biblioteca de şabloane (STL) conține și implementări de algoritmi. Până acum algoritmii au fost vizibili prin intermediul metodelor claselor container. Există însă și funcții independente care se folosesc pentru a realiza anumite prelucrări pe mai multe tipuri de containere de date. Folosirea lor implică, de cele mai multe ori, includerea fișierului *algorithm*.

ACESTE FUNCȚII AU O MAXIMĂ GENERALITATE ÎN SENSUL CĂ POT PRELUCRA ELEMENTELE MAI MULTOR CLASE CONTAINER DIN STL. GENERALITATEA ESTE ASIGURATĂ ÎN SPECIAL PRIN TRANSMITEREA ITERATORILOR CA PARAMETRI ÎN ACESTE FUNCȚII. TOT CU AJUTORUL ITERATORILOR SE DEFINEȘTE ȘI UN INTERVAL CARE CONȚINE ELEMENTELE DIN COLECȚIE PENTRU CARE SE APICĂ ALGORITMUL RESPECTIV.

Dintre cele mai importante funcții menționăm:

- ① *copy* – care realizează copierea datelor dintr-un container în altul.

De exemplu, dacă se dorește să se copieze într-un container elementele unui masiv, secvența este:

```
int a[ ]={10,5,78,33};
list<int> l(30);
copy (a, a+sizeof(a)/sizeof(int), l.begin());
```

Se observă că destinația este indicată printr-un iterator al containerului în care se vor copia elementele: *l.begin()*. Trebuie de subliniat de asemenea faptul că operația de copiere este diferită de cea de inserare, deoarece la copierea unui element nu se face și alocarea memoriei necesare lui. După

cum se observă din secvența prezentată, copierea s-a făcut după ce lista a fost declarată ca având 30 de elemente: `list<int> l(30);`

Pentru afișarea pe monitor a elementelor unui container se poate folosi funcția `copy` ca în programul:

```
#include <iostream>
#include <vector>
#include <iterator>

using namespace std;

void main()
{
    vector<int> v;
    v.push_back(45); v.push_back(67); v.push_back(32);
    ostream_iterator<int> os_it(cout, " ");
    copy(v.begin(), v.end(), os_it);
}
```

Primi doi parametri ai funcției `copy` indică câte elemente din containerul vector `v` vor fi transferate la destinație, care în acest caz este un iterator de tip `ostream_iterator` conectat la monitor (`cout`); în cazul nostru au fost transferate toate elementele colecției `v` (`v.begin() → v.end()`).

**2) `for_each`** – este o funcție care prelucrează fiecare element al colecției. Modul în care se face prelucrarea se definește prin construirea unei funcții independente care va fi trimisă ca parametru funcției `for_each`, prin intermediul unui pointer la funcție.

Ca exemplu, se vor afișa elementele unui container de tip `vector` folosindu-se funcția `for_each`. Afișarea propriu-zisă a unui element din container s-a efectuat în funcția independentă `afis`, care primește ca parametru un întreg (elementul din container).

```
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void afis(int k) { cout<<k<<endl; }

void main()
{
    vector<int> v;
    v.push_back(10); v.push_back(15); v.push_back(98);
    for_each(v.begin(),v.end(),afis);
}
```

Se observă că primii doi parametri ai funcției `for_each` sunt iteratorii care specifică câte elemente din container vor fi prelucrate de funcția `afis`; în cazul nostru se trimit spre prelucrare toate elementele (`v.begin() → v.end()`).

**3) `sort`** – definește o funcție care sortează elementele unui container; sensul sortării se precizează printr-un pointer la o funcție ce are ca argumente două variabile de tipul elementelor containerului și returnează un boolean; dacă valoarea returnată este FALSE, atunci elementele se interschimbă.

Ca exemplu, se va scrie programul care sortează crescător elementele unui container `vector` de întregi:

```
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

bool comp(int k1, int k2) { return k1<k2; }

void main()
{
    vector<int> v;
    v.push_back(10); v.push_back(150);
    v.push_back(98); v.push_back(67);
    sort(v.begin(),v.end(),comp);
    vector<int>::iterator it;
    for(it=v.begin(); it!=v.end(); it++) cout<<*it<<" ";
```

Tipurile generice din şabloanele de clase din STL pot fi substituite și cu tipuri abstracte de date, adică tipuri definite prin intermediul claselor.

Vom construi clasa `persoana` care va conține ca date membre numele și prenumele persoanei (`np`) și salariul (`sal`). Se vor defini funcții membre după cum urmează:

- constructor care să încarce date în obiectul persoana, din variabile elementare;
- supraîncărcarea operatorului `<` care returnează adevărat dacă o persoană are salariul mai mic decât o altă persoană;
- supraîncărcarea operatorului `<<` pentru afișarea unei persoane (numele și salariul).

```
class persoana
{
    char np[30];
    int sal;
```

public:

```

persoana(char *sir="Persoana", int s=0):sal(s) { strcpy(np,sir); }
bool operator<(persoana& p) { return sal<p.sal; }
friend ostream& operator<<(ostream& os, persoana& p)
{
    os<<p.np<<" "<<p.sal;
    return os;
}

```

Se va scrie programul care construiește un container de tip *vector* conținând informații despre persoane, el va fi sortat crescător în ordinea salariilor persoanelor, adică elementele vor fi afișate pe display.

```

using namespace std;
bool comp(persoana& k1, persoana& k2) { return k1<k2; }
void main()
{
    vector<persoana> v;
    v.push_back(persoana());
    v.push_back(persoana("Vasile",150));
    v.push_back(persoana("Ionescu",90));
    v.push_back(persoana("Bogdan",67));
    sort(v.begin(),v.end(),comp);
    vector<persoana>::iterator it;
    for(it=v.begin();it!=v.end();it++) cout<<*it<<"\n";
}

```

Este esențial pentru buna funcționare a programului ca în clasa *persoana* să se supraîncarce operatorul < necesar funcției *comp*, ce dă sensul sortării și operatorul <<, în vederea afișării unui obiect de tip *persoana*.

❸ *find* – este o funcție care căută un element într-o colecție și returnează un iterator, care referă acel element (prima lui apariție), în caz că există, altfel un iterator ce marchează sfârșitul colecției (egal cu valoarea *container.end()*).

Ca exemplu, să se scrie programul care afișează pentru un container de tip *vector* acele elemente care sunt mai mari sau egale cu o anumită valoare (prag), doar dacă valoarea dată ca prag există și în colecție.

```

#include <iostream.h>
#include <vector>
#include <algorithm>
using namespace std;

```

```

bool comp(int k1, int k2) { return k1<k2; }
void main()
{
    vector<int> v;
    int k;
    cout<<"\n Elementul de căutat:"; cin>>k;
    v.push_back(10); v.push_back(150);
    v.push_back(98); v.push_back(67);
    sort(v.begin(),v.end(),comp);
    vector<int>::iterator it;
    it=find(v.begin(),v.end(),k);
    if(it==v.end()) cerr<<"\n !!!Elementul <<k<< nu există!";
    else for(; it!=v.end(); it++) cout<<*it<<" ";
}

```

Se observă că:

- vectorul a fost mai întâi sortat crescător pentru a ne asigura că toate elementele ce succed elementul referit de iteratorul returnat de funcția *find()*, sunt mai mari ca el;
- funcția *find* are ca ultim parametru valoarea de căutat (*k*);
- dacă nu s-a găsit în colecția *v* nici un element având valoarea egală cu a elementului de căutat (*k*) se returnează iteratorul ce marchează sfârșitul colecției (egal cu *v.end()*).

Pentru o mai mare generalitate, funcția de căutare a fost implementată în mai multe variante; de reținut varianta *find\_if* care primește un pointer la o funcție independentă în locul valorii de căutat de la funcția *find*. Prin intermediu acestei funcții se definește condiția de căutare, prototipul ei este de forma:

*bool conditie(TIP );*

deci, returnează adevărat sau fals și primește ca parametru un element al colecției. În caz că funcția *conditie* returnează adevărat înseamnă că procesul de căutare se oprește și funcția *find\_if* returnează un iterator ce referă elementul respectiv, altfel returnează iteratorul ce marchează sfârșitul colecției.

Dacă se folosește pentru căutare funcția *find\_if*, programul anterior poate fi rescris astfel:

```

#include <iostream>
#include <vector>
#include <algorithm>

```

```

using namespace std;
int k;
bool comp(int k1, int k2) { return k1<k2; }
bool conditie(int t) { return k==t; }
void main()
{
    vector<int> v;
    vector<int>::iterator it;
    ostream_iterator<int> os_it(cout, " ");
    cout<<"\n Elementul de cautat:"; cin>>k;
    v.push_back(10); v.push_back(150);
    v.push_back(98); v.push_back(67);
    sort(v.begin(),v.end(),comp);
    it=find_if(v.begin(),v.end(),conditie);
    if(it==v.end() ) cerr<<"\n !!!Elementul <<k<< nu exista!!";
    else copy(it,v.end(),os_it);
}

```

⑤ *transform* – este o funcție cu caracter mai general și are ca obiectiv prelucrarea elementelor dintr-un container (sursă). Elementele care rezultă din prelucrare vor fi stocate într-un alt container (destinație).

Transformarea propriu-zisă a datelor se indică prin intermediul unci funcții independente care primește ca parametru un argument de tipul elementelor containerului și returnează o valoare de tip similar elementelor containerului destinație.

De exemplu, să se scrie programul care afișează modulul elementelor unui container de tip listă.

```

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void main()
{
    list<int> x;
    ostream_iterator<int> os_it(cout, " ");
    x.push_back(-10);
    x.push_back(150);
    x.push_back(-98);
    transform(x.begin(),x.end(), os_it, abs);
}

```

Observații:

- *os\_it* – este un iterator ce referă elemente din containerul destinație;
- *abs* – este o funcție de bibliotecă care returnează modulul unui număr primit ca parametru.

Această funcție mai are o formă care implică încă un container în sensul că transformă datele provenind din două containere sursă, rezultatul fiind trimis într-un alt container, destinație. Prelucrarea se face prin intermediul unei funcții independente care primește două argumente, primul de tipul elementelor primului container sursă, al doilea de tipul elementelor celuilalt container sursă și returnează o valoare având tipul elementelor containerului destinație.

Ca exemplu, vom scrie programul care înmulțește câte două elemente a doi vectori:

$$z_i = x_i * y_i \quad i=1,n$$

unde: *x*, *y*, *z* sunt obiecte colecție de tip *vector*.

```

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
double mul(int k1, double k2)
{
    return k1*k2;
}
void main()
{
    vector<int> x;
    vector<double> y, z(3);
    ostream_iterator<double> os_it(cout, " ");
    x.push_back(10); x.push_back(150); x.push_back(98);
    y.push_back(7.5); y.push_back(1.3); y.push_back(6.7);
    transform(x.begin(),x.end(),y.begin(),z.begin(),mul);
    cout<<"\n Rezultatul:\n";
    copy(z.begin(),z.end(),os_it);
}

```

Pentru a înțelege mai bine cum funcționează funcția *transform* în acest caz, trebuie subliniat că:

- $x$  – este primul container sursă și primi doi parametri ai funcției *transform* precizează elementele din colecția sursă  $x$  care urmează să fie prelucrate, adică toate ( $v.begin() \rightarrow v.end()$ );
- $y$  – este al doilea container sursă și prin iteratorul  $y.begin()$  se menționează că vor fi prelucrate elementele din  $y$ , în ordine, începând cu primul; numărul de elemente prelucrate este automat identic cu numărul de elemente prelucrate din primul container sursă ( $x$ );
- $z$  – este containerul destinație, adică cel în care se vor stoca elementele aşa cum rezultă ele din prelucrarea efectuată de funcția independentă *mul*; în containerul destinație, elementele vor fi stocate începând cu prima poziție, fapt indicat de iteratorul  $z.begin()$ , trimis ca parametru funcției *transform*;
- *mul* – este funcția independentă ce realizează prelucrarea elementelor.

În biblioteca STL există mult mai multe funcții care efectuează prelucrări asupra colecțiilor de date, care se utilizează într-o manieră similară celei prezentate și pe care le puteți afla consultând HELP-ul mediului de programare C++.

În final vă vom prezenta câteva aplicații practice ce implică folosirea unor şabloane de clase.

## 8.7 Aplicații

**1** Traversarea nerecursivă, a unui arbore binar în inordine (Stânga, Rădăcină, Dreapta - SRD) pentru afișare.

Se cunoaște că pentru a traversa nerecursiv un arbore binar este necesară pentru memorarea adreselor de revenire o structură de date auxiliară, de tip stivă. Pentru a nu mai defini structura de stivă și operațiile sale, se va recurge la folosirea şablonului *stack* din STL, care implementează această structură de date.

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

```
// structura nodului pentru arborele binar
struct arbbin
{
    int ut;
    arbbin *ss, *sd;
    arbbin(int k, arbbin *s=NULL, arbbin *d=NULL) : ut(k), ss(s), sd(d)
    {
    }
};

// functia de afisare a unui intreg
void afis(int k)
{
    cout<<setw(7)<<k;
}

// functia de traversare nerecursiva a unui arbore binar in inordine
void trav_inord(arbbin *r, void (*pf)(int))
{
    stack<arbbin*> s;
    while( r || !s.empty())
    {
        while(r)
        {
            s.push(r);
            r=r->ss;
        }
        r=s.top();
        s.pop();
        pf(r->ut); // apelul functiei de prelucrare
        r=r->sd;
    }
}

void main()
{
    // construire arbore binar
    arbbin *rad = new arbbin(50,
                            new arbbin(23,
                                       new arbbin(17, NULL, NULL),
                                       new arbbin(40,
                                                  NULL,
                                                  new arbbin(45,NULL,NULL))),
                            new arbbin(70, NULL, NULL));
    // traversare arbore in vederea afisarii
    trav_inord(rad,afis);
}
```

Se observă că funcția *trav\_inord* realizează traversarea arborelui binar în inordine (SRD) și primește ca parametri rădăcina arborelui și funcția independentă care prelucrează informația utilă din nod prin intermediul unui pointer la funcție; în cazul nostru prelucrarea presupune afișarea informației utile. În această funcție s-a folosit şablonul structurii de stiva (*stack*) având ca tip al elementelor, tipul nodului arborelui binar (*arbbin*).

Metodele care s-au folosit sunt:

- *empty* – care verifică dacă stiva este vidă;
- *push* – pentru inserarea unui element în stivă;
- *top* – metodă care returnează elementul din vârful stivei;
- *pop* – metodă care elimină elementul din vârful stivei;

## ② Determinarea cuvintelor distincte dintr-un text și a frecvențelor lor de apariție.

În vederea rezolvării acestei probleme vom folosi containerul asociativ *map*, considerând cuvântul ca fiind cheia, iar valoarea asociată este un întreg ce reprezintă frecvența sa de apariție; un astfel de container se declară în forma:

*map <string, int> f;*

Un element se definește prin intermediul unei clase pereche (*pair*) în forma:

*pair <string, int> element;*

Programul care rezolvă problema enunțată pornește de la ipotezele că:

- fișierul text pentru care se vor determina cuvintele distincte și frecvențele lor de apariție a fost deja creat;
- un cuvânt este un grup compact de caractere delimitat în text de separatorii: SPAȚIU, ENTER, ;, , TAB, ‘, “,(,),..

