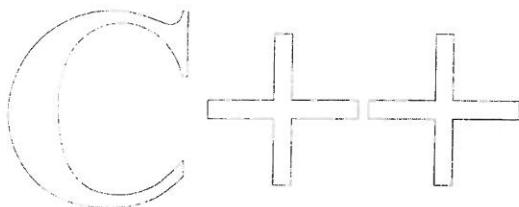


Macarie BREAZU

Programare Orientată pe Obiecte. Principii



**Editura Universității "Lucian Blaga" din Sibiu
2002**

Referenți:

Prof. dr. ing. Valer ROȘCA

Universitatea “Lucian Blaga” din Sibiu
Facultatea de Științe

Prof. dr. mat. Ioana MOISIL

Ş.I. ing. Dorin SIMA

Universitatea “Lucian Blaga” din Sibiu
Facultatea de Inginerie

Tehnoredactare: Macarie BREAZU

Copyright: Macarie BREAZU

**Descrierea CIP a Bibliotecii Naționale a României
BREAZU, MACARIE**

**Programare orientată pe obiecte : principii / Macarie
Breazu. – Sibiu : Editura Universității “Lucian Blaga” din
Sibiu, 2002.**

123 p.; 24 cm.

Bibliogr.

ISBN 973-651-465-X

004.43

In memoriam

În memoria dascălului și prietenului meu

Dan Roman [†]

[†] Dan Roman (1960-1996), șef lucrări doctorand, Universitatea Tehnică Cluj-Napoca



Prefață

Kernighan și Ritchie scriau, în prefața cărții lor *"The C Programming Language". Second Edition, Prentice Hall, 1988*, de referință în ceea ce privește limbajul C, următoarele: **"C is not a big language, and is not well served by a big book"**. Chiar dacă afirmația nu este valabilă, poate, în litera ei, și pentru limbajul C++, consider că ea rămâne valabilă, cel puțin în spiritul ei, cartea de față fiind realizată în această idee.

Nu am realizat aici o prezentare completă a limbajului C++. Oricum, nici nu mi-am propus acest lucru. Am preferat să prezint doar acelle facilități de limbaj în strânsă legătură cu programarea orientată pe obiecte. În această idee, exemplele prezentate nu sunt dorit a fi utile în sine ci doar în forma minimală, strict necesară pentru a demonstra subiectul în discuție. Totuși, exemplele sunt funcționale, rezultatele prezentate în lucrare fiind obținute prin rulare. De asemenea, am presupus o cunoaștere bună a principiilor programării structurate și, în special, a limbajului C.

Cartea de față este destinată în principal a fi suport de curs la disciplina "Programare Orientată pe Obiecte", disciplină cu rol formativ, existentă în planul de învățământ la diferite specializări în primii ani. S-a pus accentul pe o înțelegere riguroasă, profundă a principiilor programării orientate pe obiecte și a mecanismelor întâlnite. În acest fel, trecerea ulterioară la alt limbaj (de exemplu Java) nu va fi o problemă.

Trebuie atrasă de asemenea atenția asupra faptului că nu se abordează analiză, proiectare sau modelare orientată obiect, ci doar principiile programării orientate pe obiecte aşa cum apar ele în limbajul C++. Analiza, proiectarea și modelarea orientată obiect vor face, sper, subiectul unui volum ulterior.

Mulțumesc pe această cale tuturor colegilor, dascăli și studenți, care, prin observațiile lor la adresa diferitelor variante intermediere care au circulat, au făcut ca acum numărul erorilor, de editare și formulare, să fie mai redus. Cum nu pot spera că erorile au fost eliminate în totalitate și cum, cu siguranță, formulările pot fi îmbunătățite, mulțumesc anticipat pentru orice critici și sugestii în legătură cu această carte.

Macarie BREAZU

Sibiu, 01.10.2001

macarie.breazu@ulbsibiu.ro

Prefață.....	1
Cuprins	3
Cap. 1 - Încapsularea	5
1.1 Supraîncărcarea numelui funcțiilor. Funcții cu parametri având valori implicate..	5
1.2 Amintiri și generalități despre structuri.....	6
1.3 Introducerea noțiunii de clasă.....	9
1.4 Cum se leagă un apel de metodă de datele unui obiect anume (cel apelant) ?.....	10
1.5 Încapsularea datelor.....	12
1.6 Constructori și destrutori	14
1.7 Operatorii new și delete.....	16
1.8 Cazuri particulare de constructori care merită remarcate.....	17
1.9 Exemplu pentru constructori / destrutori	18
1.10 Membrii statici	19
1.11 Variabile membru de tip __property	21
Cap. 2 - Moștenirea	26
2.1 Instanțe încuibărite	26
2.2 Moștenirea	28
2.3 Constructori și destrutori în contextul derivării.....	33
2.4 Apel explicit al unui constructor al clasei de bază	34
2.5 Conversii de tip între clase derivate	34
2.6 Exemplu	35
2.7 Moștenirea multiplă.....	37
2.8 Moștenire multiplă virtuală	40
Cap. 3 - Polimorfismul	43
3.1 Supraîncărcarea metodelor în contextul moștenirii.....	43
3.2 Polimorfism – generalități	45
3.3 Legare statică și dinamică	45
3.4 Metode virtuale.....	46
3.5 Cum se implementează legarea dinamică (polimorfismul) ?	48
3.6 Ce poate fi virtual și ce nu.....	50
3.7 Un exemplu distractiv și nu numai.....	50
3.8 Metode virtuale și ierarhii de clase.....	53
3.9 Metode pure și clase abstracte.....	55
3.10 Tratarea uniformă a masivelor eterogene	56

Cap.4 - Redefinirea operatorilor.....	60
4.1 Apel prin valoare, adresă (pointer) și referință pentru tipuri de bază.....	60
4.2 Funcții friend, metode friend și clase friend (prietene)	61
4.3 Redefinirea operatorilor - introducere	64
4.4 Redefinirea operatorilor ca și metode sau ca și funcții friend	65
4.5 Distingerea între formele prefixate și postfixate ale operatorilor.....	67
4.6 Operatorul de atribuire (=)	69
4.7 Operatorii new și delete.....	72
4.8 Alte exemple de redefinire a operatorilor.....	75
4.9 Apel prin valoare,adresă (pointer) și referință pentru tipuri definite de utilizator	78
Cap. 5 - Stream-uri.....	81
5.1 Introducere.....	81
5.2 Starea unui stream	84
5.3 Operații de intrare-iesire fără format	85
5.4 Operații de intrare-iesire cu format	87
5.5 Format de inserție-extracție. Stare a formatului	88
5.6 Manipulatori predefiniți	91
5.7 Manipulatori definiți de utilizator	95
5.8 Intrare – ieșire pentru tipuri definite de utilizator	98
5.9 Legarea streamurilor (“Tying of streams”)	99
5.10 Buffer-area streamurilor	100
5.11 Streamuri și siruri	101
5.12 Streamuri și fișiere.....	103
5.13 Concluzii	105
Cap. 6 - Tratarea excepțiilor	106
6.1 Originea problemei.....	106
6.2 Soluția tradițională de tratare a erorilor și excepțiilor	106
6.3 Soluția C++ de tratare a excepțiilor.....	107
6.4 Discriminarea excepțiilor	107
6.5 Detalii despre implementarea mecanismului de tratare a excepțiilor.....	112
6.6 Tratare polimorfică a excepțiilor	114
6.7 Achiziție și eliberare de resurse.....	115
6.8 Opțiuni la tratarea unei excepții	118
6.9 Observații privind mecanismul de tratare a excepțiilor.....	119
Anexă	121
Bibliografie	123

Cap. 1 - Încapsularea

Limbajul C++ este de fapt un superset, o dezvoltare a limbajului "C", păstrând compatibilitatea "în jos", adică orice construcție valabilă în "C" rămâne valabilă și în C++, dezvoltarea constând în adăugarea de facilități noi.

1.1 Supraîncărcarea numelui funcțiilor. Funcții cu parametri având valori implicate

În "C++" este permis să avem mai multe funcții cu același nume dar diferind prin numărul sau tipul parametrilor ("supraîncărcarea numelui funcțiilor"). Această facilitate nu este strict legată de orientarea pe obiecte dar este des întâlnită, aşa încât o prezentăm în continuare.

Facilitatea este descrisă de exemplul următor, în care supraîncărcăm numele funcției Calc1:

```
long Calc1(long a, long b, long c)      // prima varianta
{
    return(a+b+c);
}

long Calc1(long a, long b)                // a doua varianta
{                                         // difera prin numarul
    return(a+b);                         // parametrilor
}

long Calc1(long a, int b)                 // a treia varianta
{                                         // difera prin tipul
    return(a+b);                         // parametrilor
}
```

compilatorul decizând în momentul apelului despre care funcție este vorba, ținând cont de numărul și tipul parametrilor implicați la apel:

```
long m=5, n=10, o=20, r;
int q=7;

r=Calc1(m, n, o);    // r=35, s-a apelat prima varianta
r=Calc1(m, n);       // r=15, s-a apelat varianta a doua
r=Calc1(m, q);       // r=12, s-a apelat varianta a treia
```

Trebuie făcute două remarci, evidente de altfel:

- compilatorul nu distinge între funcții pe baza tipului returnat (care nu se poate indica la apel).
- funcțiile supraîncărcate având același număr de parametri trebuie să difere cel puțin prin tipul unei perechi de parametri.

ambele restricții fiind necesare compilatorului pentru a putea lua decizia corectă (în caz contrar se generează eroare la compilare).

De asemenea se permite în "C++" să avem **funcții cu parametri având valori implice** (valorile implice pot fi doar constante):

```
long Calc2(long a, long b=7, long c=9) // parametrii b si c
{
    return(a+b+c); // pot lua valori
}
```

cu apeluri posibile

```
long m=5, n=10, o=20, r;
r=Calc2(m, n, o); // r=35, nu se consideră valori implice
r=Calc2(m, n); // r=24, c ia valoarea implicită
r=Calc2(m); // r=21, b și c iau valorile implice
```

Și aici rămâne de făcut o remarcă, evidentă de altfel:

- parametrii pot lua valori implice doar dinspre ultimul înspre primul, fără a sări vreunul.

Considerând atât supraîncărcarea numelui funcțiilor cât și funcțiile având valori implice, mai atragem atenția asupra faptului că:

- nu este permis să avem atât o funcție (supraîncărcată) cu un anumit număr de parametri cât și o funcție cu valori implice care să permită apel cu același număr și tip de parametri.

rezistență fiind necesară compilatorului pentru a putea lua decizia corectă (în caz contrar din nou se generează eroare la compilare).

1.2 Amintiri și generalități despre structuri

Începem introducerea noțiunii de clasă din limbajul "C++" pornind de la noțiunea de structură

Cap. 1 - Încapsularea

din limbajul "C" cu care se asemănă și pe care o considerăm (cel puțin în mare) asimilată.

Să presupunem că avem de gestionat într-un program 100 poziții în plan. Cea mai simplă soluție este de a declara două tablouri

```
int x[100] // pozitiile pe axa x
int y[100] // pozitiile pe axa y
```

Această soluție a grupat toate variabilele având aceeași semnificație în tablouri.

O soluție ceva mai avansată este de a grupa toate informațiile referitoare la un punct într-o structură SPos

```
struct SPos
{
    int x;      // pozitia pe axa x
    int y;      // pozitia pe axa y
};
```

instantierea celor 100 perechi de variabile făcându-se acum astfel:

```
struct SPos poz[100] // tablou de 100 structuri SPos
```

Această a doua soluție are avantajul de a face și la nivelul limbajului o legătură între datele unei anumite poziții (legătură evidentă la nivelul conceptual). De asemenea, suntem siguri că se aloca la fel de multe variabile x respectiv y (fapt realizabil și prin dimensiune simbolică).

Soluția se poate simplifica în "C" prin folosirea operatorului **typedef**

```
typedef struct
{
    int x;      // pozitia pe axa x
    int y;      // pozitia pe axa y
} SPos;
```

instantierea făcându-se acum mult mai simplu:

```
SPos poz[100] // tablou de 100 variabile de tip SPos
```

total operând ca și când limbajul s-a îmbogățit cu un **nou tip de date**, din care se pot face simplu instantieri de variabile.

Pentru accesarea unei variabile membru a unei structuri se folosește operatorul **"."**

```
poz[5].x // accesarea variabilei membru x din poz[5]
```

Evident, dacă avem la dispoziție adresa unei variabile de tip structură adresarea se face cu operatorul “->”.

În “C++” situația se simplifică, declararea unei structuri incluzând (implicit) și definiția unui nou tip ca în exemplul următor:

```
struct SPos
{
    int x;      // poziția pe axa x
    int y;      // poziția pe axa y
};

SPos pos[100] // tablou de 100 variabile de tip SPos
```

Reamintim faptul că dimensiunea unei structuri este egală cu suma dimensiunii elementelor sale, adică:

$$\text{dimensiune structură} = \sum \text{dimensiune membri}$$

Să presupunem că avem nevoie de o funcție care să modifice conținutul variabilelor membru ale structurii SPos (să mută poziția reprezentată de o asemenea variabilă). Deoarece un simplu *apel prin valoare* nu permite modificare valorii de apel (ci doar a copiilor de pe stiva a valorii parametrilor – problemă obligatoriu a fi deja cunoscută) soluția este de a avea un *apel prin adresă*, adică o funcție (de exemplu “Muta”) care să primească adresele variabilelor membru sau, mai simplu, adresa structurii respective (și valorile “pasului” pe cele 2 direcții)

```
void Muta(SPos *p, int dx, int dy)
{
    p->x += dx;      // mutarea pe directia x
    p->y += dy;      // mutarea pe directia y
}
```

apelul făcându-se (de exemplu) astfel (pentru a muta pe axa x cu 100 iar pe axa y cu 50)

```
Muta( &pos[5], 100, 50 );
```

Aceasta soluție (cu parametru adresa structurii asupra căreia se operează) se poate aplica tuturor funcțiilor care se referă la o asemenea structură. Soluția este puțin greoasă și nu are eleganță unei exprimări de genul

pos[5].x

1.3 Introducerea noțiunii de clasă

O nouitate adusă de limbajul "C++" este aceea că o structură poate conține nu numai date ci și funcții. Astfel funcția Muta poate fi declarată ca aparținând structurii SPos

```
struct SPos
{
    int x;                                // pozitia pe axa x
    int y;                                // pozitia pe axa y
    void Muta(SPos *p, int dx, int dy);   // functia Muta
};

void SPos::Muta(SPos *p, int dx, int dy)
{
    p->x += dx;      // mutarea pe directia x
    p->y += dy;      // mutarea pe directia y
}

SPos pos[100]           // tabel cu 100 variabile de tip SPos
```

Remarcăm diferența între *declararea* (prototipului) funcției în cadrul structurii și *implementarea* ei (indicând cu operatorul “::”, de rezoluție, structura căreia îi aparține funcția).

Aveam acum de a face atât cu **variabile membru** (ex. x și y) cât și cu **funcții membru** (ex. Muta). Funcțiile membru se numesc și *metode* (notație preferată, recomandată).

Apelul unei metode se face în acest caz ca și adresarea unei variabile membru cu operatorul “.”

```
pos[5].Muta( &pos[5], 100, 50 );
```

indicând astfel și la nivelul limbajului că avem de a face cu un apel de metodă pentru obiectul pos[5]. Se remarcă utilizarea redundantă, greoale a construcției pos[5] atât în stânga lui “.” cât și ca parametru.

Observația că formula

$$\text{dimensiune structură} = \sum \text{dimensiune variabile membru}$$

rămâne valabilă (! ! ! ! !), riguros, cât timp nu avem metode virtuale și membri statici) indică faptul că corpul metodei nu este chiar inclus în structură, neafectând dimensiunea acesteia, ci mai degrabă este ținut în zona de cod, separat, o singură dată, și nu pentru (în) fiecare structură

în parte. Apare astfel o întrebare fundamentală:

1.4 Cum se leagă un apel de metodă de datele unui obiect anume (cel apelant) ?

Rezolvarea acestei probleme are la bază generarea, de către compilator, la fiecare apel de **metodă**, a unui **parametru ascuns *this*** (cuvânt rezervat al limbajului) care este de fapt un pointer spre obiectul pentru care s-a apelat metoda. Pointerul *this* este accesibil programatorului în corpul metodei. În acest fel se permite accesarea datelor obiectului respectiv (corespunzător) prin intermediul lui *this*.

Un apel de genul:

```
obiect.metoda( parametri ... )
```

este implementat de către compilator astfel:

```
metoda( &obiect, parametri ... )
```

după cum vedem locul parametrului ascuns (dar existent!!) *this* este luat de adresa obiectului apelant. Deși pointerul *this* nu se declară explicit el se poate folosi în corpul metodei.

În acest caz este evident că metoda *Muta* nu mai are nevoie de parametrul SPos *p (locul său fiind luat de pointerul *this* ascuns) iar implementarea metodei devine:

```
void SPos::Muta(int dx, int dy)
{
    this->x += dx; // se modifica x-ul coresp. apelului
    this->y += dy; // se modifica y-ul coresp. apelului
}
```

Forma prezentată mai sus (cu acces la variabilele membru folosind *this*) este foarte utilă din punct de vedere **didactic** dar este greoaiă, incomodă pentru utilizarea ei frecventă. Pentru simplitate compilatorul generează implicit apel prin *this* pentru accesul variabilelor membru programatorul putând renunța la folosirea lui *this*:

```
void SPos::Muta(int dx, int dy)
{
    x += dx;           // se modifica x-ul coresp. apelului
    y += dy;           // se modifica y-ul coresp. apelului
}
```

Deși nu se mai poate observa direct, nu uităm rolul esențial al lui *this* în legarea apelului unei

metode de datele structurii pentru care s-a apelat metoda.

```
pos[5].Muta(100,50); // x+= 100; y+= 50 pt. struct. pos[5]
pos[3].Muta(200,150); // x+= 200; y+=150 pt. struct. pos[3]
```

Utilizarea explicită a pointerului this este esențială în situațiile în care ne interesează adresa structurii în cauză (care trebuie de exemplu introdusă într-o listă înlănțuită).