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <map>
#include <string>
using namespace std;
typedef pair<string, int> element;
```

```
void main(int argc, char *argv[])
{
    if(argc!=2)
    {
        cerr<<"\n !! Numar incorrect de argumente!!!";
        return;
    }
    ifstream fin(argv[1],ios_base::in);
    if(!fin.is_open())
    {
        cerr<<"\n Fisierul "<<argv[1]<< " inexistent!!!!";
        return;
    }
    map<string, int> f;
    map<string, int>::iterator it;
    string linie;
    char *cuv,*sep=" ;,.()\" \n";
    while(getline(fin,linie))
    {
        cuv=strtok((char*)linie.c_str(),sep);
        while(cuv)
        {
            string t(cuv);
            it=f.find(t);
            if(it!=f.end())
                (*it).second++;
            else
                f.insert(element(t,1));
            cuv=strtok(NULL,sep);
        }
    }
    for(it=f.begin(); it!=f.end(); it++)
        cout<<"\n Cuvintul "<<setw(20)<<(*it).first<<
                    " cu frecventa:"<<(*it).second;
}
```

Se observă că:

- numele fișierului de prelucrat se introduce ca argument al funcției *main()*;
- s-a citit din fișier câte o linie (*getline(fin,linie);*) și s-au extras din ea cuvintele cu funcția *strtok()*;
- pentru a identifica dacă în mulțimea *f* mai există cuvântul s-a apelat metoda *find* a clasei *f* care căută un element în mulțime având ca argument valoarea cheii; în caz că a fost găsit cuvântul în colecție s-a

- incrementat frecvența lui de apariție `(*it).second++`; altfel inserează cuvântul în colecție având frecvența `1` sc
- s-au afișat cuvintele distincte cu frecvențele lor de apariție, adică toate elementele din colecția de tip *map* *f*; folosindu-se iteratorul clasei *map* (*it*).



## **IDENTIFICAREA TIPULUI LA MOMENTUL EXECUȚIEI (RTTI - Run-Time Type Identification)**

- Cadrul general
- Operatorul *typeid*
- Operatorul *dynamic\_cast*

## 9.1 Cadrul general

Așa cum s-a văzut, la derivarea pe mai multe niveluri un pointer de clasă de bază, poate conține adresele a multor tipuri de obiecte derive din ea; apare firesc, problema identificării tipului obiectului (de bază, sau de tipuri de obiecte derive) referit la un moment dat printr-un pointer de bază. Aceasta necesitate apare deoarece, deși metodele virtuale se adaptează automat la tipul pointat, există și alte metode, nevirtuale, care nu știu cum să o facă; în general, când derivăm, nu virtualizăm toate metodele moștenite, dând forme particolare de implementare.

De asemenea, funcțiile independente pot avea în prototip ca parametru un pointer la tipul de bază, dar pot fi apelate și cu adrese de obiecte derive; cum știu ele ce tip pointat au primit, pentru a-și adapta prelucrările la tipul obiectului pe care lucrează? Acestea sunt doar câteva exemple în care apare nevoia auto-identificării clasei la momentul execuției.

O primă formă de rezolvare ar fi includerea în clasă a unei metode *isA()*, care să returneze un *id* de clasă, sau chiar numele clasei; această soluționare ridică însă probleme legate de:

- identificarea unică a clasei, printr-un întreg, citat într-un *enum*, sau printr-un text (numele clasei), lucru greu de asigurat, având în vedere că se lucrează cu multe clase, create de mai mulți programatori, la momente diferite de timp;
- asigurarea supraîncărcării metodei *isA()* pe toată ierarhia derivării, pentru adaptare la fiecare context;
- rezolvarea ambiguităților apărute la moștenirile multiple, când vom avea mai multe metode *isA()* moștenite.

O rezolvare mai elegantă oferă standardul de limbaj începând cu 1993, prin includerea unui mecanism RTTI - *Run-time type identification*.

Compilatorul Visual C++ 6.0, va include acest mecanism dacă indicați switch-ul /GR (în meniul *Project / Settings / pagina C/C++ / categoria Language C++, bifăți opțiunea RTTI*).

Mecanismul RTTI include practic trei elemente noi de limbaj:

1. operatorul *typeid*, pentru extragerea informației de tip;

2. o structură, *class type\_info*, returnată de operatorul *typeid*, după ce a fost completată cu informație suplimentară despre tip;
3. operatorul *dynamic\_cast*, pentru testarea conversiilor dinamice de tip.

## 9.2 Operatorul *typeid*

Operatorul *typeid* primește un parametru (obiect, pointer sau referință) și întoarce un obiect constant, global, de clasă *type\_info*; acesta conține mai multe informații despre clasa primită ca parametru, printre care :

- *id* – ul clasei;
- o funcție, *name()*, care returnează numele în clar, al clasei ;
- supraîncărcări pentru operatorii == și != ;
- funcția *before()* pentru a testa dacă o clasă se află înaintea alteia, în succesiunea de derivare.

Pentru a testa funcționalitatea operatorului *typeid()* vă propunem o ierarhie de clase, obținută pornind de la o clasă de bază, *persoana*:

```
#include <typeinfo.h>
#include <string.h>
#include <iostream.h>
class persoana
{
public:char nume[50];
persoana( char *n= "Anonim"
          { strcpy(nume,n); }
virtual ~persoana() { }
};

class student : public persoana
{
public:
    float media;
    student(char *n, float m) : persoana(n), media(m) { }
};

class muncitor : public persoana
{
public:
    long salariu;
```

```

        muncitor(char *n, long s) : persoana(n), salariu(s) {}

class profesor : public persoana
{
public:
    char disciplina[10];
    profesor(char *n, char *d) : persoana(n)
        { strcpy(disciplina,d); }

void main()
{
    muncitor m("Tanase V", 2500000); student s("Stancu A", 8.50);
    profesor pr( "Madgearu P", "Economie" );
    persoana p, *pp[ ] = { &p,&m,&s,&pr };
    for(int i=0; i<sizeof(pp)/sizeof(pp[0]); i++)
        cout << pp[i]->nume << " este din "
            << typeid( *pp[ i ] ).name() << endl;
}

```

Programul de mai sus gestionează printr-un vector de pointeri de persoane, obiecte derivate (muncitori, profesori, studenți) și le identifică dinamic, după tip, afișând:

```

Anonim este din class persoana
Tanase V este din class muncitor
Stancu A este din class student
Madgearu P este din class profesor

```

Se observă că `typeid()` primește în cazul nostru, obiecte (s-a extras conținutul de la adresa din `pp[i]`) și returnează obiecte `type_info`, cărora le invocă metoda `name()`, pentru a se identifica numele explicit al clasei obiectelor de intrare.

A fost nevoie de *clase polimorfice*, adică cu cel puțin o funcție virtuală (destructorul); ridicând atributul virtual, `typeid()` va răspunde că toate obiectele primite sunt de tip `class persoana`, adică aplică o *identificare statică*, bazată pe declarația de la compilare a vectorului de pointeri.

O secvență :

```
if ( typeid(p).before(typeid(s)) ) cout << " p inaintea lui s ";
```

inclusă în programul anterior, va afișa *p inaintea lui s*, confirmând că un *student* e derivat din *persoana* și nu invers.

Secvența :

`if ( typeid(100)==typeid(int) ) cout << " 100 este de tip intreg";`  
demonstrează folosirea suprăîncărării `operator==()` în clasa `type_info`, precum și funcționarea operatorului pe tipurile de bază.

Operatorul `typeid()` identifică corect chiar clasele definite și incluse în interiorul altei clase; în exemplul următor, clasa copil există doar în interiorul clasei persoana, astfel încât va fi identificată prin `class persoana::copil`, aşa cum o afișează programul:

```

#include <typeinfo.h>
#include <iostream.h>
class persoana
{
public:
    class copil { } * vect_copii;
    persoana() : vect_copii(new copil [2]) {}
    ~persoana() { delete [] vect_copii; }

void main()
{
    persoana p; cout << typeid(*p.vect_copii).name() << endl;
}

```

Folosit în clase template, operatorul `typeid()` identifică corect numele unei clase instanțiate pe baza şablonului:

```

#include <typeinfo.h>
#include <iostream.h>
class persoana
{
    char nume[10];
};

template <class T> class lista
{
public:
    lista() { cout << "Constructor "<<typeid( *this).name(); }

void main()
{
    lista<int> li;
    lista<persoana> lp;;
}

```

Rulat sub Visual C++ 6.0, constructorul de clasă va afișa:

*Constructor class lista<int>*

Constructor class lista&lt;class persoana&gt;

### 9.3 Operatorul *dynamic\_cast*

Operatorul *dynamic\_cast* rezolvă tot problema aflării tipului pointat de un pointer la un moment dat, știind că el poate referi obiecte de bază sau obiecte derivate, la rândul lor de mai multe tipuri; problema apare cu atât mai evidentă când pointerul este transmis ca parametru într-o funcție.

Maniera de rezolvare oferită prin *dynamic\_cast* este următoarea: operatorul încearcă o conversie prin cast, la momentul execuției; dacă la capătul pointerului este un obiect de tipul citat în cast și conversia este acceptată, răspunde cu adresa lui, altfel răspunde cu *NULL*:

```
T * pd = dynamic_cast <T*> (ps)
T &rd = dynamic_cast <T &> (rs)
```

*T* este un tip ce va fi înlocuit de programator cu tipul bănuit a fi pointat și care este testat prin operator, iar *ps* și *pd* sunt pointeri sursă și destinație; în general, *ps* este numele unui pointer spre obiecte de clasă de bază, iar *pd* numele unui pointer spre obiecte de clasă derivată, doarece frecvent se testează dacă un pointer de bază conține adresă de obiect derivat sau de obiect de clasă de bază.

*Downcasting*-ul de pointeri (de la bază către derivat) nu este implicit, dar este legitim dacă *pb* conține adrese de obiect derivat:

```
#include <iostream.h>

class B { public: virtual ~B(){ } };

class D : public B { };

void main()
{
    B* pb = new D;// un pointer de baza poate gestiona obiecte derivate
    D* pd;
    pd= dynamic_cast <D*>(pb); // ok: pb pointa obiect D, ce poate fi
                                // gestionat prin pd
    if( pd==NULL ) cout <<"\n Eroare conv 1";
    else cout <<"\n OK conv 1";
    delete pb;
    pb = new B;
```

```
pd = dynamic_cast <D*>(pb); //eroare: pb pointa obiect B, ce nu pot fi
                            // gestionat prin pd
if( pd==NULL ) cout <<"\n Eroare conv 2"; else cout <<"\n OK conv 2";}
```

Programul afișează:

```
OKconv 1
Eroare conv 2
```

După cum este ușor de văzut din prototip, *dynamic\_cast* lucrează și cu referințe; pentru referințe, lucrurile stau un pic mai delicat, deoarece nu se pot declara referințe decât dacă ele au fost încărcate corect de la început, dar nu se știe când conversia va reuși sau când ea va eşua!

Declarația trebuie aşadar făcută într-un bloc *try* cu *catch()*, pentru captarea eventualelor erori; când conversia eșuează, prin închiderea blocului, variabila referință este dezalocată, lucru cunoscut ca *declarare condiționată a unor variabile*:

```
#include <iostream.h>
class B { public: virtual ~B(){ } };
class D : public B { };
void main()
{
    B b; D d;
    B &rb1=b, &rb2=d;
    try { D &rd1= dynamic_cast <D &> (rb1); } // Esec !
    catch(...) { cout << "\n Conv.1 invalida"; }
    try { D &rd2= dynamic_cast <D &> (rb2); } // Ok !
    catch(...) { cout << "\n Conv.2 invalida"; }
```

După cum se vede, *rb1* a fost încărcată din start cu referință de obiect de bază, astfel încât conversia dinamică în referință de obiect derivat va eşua. În schimb, referința *rb2* deși viza tot obiecte de bază, a fost încărcată cu referință de obiect derivat, astfel încât conversia ulterioară în referință de obiect derivat devine doar o simplă chestiune semantică. Chiar și în caz de succes, folosirea ulterioară a referinței e limitată la blocul de definire.

Mai trebuie observat că aveam nevoie de o clasă polimorfică, motiv pentru care am declarat o funcție virtuală (aici destructorul).

Ne putem pune întrebarea de ce *dynamic\_cast* nu lucrează și pe obiecte, nu numai pe pointeri și referințe. Adevărul este că pe obiecte n-ar avea sens să ne întrebăm dacă o variabilă conține sau nu un anumit tip de

obiect, căci dacă o variabilă este de un anumit tip, doar acel tip îl poate conține.

Ar mai fi o observație interesantă de făcut: în ambele exemple de mai sus conversia este un *downcasting*, adică o conversie neuzuală, dinspre bază către derivat; conversia este totuși legitimă când pointerul de bază conține adresă de obiect derivat; acest lucru ne confirmă că operatorul *dynamic\_cast* a fost în principiu introdus pentru a distinge când un pointer de clasă de bază adresează obiecte derive.

În sens invers (*upcasting*), conversia este oricum implicită, dar se poate face și ea prin *dynamic\_cast*:

```
#include <iostream.h>
class BB { };
class B : public BB { };
class D : public B { };
void main()
{
    D d, * pd=&d;
    cout << "\n pd initial= " << pd;
    B* pb = dynamic_cast<B*>(pd);
    // ok: B este clasa de baza imediata pentru D;
    // pb va pointa pe subiectul B din continutul de la adresa pd
    cout << "\n pb prin dynamic_cast= " << pb;
    pb = pd; cout << " același cu pb din conversie implicită: " << pb << endl;
    BB* pbb = dynamic_cast<BB*>(pd);
    // ok: BB este baza bazei lui D
    // pbb va pointa sub_sub_obiectul BB din continutul de la adresa pd
}
```

Cum era și firesc, *dynamic\_cast* va sesiza și el toate ambiguitățile introduse prin upcasting în cazul moștenirilor multiple din clase cu bază comună, precum și eliminarea ambiguităților prin conversii în etape.

```
#include <iostream.h>
class BB { public: int x; virtual ~BB(){} };
class B1 : public BB { };
class B2 : public BB { };
class D : public B1, public B2 {
void main()
{
    D* pd = new D;
    BB* pbb = dynamic_cast<BB*>(pd);
```

```
if(!pbb) cout << "\n Ambiguitate la clasa BB";
B1* pb1 = dynamic_cast<B1*>(pd); // conversie în cascada
if(pb1) cout << "\n Ok pasul 1 al conversiei";
cout << "\n pb1: " << pb1 << " pd = " << pd;
pb1 = dynamic_cast<BB*>(pb1);
if(pb1) cout << "\n Ok pasul 2 al conversiei. Am ajuns la BB";
cout << "\n pbb pe filiera 1 = " << pbb;
B2* pb2 = dynamic_cast<B2*>(pd); // conversie în cascada
if(pb2) cout << "\n Ok pasul 1 al conversiei";
cout << "\n pb2: " << pb2 << " pd = " << pd;
pb2 = dynamic_cast<BB*>(pb2);
if(pb2) cout << "\n Ok pasul 2 al conversiei. Am ajuns la BB";
cout << "\n pbb pe filiera 2= " << pbb;
}
```

*dynamic\_cast* schimbă și adresa ca valoare și interpretarea ei; deseori prin conversie se obține aceeași valoare a adresei pentru că subiectul de bază e plasat la începutul obiectului derivat; acum însă, se poate vedea după adresa din *pbb*, că *pbb* va referi pe rând, câte unul din cele două subiecte *BB*, conținute de obiectul derivat !

*Eliminarea ambiguității în caz de moștenire multiplă pornind de la o bază comună se face prin derivarea virtuală a claselor *B1* și *B2*. Operatorul *dynamic\_cast* ne va confirma că prin derivare virtuală, facem ca *D* să conțină un singur sub-sub-obiect de tip *BB*; el ne permite în plus, identificarea adresei corecte a sub-sub-obiectului *BB*, iar *pbb* va putea primi direct adresa unui obiect derivat, fără ambiguități. Este de ajuns să derivăm sub forma :*