O altă facilitate introdusă de "C++" este **posibilitatea restricționării accesului la membri**. În acest scop fiecare membru are atributul "public" sau "private" cu următoarea semnificație:

- **public** – acești membri pot fi accesăți din "interiorul" structurii și pot fi accesăți din "exteriorul" structurii.
- **private** – acești membri pot fi accesăți din "interiorul" structurii și nu pot fi accesăți din "exteriorul" structurii.

În cele de mai sus numim "interiorul" structurii corpul metodelor structurii iar "exteriorul" structurii acele secțiuni de cod care utilizează instanțe ale structurii (dar nu sunt membre în structură).

Remarcăm că atributul public / private se referă doar la modul în care pot fi accesăți membrii structurii din exteriorul acesteia (metodele având acces la membrii indiferent de atributul membrilor).

Un exemplu de declarare a structurii SPos poate fi următorul (fără a fi încă util practic)

```
struct SPos
{
private:
    int x;                                // pozitia pe axa x
    int y;                                // pozitia pe axa y
public:
    Muta(int dx, int dy);                // metoda Muta
};
```

În care variabilele membru x și y sunt declarate private (pot fi accesate doar din Muta) iar metoda Muta este declarată publică. Remarcăm faptul că atributul privat / public nu trebuie specificat pentru fiecare membru în parte el fiind valabil pentru toți membrii următori până la stabilirea unui nou atribut. În acest caz tentativa de accesare a membrului x din afara structurii este eroare de compilare.

Cu aceste noțiuni pregătite putem introduce noțiunea de clasă:

O clasă (“class”) este o entitate asemănătoare cu o structură singura diferență constând în aceea că membrii unei structuri sunt implicit publici iar membrii unei clase sunt implicit privați.

Programarea Orientată pe Obiecte (“Object Oriented Programming” - OOP) este de fapt programare bazată pe clase (teoretic pe structuri și clase, dar practic aproape întotdeauna pe clase). De aceea limbajul “C++” se numea la început “C with Classes”.

Subliniem că toate caracteristicile prezentate anterior (în “C++”) pentru structuri sunt valabile și pentru clase. Ele au fost enunțate pentru structuri pentru ca introducerea noțiunii de clasă să se facă treptat, pe baza cunoștințelor avute despre structuri. Deși în mod intenționat am evitat până în acest punct folosirea noțiunii de clasă (preferând pe cea de structură) tot materialul anterior referitor la “C++” poate fi recitat înlocuind structură (struct) cu clasă (class).

1.5 Încapsularea datelor

Sintaxa de definire a unui nou tip devine acum:

```
specificator nume
{
    [[tip_acces:] lista_membri_1]
    [[tip_acces:] lista_membri_2]
    // ...
};
```

în care **specificator** poate fi **class**, **struct** sau **union** iar **tip_acces** poate fi **public** sau **private**. Diferența între **class**, **struct** și **union** constă (în principal) doar în tipul de acces implicit al membrilor: privat pentru class și public pentru struct și union. În expresia anterioară am indicat prin paranteze drepte cămpuri care pot lipsi.

Proprietatea unui membru de a fi privat este cea care stă la baza conceptului de **încapsulare**. Aceasta se referă la posibilitatea de a crea obiecte încapsulate, obiecte la care nu se pot accesa (vedea) anumiți membri. Se obține astfel o capsulă închisă ermetic, utilizatorul obiectului neavând acces la interiorul acestuia. Mai puțin formal, se poate face asocierea cu cutia neagră de la electrotehnica (la care accesul se face doar pe la borne – adică prin membrii publici) sau cu o nucă (la o nucă de cocos orificiile putând de asemenea fi asociate cu membrii publici).

Recomandarea este aceea de a avea **datele private** și de a asigura un **set de metode publice**

Cap. 1 - Încapsularea

necesare manipulării datelor (atât cât este permis de către proiectantul clasei).

O problemă ridicată de caracterul privat al datelor este imposibilitatea atribuirii spre ele a unor valori inițiale (în lipsa acestei inițializări datele conținând valori pe care nu ne putem baza – gunoi, “garbage”). Această problemă se poate rezolva în principiu în două feluri:

- prin utilizarea unor metode publice de atribuire / citire a valorii variabile

```
class SPos
{
    int x;                      // implicit private
    int y;                      // implicit private
public:
    void Muta(int dx, int dy);   // metoda Muta
    int GetX(){return x;}        // citire variabila x
    void SetX(int x0){x=x0;}    // atribuire variabila x
};
```

- sau prin existența unei metode publice doar de atribuire (inițializare) a datelor private

```
class SPos
{
    int x;                      // implicit private
    int y;                      // implicit private
public:
    void Muta(int dx, int dy);   // metoda Muta
    void Atribuie(int x0, int y0) // metoda de atribuire
        {x=x0; y=y0}
};
```

În primul caz avem acces complet la variabila x (chiar dacă indirect prin metodele de GetX și SetX) iar în al doilea putem doar atribui valori (inițiale) membrilor privați. Deși nu este important pentru subiectul încapsulării, implementarea metodelor imediat în cadrul declarării obiectului este echivalentă cu declararea acestor metode ca și **in-line** (care nu se apelează efectiv ci se expandează). După cum se știe, se recomandă a se implementa in-line metode (funcții) la care corpul este foarte scurt (cum ar fi GetX respectiv SetX). Am ales această soluție în exemplele anterioare doar pentru a introduce și noțiunea de in-line.

Soluția a doua de inițializare (cu o singură metodă de inițializare a întregului obiect) este **importantă** în sensul că **va evolua în constructor** (după cum vom vedea mai târziu).

Încă o remarcă: încapsularea ne permite nu numai blocarea accesului la o variabilă membru (totală) ci și posibilitatea **controlării accesului la variabilele membru** ca în exemplul următor:

```
void CPos::Atribuie(int x0, int y0)
{
    if( x0 <      0 ) x0 =      0;
    if( x0 >= 800 ) x0 = 799;
    if( y0 <      0 ) y0 =      0;
    if( y0 >= 600 ) y0 = 599;
    x=x0;y=y0;
}
```

În care am impus limite pentru valorile permise pentru x și y (de exemplu limitele coordonatelor ecranului). În acest fel se poate controla ce anume atribuiri se fac înspre variabilele obiectului (posibilitatea numitorului 0 într-o clasă CNumarRational este evident dezastruoasă) scăpând de o mulțime de griji ulterioare în ceea ce privește posibilitatea introducerii de date eronate de către utilizatorul clasei.

De cele mai multe ori se preferă această soluție de a avea o singură metodă de atribuire pentru toți membrii clasei. Acest lucru ne obligă la prezentarea următoarei facilități a limbajului C++:

1.6 Constructori și destructori

Atragem atenția asupra faptului că **declararea unei clase este echivalentă cu declararea unui nou tip de date**. Alocarea de obiecte de acest tip se face DOAR la instanțierea unor asemenea obiecte (definire de variabile de acest tip). Deci, prin **obiect** se înțelege o instanță a unei clase.

După cum am discutat și anterior, odată instanțiat un obiect **este esențială inițializarea** acestuia, fapt care impune o oarecare încălcare a principiului încapsulării prin folosirea unor metode publice pentru inițializarea variabilelor membru.

Deși facilitățile de supraîncărcare a numelui funcțiilor și funcțiile cu parametri luând valori implicate au fost prezentate pentru funcții oarecare, ele sunt **valabile și pentru metode ale obiectelor** (în fapt tot niște funcții, dar care au în plus parametrul ascuns this).

Tocmai pentru a nu ne permite să uităm apelul unei metode de inițializare C++ introduce noțiunea de constructor:

Un constructor este o metodă având același nume cu al clasei

căreia îi aparține și care prezintă următoarele proprietăți:

- nu are tip returnat (nici măcar void)
- poate beneficia de parametri implicați și de supraîncărcarea numelui (ca orice altă

Cap. 1 - Încapsularea

metodă). Putem deci avea mai mulți (oricărui) constructori pentru o clasă.

- la instanțierea unui obiect se apelează constructorul potrivit (conform parametrilor de apel).
- dacă programatorul nu declară explicit nici un constructor pentru o clasă compilatorul generează unul public, fără parametri și care nu face nimic. Din acest motiv se puteau crea instanțe structuri fără a avea constructori declarați explicit.

Remarcăm că, la orice instantiere de obiect (fie ea statică sau dinamică) se apelează implicit constructorul potrivit (conform parametrilor furnizați). Dacă acest constructor nu există se generează eroare de compilare. Constructorii variabilelor **globale** se apelează înainte de apelul lui main(), cei ai variabilelor **locale** la intrarea în funcția respectivă iar cei a variabilelor alocate **dinamic** (cu new) în momentul alocării.

Rolul principal al constructorului este acela de a inițializa obiectul pentru care a fost apelat. Acest lucru înseamnă, de obicei,

- inițializarea tuturor variabilelor membru la valori determinate (prin parametri sau implicit) aducând astfel obiectul într-o stare determinată. Memoria pentru obiectul propriu-zis (pentru variabilele membru) a fost deja alocată, acest lucru nu este sarcina constructorului.
- alte inițializări necesare obiectului (allocare pentru pointerii din cadrul obiectului, de exemplu, aici se poate aloca memorie pentru bufferele necesare).

Se numește **constructor implicit** un constructor fără parametri, obținut fie prin declararea explicită a unui constructor fără parametri fie prin generarea implicită de către compilator a unui constructor fără parametri (dacă nu este definit explicit nici un constructor). Existența unui constructor implicit este esențială pentru a putea inițializa tablouri (unde nu se pot apela explicit constructorii).

Ca și complement al constructorului s-a introdus noțiunea de destrutor:

Un destritor este o metodă având același nume cu al clasei dar precedat de ~

și care prezintă următoarele proprietăți:

- nu are tip returnat (nici măcar void)
- nu are parametri, deci nu putem deci avea mai mulți destrutori pentru o clasă.
- la distrugerea unui obiect se apelează destritorul său.

Remarcăm faptul că, la orice dispariție de obiect (fie el static sau dinamic) se apelează

IMPLICIT destructorul său. Destructorii variabilelor globale se apelează după ieșirea din funcția main(), cei ai variabilelor locale la ieșirea din funcția respectivă iar cei ai variabilelor alocate dinamic (cu new) în momentul distrugerii (eliberării) explicite a obiectului (cu delete).

Rolul principal al destructorului este opus rolului constructorului (de unde și notația sa bazată pe ~, care ca și operator are sens de **complement**), de obicei acela de eliberare a memoriei suplimentare alocate de către constructor.

Considerând o clasă CX având atât constructor fără parametri cât și constructor cu 2 parametri întregi, alocarea de obiecte statice pe baza acestor constructori se face astfel:

```
CX a, b(1,2); // a foloseste constructorul fără parametri  
// b foloseste constructorul cu 2 parametri
```

1.7 Operatorii new și delete

În "C++" se introduce **operatorul new** ca și alternativă la alocarea de memorie folosind **funcția malloc**. Putem deci aloca o variabilă întreagă astfel:

```
int *p;  
p = (int*)malloc(sizeof(int)); // folosind malloc  
p = new int; // folosind new
```

La fel se poate aloca și un obiect de tip declarat de programator (de exemplu CPos):

```
CPos *p;  
p = (CPos*)malloc(sizeof(CPos)); // folosind malloc  
p = new CPos; // folosind new
```

Deși ambele soluții instanțiază un nou obiect de tipul dorit remarcăm avantajele folosirii lui new:

- se cunoaște dimensiunea obiectului (simplificând expresia de la alocare).
- se **apeleză automat un constructor** al clasei respective, asigurând inițializarea automată a obiectului.

Pereche cu new s-a introdus **operatorul delete** în locul apelului **funcției free**:

```
free(p); // eliberare folosind free  
delete p; // eliberare folosind delete
```

Deși ambele soluții distrug obiect remarcăm avantajul folosirii lui delete:

- se apelează automat de constructorul clasei respective

1.8 Cazuri particulare de constructori care merită remarcate

Deși cazul cel mai general al unui constructor este cel care are parametri de tipuri de date oarecare (asemănător cu cei prezenți anterior) merită prezentate unele **cazuri particulare**. Considerăm o clasă în discuție CX și alte clase oarecare CT.

- **Constructorul general** este, cu aceste notații:

CX::CX(CT1, CT2, ...)

- **Constructorul implicit** este:

CX::CX()

și prezintă, după cum am amintit anterior, particularitatea că un asemenea constructor (având corpul vid) este generat **implicit** de compilator atunci când clasa CX nu are nici un constructor declarat explicit.

- **Constructorul de copiere** este:

CX::CX(CX&, ...)

unde, dacă mai există parametri, aceștia trebuie să ia valori implicate în cazul apelului respectiv. Se folosește la declarații de tipul:

```
CX a;  
CX b=a; // aici se apelează constructorul de copiere
```

sau la apeluri prin valoare și returnare de valori de tip obiect:

```
CX Functie(CX o1, ...)  
{  
CX o2;  
// ..... diverse prelucrări  
return o2;  
}
```

cu apelul posibil

```
b=Functie(a);
```

care va apela constructorul de copiere pentru **copierea obiectelor parametru** (în exemplu a în o1 și, la fel, pentru eventuale alte argumente de tip obiect) și pentru **copierea obiectului întors** (în exemplu se copiază o2 într-un obiect temporar și apoi acesta este atribuit lui b, folosind operatorul =).

Atragem atenția asupra următoarei greșeli frecvente: se confundă constructorul de copiere cu operatorul de atribuire (care va fi prezentat ulterior la redefinirea operatorului “=”).

În ceea ce privește constructorul de copiere mai facem următoarele remarcări:

- dacă nu se furnizează explicit un constructor de copiere compilatorul generează implicit unul care copiază membru cu membru datele obiectelor.
- atenție la copierea membru cu membru, care este periculoasă când se copiază membri de tip pointeri fără a face o copie zonei pointate. Această posibilitate impune o dată în plus necesitatea constructorilor de copiere (și a redefinirii operatorului de atribuire).
- **Constructorul de conversie** este:

```
CX::CX(TX&, ...)  
CX::CX(TX, ...)
```

unde, dacă mai există parametri, aceștia trebuie să ia valori implicate în cazul apelului respectiv. Este asemănător cu cel de copiere doar că obiectul parametru are alt tip. Cu cele precizate înainte considerăm că rolul și caracteristicile acestui constructor sunt evidente.

1.9 Exemplu pentru constructori / destructori

Prezentăm următorul exemplu în scopul de a demonstra apelul constructorilor și destructorilor:

```
#include <stdio.h>  
class CA  
{  
    int x,y;  
public:  
    CA()           {x=0;y=0;printf("\nConstr implicit");}  
    CA(int m,int n=0) {x=m;y=n;printf("\nConstr general ");}  
    CA(CA &o)      {x=o.x;y=o.y;printf("\nConstr copiere ");}  
    ~CA()          {printf("\nDestructor      ");}  
};  
CA a,b(1,2);                                // 2 obiecte globale  
void main()
```

```
{  
CA *c,d(2),*e,f[2],g=b,*h;           // 4 obiecte locale  
  
printf("\nInceput cod main");  
c=new CA(5);  
e=new CA;  
delete c;  
printf("\nSfirsit cod main");  
}
```

care va afișa următorul rezultat:

```
Constr implicit      // pentru a  
Constr general       // pentru b  
Constr general       // pentru d  
Constr implicit       // pentru f[0]  
Constr implicit       // pentru f[1]  
Constr copiere        // pentru g  
Inceput cod main  
Constr general        // pentru c  
Constr implicit        // pentru e  
Destructor            // pentru c  
Sfirsit cod main  
Destructor            // pentru g  
Destructor            // pentru f[1]  
Destructor            // pentru f[0]  
Destructor            // pentru d  
Destructor            // pentru b  
Destructor            // pentru a
```

1.10 Membrii statici

O altă facilitate introdusă în C++ este posibilitatea de a defini **membri statici**, o clasă putând avea atât variabile membru statice cât și funcții statice:

- **Variabilele membru statice** se instantiază o singură dată pe clasă și nu pentru fiecare instanță (obiect) în parte, deci nu măresc dimensiunea obiectului. Se pot inițializa explicit în modulul în care a fost definită clasa.
- **Funcțiile statice** nu au parametrul ascuns **this** deci preferăm să nu le numim metode. Ele se pot apela însă atât ca și metodele (pentru un obiect anume) cât și nelegate de un obiect. Neavînd pointerul **this**, nu pot accesa variabile membru nestatice (decât dacă furnizăm explicit un parametru referință spre obiect).

Principalul rol al membrilor statici este acela de a reduce variabilele și funcțiile globale care au legătură cu această clasă prin includerea lor aparentă în cadrul clasei.

Pentru exemplificare considerăm următorul program:

```
#include <stdio.h>
class X
{
    int x;          // nefolosit
    static int y;      // numar instantieri
public:
    void Met1(){printf("\nNumar instante = %d",y);}
    static void Met2(){printf("\nNumar instante = %d",y);}
//    static void Met3(){x++;} // eroare de compilare, nu se
                           // pot accesa membrii ne-statici
    X(){y++;}           // incrementare numar instantieri
    ~X(){y--;}
};

int X::y=0;        // initializare variabila membru STATICĂ
                   // se poate face chiar daca este privată !!
X a,b;

void main()
{
    X c,d,*e;

    e=new X;
    a.Met1();
    delete e;
    d.Met2(); // apel legat de un obiect
    X::Met2(); // apel nelegat de un obiect (legat de clasa)

    // X::y++;       se genereaza eroare y fiind privată
}
```

care va produce următorul rezultat:

```
Numar instante = 5      // pentru a,b,c,d,e
Numar instante = 4      // s-a distrus e
Numar instante = 4
```

Membrii statici sunt de fapt variabile și funcții globale care au însă domeniul de vizibilitate redus la clasa din care fac parte.

Considerăm că analiza rezultatelor afișate dovedește proprietățile enunțate anterior.

În problema dimensiunii unui obiect trebuie făcută următoarea detaliere:

dimensiune obiect = \sum dimensiune variabile membru
(fără a considera și variabilele membru statice)

evidență (sperăm) în contextul celor prezentate anterior.