```
class B1 :virtual public BB { };
class B2 :virtual public BB { },
```

iar programul va afișa aceeași adresă a sub- sub - obiectului *BB*, indiferent de filiera pe care se ajunge la el.

Operatorul *dynamic\_cast* devine esențial, asigurând singurul mod prin care mai putem să dacă într-un pointer de bază se află adresa unui obiect derivat (și anume care, dintr-o ierarhie pe mai multe niveluri) sau adresă de obiect de bază.

Operatorul *dynamic\_cast* este aşadar util pentru clasele polimorfice, când un pointer de bază poate conține adrese de mai multe tipuri derive, dar se poate aplica și în alte cazuri, după cum se poate testa ușor.

În majoritatea cazurilor, mecanismul *RTTI* este implementat prin includerea în tabela de funcții virtuale a unui pointer suplimentar; în cazul

operatorului *typeid()*, pointerul conduce la o funcție ce furnizează la rândul ei adresa unei structuri *type\_info*.

În cazul operatorului *dynamic\_cast*, o altă funcție de bibliotecă testează în plus, dacă destinația este de același tip cu sursa.

Lucrurile se complică în cazul derivării ierarhice pe mai multe niveluri, când trebuie invocat recursiv, același mecanism și pentru nivelurile precedente; acest lucru ne permite să identificăm când o clasă este derivată din alta, chiar când o clasă este bază a derivării, nu neapărat direct, ci pe un nivel anterior mai îndepărtat. Situația se complică și mai mult prin moșteniri multiple, când trebuie următe în sus, mai multe trasee de identificare a tipului de proveniență.

Un lucru trebuie însă precizat, în concluzie: identificarea dinamică a tipului nu trebuie să înlocuiască virtualizarea, care își are rolul și performanțele ei; cu alte cuvinte, nu trebuie ca fiecare funcție să se întrebe cu ce tip de obiect lucrează în realitate și să-și particularizeze prelucrările, când acest lucru se poate face "automat", prin virtualizarea unor metode. Când însă clasa de bază a scris-o altcineva și nu cunoaștem care din metode au fost virtualizate, verificarea tipului prin *dynamic\_cast* este o modalitatea convenabilă de asigurare a integrității tipurilor.



## **FIŞIERUL CA OBIECT**

- **Fișierul în acces secvențial și direct**
- **Fișierul în acces indexat**

În acest capitol ne propunem să implementăm fișierul ca obiect dintr-o perspectivă diferită față de cea deja existentă în biblioteca standard C++. În capitolul 4 am prezentat clasele *ifstream*, *ofstream*, *fstream* care implementau fișierul ca obiect, dar accentul era pus pe tipul de fișier (binar sau text), pe formatarea datelor, pe transferul diferitelor entități informaționale (bait, linie de text, bloc de memorie etc.), pe poziționări etc. Acum ne propunem să implementăm fișierul ca obiect prin prisma accesului la unitatea logică de informație (articoul sau înregistrarea), acces care poate fi secvențial, direct sau indexat.

## 10.1 Fișierul în acces secvențial și direct

Lucru cu fișiere presupune efectuarea unor operații bine precizate cum ar fi:

- deschiderea, respectiv închiderea;
- scrierea / citirea;
- test de fișier deschis sau sfârșit de fișier (controlul erorilor).

Pentru a asigura o maximă generalitate implementării se consideră că articolul este de tip generic deci, practic se vor defini şabioane de clase. Pentru început vom defini modul în care vrem să operăm asupra fișierelor urmând ca apoi să prezintăm implementarea propriu-zisă.

Indiferent de modul de acces implementat *deschiderea* fișierului se va face de către constructor iar *înciderea* lui va cădea în sarcina destrutorului.

Pentru **accesul secvențial** s-au supraîncărcat operatorii << și >> în vederea realizării operațiilor de scriere, respectiv citire în / din fișier. Am ales această variantă pentru că ea este deja consacrată, după cum am văzut la lucru cu stream-uri. *Scrierea secvențială* a unui articol în fișier se va face printr-o expresie de forma: **f<<art;** în timp ce *citirea* unui articol se va face în forma **f>>art;** unde, *f* este un obiect fișier, iar *art* este o variabilă de tipul articoului.

**Accesul direct** implică scrierea respectiv citirea unui articol la / de la o anumită poziție în / din fișier. Masivul unidimensional (*vectorul*) este structura a cărei exploatare se face pornind de la poziția ocupată de un element în cadrul ei. Prin comparație ne-am gândit ca accesul direct în fișier să-l realizăm manipulând fișierul ca pe un vector, sub forma:

- **f[i]=art;** pentru a scrie un articol (*art*) în fișierul *f* la poziția *i*;
- **art=f[i];** pentru a citi un articolul *i* din fișierul *f* în variabila *art*.

Când am definit clasa *vector*, pentru a accesa un anumit element am supraîncărcat operatorul [ ]. Ca să putem consulta sau modifica elementul, funcția care supraîncărca operatorul [ ] returnă o referință la element respectiv. Astfel, o expresie de genul **v[i]=k;** modifica elementul de pe poziția *i* din vectorul *v*, pe când o expresie de genul **k=v[i];** încărca variabila *k* cu valoarea elementului de pe poziția *i* din vector. Cu alte cuvinte, obținerea sau modificarea elementului se face în funcție de locul pe care-l ocupă operandul *v[i]* într-o expresie (dacă este *lvaloare* atunci el va fi modificat).

În cazul fișierelor lucrurile nu sunt la fel de simple pentru că o expresie de genul **f[i]=art;** implică apelul unei funcții de scriere în fișier, pe când expresia **art=f[i];** determină apelul unei funcții de citire din fișier. Se deduce clar că simpla supraîncărcare a operatorului [ ] în clasa fișier nu rezolvă pe deplin problema. Pentru rezolvarea acestei situații vom *privi* fișierul ca fiind o colecție de articole, adică implementarea presupune definirea a două clase:

- clasa *fișier* care modelează fișierul în ansamblul lui;
- clasa *articol* care operează asupra entității logice de bază a fișierului care este articolul sau înregistrarea.

În figura 10.1 se pune în evidență relația dintre fișier și articole, relație de tip *unu la mulți*, în sensul că un fișier are mai multe articole.



Fig. 10.1 Legătura fișier articol

Modurile de deschidere a fișierelor s-au definit prin intermediul unor constante simbolice după cum urmează:

```

#define SCRIERE 1
#define CITIRE 2
#define CITIRE_SCRIERE 3
#define SCRIERE_CITIRE 4
#define ADD 5           // adaugare
  
```

și se va lucra numai cu fișiere binare.

Articolul este de tip generic, denumit simbolic ART.  
Şablonul clasei fișier este:

```

#include <stdio.h>
#include <process.h>
// anunta folosirea clasei articol
template <typename ART> class articol;
template <typename ART> class fisier
{
    friend articol<ART>;
public:
    fisier(char *, int=SCRIERE);
    articol<ART> operator[] (long poz)
    {
        fseek(pf, poz*sizeof(ART), SEEK_SET);
        return articol<ART>(*this);
    }
    fisier& operator<<(ART a)
    {
        if(pf && mod!=2)
            fwrite(&a, sizeof(a), 1, pf);
        return *this;
    }
    fisier& operator>>(ART& a)
    {
        if(pf && mod!=1 && mod!=5)
            sf=fread(&a, sizeof(a), 1, pf);
        return *this;
    }
    int gata() { return !sf; }
    int e_deschis() { return pf ? 1 : 0; }
    ~fisier() { if(pf) fclose (pf); }
protected:
    FILE *pf;
    int sf,mod;
};

template<typename ART>
fisier<ART>::fisier(char *nume,int m):mod(m)
{
    switch(mod)
    {
        case SCRIERE:
            pf=fopen(nume,"wb"); break;
        case CITIRE:
            pf=fopen(nume,"rb"); break;
        case CITIRE_SCRIERE:
    }
}

```

```

    pf=fopen(nume,"r+b"); break;
    case SCRIERE_CITIRE:
        pf=fopen(nume,"w+b"); break;
    case ADD:
        pf=fopen(nume,"ab"); break;
    default:
        cerr<<"\n !! Mod ERONAT de deschidere fisier !!";
        exit(1);
    }
}

```

Se observă că:

- s-a definit un singur constructor care primește ca parametri numele extern al fișierului și modul de deschidere și efectuează deschiderea propriu-zisă a fișierului;
- s-au supraîncărcat operatorii << și >> pentru scrierea / citirea secvențială de articole;
- pentru semnalarea unor erori s-au definit metodele:
  - *e\_deschis()* care returnează adevărat dacă fișierul s-a deschis;
  - *gata()* care returnează adevărat dacă s-a depistat sfârșitul de fișier;
- destructorul închide fișierul;
- operatorul [ ] a fost supraîncărcat astfel încât să realizeze o poziționare în fișier, pe articolul dorit, după care să returneze un obiect de tip articol, apelând totodată constructorul lui.

Scrierea / citirea efectivă a articolului când accesul este direct se realizează în clasa *articol*, care are şablonul următor:

```

// anunta clasa fisier
template <typename ART> class fisier;
template <typename ART> class articol
{
protected:
    friend fisier<ART>;
    fisier<ART> &pfis;
public:
    void operator=(ART a)
    {
        if(pfis.pf && pfis.mod!=2)
            fwrite(&a, sizeof(a), 1, pfis.pf);
    }
    operator ART();
}

```

```

articol(fisier<ART> &f):pfis(f) { }

};

template <typename ART>
articol<ART>::operator ART()
{
    ART a;
    if(pfis.pf && pfis.mod!=1 && pfis.mod!=5)
        pfis.sf=fread(&a,sizeof(a),1,pfis.pf);
    return a;
}

```

Din descrierea clasei *articol* se observă că prin constructor se face legătura cu fișierul căruia îi aparține articolul. Realizarea operației de scriere a articolului, la poziția curentă în fișier, s-a făcut prin supraîncărcarea operatorului de atribuire (=), iar pentru a citi articolul de la poziția curentă s-a supraîncărcat operatorul de conversie explicită (*cast*) la tipul articolului (ART).

Privind descrierile celor două clase se observă că citirea unui articol, indiferent de locul unde se efectuează, setează variabila *sf* din clasa *fisier* pentru a evidenția sfârșitul de fișier; pe de altă parte, ambele obiecte își pot accesa părțile private sau protejate și asta datorită faptului că relația de prietenie (*friend*) este acordată reciproc.

În programul următor vom exploata fișiere în acces secvențial sau direct, cu diferite tipuri de articol:

```

struct pers
    { int marca; char np[30]; };
struct mat
    { int cod; char den[20]; };
void main()
{
// **** Testare fisier in acces secvențial ****
    pers aux2, vp[]={ {1000,"Vasile A."}, {1500,"Ionescu D."} };
// Creare fisier
    fisier<pers> fpers("testpers.dat");
    fpers<<vp[0]<<vp[1]; // Scriere secvențiala de articole
}
{
// Deschidere fisier in citire
    fisier<pers> fpers("testpers.dat",CITIRE);

```

```

if(!fpers.e_deschis()) // test daca exista
{
    cerr<<"\n !!!Fisier inexistent!!!";
    return;
}
// citire secvențiala cu test de sfîrsit fisier
while(fpers>>aux2, !fpers.gata())
    cout<<aux2.marca<< " <<aux2.np<<endl;
}

// ****Testare fisier in acces direct ****

int i;
double d;
mat aux1,va[]={ {100,"tabla"}, {104,"Caramida"}, {102,"tigla"}, {103,"ciment"} };
fisier<double> fd("testd.dat",SCRIERE_CITIRE);
fisier<mat> fmat("testmat.dat",SCRIERE_CITIRE);
// scriere in acces direct
fmat[0]=va[0]; fmat[1]=va[1]; fmat[2]=va[2];
fd[0]=5.7; fd[1]=8.901;
// citire in acces direct
for(i=0; aux1=fmat[i], !fmat.gata(); i++)
    cout<<aux1.cod<< " <<aux1.den<<endl;
for(i=0; d=fd[i], !fd.gata(); i++) cout<<d<<endl;
}

```

Am substituit tipul generic fie cu tip fundamental (*double*), fie cu tipuri definite de utilizator (*mat* și *pers*). Fișierele exploatați în acces direct au fost deschise în modul *SCRIERE\_CITIRE* care presupune crearea fișierului dacă nu există sau suprascrierea lui dacă există și permite atât scrierea, cât și citirea de articole.

## 10.2 Fișierul în acces indexat

### Cadrul conceptual

Accesul indexat la un fișier presupune regăsirea unui articol pornind de la valoarea unui câmp care se numește *cheie*, prin efectuarea unui singur acces la disc. Acest lucru este posibil prin întreținerea unei structuri de

regăsire, uzual arbore, care stochează valoarea cheii și poziția asociată articolului respectiv în fișier. Dacă cu ajutorul câmpului cheie se identifică în mod unic un articol, atunci cheia este se numește *cheie primară*. Astfel, codul numeric poate reprezenta cheie primară pentru identificarea unei persoane, numărul de înmatriculare pentru identificarea unui autovehicul, numărul de inventar pentru mijloace fixe, numărul matricol pentru elevi sau studenți etc.

Noi am implementat accesul indexat în ipotezele:

- cheia este primară;
- se generează două seturi de informații (fișiere):
  - fișierul care conține datele propriu-zise;
  - un alt fișier care memorează indexul, adică structura de date care face corespondență între valoarea cheii și poziția articolului respectiv în fișierul de date.

Operațiile de scriere, respectiv citire se fac specificând pe de o parte valoarea cheii iar pe de altă parte articolul care cedează respective primește informații. Am optat și în acest caz pentru o descriere cât mai naturală și anume, similar adresării în masive unidimensionale:

- pentru scriere: **f[ch] = art**; unde, *ch* este valoarea cheii articolului *art* care va fi scris în fișierul *f*;
- pentru citire: **art = f[ch]**; adică pornind de la valoarea cheii (*ch*) se obține articolul corespunzător (*art*).

Spre deosebire de accesul direct unde în loc de valoarea cheii era un întreg care indică poziția articolului, la accesul indexat tipul cheii nu este aprioric cunoscut. Rezultă de aici că şablonul clasei care implementează fișierul în acces indexat va lucra cu două tipuri generice:

- ART pentru tipul articolului;
- CH pentru tipul câmpului cheie.

## Structura de index

Indexul este construit pe baza unei structuri arborescente având nodurile memorate într-un vector pentru a permite salvarea / restaurarea lui în / din fișier. Astfel, trimiterile spre subarborei stâng și drept de la stocarea în memorie dinamică dispară, nu ar mai fi valabile într-o sesiune viitoare de lucru. Un nod al indexului conține cheia (*k*), poziția articolului în fișierul de date (*off*) și legăturile către cei doi subarbore (stâng – *ss* și drept – *sd*). Având în vedere că tipul cheii este necunoscut, s-a construit şablonul clasei nod având tipul generic *CH*:

```
template <typename CH> class arbbin;
template <typename CH> class nod
{
    friend arbbin<CH>;
protected:
    struct nd
    {
        CH k;
        long off;
        int ss, sd;
    } n;
public:
    nod(CH c, long poz, int as=-1, int ad=-1)
        { n.k=c; n.off=poz; n.ss=as; n.sd=ad; }
    void set_ss(int t) { n.ss=t; }
    void set_sd(int t) { n.sd=t; }
};
```

În această clasă se observă că informația utilă este grupată într-o variabilă de tip structură pentru a putea fi scrisă corect în fișier. Constructorul încarcă informația utilă (cheia – *c* și poziția – *poz*), precum și legăturile care au ca valori implicate pe -1, ce sugerează că este un nod frunză, deoarece arborele se memorează în vector. Metodele *set\_ss()* și *set\_sd()* modifică legăturile unui nod, din parametrul de apel.

Având în vedere că tipul cheii poate fi și de tip sir de caractere (*char\**), pentru clasa *nod* s-a definit și specializarea pentru acest tip:

```
template <> class nod<char*>
{
    friend arbbin<char*>;
protected:
    struct nd
    {
        char k[150];
        long off;
        int ss, sd;
    } n;
public:
    nod(char *c, long poz, int as=-1, int ad=-1)
        { strcpy(n.k,c); n.off=poz; n.ss=as; n.sd=ad; }
    void set_ss(int t) { n.ss=t; }
    void set_sd(int t) { n.sd=t; }
};
```

Se observă din această definiție că se poate lucra cu o cheie de maxim 149 caractere, iar pentru atribuirea de șiruri s-a folosit funcția `strcpy()` în locul operatorului = din şablonul general, în rest totul a rămas neschimbat.

Vectorul folosit pentru memorarea nodurilor arborelui nu a fost definit explicit de noi, ci s-a folosit implementarea din STL a clasei `vector`.

Şablonul care implementează arborele binar de căutare cu noduri stocate în vector are la rândul lui tipul cheii ca tip generic *CH*:

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

template <typename CH> class arbbin
{
protected:
    vector< nod<CH>*> vn;
    char nfi[100];
    int aex;
    int insarb(int &, CH, int, int =1);
    void salvare();
    long c_nod(CH,int=0);

public:
    arbbin(){ }
    void set_nume_f(char *n) { strcpy(nfi,n); }
    void restaurare();
    long cauta_nod(CH k)
    {
        if(vn.size()==0) return -1;
        return c_nod(k);
    }
    void insert_nod(CH k,int pz) { int y; aex=insarb(y,k,pz); }
    int articol_existent() { return aex; }
    ~arbbin() { salvare(); }
};

template <typename CH>
int arbbin<CH>::insarb(int &sb, CH k, int pz, int vb)
{
    static indice;
    if(vb) sb=0,indice=vn.size();
    if(indice==0||sb== -1)
    {
        sb=indice;    vn.push_back(new nod<CH>(k,pz));
        return 0;
    }
}
```

```
else
{
    if(k<vn[sb]->n.k) return insarb(vn[sb]->n.ss,k,pz,0);
    else if(k>vn[sb]->n.k) return insarb(vn[sb]->n.sd,k,pz,0);
    else return 1;
}

template <typename CH>
long arbbin<CH>::c_nod(CH k, int ind)
{
    if(ind== -1) return -1;
    else
    {
        if(k<vn[ind]->n.k) return c_nod(k, vn[ind]->n.ss);
        else if(k>vn[ind]->n.k) return c_nod(k, vn[ind]->n.sd);
        else return vn[ind]->n.off;
    }
}

template <typename CH>
void arbbin<CH>::salvare()
{
    int nr=vn.size();
    FILE *p=fopen(nfi,"wb");
    for(int i=0; i<nr; i++) fwrite(&vn[i]->n,sizeof(vn[i]->n),1,p);
    fclose(p);
    for(i=0; i<vn.size(); i++) delete vn[i];
    vn.erase(vn.begin(), vn.end());
}

template <typename CH>
void arbbin<CH>::restaurare()
{
    nod<CH>::nd aux;
    FILE *p=fopen(nfi,"rb");
    while(fread(&aux,sizeof(aux),1,p))
        vn.push_back(new nod<CH>(aux.k,aux.off,aux.ss,aux.sd));
    fclose(p);
}
```

Clasa `arbbin` gestionează nodurile prin intermediul obiectului *vn* de tip `vector`. Cum știm că în STL vectorul este dat ca şablon, precizăm că elementele lui sunt de tip pointeri la obiecte `nod`; cum și nodul este tot un şablon, obiectul *vn* a fost definit ca *template* de *template*:

```
vector< nod <CH> * > vn;
```

Pe lângă metodele clasice de lucru cu arbori binari de căutare:

- *insert\_nod()* – pentru inserarea unui nod în arbore, dând valoarea cheii și poziția articolului în fișierul de date;
- *cauta\_nod()* – care returnează poziția articolului în fișierul de date pornind de la valoarea cheii furnizată ca parametru metodei;

clasa *arbbin* mai conține și metode care realizează salvarea (*salvare()*) respectiv restaurarea (*restaurare()*) unui arbore în / din fișier. Operația de salvare este invocată de destructorul clasei în timp ce restaurarea se face doar dacă un fișier ce memorează un index există. Numele fișierului în / din care se face salvarea / restaurarea este memorat în variabila membru *nfi* și este furnizat ca parametru funcției de acces *set\_nume()*.

Operația de inserare nod poate eșua dacă un nod cu aceeași cheie există deja în arbore; poate de asemenea eșua căutarea dacă nodul care conține o anumită cheie nu există. Pentru semnalarea unor astfel de situații s-a definit funcția de acces *articol\_existent()* care returnează valoarea membrului *aex* cu semnificația de adevărat, dacă există nodul cu o anumită cheie în arbore, indiferent de operația efectuată.

Ca și în cazul nodului vom prezenta și pentru şablonul *arbbin* specializarea pentru tipul *char\**, utilă când cheia este de tip sir de caractere.

```
template <> class arbbin<char*>
{
protected:
    vector< nod<char*> * > vn;
    char nfi[100];
    int aex;
    int insarb(int &, char*, int, int =1);
    void salvare();
    long c_nod(char *,int=0);
public:
    arbbin(){}
    void set_nume_f(char *n) { strcpy(nfi,n); }
    void restaurare();
    long cauta_nod(char *k)
    {
        if(vn.size()==0) return -1;
        return c_nod(k);
    }
    void insert_nod(char *k,int pz) { int y; aex=insarb(y,k,pz); }
    int articol_existent() { return aex; }
```

```
~arbbin() { salvare(); }
```

```
};
```

```
int arbbin<char*>::insarb(int &sb, char *k, int pz, int vb)
{
    static indice;
    if(vb) sb=0,indice=vn.size();
    if(indice==0||sb== -1)
    {
        sb=indice;    vn.push_back(new nod<char*>(k,pz));
        return 0;
    }
    else
    {
        if(strcmp(k,vn[sb]->n.k)<0) return insarb(vn[sb]->n.ss,k,pz,0);
        else if(strcmp(k,vn[sb]->n.k)>0)
            return insarb(vn[sb]->n.sd,k,pz,0);
        else return 1;
    }
}
```

```
long arbbin<char*>::c_nod(char *k, int ind)
```

```
{
```

```
    if(ind== -1) return -1;
    else
    {
        if(strcmp(k,vn[ind]->n.k)<0) return c_nod(k, vn[ind]->n.ss);
        else if(strcmp(k,vn[ind]->n.k)>0) return c_nod(k, vn[ind]->n.sd);
        else return vn[ind]->n.off;
    }
}
```

```
void arbbin<char*>::salvare()
```

```
{
    int nr=vn.size();
    FILE *p=fopen(nfi,"wb");
    for(int i=0; i<nr; i++) fwrite(&vn[i]->n,sizeof(vn[i]->n),1,p);
    fclose(p);
    for(i=0; i<vn.size(); i++) delete vn[i];
    vn.erase(vn.begin(), vn.end());
}
```

```
void arbbin<char*>::restaurare()
```

```
{
    nod<char*>::nd aux;
    FILE *p=fopen(nfi,"rb");
```

```

while(fread(&aux,sizeof(aux),1,p))
    vn.push_back(new nod<char*>(aux.k,aux.off,aux.ss,aux.sd));
fclose(p);
}

```

Se observă că specializarea lui *arbbin* utilizează specializarea şablonului *nod*, iar principala diferență privind codul constă în faptul că la comparările care implică cheia se folosește funcția de bibliotecă specializată în lucru cu șiruri de caractere (*strcmp()*) și nu operatori relaționali.

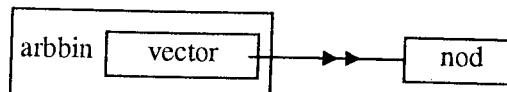


Fig. 10.2 Relații între clasele care implementează indexul

Pentru o clarificare mai bună a modului în care conlucrează clasele prezentate pentru implementarea structurii de index, în figura 10.2 sunt puse în evidență relațiile dintre clase:

- *arbbin* are inclus un obiect de tip *vector*;
- clasa *arbbin* prin intermediu vectorului este în relație de tip colecție cu clasa *nod*, în sensul că un arbore conține mai multe noduri.

### Implementarea accesului indexat la fișier

În acest moment se impune a recapitula ceea ce am implementat până acum în acest capitol. Ne amintim că am construit clasa *fisier* pentru a putea exploata fișierul atât în acces secvențial cât și în acces direct și clasa *arbbin* pentru a întreține un arbore binar de căutare folosit la implementarea structurii de index.

Pentru a realiza accesul indexat la fișier am precizat că avem nevoie de un fișier de date și de un index pentru manipularea informațiilor din fișierul de date. Cu alte cuvinte disponem de elementele de bază necesare doar că mai trebuie definită o structură care să sincronizeze fișierul de date și indexul corespunzător lui. În acest sens vom mai construi o clasă (*fisier\_indexat*) care moștenește multiplu caracteristicile celor două clase, anterior menționate, ca în figura 10.3.

Clasa *fisier\_indexat* este tot un şablon de clasă având două tipuri generice:

- ART – tipul articolului indus de clasa *fisier*;

- CH – tipul cheii indus de clasa *arbbin* care poate fi *int* sau *char\**.

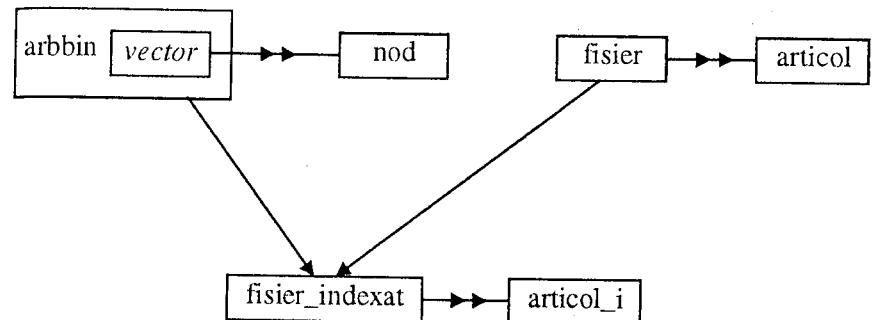


Fig. 10.3 Ierarhie de clase pentru implementarea fisierului în acces indexat

Ca și în cazul fișierului în acces direct și pentru accesul indexat s-a mai definit o clasă *articol\_i* în care se definesc operațiile ce se realizează cu un articol al fișierului indexat.

Pentru a putea folosi și chei de tip șir de caractere (*char\**) ar fi fost elegant să definim o specializare parțială pentru şablonul clasei *fisier\_indexat*. Cum însă din păcate în Visual C++ conceptul nu este implementat am uzitat de următorul truc: în clasă s-a definit variabila *sky* de tip șir de maxim 150 caractere și *ky* de tip CH, iar prin folosirea mecanismului recunoașterii dinamice a tipului la momentul execuției (RTTI) vom stoca cheia în variabila *sky* dacă este de tip *char\**, altfel în variabila *ky*. Dacă tipul cheii este *char\**, atunci clasa *fisier\_indexat* va moșteni specializarea clasei *arbbin* pentru tipul *char\**.

Şablonul clasei *fisier\_indexat* este:

```

#include <typeinfo.h>
template <typename ART, typename CH> class articol_i;
template<typename ART, typename CH>
class fisier_indexat:public fisier<ART>, public arbbin<CH>
{
protected:
    friend articol_i<ART,CH>;
    int tc;
    char sky[150];
    CH ky;
public:

```

### Fisierul ca obiect

```

fisier_indexat(char *nfd,char *nfi,int m=SCRIERE):fisier<ART>(nfd,m)
{
    tc= strcmp(typeid(CH).name(),"char *")==0 ? 1:0;
    if(tc) ky = (CH)sky;
    set_nume_f(nfi);
    if(m==CITIRE || m==CITIRE_SCRIERE) restaurare();
}
articol_i<ART,CH> operator[](CH k)
{
    tc ? strcpy(sky,(const char*)k) :
        strncpy((char*)&ky,(char*)&k,sizeof(CH));
    return articol_i<ART,CH>(*this);
}
~fisier_indexat(){}
};

Constructorul clasei primește numele fișierului de date al fișierului de index și modul de deschidere a fișierelor; dacă modul de deschidere este SCRIERE sau SCRIERE_CITIRE fișierele se vor crea concomitent.
Operatorul [ ] a fost supraîncărcat în aceeași idee ca la accesul direct, numai că de această dată se primește valoarea cheii, și se returnează un obiect de tip articol_i.

```

```

template<typename ART, typename CH> class fisier_indexat;
template<typename ART, typename CH> class articol_i
{
    friend fisier_indexat<ART,CH>;
protected:
    fisier_indexat<ART,CH> &fis_i;
public:
    articol_i(fisier_indexat<ART,CH> &f):fis_i(f) {}
    operator ART()
    {
        ART a;
        long depl=fis_i.cauta_nod(fis_i.ky);
        if(depl!=-1) // daca s-a gasit
        {
            fis_i.aex=1;
            fseek(fis_i.pf,depl,SEEK_SET);
            fread(&a,sizeof(ART),1,fis_i.pf);
        }
        else fis_i.aex=0;
        return a;
    }
};

```

### Fisierul ca obiect

```

void operator=(ART a)
{
    fseek(fis_i.pf,0,SEEK_END);
    long p=ftell(fis_i.pf);
    fwrite(&a,sizeof(ART),1,fis_i.pf);
    fis_i.insert_nod(fis_i.ky,p);
}
~articol_i(){}

```

Constructorul face legătura cu obiectul *fisier\_indexat* care a generat obiectul *articol\_i*.

Scrierea / citirea efectivă a unui articol în acces indexat se face prin supraîncărcarea operatorului = respectiv a operatorului cast la tipul articolului (ART). Se observă că scrierea articolui presupune adăugarea lui în fișierul de date, după care se inserează un nou nod în arborele binar; citirea se face prin consultarea prealabilă a indexului, rezultând poziția articolului în fișierul de date, după care printr-un singur acces la disc se obține articolul din fișierul de date.

Programul următor prezintă modul de exploatare a fișierelor în acces indexat; se vor utiliza atât articole cât și chei, de tipuri diferite.

```

#include<iostream>
#include<strstream>
struct nou { int a; char cheie[20]; double x; };
struct mat { int cod; char den[20]; };
void main()
{
// ****Testare fisier indexat ****
// creare fisier indexat
{
    mat va[]={ {100,"tabla"}, {104,"Caramida"}, {102,"tigla"}, {103,"ciment"} };
// deschidere fisier in acces indexat
    fisier_indexat<mat,int> fi_mat("testf_i.dat","fi.idx");
// adaugare de articole in acces indexat
    mat tt=va[2];
    fi_mat[tt.cod]=tt;
    fi_mat[va[0].cod]=va[0];
    fi_mat[va[3].cod]=va[3];
} // Incercarea de a mai adauga un articol avand o cheie care deja există

```

```

    fi_mat(va[0].cod)=va[0];
// test de cheie duplicata
    if( fi_mat.articol_existent() ) cout<<"\n Cheie existenta!!!";
    fi_mat(va[1].cod)=va[1];
}
// consultare fisier indexat
{
// deschidere fisier indexat pentru consultare
    fisier_indexat<mat,int> fi_mat("testf_i.dat","fi.idx",CITIRE);
    mat aux1;
    int k;
    char sk[50];
    strstream s(sk,49);
    while(cout<<"\n Dati codul de cautat:",cin.getline(sk,49))
    {
        s<<sk;      s>>k;
// citire din fisier in acces indexat si test de existenta a articolului
        if(aux1==fi_mat[k], fi_mat.articol_existent())
            cout<<aux1.cod<<" "<<aux1.den<<endl;
        else    cerr<<"\n Articolul cu codul "<<k<<" nu exista!!!";
    }
//fisier indexat cu cheie char*
// creare
{
    nou vnou[]={ {100, "ch1", 56.889}, {500, "cheia5", 45.7},
                {300, "ch3", 6} };
    fisier_indexat<nou,char*> fi_sir("testf_i_sir.dat","fsir.idx");
    fi_sir[vnou[0].cheie]=vnou[0];
    fi_sir[vnou[1].cheie]=vnou[1]; fi_sir[vnou[2].cheie]=vnou[2];
}
// consultare
{
    fisier_indexat<nou,char*> fi_sir("testf_i_sir.dat","fsir.idx",CITIRE);
    nou aux;  char kk[50];
    cin.clear();
    while(cout<<"\n Dati cheia de cautat:",cin.getline(kk,50))
    {
        if(aux==fi_sir[kk],fi_sir.articol_existent())
            cout<<aux.a<<" "<<aux.cheie<<" "<<aux.x<<endl;
        else    cerr<<"\n Articolul cu codul "<<kk<<" nu exista!!!";
    }
}

```

## **TESTE GRILĂ**

- **Întrebări**
- **Răspunsuri**

## Întrebări

1.