1.11 Variabile membru de tip `_property`

În cele ce urmează prezentăm simplificat una din extensiile limbajului C++, aşa cum a fost ea introdusă de firma Borland. Deși nu este inclusă în standardul C++, considerăm că este interesantă și merită prezentată, ea fiind valabilă (cu modificări minore) și pentru alte compilatoare (ideile bune se “regăsesc” la toată lumea).

S-a introdus un cuvânt cheie suplimentar `_property` care poate fi “aplicat” unei variabile membru. Deoarece noțiunea de proprietate se referă de obicei la toate variabilele membru ale unui obiect vom folosi pentru cele bazate pe `_property` denumirea de “variabile membru de tip `_property`” (pentru a evita eventualele confuzii care ar putea apărea).

Definirea unei variabile membru de tip `_property` se face conform următoarei sintaxe:

```
_property TipMembru NumeMembru = { read =
    MembruFolositLaCitire , write = MembruFolositLaScrisere };
```

în care `TipMembru` și `NumeMembru` au semnificația obișnuită iar între acolade se indică cum se realizează citirea și scrierea variabilei.

Pentru exemplificare (și înțelegere) prezentăm în continuare un exemplu semnificativ (pentru diferitele moduri de utilizare):

```
#include <stdio.h>

class X
{
public:
    int x,y;
private:
    int Mcitire();
    void Mscriere(int v);
    char* AflaElementul(int Index1, float Index2);
public:
    _property int v = {read = x, write = y};
    _property int Medie = {read = Mcitire, write = Mscriere};
```

Programare Orientată pe Obiecte - Principii

```
__property char* Tablou[int Index1][float Index2] =
    {read=AflaElementul};

}ob;      // aici deja se pot defini obiecte globale

int X::Mcitire()
{
    printf("\n MetCitire x=%d y=%d ret=%d",x,y,(x+y)/2);
    return (x+y)/2;
};

void X::Mscriere(int ValNouaMedie)
{
    int shift = ValNouaMedie - (x+y)/2;

    printf("\n MetScriere %d",ValNouaMedie);
    x += shift; y += shift;
};

char* X::AflaElementul(int Index1, float Index2)
{
    if(Index1==0 && Index2==0.0) return "zero";
    else if(Index1!=0 && Index2==0.0) return "intreg";
    else if(Index1==0 && Index2!=0.0) return "flotant";
    else                                return "neimpl";
}

void main()
{
    int i; float f;

    ob.x=5; ob.y=7;
    printf("\nMAIN_1_1 x=%d y=%d v=%d",ob.x, ob.y, ob.v);
    ob.v=10;
    printf("\nMAIN_1_2 x=%d y=%d ",ob.x, ob.y);
    ob.v++;
    printf("\nMAIN_1_3 x=%d y=%d \n",ob.x, ob.y);

    ob.x=5; ob.y=7;
    printf("\nMAIN_2_1 x=%d y=%d",ob.x, ob.y);
    printf("\nMAIN_2_2 Medie=%d",ob.Medie);
    ob.Medie=10;
    printf("\nMAIN_2_3 x=%d y=%d ",ob.x, cb.y);
    ob.Medie++;
    printf("\nMAIN_2_4 x=%d y=%d \n",ob.x, ob.y);

    i=2; f=3.1;
    printf("\nMAIN_3_1 %2d %5.2f %s",i,f,ob.Tablou[i][f]);
    i=0; f=0;
```

Cap. 1 - Încapsularea

```
printf("\nMAIN_3_2 %2d %.2f %s", i, f, ob.Tablou[i][f]);
i=1; f=2.0;
printf("\nMAIN_3_3 %2d %.2f %s", i, f, ob.Tablou[i][f]);
i=5; f=0.0;
printf("\nMAIN_3_4 %2d %.2f %s", i, f, ob.Tablou[i][f]);
i=0; f=3.7;
printf("\nMAIN_3_5 %2d %.2f %s", i, f, ob.Tablou[i][f]);
i=3; f=2.1;
printf("\nMAIN_3_6 %2d %.2f %s", i, f, ob.Tablou[i][f]);
};
```

La rulare se va genera următorul rezultat:

```
MAIN_1_1 x=5 y=7 v=5 // cazul variabilei v
MAIN_1_2 x=5 y=10
MAIN_1_3 x=5 y=6

MAIN_2_1 x=5 y=7 // cazul variabilei Medie
MetCitire x=5 y=7 ret=6
MAIN_2_2 Medie=6
MetScriere 10
MAIN_2_3 x=9 y=11
MetCitire x=9 y=11 ret=10 // pentru incrementare
MetScriere 11 // pentru incrementare
MAIN_2_4 x=10 y=12

MAIN_3_1 2 3.10 neimpl // cazul variabilei Tablou
MAIN_3_2 0 0.00 zero
MAIN_3_3 1 2.00 neimpl
MAIN_3_4 5 0.00 intreg
MAIN_3_5 0 3.70 flotant
MAIN_3_6 3 2.10 neimpl
```

Dacă analizăm exemplul prezentat remarcăm declararea în cadrul clasei a trei variabile membru de tip `_property` (`v`, `Medie`, `Tablou`) fiecare reprezentând un caz deosebit. Să le luăm pe rând:

În cazul variabilei v `MembruFolositLaCitire` și `MembruFolositLaScriere` sunt variabile membru ale obiectului. Remarcăm faptul că variabila membru `v` se comportă ca și un membru oarecare din punct de vedere al interfeței obiectului, ceea ce este diferit (și specific) este modul de implementare a acesteia. Pentru citire se folosește `x` (valoarea variabilei membru `x`) iar la scriere se folosește `y` (se face atribuirea în variabila membru `y`). Această observație ar justifica denumirea acestui tip de variabilă ca și variabilă virtuală. Deoarece cuvântul virtual (cuvânt cheie) are o semnificație bine definită în C++ (semnificație care nu se referă la acest aspect) vom evita folosirea să în acest context dar reținem că o variabilă de tip `_property`

este virtuală în sensul comun al cuvântului: ea există doar aparent, pe interfață, fără a exista efectiv în interiorul obiectului.

În cazul variabilei Medie se exemplifică adevărata “putere” a mecanismului care apare atunci când MembruFolositLaCitire și MembruFolositLaScriere nu mai sunt variabile membru ci sunt metode (Mcitire respectiv Mscriere). Din nou variabila de tip __property (Medie) există (doar) aparent pe interfața obiectului dar scrierea și respectiv citirea ei se fac acum prin intermediul metodelor specificate. Apelul acestora se face automat, de către compilator, la momentul necesar. În mod logic, metoda de scriere necesită un parametru de tipul membrului __property iar metoda de citire întoarce un asemenea tip.

Este posibilă și declararea unor variabile de tip __property care să folosească soluții diferite pentru implementarea scrierii și citirii (folosind o variabilă membru și o metodă).

În cazul variabilei Tablou s-a dorit declararea unui tablou de tip __property. În acest caz metodele de citire și scriere trebuie să aibă parametri suplimentari corespunzători indicilor necesari. Dacă analizăm rezultatele afișate observăm că metoda AflaElementul (având doi parametri) este cea care este apelată la citirea fiecărui element și care decide (în funcție de indecesii particulari) conținutul tabloului. Un exemplu în care acest lucru ar fi util (și elegant) ar fi cel al implementării unei matrici rare (cu elementele menținute eventual într-o listă), dar în care acest lucru este ascuns utilizatorului obiectului.

O posibilitate interesantă (valabilă pentru toate cele trei situații prezentate) este aceea de a avea specificată doar una dintre acțiuni (de exemplu read) iar cealaltă să lipsească (ca în cazul Tablou din exemplu). În acest caz tentativa de a face astfel de referiri spre acel membru (în exemplul prezentat - atribuiri) va fi detectată de către compilator ca eroare de tipul “variabila nu este accesibilă”. Acest fapt deschide perspectiva posibilității realizării unei **încapsulări controlate** (valabilă doar pentru scriere sau citire). Evident acest lucru s-ar fi putut face prin intermediul variabilei membru private și furnizarea doar a unei metode de citire sau scriere dar cu prețul sacrificării eleganței sintaxei de accesare (care acum continuă să fie cea naturală).

Atragem atenția asupra pericolului care apare atunci când în interiorul unei metode de citire sau scriere a unei variabile de tip __property s-ar accesa din nou acea variabilă. În mod logic compilatorul apelează din nou metoda respectivă efectul fiind depășirea stivei (tipic pentru situația unor apeluri recursive fără condiție de oprire).

În problema dimensiunii unui obiect trebuie făcută următoarea detaliere:

dimensiune obiect = \sum dimensiune variabile membru
(fără a considera și variabilele de tip `_property`)

evidență (sperăm) în contextul celor prezentate anterior.

Variabilele de tip `_property` ascund detaliiile de implementare. Ceea ce arată că un simplu membru poate duce în momentul accesării sale la operații complexe (de exemplu căutări în baze de date). Accentuăm încă o dată că, din punct de vedere al utilizatorului obiectului, variabilele de tip `_property` se comportă ca și variabile obișnuite. Singurul lucru care nu se poate face asupra acestora este de a se lua adresa lor și, deci, nu se pot transmite funcțiilor ca și parametri prin referință.

Amintim (anticipând polimorfismul) și faptul că metodele folosite pentru citirea și scrierea variabilelor de tip `_property` pot fi virtuale deci vor putea fi implementate diferit (polimorfic) în ierarhii de clase.

Rezumat - de tinut minte :-)

- **suprăîncărcarea** numelui funcțiilor, funcții cu parametri luând **valori implicate**
- structură – **variabile membru**
- structurile pot include și funcții \rightarrow **funcții membru = metode**
- legarea metodelor de datele unui obiect prin **pointerul ascuns this**
- atributele **public** și **private**
- **class** echivalent cu struct dar membrii implicit privați
- **încapsularea datelor** și controlul accesului
- inițializarea variabilelor membru – **constructori**
- constructori **generalii**, cazuri particulare: **impliciți**, **de copiere**, **de conversie**
- **destructori**
- membrii **statici**
- **_property** – variabile membru **aparente**

Cap. 2 - Moștenirea

2.1 Instanțe încuibărite

Deși instanțele încuibărite nu sunt un caz de moștenire, considerăm utilă explicarea pentru început a acestei facilități. În fapt nu este ceva deosebit, doar faptul că o clasă are variabilă membru o instanță a altei clase.

Să considerăm cazul următor în care clasa CB are ca și membru un obiect de tipul CA.

```
class CA
{
public:
    int x;
    CA() {x=0;}      // constructor implicit
    CA(int x0){x=x0;} // constructor explicit
};

class CB
{
public:
    CA a;           // instanta incuibarita
    int y;
    CB();           // constructor implicit
    CB(int x0,int y0); // constructor explicit
};
```

și o instanță a clasei CB

```
CB b(1,2);
```

În mod evident variabilele x și y se pot accesa folosind următoarea sintaxă (care ar fi valabilă și pentru cazul apelului eventualelor metode):

```
b.y;           // variabila membru y
b.a.x;         // variabila membru x a variabilei membru a
```

Constructorul clasei CB poate fi scris în forma:

```
CB::CB(int x0,int y0):a(x0)           // forma 1
{
    y=y0; // corpul propriu-zis al constructorului
}
```

sau în forma echivalentă:

```
CB::CB(int x0,int y0):a(x0),y(y0) // forma 2
{
}
```

În ambele forme remarcăm faptul că constructorul instanței încuibărîte *a* trebuie specificat la implementarea constructorului clasei CB conform sintaxei prezentate. Dacă acest lucru nu este făcut compilatorul folosește constructorul implicit (sau luând valori implicate) al clasei CA. Dacă acesta nu există se generează eroare la compilare.

Menționăm că construcția :a(x0) este **apel explicit de constructor** (deci nu se precizează tipul ci doar valoarea de apel).

Remarcăm forma echivalentă de declarare a constructorului (forma2) la care și inițializarea variabilei membru y se face asemănător cu inițializarea instanței încuibărîte. Se accentuează astfel faptul că tipurile predefinite și clasele definite de utilizator se comportă la fel (tipurile predefinite și clasele sunt ambele tipuri de date).

În toate situațiile constructorul instanței încuibărîte se execută înainte de execuția constructorului clasei "încuibăritoare" :-)

Dacă o clasă are mai multe instanțe încuibărîte trebuie menționat la implementarea constructorului acesteia (și nu la declararea lui) lista apelurilor constructorilor instanțelor încuibărîte. Aceștia se vor executa (într-o ordine însă nedefinită) **înainte** de a trece la constructorul clasei "încuibăritoare".

În mod evident dacă clasa "încuibăritoare" nu conține o instanță a clasei "încuibărîte" ci doar un pointer spre un asemenea obiect atunci sarcina instanțierii și distrugerii obiectului indicat de acel pointer ne revine în totalitate (de obicei problema se rezolvă pe constructorul/destructorul clasei "încuibăritoare").

Din punct de vedere al dimensiunii obiectului rezultat avem următoarele:

Dimensiunea unui obiect al clasei obținute este egală cu suma dimensiunii instanțelor încuibărite plus suma dimensiunii membrilor obișnuiți adăugați

S-a obținut astfel un obiect mai mare, care include instanțele încuibărite, dar sintaxa modului de acces la membrii astfel adăugați indică faptul că membrii obținuți din instanțele încuibărite nu sunt chiar membri direcți ai obiectului mare.

Ceea ce ar fi de dorit ar fi ca clasa "mai mare" să preia membrii clasei "mici" într-un mod asemănător cazului când aceștia ar fi membri proprii (cu sintaxa de acces cea simplă). Acest lucru este realizat prin moștenire.

2.2 Moștenirea

Reluând exemplul anterior dar considerând moștenire avem:

```
class CBaza
{
public:
    int x;
    CBaza() {x=0;} // constructor implicit
    CBaza(int x0) {x=x0;} // constructor explicit
};

class CDeriv: public CBaza // aici declarăm moștenirea
{
public:
    int y;
    CDeriv(); // constructor implicit
    CDeriv(int x0,int y0); // constructor explicit
};

CDeriv b(1,2);
```

În acest caz variabilele x și y se pot accesa folosind sintaxa normală, asemănătoare cazului în care membrii moșteniți fac parte efectiv din obiectul rezultat (sintaxa este valabilă și pentru cazul apelului eventualelor metode):

```
b.y; // variabila membru y adăugată
b.x; // variabila membru x obținută prin moștenire
```

Constructorul clasei CDeriv poate fi scris asemănător în forma:

```
CDeriv::CDeriv(int x0,int y0):CBaza(x0) // forma 1
{
```

```
y=y0;  
}
```

sau în forma echivalentă prezentată anterior (forma 2):

```
CDeriv::CDeriv(int x0,int y0):CBaza(x0),y(y0) // forma 2  
{  
}
```

Remarcăm că s-a precizat la declararea clasei CDeriv care este clasa (tipul) de bază CBaza și nu o instanță a acesteia (cum a fost în cazul anterior al instanțelor încuibărîte).

În acest mod simplu se pot obține clase noi prin preluarea (și dezvoltarea) celor existente prin adăugarea de noi variabile membru și metode.

Mai riguros declararea unei clase derivate devine:

```
spec_clasa nume_clasa_deriv [:tip_mostenire nume_clasa_baza]  
{  
    [ public:      ] lista_membri_1 ]  
    [ private:     ] lista_membri_2 ]  
    [ ..... ]  
} [lista_objekte];
```

unde "spec_clasa" poate fi "struct" sau "class" (NU și "union") iar "tip_mostenire" poate fi (deocamdată) tot "public" sau "private".

Se remarcă faptul că sintaxa prezentată este o generalizare a cazului de definire a unei clase fără a ne baza pe moștenire (caz în care clasa de bază lipsește).

Din punct de vedere al relației de moștenire clasele implicate se numesc clasă de bază și clasă derivată. Se spune că:

- clasa derivată derivează din clasa de bază, **moștenește** clasa de bază
- clasa de bază se derivează în clasa derivată, **este moștenită** de clasa derivată.

Total se petrece ca și cum obiectul de bază este îmbogățit prin derivare cu variabile membru și metode noi, păstrând ("în sămbure") și tot ce există în clasa de bază. Spre deosebire de cazul instanțelor încuibărîte, în cazul derivării membrii moșteniți sunt **membri propriu-zisi** (direcți) ai clasei rezultate (ca și când clasa derivată ar fi fost declarată de la început, conținând toți membrii – moșteniți sau adăugați).

Programare Orientată pe Obiecte - Principii

Din punct de vedere al dimensiunii obiectului rezultat prin moștenire avem următoarele:

Dimensiunea unui obiect al clasei derivate
este egală cu dimensiunea unui obiect al clasei de bază
plus suma dimensiunii membrilor noi adăugați în clasa derivată

Membrii noi adăugați în clasa derivată au drepturile de acces corespunzătoare modului de definire a lor (ca în cazul fără moștenire). O problemă deosebită este aceea a drepturilor de acces pe care **membrii moșteniți** le au în cadrul clasei derivate în funcție de drepturile de acces avute în clasa de bază și tipul moștenirii folosite. Aceste drepturi sunt prezentate în Tabelul 2.1 (varianta 1, simplificată).

tip moștenire (derivare)	Drept acces avut în clasa de bază	drept acces rezultat în clasa derivată
public	public	public
public	private	inaccesibil !!!!!!
private	public	private
private	private	inaccesibil !!!!!!

Tabelul 2.1 Drepturi de acces rezultate în urma moștenirii

Analizând datele din tabel remarcăm:

- indiferent de tipul moștenirii ce era privat este inaccesibil (în fapt esența încapsulării).
- ce era public respectă tipul moștenirii.

Apare astfel o **dilemă**: cum trebuie alese drepturile de acces în clasa de bază și tipul moștenirii:

- Membrii privați în clasa de bază nu se moștenesc.
- Membrii publici în clasa de bază nu beneficiază de încapsulare.

Deci, membrii privați nu pot fi moșteniți iar membrii publici nu asigură încapsulare, astfel încât oricum am avea de ales (deocamdată) între două reale.