Clasa:

```
class c { float a; void afis_a(); };
```

are membrii :

- a) publici;    b) privați;    c) date private și metode publice;  
 d) protected; e) descriși eronat, deoarece nu declară tipul de acces.

2.

Fie secvența :

```
class c{ ... };
void main () { c e; /* instructiuni */ }
```

În acest caz :

- a) c este obiect și e este o clasa;  
 b) c este o instanta a clasei și e este un obiect;  
 c) c este o clasa și e un obiect;  
 d) c este o instanță a clasei și e este o clasă;  
 e) descrierea este eronată deoarece se folosește același identificator pentru clasă și obiect.

3.

Având declarația :

```
class persoana {
    char nume[50];
    int virsta;
public:
    persoana ();
    int spune_virsta() { return virsta; }
};
```

Funcția int spune\_virsta( ) este :

- a) funcție inline;    b) funcție friend;    c) funcție virtuală;  
 d) descrisă eronat, deoarece nu se definesc funcții într-o clasă;

e) eronată, deoarece variabila *virsta* este privată.

4.

O funcție *friend* diferă de o metodă obișnuită a unei clase prin faptul că :

- a) nu primește pointerul implicit la obiect *this*;  
 b) nu poate accesa decât partea publică a obiectului;  
 c) nu se poate defini *inline*;  
 d) se folosește doar pentru supraîncarcarea operatorilor;  
 e) nu poate returna valori.

5.

Având urmatoarele declarații :

```
class e1 { int a, b, c ;
public :
    e1();
    e1(int , int );
    e1(int , int , int );
};

class e2 { int a, b ;
public :
    e2(const e2& );
    e2(int , int );
};
```

Care din clase are constructor de copiere ?

- a) e1;    b) e2;    c) ambele;    d) nici una;  
 e) nu se poate preciza, descrierea fiind eronată.

6.

Fie clasa :

```
class c { int a, b ;
public :
    float c (int , int );
    int det_a () { return a; }
    c(); }
```

Declarația float c (int , int) ar putea corespunde unui constructor al clasei ?

- a) da, fiind o supraîncarcare a celui existent;  
 b) nu, deoarece crează ambiguitate;  
 c) nu, deoarece constructorul nu are tip returnat;  
 d) nu, deoarece nu este friend;

e) nu, deoarece se returnează o valoare de tip real, iar datele din obiect sunt numai de tip întreg.

7.

Fie clasa :

```
class c { int a, b ;
public :
    c (int , int ) ;
    int det_a ( ) { return a ; }
    ~c () ; }
```

Semnul ~ are rolul :

- a) de a nega pe biți rezultatul returnat de metoda c( );
- b) de a nega logic rezultatul returnat de metoda c( );
- c) de a preciza existența destructorului;
- d) de a supraîncărca constructorul clasei;
- e) de a supraîncărca operatorul ~.

8.

Fie programul:

```
class c { int a;
public :
    c();
    c(const c&);
    void operator =(c&); }
void main()
{ c a;
    // instructiuni
    c b=a;
    // instructiuni }
```

Linia c b=a ; determină:

- a) execuția constructorului de copiere ;
- b) execuția metodei prin care se supraîncarcă operatorul = ;
- c) execuția atât a constructorului de copiere, cât și a metodei operator= ;
- d) eroare, deoarece nu se permite combinarea atribuirii cu o declarație;
- e) execuția constructorului implicit.

9.

De câte ori se apelează destructorul clasei în exemplul următor:

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

```
class cls { public: ~cls() { cout << "\n Destructor"; } };
void main( ) { { cls r[3]; } }
```

- a) nici o dată;    b) o dată;    c) de două ori;    d) de trei ori;    e) de patru ori;

10.

```
#include <iostream.h>
class cls { public: ~cls() { cout << "\n Destructor"; } };
void main( ) { cls *po = new cls[3]; /* ... */ delete [] po; }
```

În exemplul anterior, destructorul clasei:

- |                                |                            |
|--------------------------------|----------------------------|
| a) nu se apelează nici o dată; | b) se apelează o dată;     |
| c) se apelează eronat;         | d) se apeleză de trei ori; |
| e) se apeleză de patru ori;    |                            |

11.

O funcție independentă declarată friend în domeniul public dintr-o clasă și care primește ca parametru o referință la un obiect al clasei respective are acces:

- |   |                          |
|---|--------------------------|
| a) doar la membrii public;                                      | b) la toți membrii;      |
| c) la membrii public și protected;                              | d) la membrii protected; |
| e) la toți membrii, dar îl poate doar consulta, nu și modifica. |                          |

12.

O funcție independentă declarată friend în domeniul private dintr-o clasă și care primește ca parametru o referință la un obiect al clasei respective are acces:

- |   |                       |
|---|-----------------------|
| a) doar la membrii public;                                      | b) la toți membrii;   |
| c) la membrii public și protected;                              | d) la membrii private |
| e) la toți membrii, dar îl poate doar consulta, nu și modifica. |                       |

13.

```
#include <iostream.h>
class cls
{
public: cls() { cout << "\n Constructor"; }
        cls( cls &c) { cout << "\n Constructor copiere"; }
};
int f( cls c) { return 1; }
```

```
void main() { cls c; f(c); }
```

La execuția programului de mai sus:

- a) constructorul de clasă se apelează o dată, iar cel de copiere nici o dată;
- b) constructorii de clasă și de copiere se apeleză fiecare câte o dată;
- c) constructorul de clasă nu se apeleză nici o dată, iar cel de copiere o dată;
- d) constructorul de clasă se apeleză de două ori, iar cel de copiere niciodată;
- e) constructorii de clasă și de copiere se apeleză fiecare de câte două ori;

14.

```
#include <iostream.h>
class cls
{
    public: cls() { cout << "\n Constructor"; }
            cls( cls &c) { cout << "\n Constructor copiere"; }
};
int f( cls &c) { return 1; }
void main() { cls c; f(c); }
```

La execuția programului de mai sus:

- a) constructorul de clasă se apeleză o dată, iar cel de copiere nici o dată;
- b) constructorii de clasă și de copiere se apeleză fiecare câte o dată;
- c) constructorul de clasă nu se apeleză nici o dată, iar cel de copiere o dată;
- d) constructorul de clasă se apeleză de două ori, iar cel de copiere niciodată;
- e) constructorii de clasă și de copiere se apeleză fiecare de câte două ori;

15.

```
#include <iostream.h>
class cls
{
    public: cls() { cout << "\n Constructor"; }
            cls( cls &c) { cout << "\n Constructor copiere"; }
};
int f( cls *pc) { return 1; }
void main() { cls c; f(&c); }
```

La execuția programului de mai sus:

- a) constructorii de clasă și de copiere se apeleză fiecare de câte două ori;
- b) constructorii de clasă și de copiere se apeleză fiecare câte o dată;

- c) constructorul de clasă nu se apeleză nici o dată, iar cel de copiere o dată;
- d) constructorul de clasă se apeleză de două ori, iar cel de copiere niciodată;
- e) constructorul de clasă se apeleză o dată, iar cel de copiere nici o dată;

16.

```
#include <iostream.h>
class cls
{
    public: cls() { cout << "\n Constructor"; }
            cls( cls &c) { cout << "\n Constructor copiere"; }
};
cls f() { cls c; return c; }
void main() { cls r; r = f(); }
```

La execuția programului de mai sus:

- a) constructorul de clasă se apeleză o dată, iar cel de copiere nici o dată;
- b) constructorii de clasă și de copiere se apeleză fiecare câte o dată;
- c) constructorul de clasă nu se apeleză nici o dată, iar cel de copiere o dată;
- d) constructorul de clasă se apeleză de două ori, iar cel de copiere o dată;
- e) constructorii de clasă se apeleză o dată, iar cel de copiere se apeleză de două ori;

17.

```
#include <iostream.h>
class cls
{
    public: cls() { cout << "\n Constructor"; }
            cls( cls &c) { cout << "\n Constructor copiere"; }
};
cls &f() { static cls c; return c; }
void main() { cls r; r = f(); }
```

La execuția programului de mai sus:

- a) constructorul de clasă se apeleză o dată, iar cel de copiere nici o dată;
- b) constructorul de clasă și de copiere se apeleză fiecare câte o dată;
- c) constructorul de clasă nu se apeleză nici o dată, iar cel de copiere o dată;
- d) constructorul de clasă se apeleză de două ori, iar cel de copiere nici o dată;
- e) constructorii de clasă se apeleză o dată, iar cel de copiere se apeleză de două ori.

18.

```
#include <iostream.h>
class cls { public: ~cls() { cout << "Destructor"; } };
cls f() { cls c; return c; }
void main() { { cls r; r = f(); } }
```

În programul de mai sus, destruitorul de clasă:

- a) se apelează o dată;
- b) nu se apelează nici o dată;
- c) se apelează de două ori;
- d) se apelează de trei ori;
- e) este eronat

19.

```
#include <iostream.h>
class cls
{ public:
    auto int i;
    static int s;
    register int r;
    extern int e;
};
```

Care din clasele de memorie 1- automatic, 2- static, 3-register, 4 – extern, nu pot fi aplicate membrilor unei clase de obiecte ?

- a) 3+4
- b) 1+2
- c) 1+2+3
- d) 4
- e) 1+3+4

20.

```
#include <iostream.h>
class cls { public: static int s; };
int cls::s=0;
void main() { int i=7; cls::s=i; cout << cls::s; }
```

Utilizarea lui s este:

- a) ilegală, deoarece nu există nici un obiect creat;
- b) ilegală, deoarece s este incomplet specificat;
- c) corectă, deoarece membrii statici există înainte de a se crea obiecte din clasă;
- d) ilegală, deoarece variabilele statice pot fi doar *private*;
- e) ilegală, deoarece s este dublu definit, în clasă și în afara ei;

21.

Fiind date clasele :

```
class ex1 ;
```

```
class ex2 {
public:
    friend class ex1 ;
    ...
};

class ex1 { ... };
```

Relația dintre clase este :

- a) clasa ex1 este *friend* al clasei ex2;
- b) clasa ex2 este *friend* al clasei ex1;
- c) relația de prietenie este reciprocă;
- d) eronată, pentru că numai funcțiile pot fi *friend*;
- e) eronată, deoarece se face în secțiunea publică.

22.

O metodă statică a unui obiect se caracterizează prin faptul că :

- a) nu primește pointerul la obiect *this*;
- b) poate fi apelată doar de către metodele obiectului;
- c) folosește numai datele private;
- d) nu poate fi definită decât *inline*;
- e) folosește numai datele publice.

23.

Fie clasa:

```
class persoana {
    int npers;
    char numep[50];
public :
    persoana ();
    int nr_persoane() {return npers;} }
```

Dacă se dorește ca variabila *npers* să conțină numărul tuturor obiectelor care aparțin clasei persoana atunci ea trebuie definită ca variabilă:

- a) *friend*;
- b) *virtuală*;
- c) *statică*;
- d) *inline*;
- e) *register*.

24.

Fie programul:

```
class c { int a;
public :
    void init( int nr=0) { a=nr; }
```

```

int inc() {return a++;}
};

typedef int (c::*PM)();

void main() {
    c a,*pa=&a;
    PM f=c::inc;
    a.init(1);      pa->init(3);
    int i=(a.*f)(), j=(pa->*f)();    }

```

Cu ce valori sunt initializate variabilele *i* și *j*?

- a) i=1,j=3 ;    b) i=4,j=5 ;    c) i=2,j=4 ;    d) i=1,j=2 ;    e) i=3,j=4.

25.

Fiind dată următoarea secvență:

```

class c { int a;
public :
    void cit();    void prel();    void afis(); } ;
void main()
{ void ( c::*v[ ] ) () = { c::cit, c::prel, c::afis };    }

```

*v* reprezintă:

- a) pointer la metodă;                 b) vector de pointere la metode;  
 c) vector de obiecte ale clasei *c*;  
 d) vector de pointere la obiecte ale clasei *c* ;  
 e) o declarație eronată sintactic.

26.

```

#include <iostream.h>
struct persoana
{
    char nume[50];
    struct copil { char prenume[10]; int virsta; } c[3];
} p1,*pp;
void main()
{
    pp=&p1; p1.c[0].virsta=5;
    cout << pp->c->virsta;           // 1
    cout << pp->c[0].virsta;         // 2
    cout << (pp->c+1)->virsta;     // 3
    cout << ((*pp).c)->virsta;     // 4
    cout << (*pp).c[1].virsta;       // 5
}

```

```

cout << (*pp).c->virsta;           // 6
}

```

Care din expresiile:

```

pp->c->virsta;                  // 1
pp->c[0].virsta;                // 2
(pp->c+1)->virsta;            // 3
((*pp).c)->virsta;             // 4
(*pp).c[1].virsta;              // 5
(*pp).c->virsta;               // 6

```

sunt corecte?

- a) 2+5                 b) toate                 c) 2+5+6                 d) 1+2+5+6                 e) 1+2+3+5+6

27.

```

#include <string.h>
#include <iostream.h>
class copil
{
    int virsta;
public:
    char prenume[20];
    copil( int v = 1, char *pren="Puiu") : virsta(v)
    { strcpy(prenume,pren); }
class persoana
{
    int virsta;
public:
    copil c;
    persoana(int v1=18, int v2=1, char pren[]="Puiu") :
        virsta(v1), c( v2,pren ) {}
};
void main()
{
    persoana p( 30,5, "Sandu");
    cout << p.c.prenume << " are " << p.c.virsta << " ani";    }

```

- a) programul afișează "Puiu are 0 ani"  
 b) programul afișează "Sandu are 5 ani"  
 c) obiectul *persoana* nu are acces pe zona private a clasei *copil*, inclusă  
 d) programul este eronat, clasa *copil* trebuie să fie declarată în clasa *persoana*

- e) nu este permisa includerea unor obiecte în altele

28.

Programul:

```
#include <iostream.h>
class C { public: int x,y; C( int i = 0, int j = 0 ) : x( i ), y( j ){} }
void main() { C a,b,c[ ] = { C(1,1), C(2,2), a }; cout << c[2].x; }
```

- a) inițializează incorect un vector de obiecte  
 b) afișează 1;  
 d) declară incorect un vector de obiecte  
 c) afișează 2;  
 e) afișează 0

29.

Fiind dată următoarea secvență:

```
#include <iostream.h>
class c
{ int a;
public :
    void cit(){cout << "cit" << endl;};
    void prel(){ cout << "prel" << endl;};
    void rez(){cout << "rez" << endl;};
};
void main()
{
    c ob;
    void ( c::*v[ ] ) () = { c::cit, c::prel, c::rez };
    void ( c::*x )[3](); x=&v; (ob.*x[0][2])();
}
```

În declarația `void ( c::*x )[3]();` `x` reprezintă:

- a) pointer la vector de pointeri la metode ale clasei `c`;  
 b) pointer la o metodă a clasei `c`;  
 c) vector de pointeri la metode ale clasei `c`;  
 d) vector de pointeri la funcții ce returnează obiecte de tip `c`;  
 e) pointer la funcție ce primește ca parametru un vector de obiecte de tip `c` și returnează `void`.

30.

```
#include <iostream.h>
class X
{
    int x;
public:
```

```
X(int n=0):x(n){}
friend class Y;
```

```
};

class Y
{
    int y; friend class Z;
};

class Z
{
public: void f(X ox) { cout << ox.x; }
};

void main() { X ox; Z oz; oz.f(ox); }
```

- a) apelul funcției `f()` nu concordă cu prototipul  
 b) `ox.x` nu este accesibil, căci atributul `friend` nu este tranzitiv  
 c) clasa `Y` acționează ca mediator de acces între clasele `Z` și `X`  
 d) relația de `friend` este reciprocă  
 e) variabila `ox.x` este de acces `public`

31.

Se dă clasa :

```
class c { double a, b ;
public :
    friend c operator + (c &, double );
    friend c operator + (double, c & );
};
```

`operator+` se suprăîncarcă în două moduri deoarece :

- a) se permite în acest fel comutativitatea operației de adunare între un obiect și un `double`;  
 b) există doi operatori în limbaj, unul unar (de semn) și altul binar pentru adunare;  
 c) se permite în acest fel asociativitatea operației de adunare a obiectelor;  
 d) operația este eronată, deoarece se folosesc funcții `friend`;  
 e) descrierea este eronată, deoarece în aceste cazuri se folosesc doar metode statice.

32.

```
#include <iostream.h>
class ex1;
class ex2
{
    int a;
public :
    int b; friend class ex1 ;
```

```

protected:
    int c;
};

class ex1 { public: void f(ex2 &ob){ cout << ob.a << ob.b << ob.c; } };
void main() { ex1 ob1; ex2 ob2; ob1.f(ob2); }

Precizați care din afirmațiile de mai jos este cea corectă.

```

- a) *f()* are acces la toți membrii clasei *ex2*;  
b) *f()* are acces doar la variabila publică *b*, deoarece declarația de *friend* este dată în domeniul public;  
c) *f()* are acces doar la variabilele publice și *protected* (variabilele *b* și *c*);  
d) *f()* nu are acces la nici una din variabilele *a,b,c*.  
e) funcția *f()* este incorrect apelată.

33.

```

#include <iostream.h>
class C
{
public:
    int x;
    C(int v):x(v) { }

    double operator+( C &c, double d) { return c.x+d; }
    double operator+( double d, C &c) { return c.x+d; }

    void main() { C c(5); cout << 2 + c + 3; }
}

```

- a) supraîncărările operator+() trebuiau declarate *friend* în clasa *C*  
b) adunarea obiectelor cu double nu este comutativă  
c) programul afișează 10  
d) supraîncărările operator+() nu se justifică, deoarece au același cod;  
e) adunarea obiectelor cu double nu este asociativă (supraîncărările trebuiau să returneze referințe)

34.

Fiind dat programul:

```

#include <iostream.h>
class ex1 { int x; int y; };
class ex2 { int x; static int y; };
int ex2::y;
void main()
{ ex1 ob1; ex2 ob2; cout << sizeof(ob1)<< " " << sizeof(ob2); }

```

Care afirmație este adevărată ?

- a) *sizeof(ob1) < sizeof(ob2)*      b) *sizeof(ob1) > sizeof(ob2)*  
c) *sizeof(ob1) == sizeof(ob2)*  
d) *sizeof* nu se aplică obiectelor, ci claselor  
e) nu se poate aplica operatorul *sizeof()* unui obiect fără a fi supraîncărcat.

35.

```

#include <iostream.h>
class persoana
{
    int virsta; int salariu;
    friend ostream & operator<<(ostream &out,persoana p)
    { out << p.virsta << " " << p.salariu; return out; }

    persoana(int v) : virsta(v), salariu(0) { }
    persoana() : virsta(0), salariu(0) { }

    void main() { persoana p=1; cout << p; }
}

```

Programul de mai sus:

- a) afișează 1 0      b) afișează 0 0  
c) conține erori de compilare      d) afișează 1 1      e) afișează 0 1

36.

```

#include <iostream.h>
class cls { public: int x; friend void f(cls &c) { } };

void main()
{ const cls c; f(c); }

```

- a) *f()* nu poate fi apelată cu referință de obiect constant;  
b) *f()* este corect apelată, deoarece nu modifică obiectul *c*;  
c) *f()* este incorrect definită în interiorul unei clase;  
d) *f()* este incorrect apelată, deoarece nu se precizează obiectul care a apelat-o;  
e) *f()* este eronat definită deoarece atributul *friend* nu se aplică decât claselor.

37.

Care din variantele de folosire a obiectelor și pointerilor de obiecte constante sunt corecte ?

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

```

class C
{
    public:
        int x;
        C(int v=0): x(v){}
        int get_x() { return x; }
};

void main()
{
    C c1, c2;
    const C c3(10);
    const C *p1 = &c1;
    C * const p2=&c2;
    *p1 = c2;           // varianta 1
    *p2 = c1;           // varianta 2
    c3.get_x();          // varianta 3
}

```

- a) doar varianta 1  
b) doar variantele 2 si 3  
c) doar varianta 2  
d) toate variantele  
e) doar variantele 1 si 3

38.

```

#include <iostream.h>
class cls
{
    int x;
    friend void f(cls);
};
void f(cls c) { cout << c.x<<endl; }
void f(cls c, int i) { cout << c.x<<endl; }
void main() {   cls c; f(c); f(c,1); }

```

- a) friend acționează implicit pentru toate versiunile lui f  
b) nu se pot declara friend funcții ce returnează void  
c) fiecare versiune a unei funcții poate fi declarată friend  
d) funcțiile supraîncărcate nu pot fi declarate friend  
e) exemplul funcționează fără nici o modificare

39.

```

class X
{
    public:

```

```

X() {}
X(X x) {}

};

void main() { X ox; }

```

- a) clasa X are doi constructori corecți, unul de clasă, altul de copiere  
b) cei doi constructori ai clasei X generează ambiguitate  
c) constructorul de copiere este eronat (acceptat în această formă ar produce recursivitate infinită)  
d) constructorul de copiere poate primi obiectul sursă, prin valoare  
e) constructorul de copiere poate primi obiectul sursă doar în forma:  
const & X

40.

Fiind dată următoarea clasă:

```

class ex { int a,b,c; public :int operator . () ;};

Funcția realizează supraîncărcarea operatorului ". " ?
```

- a) da, deoarece returnează un *int* și există o dată de tip *int* în cadrul obiectului;  
b) nu, deoarece operatorul . se supraîncarcă numai printr-o funcție *friend*;  
c) nu, deoarece nu există o variabilă de tip *struct* în obiect;  
d) nu, deoarece operatorul . nu poate fi supraîncărat;  
e) da, respectă regulile de supraîncărcare a operatorilor.

41.

Fie clasa :

```

class v {
    int v[50], nr_c; // vectorul și numarul de componente
public :

```

```

    int& operator[](int i) { return i>0 && i<nr_c ? v[i] : v[nr_c-1]; }
};

```

Supraîncărcarea operatorului [] realizează:

- a) modificarea valorii elementului ;      b) determinarea valorii elementului;  
c) modificarea și determinarea valorii elementului;  
d) o definire eronată, deoarece returnează o referință ;  
e) o altă operație decât cele propuse.

42.

Dupa o declarație:

```
#include <iostream.h>
class vector
{
public:
    double x[10];
    double & operator[](int i) { return x[i]; }
};

v;
```

Care din urmatoarele combinații de adresări sunt corecte:

```
v[4] = 4; cout << v[4] << " ";           // varianta 1
v.x[4]=100; cout << v.x[4] << " ";       // varianta 2
v[4].x[4]=100; cout << v[4].x[4] << " "; // varianta 3
```

- a) doar variantele 1 și 2
- b) doar varianta 1
- c) doar varianta 2
- d) toate variantele
- e) doar variantele 2 și 3

43.

În clasa :

```
class M {
    int v[50][50], nr_l, nr_c; // matricea, nr. de linii și coloane
public:
    int& operator[](int i, int j)
        { return i>0&&i<nr_l&&j>0&&j<nr_c?v[i][j]:v[nr_c-1][nr_l-1]; }
};
```

Supraîncărcarea operatorului [] se face corect în acest caz ?

- a) da;
- b) nu, metoda neacceptând doi parametri;
- c) nu, deoarece metoda returnează o referință;
- d) nu, deoarece trebuia definită ca funcție *friend*;
- e) nu, deoarece metoda trebuia definită statică.

44.

```
#include <iostream.h>
class vector
{
    int *pe, nr_c;
public:
    operator int () { return nr_c; }
    vector(int );
};
```

```
vector:: vector(int n)
{
    pe=new int[n];nr_c=n;
    while(n--) pe[n]=n;
}
void f( int i) { cout << i << endl; }
void main()
{
    vector x(10);          f(x);
}
```

Programul afișează:

- a) 10 ;
- b) 9;
- c) numerele de la 1 la 10;
- d) numerele de la 0 la 10;
- e) numerele de la 0 la 9.

45.

```
#include <iostream.h>
class cls
{
public: void *operator new( size_t dim)
        { cls *po = new cls[dim]; cout << "\n Aloca obiect";
        return po; }
};
```

În exemplul anterior, suprascrierea operatorului *new* afișează “Aloca obiect”:

- a) de două ori, câte o dată pentru fiecare pointer;
- b) o dată, căci pentru vectori se folosește varianta inițială a lui *new*;
- c) de zero ori, căci nu se folosește niciodată în acest context varianta suprascrisă;
- d) de șase ori, câte o dată pentru fiecare obiect alocat;
- e) de zero ori, căci operatorul *new* nu se suprascrue.

46.

Fiind dată clasa:

```
#include <iostream.h>
class persoana
{
public:    float salariu;
           persoana(float s=0): salariu(s){ }
operator float() { return salariu; }
float indexare(float coef)
           { return salariu *(1+coef/100); }
};
```

```
void main()
{ persoana p(100); cout << p.indexare(p); }
```

Apelul funcției *indexare()*:

- a) generează eroare, nexistând o supraîncărcare ce primește obiect persoana în intrare;
- b) generează eroare, prin folosirea recursivă a obiectului p;
- c) folosește *cast*-ul definit de programator;
- d) se traduce prin *indexare( int )*;
- e) se traduce prin *indexare( void )*;

47.

```
#include <iostream.h>
class persoana { public: int virsta; persoana(int v=30) : virsta(v){} };
class profesor
{
    public:int virsta;           profesor(int v=20) : virsta(v){}
    operator persoana()         { persoana p; p.virsta = virsta; return p; }
};
persoana f(persoana p) { p.virsta++; return p; }
void main()
{
    persoana p; f(p); cout << endl << p.virsta;
    profesor prof; f(prof); cout << endl << prof.virsta;
}
```

Vârstele afișate la rularea programului de mai sus sunt:

- a) 31 21, datorită incrementărilor din funcție;
- b) 30 20, ambele obiecte fiind temporare, la transmiterea prin valoare;
- c) 31 20, *profesor* fiind temporar, datorită conversiei prin *cast*;
- d) 30 21, *persoana* fiind temporar, datorită conversiei prin *cast*;
- e) 0 0, deoarece pentru obiecte temporare s-au apelat constructori de copiere.

48.

Programul

```
#include <iostream.h>
class persoana
{
    int virsta;
    public:
        persoana(int v = 18): virsta(v){}
```

```
operator int( ) { return virsta; }
persoana& operator++( )
{ virsta++; return *this; }
persoana operator++(int)
{ persoana aux = *this; virsta++; return aux; }
```

```
};
```

```
void main()
```

```
{
```

```
persoana p(20);
int x = p++, y = ++p;
cout << "x = " << x << " y = " << y << endl;
```

} afișează:

- a) x = 20 y = 20, pre / postincrementarea neoperând în cazul obiectelor
- b) x = 20 y = 21, datorită post - și respectiv preincrementării
- c) x = 21 y = 22, datorită incrementărilor în cascadă
- d) x = 20 y = 22, datorită dublei incrementări a lui p
- e) x = 19 y = 20, datorită valorii implicate a parametrului din constructor

49.

Având o clasă definită astfel:

```
class ex { int a ;
public :
```

```
friend istream& operator >> (istream& , ex & );
```

Funcția friend istream& operator >> (istream& , ex & ) realizează supraîncărcarea operatorului >> ?

- a) da, pentru implementarea operatiei de deplasare la dreapta;
- b) da, în scopul citirii datelor obiectului, de la tastatură;
- c) da, în scopul afișării datelor obiectului;
- d) nu, operatorul >> nu se poate supradefini;
- e) nu, neavând suficienți parametri.

50.

Potibilitatea definirii unui obiect ca fiind o extensie a altuia este:

- |   |                       |
|---|-----------------------|
| a) dată de încapsulare;                 | b) dată de moștenire; |
| c) dată de polimorfism;                 |                       |
| d) dată de existența claselor virtuale; | e) nepermisă.         |

51.

Fie clasa *D*, derivată public din clasa *B*. În acest caz, o metodă publică din clasa *D* poate accesa o dată din secțiunea privată a clasei *B*?

- a) da, în orice condiții;                            b) da, dar fără să-i modifice valoarea;
- c) da, doar dacă metoda se definește *inline*;            d) nu;
- e) nu este permisă derivarea publică a unei clase dintr-o altă clasă.

52.

```
#include <iostream.h>
class persoana
{
    private:      int virsta ;
    public :
        persoana (int v = 20) :virsta(v) {}
        int spune_virsta() { return virsta ; }
};
class muncitor : public persoana
{
    public: int f() { return spune_virsta(); }      };
void main () { muncitor m ; cout <<m.f(); }
```

Funcția *f()* din clasa *muncitor* are acces pe o zonă privată din clasa de bază?

- a) da, doar printr-o metodă *protected*, moștenită;
- b) da, dar fără să-i modifice valoarea;
- c) da, dar prin metode *public* sau *protected*, moștenite;
- d) nu;
- e) da, doar printr-o metodă *publică*, moștenită;

53.

Fie declaratia :

```
#include <iostream.h>
class c1 { int a; };
class c2 : public c1
{
    public:
        int b;
        void scrie_a(){ cout << "a = " << a << endl;}
};
void main()
{   c2 ob; ob.scrie_a (); }
```

Selectați afirmația corectă:

- a) funcția *scrie\_a()* nu are drept de acces asupra unui membru privat;

- b) programul afișează valoarea lui *a*;
- c) funcția de acces are doar acces *read-only* asupra unui membru privat;
- d) derivarea publică este incorrect realizată
- e) prin derivare publică, accesul la membrii moșteniți devine public.

54.

Fie declarația :

```
class c1 { /* ... */ };
class c2 : public c1 /* ... */;
```

Atunci clasa *c2* față de *c1* este:

- a) derivată;      b) de bază;                    c) friend;            d) virtuală;
- e) declarată eronat, în loc de semnul : trebuie pus operatorul de rezoluție :: .

55.

```
#include <iostream.h>
class B
{
    int x;
    friend void f(B);
    public: B(int i=0):x(i){}
};
class D :public B
{
    int y;
    public: D(int i=0, int j=1):B(i),y(j){}
};
void f(B b) { cout << b.x<<endl; }
void main() { B b; D d; f(b); f(d); }
```

O funcție declarată *friend* în clasa de bază:

- a) rămâne *friend* în clasa derivată, pentru partea moștenită
- b) are acces pe toată clasa derivată
- c) nu are acces pe zona privată a clasei derive
- d) are acces pe zonele *public* și *protected* ale clasei derive
- e) nu are acces pe zonele private și *protected* ale clasei derive

56.

```
#include <iostream.h>
class D;
class B
{
    int x;
```

```

friend void f(B, D);
public: B(int i=0):x(i){}
};

class D :public B
{
    int y;
public: D(int i=0, int j=1):B(i),y(j){}
};

void f(B b, D d) { cout << b.x << " " << d.y << endl ; }
void main() { B b; D d; f(b, d); }

```

O funcție declarată friend în clasa de bază:

- a) rămâne friend pentru întreaga clasă derivată
- b) are acces pe toate zonele moștenite din clasa de bază
- c) pierde drepturile de acces, dacă derivarea se face private
- d) pierde drepturile de acces, dacă derivarea se face private sau protected
- e) nu are acces pe zonele private și protected ale clasei derivate

57.