În această dilemă privat / public am avea nevoie de un tip de acces care să se comporte ca și privat pentru încapsulare (pentru exteriorul clasei) dar ca și public pentru moștenire. În acest scop s-a introdus tipul de acces protected și tipul de moștenire protected. Tabelul anterior devine acum Tabelul 2.2 (varianta finală):

Avem deci trei tipuri de derivare:

- **derivare publică** - membrii privați rămân inaccesibili, ceilalți își păstrează tipul.
- **derivare protejată** - membrii privați rămân inaccesibili, ceilalți devin protected.

Cap. 2 – Moștenirea

tip moștenire (derivare)	Drept acces avut în clasa de bază	drept acces rezultat în clasa derivată
public	public	public
public	protected	protected
public	private	inaccesibil !!!!!!!
protected	public	protected
protected	protected	protected
protected	private	inaccesibil !!!!!!!
private	public	private
private	protected	private
private	private	inaccesibil !!!!!!!

Tabelul 2.2 Drepturi de acces rezultate în urma moștenirii

- **derivare privată** - membrii privați rămân inaccesibili, ceilalți devin privați.

Se recomandă folosirea atributului **protected în clasa de bază** (evident, pentru membrii care nu dorim să fie oricum publici) și **derivare publică sau privată** după cum dorim să permitem sau nu derivări ulterioare ale clasei obținute prin derivare.

Subliniem că prin moștenire **nu se modifică tipul de acces la membrii pentru clasa de bază** ci doar accesul la aceștia ca și membri ai clasei derivate.

Dacă considerăm două clase ca în secvența următoare:

```

class CBase
{
private:
    // a      // membrii privati in CBase
protected:
    // b      // membrii protected in CBase
public:
    // c      // membrii publici in Cbase
};

class CDerived: tip_derivare CBase
{
private:
    // d      // membrii privati adaugati in CDerived
protected:
    // e      // membrii protected adaugati in CDerived
public:
    // f      // membrii publici adaugati in CDerived
};

```

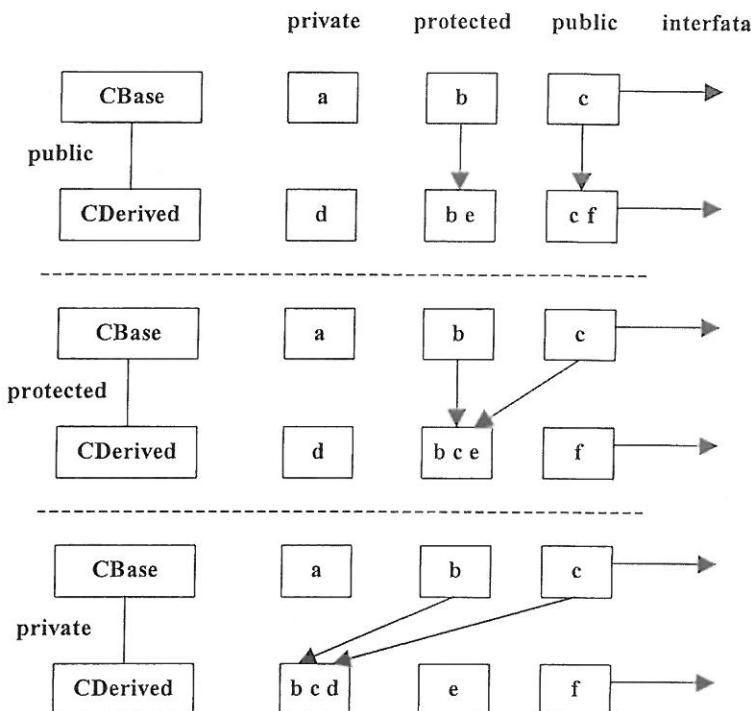


Figura 2.1 “Soarta” membrilor în cazul derivării publice, protejate și private

În Figura 2.1 se prezintă **toți** membrii clasei CDerived și tipul de acces al acestora în funcție de tipul derivării folosite.

Dintre avantajele derivării amintim:

- **economia și reutilizarea de cod** – se pot folosi ușor secvențe de cod gata scrise, timpul de testare scăzând, clasele de bază fiind probabil bine testate. Mai mult, se pot refolosi clase chiar și fără o cunoaștere a modului cum a fost scris codul din spatele lor (și, de obicei, aşa se întâmplă) fiind suficientă descrierea funcționării acelui cod.
- **extensibilitatea** – o clasă disponibilă poate fi extinsă prin derivări ulterioare. Chiar și clase de bibliotecă pentru care nu este disponibil codul sursă pot fi (și de obicei sunt) extinse.
- **polimorfism** – se oferă suport pentru polimorfism, cu toate avantajele ce decurg din aceasta. Se vor trata în capitolul următor.

- **compromisul închis-deschis** – se asigură un compromis între cele două tendințe esențiale și contradictorii ale programării orientate pe obiecte: o clasă trebuie să fie **închisă**, manipulabilă doar prin intermediul unei interfețe bine precizate având detaliile de implementare ascunse, dar în același timp și **deschisă**, permitând extinderea ei ulterioară.

Derivarea are și unele particularități în sensul că nu se moștenesc:

- constructorii și destructorii.
- funcțiile, metodele și clasele friend (se vor prezenta într-un capitol următor).

Riguros, declararea unei clase derivate devine acum:

```
spec_clasa nume_clasa_deriv [:tip_mostenire nume_clasa_baza]
{
    [ public:      ] lista_membri_1 ]
    [ protected:   ] lista_membri_2 ]
    [ private:    ] lista_membri_3 ]
    [ .....        ]
} [lista_objekte];
```

unde "spec_clasa" poate fi "struct" sau "class" (NU și "union") iar "tip_mostenire" poate fi "public", "protected" sau "private". Dacă spec_clasa este "struct" respectiv "class" tip_mostenire implicit este "public" respectiv "private" (la fel ca tipul implicit al membrilor).

2.3 Constructori și destructori în contextul derivării

O problemă care se pune în cazul obiectelor obținute prin derivare este modul cum se construiesc și se distrug aceste obiecte. Regula spune că:

- la instantierea unui obiect dintr-o clasă derivată (global, local sau dinamic, nu are importanță), întâi se execută (implicit sau explicit) corpul constructorului potrivit al clasei de bază și abia apoi corpul constructorului potrivit al clasei derivate.
- la distrugerea unui obiect dintr-o clasă derivată, întâi se execută corpul destructorului clasei derivate și abia apoi destructorul clasei de bază.

Deci, întotdeauna *obiectele se construiesc dinspre interior ("sâmbure") spre exterior ("coajă") și se distrug dinspre exterior spre interior.*

Această ordine de execuție existentă (impusă de limbaj) este singura utilă pentru ca constructorul clasei derivate să se poată baza pe membrii moșteniți din clasa de baza gata

initializați (de către constructorul clasei de bază). Deoarece constructorul clasei de bază nu cunoaște nimic despre eventuala moștenire și existența a membrilor adăugați în clasa derivată cealaltă ordine de initializare (dinspre exterior spre interior) nu ar aduce nici un avantaj.

Dacă există și instanțe încuibările constructorii / destructorii acestora se apelează între apelul constructorilor / destructorilor claselor de bază și a celor derive. Ordinea relativă a apelului constructorilor / destructorilor instanțelor încuibările este nedefinită.

Dacă la implementarea constructorului clasei derive nu se specifică un apel al constructorului clasei de bază compilatorul generează implicit (dacă poate, dacă nu dă eroare) un apel al constructorului implicit sau luând valori implicate al clasei de bază. De cele mai multe ori dorim însă să inițializăm explicit obiectul de bază și atunci trebuie să facem:

2.4 Apel explicit al unui constructor al clasei de baza

conform sintaxei:

```
nume_cl_deriv::nume_cl_deriv([lista_param_1])
    :nume_cl_baza([lista_param_2])
{
    // constructor propriu-zis al clasei derive
    // .....
}
```

Subliniem încă o dată că lista_param_1 este o declarare de parametri (au deci și tip) iar lista_param_2 corespunde unui **apel de metodă** (constructor), deci se pun doar valori (nu și tipuri).

Apelul explicit al constructorului clasei de bază este **obligatoriu** atunci când clasa de bază nu are constructor implicit sau constructor cu toți parametrii luând valori implicate.

Pentru exemple de apel explicit al constructorilor clasei de bază se va urmări exemplul prezentat după capitolul următor legat de conversii.

2.5 Conversii de tip între clase derive

Deoarece o clasă derivată poate fi privită ca o extensie a clasei de bază C++ permite conversia unui obiect de clasă derivată la un obiect de clasă de bază. De asemenea, un pointer la clasa de bază poate ține adresa unei instanțe a clasei derive.

```
class CA           // clasa de baza
{};
class CB:public CA // clasa derivata
{};

void main()
{
CA a;           // doua instantieri automatice (locale)
CB b;

CA* pa = new CA;          // doua instantieri dinamice
CB* pb = new CB;

// conversii corecte
a=b; // operator de atribuire definit implicit de catre C++
pa=pb;

// conversii ilegale !!!!!!!!
b=a;
pb=pa;
}
```

Asemenea conversii se încearcă a se realiza, implicit, la apelul funcțiilor cu parametri de tip diferit de cel așteptat.

Remarcăm un fapt interesant. Dacă într-o clasă de bază un membru era public și, prin derivare, el devine privat, el poate fi totuși accesat (!!!) prin conversia forțată (cast) a unui pointer spre obiectul derivat către pointer spre clasa de bază. Chiar dacă acest lucru este posibil, el **nu este recomandat**.

2.6 Exemplu

Pentru exemplificarea apelului constructorilor / destructorilor în contextul moștenirii prezentăm următorul exemplu:

```
#include <stdio.h>
class CA
{
protected:
    int x,y;      // pentru a putea fi moșteniti
public:
    CA() {x=0; y=0; printf("\nConstr A implicit");}
    CA(int x0,int y0=0)
        {x=x0;y=y0;printf("\nConstr A general");}
```

Programare Orientată pe Obiecte - Principii

```
~CA()      { printf("\nDestructor A      "); }

class CB: public CA      // CB se obtine prin derivare
{
    int z,t;           // membrii noi
public:
    CA a;             // clasa incuibarita
    CB(){z=0;t=0; printf("\nConstr B implicit");}
    CB(int x0,int y0,int z0,int t0=0);
    CB(int z0,int t0);
    ~CB()            { printf("\nDestructor B"); }
};

CB::CB(int x0,int y0,int z0,int t0):CA(x0,y0),z(z0)
{
    t=t0;printf("\nConstr B general 4 parametri");
}

CB::CB(int z0,int t0):z(z0),t(t0)
{
    printf("\nConstr B general 2 parametri");
}

CA a,b(1);           // obiecte globale
CB c,d(2,3),e(4,5,6);

void main()
{
    CA f(7,8),*g;        // obiecte locale
    CB h(9,10,11,12),*i,j[2];

    printf("\nInceput cod main");
    g=new CA(13);        // obiecte dinamice
    i=new CB(14,15,16);
    delete g;
    printf("\nSfirsit cod main");
}
```

care va produce următorul rezultat (se va urmări și înțelege)

Constr A implicit	// pentru instanta a
Constr A general	// pentru instanta b
Constr A implicit	// pentru instanta c
Constr A implicit	// pentru instanta c.a
Constr B implicit	// pentru instanta c
Constr A implicit	// pentru instanta d
Constr A implicit	// pentru instanta d.a
Constr B general 2 parametri	// pentru instanta d

```

Constr A general           // pentru instanta e
Constr A implicit          // pentru instanta e.a
Constr B general 4 parametri // pentru instanta e
Constr A general           // pentru instanta f
Constr A general           // pentru instanta h
Constr A implicit          // pentru instanta h.a
Constr B general 4 parametri // pentru instanta h
Constr A implicit          // pentru instanta j[0]
Constr A implicit          // pentru instanta j[0].a
Constr B implicit          // pentru instanta j[0]
Constr A implicit          // pentru instanta j[1]
Constr A implicit          // pentru instanta j[1].a
Constr B implicit          // pentru instanta j[1]

Inceput cod main
Constr A general           // pentru instanta g
Constr A general           // pentru instanta i
Constr A implicit          // pentru instanta i.a
Constr B general 4 parametri // pentru instanta i
Destructor A               // pentru instanta g

Sfirsit cod main
Destructor B               // pentru instanta j[1]
Destructor A               // pentru instanta j[1].a
Destructor A               // pentru instanta j[1]
Destructor B               // pentru instanta j[0]
Destructor A               // pentru instanta j[0].a
Destructor A               // pentru instanta j[0]
Destructor B               // pentru instanta h
Destructor A               // pentru instanta h.a
Destructor A               // pentru instanta h
Destructor A               // pentru instanta f
Destructor B               // pentru instanta e
Destructor A               // pentru instanta e.a
Destructor A               // pentru instanta e
Destructor B               // pentru instanta d
Destructor A               // pentru instanta d.a
Destructor A               // pentru instanta d
Destructor B               // pentru instanta c
Destructor A               // pentru instanta c.a
Destructor A               // pentru instanta c
Destructor A               // pentru instanta b
Destructor A               // pentru instanta a

```

2.7 Moștenirea multiplă

Deși până acum nu am discutat decât cazul în care o clasă moștenește o singură clasă de bază în C++ s-a introdus și conceptul de moștenire multiplă în care o clasă poate **moșteni mai multe clase de bază**.

Din punct de vedere al dimensiunii obiectului rezultat prin moștenire multiplă (fără a considera și clase virtuale) avem următoarele:

Dimensiunea unui obiect al clasei derivate cu moștenire multiplă este egală cu suma dimensiunii obiectelor de bază plus suma dimensiunii membrilor noi adăugați în clasa derivată

Prezentăm un exemplu în care clasa derivată moștenește două clase de bază:

```
class CBase1
{
public:
    int x;
    int y;
};

class CBase2
{
public:
    int x;
    int z;
};

class CDeriv:public CBase1,public CBase2
{
public:
    int t;
};
```

considerăm o instanță de CDeriv

```
CDeriv d;
```

În acest caz membrii se pot accesa în felul următor (simplu):

```
d.t  
d.y  
d.z
```

Deoarece ambele clase de bază au un membru x accesul următor este găsit eronat de către compilator datorită **ambiguității** apartenenței lui x – la Cbase1 sau Cbase2:

```
d.x
```

Soluția eliminării ambiguității constă în prefixarea membrului cu numele clasei căreia îi aparține, folosind “scope access operator” (de rezoluție):

```
d.CBase1::x      // un membru x  
d.CBase2::x      // celălalt membru x
```

Deși exemplul a fost prezentat doar cu variabile membru totul se petrece la fel și cu metode. Toată discuția legată de drepturi de acces în clasa de bază și în clasa derivată în funcție și de tipul derivării rămâne valabilă de la moștenirea simplă.

Evident metodele fiecărei clase de bază își acceseză în continuare fără probleme propriile variabile membru.

Ca și restricție, o clasă nu poate moșteni o aceeași clasă de bază de mai multe ori, următoarele exemple fiind **eroneate**

```
class CA  
{};  
  
class CB:public CA  
{};  
  
class CC:public CA,public CA      // eronat  
{};  
  
class CD:public CA,public CB      // eronat  
{};
```

dar situația următoare este considerată **validă** de către compilator:

```
class CA  
{  
public:  
    int x;  
};  
  
class CB:public CA  
{};  
  
class CC:public CA  
{};  
  
class CD:public CB,public CC      // corect  
{  
    int a,b;  
};
```

Din nou avem două copii ale clasei CA care însă **pot fi diferențiate** folosind operatorul de

rezoluție “::” și numele clasei (CB sau CC).

Pentru a realiza o moștenire multiplă este suficient să se specifice după numele clasei derivate o listă de clase de bază, separate prin virgulă (evident, acestea pot fi prefixate fiecare cu tipul moștenirii). Atragem atenția însă că **ordinea claselor de bază nu este indiferentă, apelul constructorilor claselor de bază făcându-se în ordinea enumerării claselor de bază** (indiferent de ordinea apelurilor explicite a constructorilor claselor de bază).

Pentru exemplul anterior în cazul declarării unui constructor în felul următor:

```
CD::CD(): a(1),CC(2), b(3), CB(4)
{
    // .....
}
```

succesiunea operațiilor va fi

- apelul constructorului CB(4) (evident, presupunem că el există)
- apelul constructorului CC(2) (evident, presupunem că el există)
- inițializarea membrilor a și b cu 1 și 3

2.8 Moștenire multiplă virtuală

Dacă dorim ca să avem o singură copie a clasei de bază moștenită multiplu atunci moștenirea trebuie făcută **virtuală** ca în exemplul următor:

```
class CB
{
public:
    int x;
};

class CD1:virtual public CB      // moștenire virtuală
{
};

class CD2:virtual public CB      // moștenire virtuală
{
};

class CM:public CD1,public CD2
{
};
```

```
#include <stdio.h>

void main()
{
    CM m1;

    m1.x=0;      printf("\n%d %d %d",m1.x,m1.CD1::x,m1.CD2::x);
    m1.CD1::x=1; printf("\n%d %d %d",m1.x,m1.CD1::x,m1.CD2::x);
    m1.CD2::x=2; printf("\n%d %d %d",m1.x,m1.CD1::x,m1.CD2::x);
}
```

La execuție programul tipărește:

```
0 0 0
1 1 1
2 2 2
```

ceea ce dovedește că există o singură copie a variabilei membru **x** care poate fi accesată prin cele trei modalități: `m1.x`, `m1.CD1::x`, `m1.CD2::x`.

Dacă cumva “pe o ramură” moștenirea nu este virtuală problema ambiguității rămâne în actualitate ceea ce impune folosirea operatorului “`::`” și prefixarea cu numele clasei.

În cazul existenței moștenirii virtuale, ordinea de apel a constructorilor claselor de bază este:

- constructorii claselor de bază virtuale.
- constructorii claselor de bază ne-virtuale.

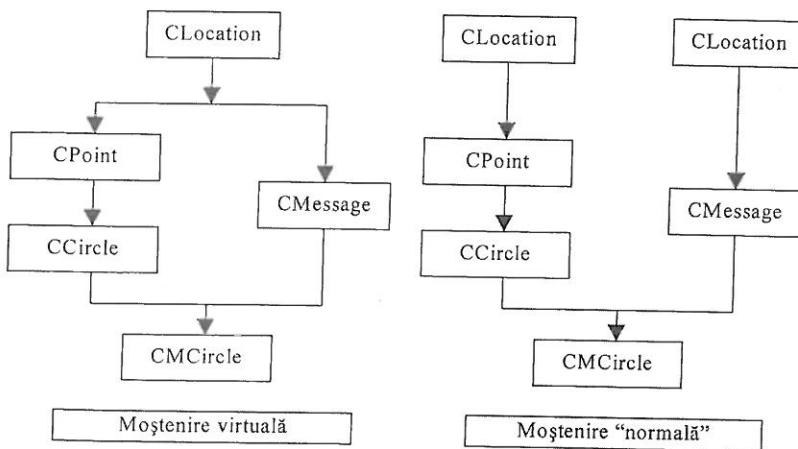


Figura 2.2 - Exemplu de moștenire multiplă

Constructorul claselor de bază se execută:

- o singură dată pentru clasele virtuale.
- câte o dată pentru fiecare clasă de bază nevirtuală din ierarhia respectivă.

Ca și un exemplu de moștenire multiplă am prezentat ierarhia de clase din Figura 2.2. În aceasta cazul din stânga presupune moștenire virtuală a clasei CLocation (cercul și mesajul au aceeași poziție) iar cazul din dreapta presupune moștenire normală (cercul și mesajul au poziții diferite). Pe Figura 2.2 am încercat să sugerăm instanțele existente nu strict relațiile de moștenire dintre clase.

Rezumat - de ținut minte :-)

- instanțe **încuibările**
- **moștenire** - declarare și semnificație
- **tip de acces rezultat** pentru membrii moșteniți
- necesitatea și rolul apariției tipului de acces **protected**
- constructori și destrutori în contextul derivării
- **apel explicit** constructor clasă de bază
- **conversii valide** între clase derivate
- moștenirea **multiplă** - ambiguitatea și operatorul **::**
- moștenirea multiplă **virtuală** - o singură copie a clasei de bază

Cap. 3 - Polimorfismul

3.1 Supraîncărcarea metodelor în contextul moștenirii

Atragem atenția ca cele prezentate în acest subcapitol nu reprezintă polimorfism ci doar explică contextul în care acesta va fi introdus.

Să considerăm exemplul următor:

```
#include <stdio.h>

class CA
{
public:
    met_1(){printf("\nMetoda 1 clasa CA");}
    met_2(){printf("\nMetoda 2 clasa CA");}
    met_3(){printf("\nMetoda 3 clasa CA");met_1();met_2();}
};

class CB:public CA
{
public:
    met_1(){printf("\nMetoda 1 clasa CB");}
    met_2(){printf("\nMetoda 2 clasa CB");}
};

void main()
{
CA a;
CB b;

a.met_1();
a.met_2();

b.met_1();
b.met_2();

a.met_3();
b.met_3();      // disponibila prin moștenire
}
```

care va produce următorul rezultat:

```
Metoda 1 clasa CA          // a.met_1()
Metoda 2 clasa CA          // a.met_2()
Metoda 1 clasa CB          // b.met_1()
Metoda 2 clasa CB          // b.met_2()
Metoda 3 clasa CA          // a.met_3()
Metoda 1 clasa CA          //
Metoda 2 clasa CA          //
Metoda 3 clasa CA          // b.met_3()
Metoda 1 clasa CA          // !!!!
Metoda 2 clasa CA          // !!!!
```

Remarcăm faptul că în CA avem definite metodele met_1(), met_2() și met_3(). Clasa CB **supraîncarcă** (înlocuiește) metodele met_1() și met_2() și păstrează prin moștenire met_3(). Reamintim că, dacă noile metode prezintă semnături diferite (alta valoare returnată sau alt număr și tip de parametri) clasa CB va avea prin moștenire și vechile variante ale metodelor iar compilatorul va decide prin diferențele de semnătură care variantă este cea apelată.

Clasele din exemplul anterior au deci fiecare câte trei metode:

CA	met_1()	obținută prin definire
	met_2()	obținută prin definire
	met_3()	obținută prin definire
CB	met_1()	obținută prin supraîncărcare
	met_2()	obținută prin supraîncărcare
	met_3()	obținută prin moștenire

Interpretând rezultatele rulării remarcăm că apelurile către met_1() și met_2() și chiar a.met_3() au avut loc conform așteptării. Rezultatul interesant este acela al apelului b.met_3() care, deși este metodă în CB și este apelată pentru o instanță a clasei CB, apelează met_1() și met_2() ale clasei de bază. Cauza acestei comportări este legarea statică a metodelor, după cum vom explica în continuare.

Dacă se dorește apelarea metodei met_1() din clasa CA pentru obiecte ale clasei CB se poate folosi ”scope access operator” ca în exemplul următor:

```
b.CA::met_1();
```

care va produce rezultatul:

```
Metoda 1 clasa CA           // b.CA::met_1()
```

3.2 Polimorfism – generalități

Pentru abordarea polimorfismului trebuie plecat de la originea greacă a cuvintelor ce îl compun: “poli” înseamnă “multe” și “morphos” înseamnă “formă”. În domeniul despre care discutăm (al programării orientate pe obiecte) acest termen descrie proprietatea unei aplicații de a se comporta diferit (în mai multe forme) în funcție de condițiile **din momentul rulării**.

Mai specific, problema apare când se apelează o metodă (cu nume și parametri cunoscuți) a unui obiect dar nu se cunoaște exact tipul obiectului. Chiar dacă problema pare puțin ciudată (**putem avea instanțiate obiecte fără a le cunoaște tipul ?**) ea este reală, existând un (singur) caz în care acest lucru este posibil: acela al pointerilor la obiecte ale unei clase de bază care pot ține adresa atât a unor obiecte ale clasei de bază cât și (mai corect spus sau) a oricărora obiecte din clase derivate din această clasă de bază.

Rezultă de aici o observație esențială: în **C++ polimorfismul se manifestă doar în contextul moștenirii**. Acest fapt justifică și ordinea în care noțiunile de încapsulare, moștenire și polimorfism au fost introduse.

3.3 Legare statică și dinamică

În acest context numim **legare** (“binding”) mecanismul prin care compilatorul generează cod pentru apelul unei anumite metode (cum face legătura între numele metodei și codul acesteia).

Cele două soluții consacrate pentru această problemă sunt:

legarea statică (timpuriu – “early binding”)

În acest caz compilatorul și editorul de legături fac corespondență între numele metodei și adresa acesteia din segmentul de cod. Odată fixată această legătură **în momentul compilării / link-editării** aceasta este definitivă în tot timpul rulării programului. La nivelul cel mai de jos, legarea statică corespunde apelului direct al unei proceduri (folosind de exemplu o instrucție de tip CALL direct). Aceasta este modul de legare clasic folosit de toate limbajele structurate (ne-obiectuale) și de limbajele obiectuale pentru metodele ne-virtuale (clasice).

legarea dinamică (târzie – “late binding”)

În acest caz compilatorul și editorul de legături fac corespondența între numele metodei și adresa acesteia din segmentul de cod prin intermediul unui tabel de adrese posibile de salt. Alegerea adresei efective de apel se face **în momentul rulării**, în funcție de obiectul pentru care se apelează (deci, decizia legării este întârziată). La nivelul cel mai de jos, legarea dinamică corespunde apelului indirect al unei proceduri (folosind de exemplu o instrucțiune de tip CALL indirect). Acesta este modul de legare folosit de limbajele obiectuale pentru metodele virtuale.

Există o butadă în știința calculatoarelor conform căreia evoluția sa are la bază doar două alternative: utilizarea unei memorii cache și adăugarea unui nivel suplimentar de indirectare.

Polimorfismul fiind un exemplu de soluție având la bază indirectare prezintă binecunoscutele avantaje și dezavantaje ale acestor metode:

- avantaje: se asigura (introduce) un grad de **flexibilitate**
- dezavantaj: ca în orice apel indirect, se înrăutățesc performanțele de **viteză**

Compromisul avantaje / dezavantaje (flexibilitate / viteză) se poate stabili de către programator, prin posibilitatea alegерii, pentru fiecare metodă în parte, a tipului de legare: statică sau dinamică.

3.4 Metode virtuale

După cum rezultă din cele de mai înainte trebuie să existe o posibilitate de a stabili tipul legării folosite pentru fiecare metodă în parte. Aceasta are la bază cuvântul cheie virtual care prefixează definirea unei metode. Avem deci:

- **Metode ne-virtuale** (normale, clasice) la care legarea este statică. Sunt definite fără a se folosi cuvântul virtual.
- **Metode virtuale** la care legarea este dinamică. Se definesc prin prefixarea definirii metodei cu cuvântul cheie virtual.

Reluăm exemplul de la începutul materialului cu singura modificare că met_1() se va defini virtuală:

```
#include <stdio.h>

class CA
```

```

{
public:
virtual met_1(){printf("\nMetoda 1 clasa CA");}
    met_2(){printf("\nMetoda 2 clasa CA");}
    met_3(){printf("\nMetoda 3 clasa CA");met_1();met_2();}
};

class CB:public CA
{
public:
    met_1(){printf("\nMetoda 1 clasa CB");}
    met_2(){printf("\nMetoda 2 clasa CB");}
};

void main()
{
CA a;
CB b;

a.met_1();
a.met_2();

b.met_1();
b.met_2();

a.met_3();
b.met_3(); // disponibila prin mostenire
}
}

```

care va produce următorul rezultat

Metoda 1 clasa CA	// a.met_1()	legare dinamica
Metoda 2 clasa CA	// a.met_2()	legare statica
Metoda 1 clasa CB	// b.met_1()	legare dinamica
Metoda 2 clasa CB	// b.met_2()	legare statica
Metoda 3 clasa CA	// a.met_3()	
Metoda 1 clasa CA	//	
Metoda 2 clasa CA	//	
Metoda 3 clasa CA	// b.met_3()	
Metoda 1 clasa CB	// !!!! aici apare modificarea	
Metoda 2 clasa CA	// !!!! aici nu apare modificare	

Remarcăm faptul că, în acest caz, metoda met_1() a fost legată dinamic (în implementarea metodei met_3() din CA) și, pentru obiectul CB, a fost înlocuită de noua versiune a acesteia. Astfel, totul se petrece ca și când clasa derivată a modificat cod scris într-o clasă de bază !!! (la care evident nu are acces, putând fi chiar scris de altcineva și disponibil doar sub forma de bibliotecă). Pentru ca acest lucru să poată avea loc, este strict necesar "ajutorul" clasei de bază,

care a "consumat" (prin declararea metodei ca virtuală) la legarea dinamică a acestei metode.

Atragem atenția asupra faptului că "atributul" virtual se aplică individual funcțiilor și nu se "propagă în jos" cum se întâmplă cu public, protected și private. Deci, în exemplul anterior met_2() nu este virtual.

Această comportare deosebită a metodelor legate dinamic (virtuale) ridică următoarea problemă:

3.5 Cum se implementează legarea dinamică (polimorfismul) ?

Pentru a explica acest lucru să revenim la exemplul cu clasele CA și CB pe care îl mai dezvoltăm puțin.

```
class CA
{
public:
    virtual met_1() { /* ... */ }
    virtual met_2() { /* ... */ }
    met_3() { /* ... */ }
    virtual met_4() { /* ... */ }
    met_5() { met_1(); met_2(); met_3(); met_4(); }
    // variabile membru CA
};

class CB: public CA
{
public:
    met_1() { /* ... */ }
    met_3() { /* ... */ }
    met_4() { /* ... */ }
    // variabile membru CB
};

CA a,a1,a2; CB b,b1;      // cîteva instantieri
```

Pentru fiecare clasă compilatorul construiește o tabelă cu metodele virtuale ale clasei respective tabelă numită VMT (Virtual Method Table) și include în fiecare instanță (obiect) a clasei un pointer VMT_ptr spre tabela VMT corespunzătoare clasei.

Accentuăm faptul că pentru o metodă ne-virtuală legarea este necondiționat statică iar pentru o metodă virtuală legarea este necondiționat dinamică (âtât la apel direct prin obiect sau prin pointer la obiect cât și la apel din cadrul altor metode).

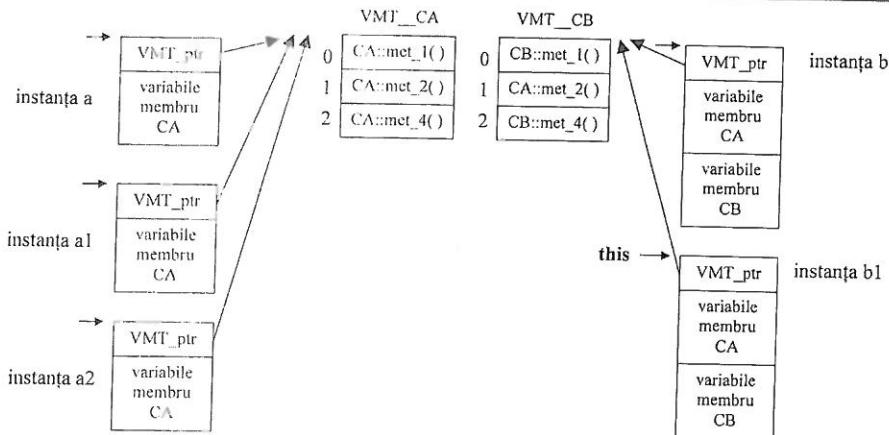


Figura 3.1 Tabelele VMT

Având în vedere schema de mai sus putem înțelege modul cum se implementează, de către compilator, metoda **met_5()**:

```

VMT[0]()      // apel metoda prin VMT  (legare dinamica)
VMT[1]()      // apel metoda prin VMT  (legare dinamica)
CA::met_3()    // apel metoda          (legare statica)
VMT[2]()      // apel metoda prin VMT  (legare dinamica)
  
```

Pentru **a.met_5()** acesta este echivalent cu:

```

CA::met_1();    // apel metoda legata prin VMT
CA::met_2();    // apel metoda legata prin VMT
CA::met_3();    // apel metoda legata static
CA::met_4();    // apel metoda legata prin VMT
  
```

iar pentru **b.met_5()** este echivalent cu:

```

CB::met_1();    // apel metoda legata prin VMT inlocuita
CA::met_2();    // apel metoda legata prin VMT neinlocuita
CA::met_3();    // apel metoda legata static
CB::met_4();    // apel metoda legata prin VMT inlocuita
  
```

Mai detaliat avem: pentru apelul **VMT[i]** se accesează, prin intermediul lui **this**, zona de date a obiectului și se obține **VMT_ptr**. Prin intermediul lui **VMT_ptr** se accesează tabela VMT din care, prin indexare, se determină adresa metodei **VMT[i]**.

Accentuăm faptul că **polimorfismul se manifestă la nivel de clasă și nu la nivel de instanță.**

De asemenea, acum putem explica dimensiunea unui obiect având metode virtuale: indiferent de numărul metodelor virtuale ale clasei respective (evidenț, cel puțin una).

Dimensiunea unui obiect cu metode virtuale este egală cu dimensiunea variabilelor membru (adăugate sau moștenite) plus dimensiunea unui pointer (VMT_ptr)

3.6 Ce poate fi virtual și ce nu

Deși atributul virtual prefixează declarații de metode, nu orice metodă poate fi declarată virtuală. Astfel:

- **Constructorii nu pot fi virtuali.** Oricum, prin mecanismul apelului constructorului clasei derivate ulterior apelului constructorului clasei de bază, se poate modifica comportamentul obiectului rezultat (anulând eventuale acțiuni nedorite ale constructorului clasei de bază). Deși constructorii virtuali ar fi utili pentru evitarea apelului constructorului clasei de bază, lipsa constructorilor virtuali poate fi suplinită prin apelarea unei metode virtuale în corpul constructorului nevirtual.
- **Destructorii pot fi virtuali.** Neavând tip returnat și nici parametrii este evident că el este unic pentru fiecare clasă. Această facilitate este salvatoare în cazul distrugerii uniforme de masive eterogene.
- **Metodele statice nu pot fi virtuale.** Acest lucru este evident, având în vedere faptul că ele nu sunt metode propriu-zise, deci nu au pointerul this și deci nu au acces la VMT.
- **Metodele friend pot fi virtuale** dar nu și funcțiile friend (care, ca și metodele statice, nu aparțin unui obiect).
- **Metodele operator pot fi virtuale** dar nu și operatorii redefiniți prin funcții friend.

3.7 Un exemplu distractiv și nu numai....

Să considerăm următorul exemplu care, chiar dacă aparent este doar distractiv, evidențiază interacțiunea metodelor virtuale cu celelalte tipuri de metode într-o ierarhie de clase.

```
#include <stdio.h>

class fairy_tale
{
public:
    void act0(){printf("0 Inceput poveste originala\n");           act1();}
    virtual void act1(){printf("1 Printesa intinde bruscoul\n");      act2();}
    void act2(){printf("2 Printesa saruta bruscoul\n");            act3();}
}
```

Cap. 3 - Polimorfismul

```
virtual void act3(){printf("3 Broscoiul se transforma in Print\n"); act4();}
virtual void act4(){printf("4 Si au trait multi ani fericiti\n"); act5();}
void act5(){printf("5 Sfirsit fericit\n");}
};

class unhappy_tale: public fairy_tale
{
public:
    void act0(){printf("0 Inceput poveste modificata\n");
    void act2(){printf("2 Printesa atinge broscoiul\n");
    void act3(){printf("3 Broscoiul ramine broscoi\n");
    void act4(){printf("4 Printesa fugе ingrozita\n");
    void act5(){printf("5 Sfirsit nu prea fericit\n");
};

void main()
{
fairy_tale *tale;

printf("FAIRY TALE\n");
tale = new fairy_tale;
tale->act0();
delete tale;

printf("UNHAPPY TALE\n");
tale = new unhappy_tale;
tale->act0();
delete tale;
}
```

care va produce următorul rezultat

```
FAIRY TALE
0 Inceput poveste originala
1 Printesa intilneste broscoiul
2 Printesa saruta broscoiul
3 Broscoiul se transforma in Print
4 Si au trait multi ani fericiti
5 Sfirsit fericit
UNHAPPY TALE
0 Inceput poveste originala
1 Printesa intilneste broscoiul
2 Printesa saruta broscoiul
3 Broscoiul ramine broscoi
4 Printesa fugе ingrozita
5 Sfirsit nu prea fericit
```

Analizând rezultatul remarcăm următoarele:

- **funcțiile virtuale nu trebuie neapărat să fi înlocuite.** Clasa unhappy_tale nu furnizează înlocuitor pentru act1() dar aceasta este totuși disponibilă prin moștenire.