Programul

```

#include <iostream.h>
class persoana
{
    int vîrstă;
public:
    persoana(int v=18): vîrstă(v){}
    friend int get_vîrstă(persoana p) { return p.vîrstă;}
};

class student: public persoana
{
    float media;
public:
    student(int v = 18, float m = 0.0):persoana(v),media(m) {}
};

void main() { student s(20, 9.50); cout << get_vîrstă(s); }

```

- a) semnalează eroare, vîrstă student nefiind accesibilă printr-o funcție friend în clasa persoana
- b) semnalează eroare, neexistând prototipul int get\_vîrstă( student )
- c) semnalează eroare, membrii private în clasa de bază fiind totdeauna inaccesibili în clasa derivată

- d) afișează 18, vîrstă implicită a obiectului persoana rezultat din conversia cerută de adaptarea la prototipul funcției get\_vîrstă()
- e) afișează 20, vîrstă studentului primit ca parametru în funcția de acces

58.

Fiind date două clase :

```

class ex1 { ... };
class ex2 : public ex1 { ... };

```

Atunci în funcția :

```

void main () { ex1 *p1; ex2 *p2; p1=p2; }
Atribuirea p1 = p2 este posibilă ?

```

- a) nu, deoarece sunt variabile de tipuri diferite;
- b) nu, deoarece trebuie folosit operatorul cast;
- c) da, deoarece p1 este pointer la bază și p2 este pointer la derivat;
- d) nu, deoarece nu sunt permise atribuiri între obiecte, care sunt în relații de moștenire;
- e) da, doar ce ambii sunt pointeri.

59.

Având clasele:

```

class cb { protected: int a; public: cb() {a=7;} };
class cd : public cb { int b; public: cd() {b=a+3;} };

```

Declarația:

```

void main() { cd od; }

```

determină ca variabila b să ia valoarea 10?

- a) da, deoarece mai întâi se execută constructorul bazei și apoi cel al clasei derivate;
- b) nu, deoarece mai întâi se execută constructorul clasei derivate și apoi cel al clasei de bază;
- c) nu, deoarece ar trebui transmis constructorul clasei derivate prin parametru;
- d) da, deoarece clasa de bază cb nu are destructor;
- e) nu, deoarece nu a fost definit în prealabil un obiect al clasei cb.

60.

Fie o clasa CD care moștenește clasa CB, ambele clase având cîte un destructor. Să se precizeze, în cazul definirii unui obiect de tipul CD, care destructor se va executa primul ?

- a) al clasei *CD*;  
 b) al clasei *CB*;  
 c) ai clasei care are definiți mai mulți constructori ;  
 d) se va preciza în program în mod obligatoriu, altfel rezultă eroare;  
 e) nu există un criteriu precis, acest lucru se face aleator.

61.

```
#include <iostream.h>
class B
{
    int x;
public:
    B (int v) : x(v){}
    int get_x() { return x; }
};
class D : private B
{
    int y;
public:
    D (int v) : B(v){}
    int get_x() { return B::get_x(); }
};
void main()
{   D d(10); cout << d.get_x(); }
```

- a) variabila *x* nu este accesibila deoarece derivarea lui *D* este private  
 b) variabila *x* nu este accesibila deoarece este private in *B*  
 c) transferul de sarcini între constructori nu este permis, *B()* devenind private  
 d) constructorul *D* este apelat eronat  
 e) programul afișează 10

62.

În ipoteza că există clasele de bază *B1*, *B2*, *B3* și *B4*, declarația:  
 class *D*: public *B1*, private *B2*, protected *B3*, *B4* { int *m*; };

este:

- a) incorectă, nexistând derivare de tip *protected*;  
 b) incorectă, pentru că la *B4* nu se specifică tipul derivării;  
 c) perfect validă;  
 d) incompletă, deoarece clasa *D* nu are constructori;  
 e) incompletă, deoarece clasa *D* nu are metode specifice.

63.

```
#include <iostream.h>
class B { public: int fb() { return 1; } };
class D: public B { public: int fd() { return 2; } };
void main()
{ int (B::*pf)( ) = B::fb, (D::*pf)( ) = D::fd; pf=pf; }
```

În exemplul de mai sus, este posibilă atribuirea *pf*=*pf* ?

- a) da, singura direcție de conversie admisă fiind de la derivat către bază;  
 b) nu, singura direcție de conversie admisă fiind cea inversă, *pf*=*pf*;  
 c) da, obiectele de bază pot fi convertite în obiecte derivate;  
 d) da, doar dacă *pf* pointează spre o funcție moștenită din *B*;  
 e) nu, conversiile de pointeri spre funcții membre nu sunt admise în nici un sens.

64.

Fie dată următoarea ierarhie

```
class B { /*.....*/ };
class D1:B { /*.....*/ };
class D2:B { /*.....*/ };
class M1:D1, public D2 { /*.....*/ };
class M2:virtual D1, virtual public D2 { /*.....*/ };
```

Precizați care afirmații sunt corecte:

- 1) clasa *M1* va moșteni un obiect de tip *B*;
- 2) clasa *M1* va moșteni două obiecte de tip *B*;
- 3) clasa *M2* va moșteni un obiect de tip *B*;
- 4) clasa *M2* va moșteni două obiecte de tip *B*;

- a) 2+3      b) 1+2      c) 2+4      d) 1+3      e) 1+4

65.

```
#include <iostream.h>
class B1 { int x; };
class B2 { int y; };
class B3 { int z; };
class B4 { int t; };
class D: public B1, private B2, protected B3, B4 { public: int m; };
void main()
{
    D d;
    cout << d.m; // varianta 1
    cout << d.x; // varianta 2
```

```

cout << d.y; // varianta 3
cout << d.z; // varianta 4
cout << d.t; // varianta 5
}

```

Variantele care permit accesul la variabile, pentru afișare sunt:

- a) 1 + 2 + 4 + 5
- b) 1 + 2
- c) 1 + 2 + 4
- d) 1
- e) 1 + 2 + 5

66.

```

#include <iostream.h>
#include <stdarg.h>
class cls
{
    public: int a; cls(int v):a(v){ } void g(int); };
class sbcls
{
    int y; public: sbcls(int v=2):y(v) { } void f(int,...); };
void cls::g(int n)
{
    sbcls s; s.f(1,this); }
void sbcls::f(int t,...)
{
    va_list lv; va_start(lv,t);
    cls *h;
    if(t == 1) h = (cls *) va_arg(lv,void *);
    cout << h->a; va_end(lv);
}
void main() { cls t(7); t.g(1); }

```

În contextul de mai sus, *this* reprezintă:

- a) pointer la un obiect de tip cls;
- b) obiect de tip cls;
- c) pointer la un obiect de tip sbcls;
- d) obiect de tip sbcls;
- e) pointer la ferestra principala a aplicației.

67.

Un instrumentul performat prin care se realizează polimorfismul îl constituie :

- a) funcțiile friend;
- b) funcțiile inline;
- c) constructorii;
- d) funcțiile virtuale;
- e) destructorii.

68.

Fiind dat programul:  
`#include <iostream.h>`

```

class c1 { public: int x; void f() {} };
class c2 { public: int x; virtual void f() {} };
void main()
{
    c1 o1; c2 o2; cout << sizeof(o1) << " " << sizeof(o2);
}

```

Care dintre următoare afirmațiile sunt adevărate?

- a) sizeof(o1) < sizeof(o2)
- b) sizeof(o1) > sizeof(o2)
- c) sizeof(o1) = sizeof(o2)
- d) nu se pot declara obiecte de tip c2, deoarece f() este virtuală pură
- e) nu se pot declara obiecte de tip c1 și c2, deoarece f() nu are corp executabil

69.

```

class c { int a;
public:
    virtual void metoda1() = 0;
    virtual void metoda2(int) = 0; };

```

```

void main()
{
    c *pob; // declaratia 1
    c ob; // declaratia 2
    c *vpob[5]; // declaratia 3
    c vob[5]; // declaratia 4
}

```

Declaratiile admise în acest caz sunt:

- a) 1+2+3
- b) 1+2;
- c) 1 + 3;
- d) 2+4;
- e) 1+2+3+4.

70.

```

class c { int a;
public:
    virtual void metoda1()=0;
    virtual void metoda2(int)=0; };

```

- a) este o clasă virtuală;
- b) este o clasă incomplet definită;
- c) este o clasă abstractă;
- d) realizează incorect definirea funcțiilor virtuale;
- e) este o clasă pură.

71.

Fie clasa:

```
class C { int a;
public:
    virtual int f()=0;
};
```

Metoda *f* este:

- a) virtuală;    b) virtuală pură;    c) declarată eronat sintactic;  
d) abstractă;    e) friend.

72.

Clasele ce permit parametrizarea tipurilor de date asociate unor variabile membru sunt numite:

- a) *virtuale*;    b) *friend*;    c) *template*;  
d) *derivate*;    e) *complexe*.

73.

Se dă clasa:

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

Considerând declarația *A \*p*; care este diferența dintre ceea ce realizează instrucțiunile: *p = new A;* și *p = (A\*) malloc(sizeof(A));*

- a) nu există nici o diferență, ambele au ca scop alocarea memoriei pentru un obiect al clasei *A*;  
b) nu există nici o diferență, ambele au ca scop alocarea memoriei și execuția constructorului implicit, pentru un obiect al clasei *A*;  
c) prima instrucțiune alocă memorie și execută și constructorul implicit în timp ce a doua alocă doar memorie;  
d) prima instrucțiune alocă memorie pentru obiect, în timp ce a doua alocă memorie și execută și constructorul implicit;  
e) prima este incorectă sintactic, deoarece lipsește operatorul *cast* (*A\**) înaintea operatorului *new* iar a doua instrucțiune este corectă.

74.

```
template <class INTEGER, int k>
class A{
    INTEGER v[k];
};
```

Sablonul clasei *A*:

- a) parametrizează tipul elementelor masivului *v*;  
b) parametrizează tipul elementelor masivului *v* și numărul de elemente;  
c) parametrizează numărul de elemente a masivului *v*;  
d) este incorrect definit, deoarece tipul *INTEGER* nu există în limbajul C;  
e) este incorrect definit, deoarece şablonanele de clase nu parametrizează decât tipuri de date și nu variabile.

75.

În programul:

```
#include <iostream.h>
struct test
{
    int x;
public: int y;
private: int z;
protected: int v;
} t;
void main() { cout << t.v; cout << t.x; }
```

- a) variabila membru *x* este inaccesibilă;  
b) variabila membru *v* este inaccesibilă;  
c) structurile nu pot conține domenii de acces  
d) structurile nu pot conține domeniul de acces *protected*;  
e) programul este corect în totalitate;

76.

```
#include <iostream.h>
class I
{ public: int & g() { static int i = 10; return i; } };
class C
{ public: I & f() { static I o; return o; } };
void main() { C c; cout << c.f().g(); }
```

Secvența de mai sus:

- a) folosește incorect atributul static într-o clasă  
b) adresează eronat un membru în clasă;  
c) afișează 0;  
d) afișează 10  
e) folosește incorect referința de obiect.

```

77.
#include <iostream.h>
class matrice
{
public:
    int a[2][2];
    int * operator[](int i) { return a[i]; }
    matrice()
    { a[0][0]=00; a[0][1]=01; a[1][0]=10; a[1][1]=11; }
};

void main()
{
    matrice m;
    cout << "\n " << m[1][1]; // varianta 1
    cout << "\n " << m.a[1][1]; // varianta 2
    cout << "\n " << m.operator[](1)[1]; // varianta 3
}

```

Precizați care variante afișează corect elementul a[1][1] al matricei:

- a) 1+2+3;    b) 1+2;    c) 1;    d) 2;    e) 2+3;

## Răspunsuri

1 / b

Dacă nu apar explicit domeniile de acces, implicit toți membrii clasei sunt privați.

2 / c

O clasa dă structura generală a unui obiect; variabilele de tipul introdus prin clasa respectivă sunt instanțe ale clasei și se numesc obiecte.

3 / a

Funcțiile reduse ca dimensiune, cu o mare stabilitate în timp și cu frecvență mare de apel, pot fi declarate *inline*, apelul lor înlocuindu-se cu codul executabil. Implicit, compilatorul încearcă să expandeze inline funcțiile descrise în interiorul clasei.

4 / a

Chiar când apare descrisă în interiorul clasei (lucru acceptat de multe compilatoare), o funcție *friend* rămâne independentă de clasă, ea primind doar drepturi speciale de acces la membrii clasei. Pentru a lucra pe obiecte, ea primește obiectul ca parametru, nu implicit prin pointer *this*, cum este cazul funcțiilor membre nestatiche.

5 / c

Atunci când programatorul nu scrie un constructor de copiere, compilatorul pune el unul care inițializează obiectul prin copiere bătă de bătă dintr-un obiect existent.

6 / c

Numele clasei este identificator unic și nu poate fi folosit pentru metode sau funcții independente; folosit pentru o funcție cu tip returnat, generează eroare de compilare, deoarece un constructor nu poate avea tip returnat, obiectul clasei fiind rezultatul implicit al rulării constructorului.

7 / c

Destructorul este apelat implicit la distrugerea obiectelor create într-un bloc; semnul ~ care prefațează numele clasei pentru a desemna destructorul clasei, în general nu crează confuzii cu supraîncărcarea operatorului ~ sau cu negarea pe biți (~).

8 / a

Când operatorul = apare la declarația unui obiect atunci se execută constructorul de copiere; construcția c b=a; se mai poate scrie c b(a); ceea ce sugerează mai clar execuția constructorul de copiere.

9 / d

Fiind declarat un vector de trei obiecte, tot atâtea se vor și distruge.

10 / d

Alocarea dinamică de memorie pentru obiecte cu operatorul new determină și execuția constructorului iar dezalocarea cu delete determină execuția destructorului. Fiind vorba de dezalocarea cu delete a masivului de trei obiecte se va executa destructorul de trei ori.

11 / b

Fiind funcție friend poate accesa toți membrii, prin intermediul referinței la obiectul primit ca parametru în funcție, fără nici o restricție, indiferent în ce secțiune a fost declarată.

12 / b

Aceeași explicație ca la întrebarea 11.

13 / b

La declararea obiectului c se execută constructorul de clasă (implicit) iar la transmiterea obiectului, prin valoare, ca parametru funcției f() se execută constructorul de copiere pentru a se copia parametrul (obiectul) pe stivă.

14 / a

Spre deosebire de întrebarea anterioară, se transmite funcției o referință la obiect, adică o adresă și aceasta se va copia pe stivă, nu obiectul; deci constructorul de copiere nu se execută.

15 / e

Explicația de la întrebarea anterioară rămâne valabilă singura deosebire constă în faptul că se transmite funcției adresa obiectului ceea ce nu implică copierea pe stivă a obiectului, deci nu se execută constructorul de copiere.

16 / d

La declarația obiectului r se execută constructorul de clasă (1); în funcția f() se declară obiectul c pentru care iar se execută constructorul de clasă (2). La returnarea din funcție a obiectului c se execută constructorul de copiere (1).

17 / d

Aceeași explicație ca la întrebarea precedentă doar că se returnează din funcție o referință la obiect ceea ce nu mai determină execuția constructorului de copiere.

18 / d

Destructorul se execută pentru obiectul r (1) pentru obiectul c declarat local în funcția f() (2) iar pentru copia care este construită când se execută return c; se mai execută încă odată destructorul clasei (3).

19 / e

Singura clasă de memorie care se poate asocia unui membru al unei clase este static.

20 / c

Explicația este conținută în răspunsul corect.

21 / a

Drepturile conferite prin friend nu sunt reciproce; clasa care acordă drepturile face declarația de friend pentru o altă clasă sau pentru funcții independente.

22 / a

Metodele statice nu țin de un obiect anume, ci de clasă în genere; în consecință, ele nu primesc implicit pointerul la obiectul curent; când prelucră obiecte, metodele statice primesc obiectele ca parametri expliciti.

23 / c

Variabilele care nu iau valori specifice fiecărui obiect în parte se declară statice; ele aparțin clasei prin faptul că informația conținută se referă la clasă, în ansamblul ei.