VMT fairy_tale		VMT unhappy_tale	
act1()	fairy_tale::act1()	act1()	fairy_tale::act1()
act3()	fairy_tale::act3()	act3()	unhappy_tale::act3()
act4()	fairy_tale::act4()	act4()	unhappy_tale::act4()

Figura 3.2 Tabelele VMT pentru "povești"

- **funcțiile virtuale pot apela funcții ne-virtuale.** act1() virtuală apelează act2() ne-virtuală. Deși obiectul apelat este o instanță unhappy_tale se apelează tot act2() de la happy_tale. Cauza: legarea statică pentru act2().
- **funcțiile ne-virtuale pot apela funcții virtuale.** act2() ne-virtuală apelează act3() virtuală. act3() efectiv apelată depinde de obiectul pentru care se efectuează apelul. Cauza: legarea dinamică pentru act3().
- **funcțiile virtuale pot apela alte funcții virtuale.** act3() virtuală apelează act4() virtuală. act4() efectiv apelată depinde de obiectul pentru care se efectuează apelul. Cauza: legarea dinamică pentru act4().
- **funcțiile virtuale pot apela funcții supraîncărcate.** act4() virtuală apelează act5() care este supraîncărată. act5() efectiv apelată depinde de obiectul în care se efectuează (implementează) apelul. Cauza: legarea statică diferită în cele două implementări ale lui act4(). Atenție, nu avem legare dinamică pentru act5().
- **funcțiile supraîncărcate pot fi apelate direct.** act0() supraîncărată este apelată direct. În acest caz avem legare statică a metodei dependentă de tipul pointerului implicat după cum rezultă din exemplul următor de apel:

```

unhappy_tale *tale;

printf("UNHAPPY TALE\n");
tale = new unhappy_tale;
tale->act0();
delete tale;

```

care tipărește:

```

UNHAPPY TALE
0 Inceput poveste modificata      // aici apare modificarea
1 Printesa intilneste broscoiul
2 Printesa saruta broscoiul
3 Broscoiul ramine broscoi
4 Printesa fuge ingrozita
5 Sfirsit nu prea fericit

```

Pentru înțelegerea acestor situații recomandăm vizualizarea tabelelor metodelor virtuale pentru clasele `fairy_tale` și `unhappy_tale` (prezentată în Figura 3.2).

Având această viziune a tabelelor VMT și bazându-ne pe legarea dinamică a metodelor virtuale să se explice (înțeleagă) comportamentul claselor `fairy_tale` și `unhappy_tale` prezentat anterior.

Ca și regulă putem spune că, la apelul unei metode a unui obiect prin intermediul unui pointer spre obiect, avem:

- dacă legarea metodei este **statică** contează **tipul pointerului** (și nu tipul **obiectului** a cărui adresă o reține la acel moment pointerul).
- dacă legarea metodei este **dinamică** contează **tipul obiectului** a cărui adresă o reține la acel moment pointerul (și nu tipul pointerului - care ar putea fi de tipul obiectului sau de tipul unei clase de bază).

3.8 Metode virtuale și ierarhii de clase

În cele ce urmează ne propunem să analizăm modul în care se “propagă” caracterul de virtualitate al metodelor într-o ierarhie de clase. Pentru aceasta considerăm următorul exemplu:

```
#include <stdio.h>

class CA
{
public:
    virtual void met_1(void)    {/*...*/}
    virtual void met_2(void)    {/*...*/}
    virtual void met_3(void)    {/*...*/}
    virtual void met_4(void)    {/*...*/}
};

class CB: public CA
{
public:
    virtual void met_1(void)    {/*...*/}
    void met_2(void)           {/*...*/}
    void met_4(int)            {/*...*/}
};

class CC: public CB
{
public:
    virtual void met_4(int)    {/*...*/}
```

};

și un program principal care să instanțieze asemenea clase:

```
void main()
{
CA a,*pa;CB b,*pb;CC c;

    a.met_4();      // corect
    pa=new CA;
    pa->met_4();  // corect
    delete pa;

//  b.met_4();      // incorrect, met_4(void) nu e vizibila
//  pb=new CB;
//  pb->met_4();  // incorrect, met_4(void) nu e vizibila
    delete pb;

    pa=new CB;
    pa->met_4();  // corecta !!!!, foloseste VMT-ul lui CB
    delete pa;
}
```

Tabelele VMT pentru clasele în discuție arată astfel:

VMT CA
CA::met_1(void)
CA::met_2(void)
CA::met_3(void)
CA::met_4(void)

VMT CB
CB::met_1(void)
CB::met_2(void)
CA::met_3(void)
CA::met_4(void)

VMT CC
CB::met_1(void)
CB::met_2(void)
CA::met_3(void)
CA::met_4(void)
CC::met_4(int)

Din exemplul prezentat putem trage următoarele concluzii:

- redefinirea unei metode virtuale tot cu una având atributul virtual înlocuiește în VMT cea veche cu cea nouă. Exemplul: met_1.
- redefinirea unei metode virtuale cu una neavând atributul virtual se comportă la fel ca în cazul anterior. Exemplul: met_2. Deci, odată impus caracterul virtual, acesta "nu se mai pierde". Se recomandă totuși rescrierea cuvântului virtual și în clasele derivate (deși este redundant) pentru o înțelegere mai rapidă la citirea codului.
- neredefinirea unei metode virtuale păstrează în VMT ultima variantă disponibilă. Exemplu: met_3.
- redefinirea unei metode virtuale cu una având același nume dar parametrii diferiți nu o

pune pe cea nouă în VMT (deci aceasta nu este virtuală). Exemplu: met_4(int).

- met_4(int) poate fi redefinită ulterior ca fiind virtuală. Exemplu: met_4(int) în CC.
- redefinirea unei metode virtuale cu una având același nume dar parametrii diferiți ascunde pe cea veche (din punct de vedere al domeniului de vizibilitate și nu din acela al VMT). Exemplu: liniile eronate din programul prezentat. Totuși, metoda veche încă există în VMT și poate fi accesată prin pointer la clasa de bază (unde metoda era vizibilă). Exemplu: ultimul apel din main.

Mai amintim faptul că o metodă (deci și o metodă virtuală) nu poate fi redefinită cu aceiași parametrii dar având tip returnat diferit.

Dacă dorim să evităm mecanismul de polimorfism avem întotdeauna la dispoziție operatorul "scope access resolution" prin care putem impune metoda apelată (fortând legare statică).

3.9 Metode pure și clase abstracte

Uneori se pune problema să proiectăm clase cu un caracter de generalitate, fără a ne referi la o particularizare a acestora. De exemplu, putem avea în vedere proiectarea unei ierarhii de clase destinață gestionării de obiecte cum ar fi cerc, pătrat, triunghi, etc. Toate acestea au în comun faptul că au o poziție, o dimensiune, o culoare și diferă doar prin modul în care se afișează și se sterg. Este de dorit să "prindem" într-o clasă de bază comportamentul comun și, prin derivare și polimorfism, să particularizăm comportamentul pentru clasele derivate specializate. Clasa de bază ar putea fi scrisă astfel

```
class CFigura
{
    int x,y,cul, dim;           // poz x, y, culoarea si dimensiunea
    void Muta(int dx,int dy);
    virtual void Afiseaza();    // există pentru a putea fi apelate de Muta
    virtual void Sterge();      // fi apelate de Muta
    // alte metode, constructori și destrutori
};
```

Comportamentul comun al tuturor claselor derivate din CFigura poate fi "prins" în metode ale acestei clase de bază. Indiferent de ce tip de figură este vorba, mutarea se face la fel

```
void CFigura::Muta(int dx,int dy)
{
    Sterge();                  // stergere
    x+=dx;y+=dy;              // mutare propriu-zisa
```

```
Afiseaza(); // afisare  
}
```

Prin derivări ale clasei CFigura se pot defini figuri diferite care își particularizează afișarea și stergerea dar care păstrează metoda de mutare din clasa de bază ("modificată" prin polimorfism).

Soluția are dezavantajul că permite instanțierea de obiecte din clasa CFigura care nu sunt complet definite. Pentru a evita instanțierea de obiecte din clase "neterminate" s-a introdus noțiunea de metodă pură pentru metodele virtuale.

O metodă virtuală se definește ca fiind **pură** dacă la declarare corpul ei se face egal cu 0 ca în exemplul următor:

```
class CFigura  
{  
    int x,y,cul,dim; // poz x, y, culoarea si dimensiunile  
  
    void Muta(int dx,int dy);  
    virtual void Afiseaza()=0; // aici se defineste ca pură  
    virtual void Sterge()=0; // aici se defineste ca pură  
    // alte metode, constructori si deconstructori  
};
```

metoda Muta rămânând neschimbată.

O metodă definită pură nu se implementează în clasa respectivă. Rolul ei este doar acela de a fi înlocuită într-o clasa derivată. Alte metode ale clasei însă o pot folosi ca și cum ar exista deja implementață.

O clasă cu cel puțin o metodă pură se numește **clasă abstractă**. Clasele abstracte sunt importante deoarece **compilatorul nu permite instanțierea unei clase abstracte**.

Rolul unei clase abstracte este de a fi moștenită într-o clasă derivată care să înlocuiască **toate** metodele "pure". În general asemenea clase se găsesc înspre "rădăcina" ierarhiei de clase.

3.10 Tratarea uniformă a masivelor eterogene

Una din **cele mai importante aplicații ale polimorfismului** este tratarea uniformă a unor masive (tablouri, liste, etc...) eterogene în sensul că sunt constituite din obiecte de tipuri diferite. Soluția se bazează pe posibilitatea unui pointer spre o clasă de a reține adresa unor

Cap. 3 - Polimorfismul

instanțe ale acelei clase dar și a unor instanțe ale unor clase deriveate. În acest context polimorfismul permite o tratare uniformă a unor asemenea masive prin legarea dinamică pe care o presupune.

În continuare exemplificăm această posibilitate:

```
#include <stdio.h>

class CA
{
public: int x;
    CA(int x0){x=x0;}
    virtual ~CA(){printf("\nDestr CA (%d)",x);}
    virtual met(){printf("\nMetoda CA (%d)",x);}
};

class CB:public CA
{
public: int y;
    CB(int x0,int y0):CA(x0){y=y0;}
    ~CB(){printf("\nDestr CB (%d,%d)",x,y);}
    met(){printf("\nMetoda CB (%d,%d)",x,y);}
};

void main()
{
    CA* tblEt[6];

    tblEt[0] = new CA(1);      // construim un masiv eterogen
    tblEt[1] = new CA(2);
    tblEt[2] = new CB(3,4);
    tblEt[3] = new CA(5);
    tblEt[4] = new CB(6,7);
    tblEt[5] = new CB(8,9);

    for(int k=0;k<6;k++)      // aici tratarea e uniformă,
       tblEt[k]->met();      // comportamentul e polimorf

    for(k=0;k<6;k++)
        delete tblEt[k];      // distrugere polimorfa
}
```

Prezentăm în ceea ce urmează rezultatele rulării programului în cazul în care metoda met() și destructorul nu sunt virtuale (deci nu ne bazăm pe polimorfism) și în cazul în care acestea sunt virtuale (deci ne bazăm pe polimorfism).

Programare Orientată pe Obiecte - Principii

<u>met()</u> și ~CA() <u>nevirtuale</u>	<u>met()</u> și ~CA() <u>virtuale</u> (ca în exemplu)
Metoda CA (1) // [0]	Metoda CA (1) // [0]
Metoda CA (2) // [1]	Metoda CA (2) // [1]
Metoda CA (3) // [2]	Metoda CB (3,4) // [2]
Metoda CA (5) // [3]	Metoda CA (5) // [3]
Metoda CA (6) // [4]	Metoda CB (6,7) // [4]
Metoda CA (8) // [5]	Metoda CB (8,9) // [5]
Destr CA (1) // [0]	Destr CA (1) // [0]
Destr CA (2) // [1]	Destr CA (2) // [1]
Destr CA (3) // [2]	Destr CB (3,4) // [2]
Destr CA (5) // [3]	Destr CA (3) // [2]
Destr CA (6) // [4]	Destr CA (5) // [3]
Destr CA (8) // [5]	Destr CB (6,7) // [4] Destr CA (6) // [4] Destr CB (8,9) // [5] Destr CA (8) // [5]

Din analiza rezultatelor observăm că:

- în lipsa polimorfismului toate obiectele se consideră a fi de tipul pointerului (CA), funcționarea fiind deci "defectuoasă". O posibilă soluție ar putea fi menținerea în masiv (tablou) și a unui indicator al tipului obiectului și apelul explicit al metodelor clasei deriveate când este cazul (folosind conversia explicită de tip pentru pointerul respectiv – "cast"). Considerăm soluția neelegantă și greoie, tratarea nefiind uniformă.
- în prezența polimorfismului se apelează metoda met() corespunzătoare instanței efective (CA sau CB). În acest fel s-a reușit o tratare uniformă a masivului (din punct de vedere al sursei programului). La baza soluției stă legarea dinamică a metodelor virtuale (deci polimorfismul).

Un caz deosebit este acela al destructorului virtual când, tot prin polimorfism (legare dinamică), se apelează destructorul corespunzător instanței respective, distrugerea obiectelor făcându-se corect. Remarcăm faptul că, după execuția destructorului clasei deriveate (legat dinamic), se apelează și destructorul clasei de bază. Aceasta dovedește că, în "interiorul" (la finalul) destructorului clasei deriveate apelul destructorului clasei de bază se face prin **legare statică**.

După cum am amintit și la prezentarea polimorfismului, soluția prezentată de tratare uniformă a masivelor eterogene este valabilă doar în cazul în care masivul reține pointeri către o clasă de bază iar "eterogenitatea" masivului apare în aceea că instanțele menținute (gestionate) sunt

Cap. 3 - Polimorfismul

instanțe ale clasei de bază sau ale unor clase deriveate (de oricără ori) din aceasta (pentru ca polimorfismul -- care a fost cheia soluției - să funcționeze). Evident, tratarea uniformă se referă doar la apelul unor metode existente în clasa de bază (fie ele și pure), eventual înlocuite în clasele deriveate.

Rezumat - de finut minte :-)

- supraîncărcarea metodelor în contextul moștenirii – nu e încă polimorfism
- legarea statică și legarea dinamică a metodelor
- cuvânt cheie **virtual** => legare dinamică => **polimorfism**
- polimorfism – doar în **contextul moștenirii**
- implementarea legării dinamice (polimorfismului) - **tabelele VMT**
- metode virtuale în icerarhii de clase
- **metode pure** (încă neimplementate) și **clase abstracte** (cu funcții pure, neinstantiabile) masive eterogene – **tratare uniformă** prin polimorfism

Cap.4 - Redefinirea operatorilor

4.1 Apel prin valoare, adresă (pointer) și referință pentru tipuri de bază

Pentru început, ne propunem să analizăm modul de transmitere a parametrilor spre funcții (incluzând în această discuție și metode) pentru cazul transmiterii de date având **tipuri predefinite**.

După cum se cunoaște din limbajul C parametrii se pot transmite prin valoare și prin adresă (pointer). Situația este descrisă în tabelul următor:

	apel prin valoare	apel prin adresă (pointer)	apel prin referință
sintaxă declarare parametru formal	functie(tip Nume,...)	functie(Tip *Nume,...)	functie(Tip &Nume,...)
sintaxă apel	functie(variabila,...)	functie(&variabila,...)	functie(variabila,...)
utilizare variabilă în cadrul funcției	Nume	*Nume	Nume
ce se transmite efectiv pe stivă	valoarea variabilei parametru	adresa variabilei parametru	adresa variabilei parametru
modificarea parametruului	NU	DA	DA

Remarcăm următoarele:

- apelul prin valoare este comod din punct de vedere al sintaxei (nu e nevoie de * și &) dar nu permite modificarea în cadrul funcției a variabilei trimise parametru (se modifică de fapt doar copia de pe stivă a acesteia).
- apelul prin adresă permite modificarea variabilei propriu-zise (având la dispoziție adresa ei) dar sintaxa este îngreunată.

Cap. 4 – Redefinirea Operatorilor

Pentru rezolvarea acestei “dileme” s-a introdus în C++ tipul referință care combină avantajele celor două situații anterioare. Definirea, apelul și utilizarea unui parametru referință sunt descrise în tabelul anterior.

În continuare prezentăm un exemplu de apel al unei funcții având 3 parametri întregi transmiși prin cele 3 metode descrise anterior.

```
#include <stdio.h>

int functie(int x,int *y,int &z)    // cele 3 posibilitati

{
int t;

    printf("\nAm apelat %d %d %d",x,*y,z);
    x++; (*y)++; z++;
    t=x+*y+z;
    printf("\nRevin      %d %d %d",x,*y,z);
    return t;
}

void main()
{
int a=2,b=5,c=3,r=11;

    printf("\nApelez      %d %d %d %d",a,b,c,r);
    r=functie(a,&b,c);
    printf("\nAm revenit %d %d %d %d",a,b,c,r);
}
```

Programul va tipări următoarele:

```
Apelez      2 5 3 11
Am apelat  2 5 3
Revin      3 6 9
Am revenit 2 6 9 18
```

Atragem încă o dată atenția asupra sintaxei declarării, apelului și utilizării variabilelor parametru și asupra valorii variabilelor locale funcției main după întoarcerea din funcție.

4.2 Funcții friend, metode friend și clase friend (prietene)

Necesitatea mecanismului de “friend” provine de la imposibilitatea unei metode de a fi membră în mai multe clase. S-a impus existența unei soluții ca o funcție ne-membră să poată accesa membrii ne-publici ai unei clase. Acest lucru se realizează prin prefixarea declarației cu

cuvântul cheie “**friend**”.