24 / e

Prin apelul succesiv al metodei *init()*, *a* va lua valoarea 3. Se apelează metoda *inc()* prin intermediul unui pointer la o funcție membră și *i* va lua valoarea 3 după care *a* devine 4, fiind vorba de o postincrementare, apoi *j* ia valoarea 4 după care *a* devine 5 dar nu mai contează.

25 / b

Din evaluarea declarației se observă că este vorba de un vector de pointeri la metode, lucru confirmat și de modul în care a fost inițializat masivul.

26 / b

Exemplul demonstrează multitudinea formelor de adresare a unui membru dintr-o clasă inclusă într-o altă clasă.

27 / c

O clasă nu dobândește drepturi speciale de acces asupra zonelor protejate ale unei clase pe care o include.

28 / e

La inițializarea obiectelor pot fi folosite obiecte anterior definite; obiectul *a* conține membrii inițializați cu valori implicate; el este folosit în continuare la inițializarea lui *c*[2].

29 / a

Exemplul ilustrează complexitatea declarațiilor din limbajul C++, sporită și prin combinarea celor doi operatori `::` și `*`.

30 / b

Atributul friend nu este simetric și nici tranzitiv; faptul că *Y* este friend pentru *X* și *Z* pentru *Y*, nu înseamnă că și *Z* este friend pentru *X*. Clasa *Y* ar putea acționa ca mediator dacă și-ar folosi efectiv drepturile sale de acces pe *X*, furnizând obiectelor *oz*, la cerere, valoarea lui *ox.x*.

31 / a

Prin funcții *friend* poate fi descrisă și acțiunea unui operator binar, atunci când operandul stâng este de tip fundamental; când operandul stâng este de tip obiect putem descrie acțiunea operatorului și prin funcție membră (nu numai prin funcție *friend*), dar când este de tip fundamental singura modalitate o reprezintă funcțiile *friend*. În acest mod putem descrie operații comutative: obiect cu tip de bază, dar și tip de bază cu obiect.

32 / a

Prin intermediul parametrului de apel (ob2) a metodei *f()* a clasei *ex1* se pot accesa toți membrii clasei *ex2* deoarece clasa *ex1* este clasă friend pentru *ex2*.

33 / c

Deoarece metodele care supraîncarcă operatorul `+` nu accesează partea privată a clasei *C* ele nu trebuie să fie declarate friend; funcțiile sunt corect definite.

34 / b

Membrii statici ai unei clase nu ocupă memorie în cadrul clasei; în consecință, obiectele de tip *ex2* ocupă memorie mai puțină. Operatorul *sizeof()* se aplică atât variabilelor sau constantelor, cât și expresiilor și tipurilor, în general.

35 / a

Pentru inițializarea lui *p*, compilatorul caută să rezolve un apel persoana(*int*) și va identifica prima versiune de constructor. În continuare, constructorul implicit de copiere inițializează noul obiect cu informațiile din obiectul constant.

36 / a

Funcția *f()* nu poate fi apelată cu referință de obiect constant, dacă nu declară și ea referință constantă, sub forma:

```
friend void f(const cls &c) { }
```

37 / c

Atribuirea de la varianta 2 este posibilă pentru că doar pointerul este constant, conținutul zonei de memorie poate fi alterat.

38 / c

Atributul de friend are efect numai pentru versiunea de supraîncărcare pentru care a fost declarat.

39 / c

Constructorul de copiere trebuie să primească obiectul sursă prin referință ( se recomandă chiar referință constantă); compilatoarele verifică acest lucru, deoarece transferând obiectul prin valoare s-ar invoca din nou constructorul de copiere pe stivă, conducând la recursivitate infinită.

40 / d

Operatorul . nu se poate supraîncărca datorită semnificației pe care o are deja în cadrul programării orientate obiect.

41 / c

Returnând o referință la o zonă de memorie, se permite atât modificarea cât și determinarea conținutului ei.

42 / a

În varianta 1 se apelează supraîncărcarea operatorului []; în varianta 2 se califică membrul public x care este de tip masiv după care se accesează o componentă; varianta 3 este incorrectă pentru că din folosirea operatorului [] rezultă o referință la double căreia nu își poate aplica operatorul .

43 / b

Nu se permite supraîncărcarea operatorului [ ] cu doi parametri pentru că în limbajul C el a fost definit astăzi încât să primească doar un singur operand între parantezele drepte.

44 / a

Funcția f() prin definire primește un întreg dar la apel se transmite un vector. Este posibil acest lucru pentru că prin supraîncărcarea operatorului cast, un vector știe să se convertească într-un întreg.

45 / b

Explicația este dată în răspunsul corect.

46 / c

Funcția indexare() prin definire primește un float dar la apel se transmite o persoană. Este posibil acest lucru pentru că prin supraîncărcarea operatorului cast, o persoană știe să se convertească într-un float.

47 / b

Fiind vorba de transfer prin valoare se modifică copiile obiectelor; valorile din obiectele originale rămân neschimbate.

48 / d

Dubla incrementare este realizată prin supraîncărcările operatorului ++ în forma pre și post.

49 / b

Operatorul binar >> este ușual supraîncărcat pentru a realiza intrări la nivel de obiect, continuând într-un fel semnificația atribuită în cazul tipurilor de bază în C++. Acest lucru este confirmat și prin faptul că se primește prin referință un parametru de tip istream și se returnează tot o referință la istream.

50 / b

Un obiect derivat moștenește toate atributele și metodele obiectului de bază ; în plus, el poate adapta metodele moștenite, poate aduce noi metode și caracteristici, apărând deci ca o extensie a obiectului de bază.

51 / d

Prin derivare nu se câștigă drepturi de acces pe clasa de bază, ci eventual se introduc noi restricții prin tipul derivării. Zona privată a clasei de bază este inaccesibilă direct din clasa derivată, dar poate fi accesată indirect, cu funcții de acces public sau protected, moștenite.

52 / c

Zona privată a clasei de bază poate fi accesată indirect, cu funcții de acces public sau protected, moștenite.

53 / a

Membrul a al clasei c1 este implicit privat ; neexistând nici o funcție de acces, el nu este nici direct, nici indirect accesibil.

54 / a

Clasele obținute pornind de la clase existente se numesc deriveate.

55 / a

Atributul de friend se transmite prin derivare, dar drepturile de acces se referă numai la porțiunea din clasa derivată moștenită din clasa de bază (care i-a acordat privilegiile de acces); în consecință f() poate afișa valoarea lui x, atât din obiecte de bază, cât și din obiecte derivate, dar nu poate afișa valoarea lui y.

56 / b

Drepturile de acces conservate prin derivare nu depind de tipul derivării și afectează doar parte din clasa derivată moștenită din clasa de bază; altfel s-ar ajunge la situația în care nu poți proteja datele dintr-o clasă derivată, numai pentru că producătorul clasei de bază a acordat privilegii de acces multor funcții. cout << b.x << " << d.y; nu este acceptat de compilator din cauza lui y inaccesibil, în timp ce cout << b.x << " << d.x; ar fi fost corect.

57 / e

Explicația este dată de faptul că unui obiect de bază îl se poate atribui un obiect al clasei derivate (funcția get\_virsta() a fost declarată având un parametru de tip clasă de bază și a primit la apel un obiect al clasei derivate).

58 / c

Conversia pe direcția "is a" (de la derivat către bază, upcasting) este implicită pentru obiecte, pointeri sau referințe.

59 / a

Explicația se află în răspunsul corect.

60 / a

Regula de execuție a destructorilor este: mai întâi se execută destructorul clasei derivate și apoi cel al clasei de bază (învers ca la constructori).

61 / e

Fiind vorba de moștenire privată, partea publică a clasei de bază se comportă ca și cum ar fi privată pentru clasa derivată, deci se poate accesa prin funcții de acces.

62 / c

63 / b

Explicația se află în răspunsul corect

64 / a

Clasa M1 și M2 moștenește două clase D1 și D2 care au o bază comună B. M1 nu le moștenește virtual și atunci va moșteni două obiecte de bază B unul pe filiera lui D1 altul via D2. Pentru că M2 este derivată virtual din D1 și D2 moștenește doar un singur obiect al clasei B.

65 / d

Toți membrii claselor de bază sunt privați, deci inaccesibili direct dintr-un obiect derivat. Membrul m din clasa derivată (D) este public, deci accesibil direct prin intermediul obiectului derivat d.

66 / a

Fiind folosită într-o metodă a clasei cls, this este un pointer la un obiect de tip cls.

67 / d

Polimorfismul se realizează prin două mecanisme: prin supraîncărcare de funcții sau operatori, respectiv prin virtualizare. Constructorii, ca orice funcție, pot fi supraîncărați, dar ei în sine nu stau la baza realizării polimorfismului.

68 / a

Clasele polimorfe ocupă mai multă memorie, datorită pointerului la tabela de funcții virtuale.

69 / c

Pentru o clasă abstractă nu se pot defini obiecte dar se pot declara pointeri la obiecte, deci și masive de pointeri.

70 / c

O clasă care conține cel puțin o metodă virtuală pură se numește clasă abstractă.

71 / b

Un prototip de funcție virtuală urmat de = 0 este o funcție virtuală pură.

72 / c

Sunt așa numitele şabloane de clase sau clase *template*.

73 / c

Operatorul new pentru tipul class permite pe lângă alocarea dinamică de memorie și executarea unui constructor; dacă nu se specifică nici unul se execută cel implicit. Funcția malloc() face doar alocarea dinamică de memorie.

74 / b

Şabloanele de clasă pot descrie clase parametrice atât la nivelul tipurilor de date pentru unele date membre cât și la nivelul unor constante folosite în cadrul clasei.

75 / b

Ca și la clase, membrii private sau protected dintr-o structură sunt inaccesibili direct din exterior. Spre deosebire de clase, structurile au membrii implicit publici.

76 / d

A nu se confunda semnificația clasei de memorie static atribuită unui membru al clasei cu cea atribuită unei variabile locale dintr-o metodă chiar dacă e vorba de un obiect.

77 / a

Operatorul [] supraîncărcat returnează adresa primului element al unei anumite linii. Aplicând iar operatorul [] pointerului, obținem chiar elementul. În variantele 1 și 3 se apelează în două forme (echivalente) metoda care supraîncarcă operatorul [] și apoi se accesează un element din matrice. În varianta 2 se referă direct un element din matrice acest lucru fiind permis pentru că membrul matrice este public.

## ***INDEX***

---

*Index*

.	14
.*	39
:	18
::	14,215
->	14,29
->*	39
<b>A</b>	
abstractizare	10
adjustfield	143
at	223
<b>B</b>	
bad	145
badbit	144
basefield	143
before	261
begin	224
<b>C</b>	
capacity	225
cin	13,63,134
clasă abstractă	122
clasă de bază	108
clasă derivată	108
clasă virtuală	129
clase cu membri pointeri	22
clase incluse	30
clase template	188
class	11
clear	145
componerea claselor	30,116
componerea şablonelor	209
const	33,46
constantă asociată clasei	19
constante în template	192
constructor de copiere	21,23,111
constructor implicit	18
constructori	17,110,125
copy	247
cout	13,63,134
<b>D</b>	
dec	140
delete	29
deque	226
derivare private	108
derivare protected	108
derivare public	108
derivare virtuală	129
derivarea de şablonane	204
<b>E</b>	
destructor	22
dynamic_cast	264
<b>F</b>	
empty	235
end	224
endl	140
ends	140
eof	154
eofbit	154
equal_range	231
erase	224
explicit	88
<b>G</b>	
fail	145
failbit	144
fill	138
find	230,250
first	233
floatfield	143
flush	140
for_each	248
friend	44,114
fstream	147
fstream.h	147
funcție polimorfică	52
funcție virtuală	116
funcție virtuală pură	121
funcții de acces	27
funcții statice	34
funcții template	186
<b>H</b>	
hex	140
<b>I</b>	
ifstream	63,147
ignore	147
inline	12,16
insert_iterator	246
interfață publică	16
iomanip.h	140
ios	137
ios::app	148
ios::ate	148

---

*Index*

ios::beg	154	multiset	228
ios::binary	148	<b>N</b>	
ios::cur	154	name	261
ios::dec	142	namespace	214
ios::end	154	new	29
ios::fixed	142	<b>O</b>	
ios::hex	142	obiect	13
ios::in	148	obiectul arbore binar de căutare	176
ios::internal	142	obiectul coada	174
ios::left	142	obiectul lista	162
ios::nocreate	148	obiectul stiva	172
ios::noreplace	148	oct	140
ios::oct	142	ofstream	63,147
ios::out	148	open	148
ios::right	142	operator ()	75
ios::scientific	142	operator ,	74
ios::showbase	142	operator []	64
ios::showpoint	142	operator +	58
ios::showpos	142	operator ++	56
ios::skipws	142	operator +=	58
ios::stdio	142	operator <<	62
ios::trunc	148	operator =	23,85,114
ios::unitbuf	142	operator ->	77
ios::uppercase	142	operator >>	62
iostream	137	operator cast	71,114
iostream.h	14,134	operator delete	68
is_open	153	operator new	68
istream	63,134	ostream	63,134
istream_iterator	244	ostream_iterator	243
istrstream	158	ostrstream	158
iterator	223	<b>P</b>	
<b>L</b>		pair	231
legătură dinamică	117	pointer de membru	38
legătură statică	116	pointer la obiect	28
list	225	pop	235
listă inițializatori	18	pop_back	224
lower_bound	232	precision	138
<b>M</b>		priority_queue	236
manipulator	140	private	11,14,15
map	232	protected	12,14,15
masive de obiecte	29	public	12,14,15
membru pointer	38	publicizare	109
metodă	11	push	235
moștenire multiplă	124	push_back	223,226
moștenirea multiplă a	207	push_front	226
şablonelor		put	136,151
multimap	232		

			Index
<b><i>Q</i></b>			
queue			235
<b><i>R</i></b>			
rdstate			145
read			151
reserve			224
resetiosflags			140
resize			225
reverse_iterator			245
RTTI			260
<b><i>S</i></b>			
second			233
seekg			154
seekp			154
set			228
setbase			140
setf			143
setfill			140
setiosflags			140
setprecision			140
setw			140
size			223
sort			249
specializare parțială			202
specializare totală			200
specializări			197
specializările metodelor			198
stack			234
static			33
STL			219
str			158
streambuf			134
streamoff			154
streampos			154
strstrea.h			158
strstream			159
supraîncârcare			52
<b><i>T</i></b>			
tellg			155
tellp			155
template			187
template cu valori			196
implicite			
this			26
tip abstract			10,15
tip încapsulat			11,15
transform			252
typeid			261
typename			187
<b><i>U</i></b>			
unsetf			144
upper_bound			232
using			216
<b><i>V</i></b>			
vector			222
<b><i>W</i></b>			
width			138
write			136,151
ws			140

## *B i b l i o g r a f i e*

- Andonie, R., Gârbacea, I., *Algoritmi fundamentali o perspectivă C++*, Ed. Libris, Cluj-Napoca, 1995.
- Barkakati, N., Hipson, D.P., *Visual C++ Developer's Guide*, SAMS Publishing, Indiana, USA, 1993.
- Deitel, H.M., Deitel, P.J., *C++ How to program*, Prentice Hall Inc., New Jersey, 2001.
- Eckel, B., *Thinking in C++*, Prentice Hall Inc., New Jersey, 1999.
- Gurewich, O., Gurewich, N., *Borland C++ Multimedia Programming*, SYBEX, San Francisco, 1994.
- Murray, R., *C++ Strategies & Tactics*, Addison-Wesley, 1993.
- Mușlea, I., *Inițiere în C++*. Programare orientată pe obiecte, Ed. Microinformatica, Cluj-Napoca, 1992.
- Prata, S., *Manual de programare în C++*, Ed. Teora, București, 2001.
- Schild, H., *C++ manual complet*, Ed. Teora, București, 1991.
- Smeureanu, I., Ivan, I., Dârdală, M., *Structuri și obiecte în C++*, Ed. CISON, București, 1998.
- Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986.
- Wyk, V.J.C., *Data Structures and C Programs*, Addison-Wesley Publishing Company, 1988.