Chiar dacă sunt declarate în cadrul clasei, ele **nu sunt metode**, deci nu li se transmite pointerul ascuns **this**. Esențial este că ele **au acces la membrii ne-publici ai clasei cu care sunt prietene**. Evident, funcțiile friend **încalcă (controlat) principiul încapsulării**. Nefiind membre ale clasei, accesibilitatea lor nu este afectată de secțiunea public, protected sau private în care sunt declarate (fapt aparent surprinzător). Încalcarea controlată a încapsulării se referă la faptul că funcțiile friend trebuie declarate în cadrul clasei, deci de către proiectantul clasei. Clasa își stabilește funcțiile care îi sunt prietene și nu funcțiile clasele cu care să fie prietene, adică nu putem fi prieteni cu o clasă decât dacă proiectantul acesteia consimte acest lucru.

În exemplul următor prezentăm modul de declarare a unei funcții **MULTIPLY** de înmulțire a unui vector cu o matrice, funcție declarată ca și funcție friend:

```
class CMatrice;

class CVector
{
    int v[10];
    friend CVector Multiply(CMatrice &m,CVector& v);
    // . . . .
};

class CMatrice
{
    int m[10][10];
    friend CVector Multiply(CMatrice &m,CVector& v);
    // . . . .
};

CVector Multiply(CMatrice &m,CVector& v)
{
    CVector rez;

    for(int i=0;i<10;i++)
    {
        rez.v[i]=0;                                // membrul v privat
        for(int j=0;j<10;j++)
            rez.v[i]+=m.m[i][j]*v.v[j]; // membrii v si m privati
    }
    return rez;
}
```

În lipsa mecanismului funcțiilor friend, înmulțirea ar fi putut fi făcută doar prin:

- punerea variabilelor membru ale matricii și vectorului publice, ceea ce ar fi impus renunțarea completă la avantajele încapsulării (inacceptabil).
- dotarea claselor cu metode de acces la membri (citire și scriere a lor) fapt care ar fi compromis eleganța și performanțele de viteză (orice acces la o variabilă însemnând apeluri suplimentare de metode).

Cum ambele soluții prezintă dezavantaje majore mecanismul funcțiilor friend este justificat. Remarcăm și faptul că funcțiile friend au acces la membrii ne-publici ai tuturor obiectelor de tipul respectiv, nu doar la variabilele primite parametru (cum, eronat, s-ar putea trage concluzia din exemplul anterior). De cele mai multe ori însă, se folosesc parametri pentru a suplini lipsa lui this.

Nu numai funcțiile ci și metodele pot fi declarate **friend**, ca în cele ce urmează:

```
class CX
{
    // ...
    void o_metoda(/*....*/);
};

class CY
{
    // .... clasa CY propriu-zisa
    friend void CX::o_metoda(/*....*/);
};
```

În acest exemplu metoda clasei CX devine prietenă cu clasa CY. În plus, pentru a evita declararea individuală a fiecărei metode a unei clase ca fiind prietenă cu o altă clasă se poate declara în acest scop întreaga clasă ca fiind prietenă cu altă clasă ca în exemplul următor:

```
class CX
{
    // .... clasa CX propriu-zisa
};

class CY
{
    // .... clasa CY propriu-zisa
    friend class CX;
};
```

efectul fiind acela că toate metodele lui CX devin prietene cu CY. Relația de friend între clase nu este nici comutativă și nici tranzitivă (fără declararea explicită a tuturor relațiilor de friend).

4.3 Redefinirea operatorilor - introducere

Pentru **tipurile predefinite** semnificația operatorilor este binecunoscută, de exemplu este evident ce înseamnă + pentru întregi. O caracteristică importantă a programării obiectuale este aceea că permite programatorului să își definească propriile clase, în sapt **propriile tipuri de date**. În sensul comportării cât mai naturale a noilor tipuri introduse, devine evidentă utilitatea definirii și pentru aceste noi tipuri a unor operatori.

Termenul original este acela de “**Operators Overloading**”, adică **supraîncărcarea operatorilor**, pentru a indica dreptul programatorului de a atribui sensuri noi operatorilor existenți. Din această cauză se folosește de multe ori termenul de **redefinire a operatorilor** pentru a indica acest fapt.

În limbajele de programare nu există, de exemplu, un tip număr complex, cu toate că, în matematică, acest tip este bine definit. Putem declara o clasă complex pentru care să avem definită adunarea astfel:

```
class complex
{
    double real, imag;
    // ...
    complex Adunare(complex &arg);      // prima varianta
    complex operator+(complex &arg);     // a doua varianta
};
```

în care ambele metode adună separat partea reală și cea imaginară, cu posibilele utilizări:

```
complex a,b,c;

c=a.Adunare(b);      // foarte incomod
c=a.operator+(b);   // ceva mai clar, dar tot incomod
c=a+b;               // in sfirsit natural
```

În exemplul anterior remarcăm forma incomodă cu apel explicit de metodă (mai ales în cazul unor expresii complicate, cu adunări multiple) și ultima formă, în sfârșit naturală (ultimele două forme de scriere sunt echivalente). A doua formă indică faptul că avem de a face de fapt cu **funcții (metode) cu nume de operator** (“operator” este cuvânt cheie al limbajului).

Un exemplu mai complet este prezentat în continuare:

```
#include <stdio.h>

class complex
{
public:
    double real;
    double imag;
    complex(double re,double im){real=re;imag=im;}
    complex(){real=0;imag=0;};

    complex operator+(complex &arg)
    {
        complex rez;
        rez.real=real+arg.real; // rez.real=this->real+arg.real
        rez.imag=imag+arg.imag; // rez.imag=this->imag+arg.imag
        return rez;
    }
};

void main()
{
    complex c1(1,2),c2(5,7),c3;

    printf("\nc1 ( %lf , %lf )",c1.real,c1.imag);
    printf("\nc2 ( %lf , %lf )",c2.real,c2.imag);
    c3=c1+c2;
    printf("\nc3 ( %lf , %lf )",c3.real,c3.imag);
}
```

care produce următorul rezultat:

```
c1 ( 1.000000 , 2.000000 )
c2 ( 5.000000 , 7.000000 )
c3 ( 6.000000 , 9.000000 )
```

În mod evident adunarea este acum mult mai naturală astfel încât programul poate fi înțeles mult mai ușor și ne putem concentra asupra prelucrării propriu-zise. Pentru o implementare reală, utilă clasa complex trebuie completată cu o mulțime de alți operatori. Modul cum se întoarce efectiv rezultatul va fi discutat mai târziu.

4.4 Redefinirea operatorilor ca și metode sau ca și funcții friend

Deși a fost trecut anterior cu vederea, redefinirea operatorilor se poate face atât **prin metode** cât și **prin funcții friend**. Apelul funcțiilor și metodelor operator se face de către compilator conform următoarelor echivalări:

aritate	sintaxa	apel pentru operator implementat prin metodă	apel pentru operator implementat prin funcție friend
binar	$x \otimes y$	$x . \text{operator} \otimes (y)$	$\text{operator} \otimes (x, y)$
unar	$\oplus x$	$x . \text{operator} \oplus ()$	$\text{operator} \oplus (x)$

în care am notat generic prin \otimes orice **operator binar** (cu doi operanzi) și prin \oplus orice **operator unar** (cu un singur operand).

Prezentăm un exemplu în care au fost redefiniți 4 operatori acoperind toate situațiile metodă - funcție friend și unar - binar:

```
#include <stdio.h>
class CX
{
public:
    int x;
    void operator++(); // unar, metoda
    void operator+(int arg2); // binar, metoda
    friend void operator--(CX&arg1); // unar, friend
    friend void operator-(CX&arg1,int arg2); // binar, friend
};

void CX::operator++() {x++;} // this->x++
void CX::operator+(int arg2) {x+=arg2;} // this->x+=arg2
void operator--(CX&arg1) {arg1.x--;} // nu există
void operator-(CX&arg1,int arg2) {arg1.x-=arg2;} // this

void main()
{
CX a;
    a.x=5; printf("\na.x=%d",a.x); // aici = este de la intregi
    a++; printf("\na.x=%d",a.x);
    a+2; printf("\na.x=%d",a.x);
    a--; printf("\na.x=%d",a.x);
    a-3; printf("\na.x=%d",a.x);
}
```

care tipărește:

```
a.x=5 // la inceput
a.x=6 // dupa operatorul ++
a.x=8 // dupa operatorul +
a.x=7 // dupa operatorul --
a.x=4 // dupa operatorul -
```

Deoarece redefinirea operatorilor are de sapt la bază funcții (metode) cu nume de operator putem beneficia de **supraîncărcarea numelui funcțiilor**, adică putem defini **diferit același operator** pentru **argumente (operanzi)** de **tipuri diferite** (putem, pentru o clasă, defini adunarea într-un fel pentru al doilea operand întreg și în alt fel pentru al doilea operand de tip long). Această posibilitate este valabilă, evident, doar pentru **operatorii binari**, unde avem libertatea alegerii tipului celui de al doilea operand, primul operand este impus a fi de tipul clasei pentru care redefinim operatorul.

Cu toate că funcțiile operator îmbunătățesc versatilitatea programelor, trebuie respectate anumite **restrictii**:

- nu se pot defini operatori **noi** (de aceea se și numește **redefinirea operatorilor**).
- se pot redefini **toți** operatorii existenți în C++ cu **excepția** următorilor:

. C++ selectie directă a membrilor
. * C++ dereferentiere pointeri la metode
:: C++ "scope access" / rezolutie
?: Operatorul conditional

- nu se poate schimba aritatea și prioritatea (**precedența**) operatorilor. Operatorii unari trebuie să rămână unari, operatorii binari trebuie să rămână binari iar precedența operatorilor nu poate să schimbe, (adică, de exemplu, * rămâne priorității lui +).
- comportamentul operatorilor pentru **tipurile predefinite nu poate fi schimbat**, adică primul operand operatorului redefinit trebuie să fie o clasă (cea în discuție).
- redefinirea unor operatori nu redifinește și operatorii aparent combinați, adică redefinirea lui + și a lui = nu redifinește și +=. Dacă îl dorim și pe cel combinat, trebuie să îl redefinim explicit.
- operatorii =, [] și -> trebuie să fie funcții membru ne-statice.

Scopul acestor restricții este acela de a asigura că:

- compilatorul de C++ nu trebuie să își modifice regulile sintactice.
- semnificația operatorilor predefiniți nu poate fi schimbată.

4.5 Distingerea între formele prefixate și postfixate ale operatorilor

O problemă apărută în primele versiuni de C++ era imposibilitatea distingerea între formele prefixate și postfixate ale operatorilor unari. Problema s-a rezolvat în sensul că, prin convenție:

- varianta prefixată se redifinește "normal" (majoritatea operatorilor unari au doar forma prefixată).

- varianta postfixată se redefineste având suplimentar un parametru de tip întreg.

Parametrul suplimentar din forma postfixată nu este efectiv folosit, scopul său este doar de a asigura o **diferență de semnătură**. Prezentăm un exemplu în care operatorii de preincrementare și postincrementare se redefinesc ca și metode. Dacă s-ar defini ca și funcții friend ar exista și un prim parametru referință la un obiect al clasei respective (din acest motiv s-a folosit exprimarea de “suplimentar un parametru” pentru forma postfixată). În exemplul prezentat operatorul întoarce valoarea dintr-un tablou și incrementează indicele în acesta înainte sau după acces (pre/post incrementare).

```
#include <stdio.h>

class CX
{
    int tab[10];
public:
    int poz;
    int operator++(int); // varianta postfixata
    int operator++(); // varianta prefixata
    CX();
};

CX::CX()
{
    for(int i=0;i<10;i++)
        tab[i]=10*i;
    poz=3; // initial in pozitia 3
}

int CX::operator++() // varianta prefixata
{
    poz++;poz%=10; // intii incrementam
    return(tab[poz]); // apoi luam valoarea
}

int CX::operator++(int) // varianta postfixata
{
    int t=poz; // pastram pozitia veche
    poz++;poz%=10; // incrementam
    return(tab[t]); // luam valoarea din pozitia veche
}

void main()
{
CX a;
int pold,v,pnew;
```

```
pold=a.poz; v=a++; pnew=a.poz;
printf("\npoz_old=%d value=%d poz_new=%d",pold,v,pnew);

pold=a.poz; v+=a; pnew=a.poz;
printf("\npoz_old=%d value=%d poz_new=%d",pold,v,pnew);

printf("\npoz_NEW=%d value=%d poz_OLD=%d",a.poz,a++,a.poz);
printf("\npoz_NEW=%d value=%d poz_OLD=%d",a.poz,++a,a.poz);
}
```

Din rezultatele pe care le tipărește programul:

```
poz_old=3 value=30 poz_new=4
poz_old=4 value=50 poz_new=5
poz_NEW=6 value=50 poz_OLD=5
poz_NEW=7 value=70 poz_OLD=6
```

remarcăm:

- funcționarea corectă a celor două forme pre și post fixate.
- analizând ultimele două linii tipărite remarcăm și ordinea în care se evaluatează argumentele în cazul unui apel de funcție: dinspre ultimul înspre primul, adică în ordinea punerii lor pe stivă (conform convenției de apel C). Fiind un aspect delicat, nu se recomandă să ne bazăm pe el, și este preferabilă soluția cu apel anterior și variabile temporare.

4.6 Operatorul de atribuire (=)

Un operator de o importantă deosebită este operatorul = (de atribuire). El trebuie să primească parametru o referință la un obiect de același tip pe care îl copiază în obiectul curent. Prezintă următoarele particularități:

- este folosit în cazul atribuirii de obiecte de acel tip (inclusiv a obiectelor întoarse de către funcții).
- dacă nu este definit de către utilizator **compilatorul generează implicit unul, care copiază membru cu membru** ("bit cu bit") conținutul obiectelor.

Din **copierea membru cu membru implicită** provine și principala sa **problemă**. Să considerăm o clasă care are ca și membri un întreg "x" și un pointer "sir" spre un sir de caractere alocat dinamic și următorul scenariu (prezentat în Figura 4.1):

- la început avem două obiecte a și b, ca în prima parte a figurii. Situația este în regulă.
- după atribuirea a = b prin copierea adresei membrului sir (la copierea membru cu membru)

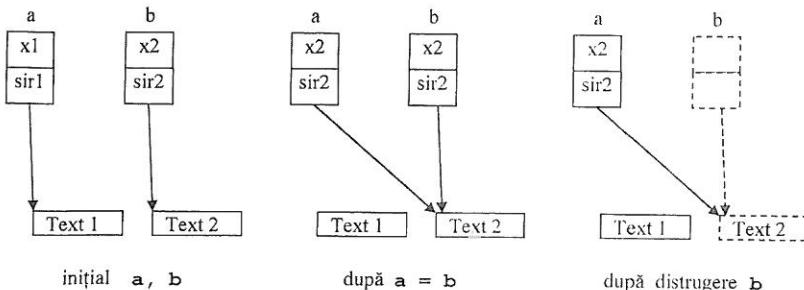


Figura 4.1 Descrierea scenariului "problemă"

unul din şiruri devine "pierdut" în memorie (nu mai avem adresa sa pentru a îl elibera) iar ambele obiecte indică spre acelaşi şir (o schimbare în unul afectând și celălalt obiect).

- după distrugerea unuia din obiecte (în figură a obiectului b) obiectul rămas conține un pointer "eronat", care indică spre o zonă de memorie deja eliberată (cu consecințe dintre cele mai grave, de exemplu blocări).

Problema apare deci datorită copierii membru cu membru a unor membrii a căror semnificație nu este doar valoarea lor propriu zisă, ci ea reprezintă "identificatorul" unei resurse (de exemplu zonă de memorie alocată, fișier deschis, etc.). Pentru a nu pierde / dupica asemenea resurse se impune redefinirea operatorului = în cazul în care obiectul are în componență asemenea membri. Operatorul = astfel redefinit trebuie să elibereze resursa veche și să duplice resursa nouă (de obicei prin alocarea uneia noi și copierea conținutul acesteia). Un exemplu în care resursa "vulnerabilă" este adresa unui şir de caractere (caz descris anterior) este prezentat în continuare:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

class CX
{
public:
    int x;
    char *Sir;
    CX(int x0,char *Sir0);      // constructor
    ~CX();                      // destructor
    CX(CX&v);                  // constructor copiere
    void operator=(CX&v);     // operatorul de atribuire
    void Copiaza(CX&v);       // va face copierea "cu grija"
};


```

```
CX::CX(int x0,char *Sir0)      // constructor
{
    printf("\nConstructor general %d,%s",x0,Sir0);
    x=x0;
    Sir=(char*)malloc(strlen(Sir0)+1); // alocare pentru text
    strcpy(Sir,Sir0);                // copiere sir text
}

CX::~CX()                      // destructor
{
    printf("\nDestructor %d,%s",x,Sir);
    free(Sir);                     // eliberare sir text
}

CX::CX(CX&v)                  // constructor copiere
{
    printf("\nConstructor copiere %d,%s",v.x,v.Sir);
    Copiaza(v);                  // copiere "cu grija"
}

void CX::operator=(CX&v)        // operator = redefinit
{
    printf("\nOperator= %d,%s",v.x,v.Sir);
    free(Sir);                   // eliberare sir vechi
    Copiaza(v);                 // copiere "cu grija"
}

void CX::Copiaza(CX&v)          // copierea "cu grija"
{
    x=v.x;                       // copiere simpla
    Sir=(char*)malloc(strlen(v.Sir)+1); // alocare sir
    strcpy(Sir,v.Sir);           // copiere sir
}

void main()
{
    CX a(1,"sir_1"),b(2,"sir_2"),c(3,"sir_3");
    CX d=a,*e;                   // constr. copiere pentru d

    b=c;                         // operator atribuire
    e=new CX(4,"sir_4_Dinamic"); // NU ESTE OPER. REDEFINIT
                                  // SE ATRIBUIE POINTERI
    c=*e;                         // operator atribuire
    delete e;
    printf("\nb.x=%d b.Sir=%s",b.x,b.Sir);
    printf("\nc.x=%d c.Sir=%s",c.x,c.Sir);
}
```

Rezultatul rulării acestui program se prezintă în continuare:

```
Constructor general 1,sir_1          // pentru a
Constructor general 2,sir_2          // pentru b
Constructor general 3,sir_3          // pentru c
Constructor copiere 1,sir_1          // pentru d=a
Operator= 3,sir_3                  // pentru b=c
Constructor general 4,sir_4_Dinamic // pentru e
Operator= 4,sir_4_Dinamic          // pentru c=*e
Destructor 4,sir_4_Dinamic          // pentru e
b.x=3 b.Sir=sir_3
c.x=4 c.Sir=sir_4_Dinamic
Destructor 1,sir_1                  // pentru d    !!!! 
Destructor 4,sir_4_Dinamic          // pentru c    !!!! 
Destructor 3,sir_3                  // pentru b    !!!! 
Destructor 1,sir_1                  // pentru a
```

Remarcăm:

- alocarea pe constructor și eliberarea pe destructor a memoriei pentru “resursele” sir.
- copierea “cu grijă” se face alocând în prealabil o nouă “resursă” și abia apoi copiind conținutul acesteia. Evident, ne rămâne și sarcina copierii celorlalți membri (“normali”), deoarece operatorul de copiere membru cu membru nu se mai generează.
- probleme asemănătoare (**datorate copierii membru cu membru**) se pun și pentru **constructorul de copiere** soluția fiind identică (doar că “resursa” veche nu este încă alocată deci nu trebuie întâi eliberată – fiind constructor abia acum se aloca prima dată).
- diferența între **constructorul de copiere** (utilizat pentru variabila d, când = egalul apare chiar la declararea variabilei destinație) și **operatorul de atribuire** (când = apare ulterior declarării variabilei destinație).

4.7 Operatorii new și delete

Alți doi operatori ce merită remarcări sunt operatorii new și delete. Rolul acestora este acela ca proiectantul clasei să poată **decide locul în care se face alocarea de memorie pentru obiectele dinamice** ale acelei clase. De cele mai multe ori se alocă o zonă de memorie mai mare care se se gestionează de către acea clasă “scurtcircuitând” mecanismul clasic de alocare (câștigul fiind în principal de viteză). Operatorii trebuie să fie **metode** (nu funcții friend) **static**e (deci nu pot fi virtuali) și trebuie să aibă următorii parametrii:

```
void* operator new(size_t nr);
void operator delete(void* adr, size_t nr);
```

în care:

- nr este dimensiunea memoriei solicitate / eliberate (este stabilită de către compilator). Prezența sa, aparent inutilă deoarece putem cunoaște dimensiunea clasei (folosind operatorul sizeof), este necesară în cazul obiectelor derivate din aceasta, obiecte care pot avea altă dimensiune (evident, doar mai mare).
- adr este adresa de memorie care se eliberează (adresa obiectului ce se distrugе).

Operatorul new întoarce adresa zonei unde s-a alocat memorie pentru această instanță a clasei (unde se va construi obiectul).

Operatorul new se apelează înaintea apelului constructorului instanței respective iar operatorul delete după apelul destructorului. Dacă **operatorul new** întoarce **NULL** (nu s-a putut aloca memorie) nu se mai apelează **constructorul** (care ar face în acel caz accese de memorie eronate).

La o alocare dinamică de obiect lucrurile se petrec deci astfel:

- se apelează operatorul new (redefinit) care întoarce o adresă (să o notăm AdrOb).
- dacă AdrOb este diferită de NULL
 - this = AdrOb
 - apel constructor corespunzător (cu this parametru ascuns)
- se întoarce adresa AdrOb.

În exemplul simplu prezentat în continuare clasa X dispune de un buffer static (în care se pot aloca doar două obiecte dinamice) și de două variabile care indică starea de alocare a fiecărei jumătăți a bufferului.

```
#include <stdio.h>
class X
{
    int x,y;      // elemente utile
public:
    X::X()          {printf("\nConstr. implicit");x=0;y=0;}
    X(int x0,int y0) {printf("\nConstr. %d %d",x0,y0);x=x0;y=y0;}
    ~X()           {printf("\nDestr. %d %d",x,y);}
    void* operator new(size_t nr);
    void operator delete(void* adr, size_t nr);
private:
    static char Buffer[];
    static int liber1,liber2;
};

char X::Buffer[2*sizeof(X)];      // memorie pentru 2 alocari
```

```

int X::liber1=1,X::liber2=1;      // initial libere

void* X::operator new(size_t nr)
{
    printf("\nOperator new %d",nr);
    if(X::liber1) return(X::liber1=0,Buffer); // poz1
    if(X::liber2) return(X::liber2=0,Buffer+sizeof(X)); // poz2
    return NULL;                                // nu s-a putut aloca
}

void X::operator delete(void* adr,size_t nr)
{
    printf("\nOperator delete %d",nr);
    if(adr==Buffer+sizeof(X)) X::liber2=1; // poz2 eliberata
    if(adr==Buffer)           X::liber1=1; // poz1 eliberata
}

void main()
{
X a(0,1),*b,*c,*d; // a este obiect STATIC

    b = new X(2,3); // alocare in pozitia 1
    c = new X();     // alocare in pozitia 2
    d = new X(5,6); // alocarea va esua
    delete b;        // eliberare pozitia 1
    d = new X(7,8); // alocare in pozitia 1
}

```

Corespunzător, programul va tipări:

```

Constr. 0 1          // pentru a
Operator new 4        // pentru b=new X(2,3)
Constr. 2 3          //      ---- ' '
Operator new 4        // pentru c=new X()
Constr. implicit      //      ---- ' '
Operator new 4        // pentru d=new X(5,6), fara constructor
Destr. 2 3           // pentru delete b
Operator delete 4     //      ---- ' '
Operator new 4        // pentru d=new X(7,8)
Constr. 7 8           //      ---- ' '
Destr. 0 1           // pentru a

```

Pentru înțelegerea acestor rezultate trebuie precizat că programul a fost rulat pe un compilator pentru care dimensiunea reprezentării unui întreg este 2 octeți (deci 4 reprezintă dimensiunea obiectelor de tip X - adică 2 întregi).

Atragem atenția asupra faptului că, pentru alocare dinamică de tablouri, trebuie redefiniți

operatori specifici:

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

diferit față de cei anteriori (pentru elemente individuale), la care semnificația parametrilor este aceeași.

4.8 Alte exemple de redefinire a operatorilor

Primul este “haziu”, redefinind operatorul virgulă, cel mai simplu (și cel mai neglijat) operator:

```
#include <stdio.h>
class CX
{
public:
    int operator,(int x)
        {printf("\nOP virgula param=%d",x);return(x+3);}
}t;

void main()
{
    printf("\nRezultatul=%d", (t,2));
}
```

secvență care va tipări:

```
OP virgula param=2
Rezultatul=5
```

Putem remarcă încă o dată distincția între **operatorul virgulă** și **separitorul virgulă** utilizat pentru separarea argumentelor unei funcții (situație în care nu este operator).

Un alt exemplu interesant este următorul, în care se redifineste operatorul de indexare [] pentru indexarea într-un tablou folosind “indici” de natură diferită:

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

class Ctst
{
public:
    char* Nume[4]; // aici se cauta
```

```

int Matricol[4];           // ---- ' ----
int operator[](char *nume); // prima varianta operator
int operator[](int matricol); // a doua varianta operator
}tstObj;

int Ctst::operator[](char *nume) // indexare dupa nume
{
    for(int x=0;x<3;x++)
        if(strcmp(Nume[x],nume)==0) // e cel cautat ?
            return x;             // gasit aici
    return -1;                  // negasit
}

int Ctst::operator[](int matricol) // indexare dupa matricol
{
    for(int x=0;x<3;x++)
        if(Matricol[x]==matricol) // e cel cautat ?
            return x;             // gasit aici
    return -1;                  // negasit
}

void main()
{
    tstObj.Nume[0]="unu";    tstObj.Matricol[0]=1; // initializari
    tstObj.Nume[1]="doi";    tstObj.Matricol[1]=5;
    tstObj.Nume[2]="trei";   tstObj.Matricol[2]=2;
    tstObj.Nume[3]="patru";  tstObj.Matricol[3]=9;

    printf("\nRezultate cautare\n %2d \n %2d \n %2d \n %2d",
          tstObj["unu"],tstObj[2],tstObj["trei"],tstObj[7]);
}

```

Operatorii [] redefiniți în exemplul anterior întorc indexul în tablou al elementului căutat sau -1 dacă nu este găsit, programul tipărind:

```

Rezultate cautare
0      // gasit in pozitia 0
2      // gasit in pozitia 2
-1     // negasit
-1     // negasit

```

Probleme deosebite pune redefinirea operatorului -> datorită unarității lui și modului de interpretare. Astfel, compilatorul interpretează expresia **a->m** ca **(a.operator->())->m**. Din acest motiv operatorul trebuie să întoarcă o entitate asupra căreia se poate aplica apoi din nou operatorul -> deci trebuie să întoarcă fie un pointer (când următorul -> este cel **standard**) fie un **obiect** (când următorul -> este un **operator redefinit** pentru acel obiect).

Un exemplu simplu de aplicare este prezentat în continuare:

```
#include <stdio.h>
class CElem
{
public:
    int      x;
    static int  comuta;
    CElem   *frate;           // adresa obiectului "frate"
    CElem* operator->()
    {return (comuta)?frate:this;} // fratele sau el
};

int CElem::comuta=0;      // fanionul pentru comutare

void main()
{
CElem a,b,*pa,*pb;

a.x=1; a.frate=&b; pa=&a;
b.x=3; b.frate=&a; pb=&b;

printf("\ncomuta=%d a.x=%d a->x=%d pa->x=%d b.x=%d b->x=%d pb->x=%d",
       a.comuta, a.x, a->x, pa->x, b.x, b->x, pb->x );

a.comuta=1;
printf("\ncomuta=%d a.x=%d a->x=%d pa->x=%d b.x=%d b->x=%d pb->x=%d",
       a.comuta, a.x, a->x, pa->x, b.x, b->x, pb->x );
}
```

După cum se poate remarcă din codul anterior, am redefinit operatorul `->` pentru comutarea între cele două obiecte definite, în funcție de o variabilă statică. În esență, se permite astfel existența unui nivel de **indirectare**, aplicarea lui `->` putând face referire indirect la alt obiect.

Funcționarea operatorului redefinit este demonstrată de rezultatul rulării programului și anume:

```
comuta=0 a.x=1 a->x=1 pa->x=1 b.x=3 b->x=3 pb->x=3
comuta=1 a.x=1 a->x=3 pa->x=1 b.x=3 b->x=1 pb->x=3
```

Se remarcă faptul că operatorul `->` redefinit se referă doar la cazul **aplicării lui asupra obiectelor** de acel tip **nu și aplicării lui asupra pointerilor** de acel tip.

Redefinirea operatorilor << și >> stă la baza bibliotecii de intrare / ieșire a limbajului C++ (biblioteca de lucru pe stream-uri) care va fi tratată separat.

4.9 Apel prin valoare, adresă (pointer) și referință pentru tipuri definite de utilizator

Revenim asupra exemplului prezentat la început, dar în acest caz considerând tipul implicat (ca și parametru și valoare întoarse) unul definit de utilizator (o clasă). După cum vom constata este util pentru înțelegere să avem definit constructorul de copiere și redefinit operatorul = (de copiere).

```
#include <stdio.h>
#include <conio.h>

class CX
{
public:
    int a;

    CX()          {printf("\nConstructor implicit");      a=0;    }
    CX(CX&m)    {printf("\nConstructor copiere %d",m.a);a=m.a;}
    CX(int a0)   {printf("\nConstructor %d",a0);           a=a0;   }
    ~CX()        {printf("\nDestructor %d",a);             a=0;    }
    void operator=(CX &d)
        {printf("\nOperator= %d->%d",d.a,a); a=d.a;}
};

CX functie(CX x,CX *y,CX &z)          // cele 3 posibilitati
{
    CX t;

    printf("\nAm apelat %d %d %d",x.a,y->a,z.a);
    x.a++; y->a++; z.a++;
    t.a = x.a+y->a+z.a;
    printf("\nRevin      %d %d %d",x.a,y->a,z.a);
    return t;
}

void main()
{
    CX a(2),b(5),c(8),r(11);

    printf("\nApelez      %d %d %d %d",a.a,b.a,c.a,r.a);
    r=functie(a,&b,c);
    printf("\nAm revenit  %d %d %d %d",a.a,b.a,c.a,r.a);
}
```

La rulare, programul tipărește următoarele:

```

Constructor 2           // pentru a
Constructor 5           // pentru b
Constructor 8           // pentru c
Constructor 11          // pentru r
Apelez    2 5 8 11
Constructor copiere 2   // pentru x (parametru)
Constructor implicit    // pentru t
Am apelat  2 5 8
Revin      3 6 9
Constructor copiere 18  // pentru obiectul intors (temporar)
Destructor  18          // pentru t
Destructor  3           // pentru x (parametru)
Operator= 18=>11       // copiere obiect intors in r
Destructor  18          // pentru obiectul intors (temporar)
Am revenit  2 6 9 18
Destructor  18          // pentru r
Destructor  9           // pentru c
Destructor  6           // pentru b
Destructor  2           // pentru a

```

Interpretând acestea remarcăm (în ordinea “desfășurării evenimentelor”):

- se instanțiază obiectele locale funcției main (normal).
- se execută corpul funcției main și se apelează funcția (cu obiectele având valorile așteptate).
- se instanțiază un obiect temporar folosind **constructorul de copiere** (o copie temporară a parametrului transmis prin valoare).
- se instanțiază variabila locală (cu constructorul corespunzător declarării).
- se începe execuția corpului funcției, parametrii având valorile așteptate.
- se efectuează acțiunea efectivă a funcției - se incrementează toate variabilele membru ale obiectelor implicate și se determină rezultatul.
- se părăsește corpul funcției, parametrii având valorile actualizate.
- se instanțiază, folosind **constructorul de copiere**, un obiect temporar în care se întoarce rezultatul (corespunzător întoarcerii prin valoare).
- se distrugе variabila locală (rezultatul).
- se **distrugе obiectul temporar** corespunzător parametrului **transmis prin valoare**.
- se copiază, folosind **operatorul de atribuire**, obiectul creat temporar pentru întoarcerea rezultatului în obiectul destinație.
- se **distrugе variabila temporară** în care s-a întors **rezultatul**.
- se revine în main, obiectul parametru **transmis prin valoare** fiind însă **nemodificat**.
- se distrug obiectele locale funcției main (normal).

Dacă însă funcția întoarce rezultat prin pointer sau prin referință, este evident că constructorul

de copiere și operatorul egal nu se mai folosesc (în momentul întoarcerii din funcție). În acele cazuri trebuie însă avut grijă ca adresa întoarsă să nu refere un obiect care va fi distrus la părăsirea funcției.

Analiza atentă a remarcilor de mai sus, a codului și a rezultatelor tipărite permit înțelegerea modului cum se face apelul prin valoare, pointer și referință. Revăzând exemplul prezentat chiar la începutul capitolului (pentru tipurile predefinite) putem înțelege într-o lumină nouă (mai generală) acel exemplu. Remarcăm încă o dată **comportarea uniformă** a tipurilor predefinite și a celor definite de utilizator și **importanța deosebită a constructorului de copiere și a operatorului de atribuire** în cazul apelului funcțiilor și întoarcerii rezultatului prin valoare.

Rezumat - de ținut minte :-)

- apel prin **valoare**, **adresa** și **referință** pt. tipuri de bază - particularități
- **friend** – funcții, metode și clase – au acces la membrii ne-publici
- **redefinirea operatorilor = metode** (sau funcții friend) cu **nume de operator**
- **restrictii** la redefinire operatori
- operatorul de **atribuire** (și constructorul de copiere) – probleme specifice
- operatorii **new** și **delete**, virgulă, [], ->
- apel prin **valoare**, **adresa** și **referință** pt. tipuri definite de utilizator, **rolul operatorului de atribuire și constructorului de copiere** în acest caz

Cap. 5 - Stream-uri

5.1 Introducere

După cum se cunoaște, **limbajul C** în sine nu includea nici o instrucțiune pentru operațiile de intrare-iesire. Toate funcțiile aferente acestor operații sunt disponibile sub forma unei biblioteci, deci nu sunt parte propriu-zisă a limbajului. Cu toate acestea și biblioteca de intrare-iesire este standardizată, fiind disponibilă în forma respectivă pe toate implementările existente (fapt care duce la ideea că ar fi parte a limbajului, când de fapt avem de a face doar cu funcții de bibliotecă). Pentru utilizarea funcțiilor de intrare-iesire trebuie inclus antetul `<stdio.h>` (STDandard Input Output).

Limbajul C++ urmează aceiași cale și nu include operațiile de intrare-iesire în limbaj ci într-o bibliotecă creată chiar cu ajutorul limbajului. Deci, în cele ce urmează, ne îndepărtem de la prezentarea principiilor programării obiectuale pentru a lua în discuție o anumită bibliotecă. Acest fapt este justificat de:

- necesitatea și utilitatea acestei biblioteci, cu facilități inexistente în C-ul standard.
- biblioteca în sine reprezintă un exemplu foarte bun de implementare eficientă și elegantă a unei ierarhii de clase.

Modelul fundamental pentru construcția bibliotecii de I/O este cel de “**stream**”, văzut ca și flux de date de la o sursă spre o destinație. Chiar dacă termenul a apărut încă din C-ul standard consacrarea sa a venit odată cu biblioteca C++.

Prezentăm în continuare un exemplu simplu de tipărire bazat atât pe biblioteca standard C cât și pe cea C++:

```
#include <stdio.h>      // pentru funcția printf
#include <iostream.h>     // pentru obiectul cout

char    c='c';
int     i=123;
long    l=12345678;
float   f=12.3456;
double  d=1234.456789;
```

```
main()
{
    printf("%c %d %ld %f %lf\n", c, i, l, f, d);
    cout<<c<<" "<<i<<" "<<l<<" "<<f<<" "<<d;
}
```

care va produce următorul rezultat:

```
c 123 12345678 12.345600 1234.456789
c 123 12345678 12.3456 1234.456789
```

Din acest exemplu simplu constatăm:

- biblioteca standard C este în continuare disponibilă.
- pentru utilizarea bibliotecii C++ trebuie inclus <iostream.h>.
- există un obiect cout instantiat implicit.
- ieșirea în C++ se bazează pe operatorul redefinit << (intrarea se va baza pe >>).
- funcționarea este aproape identică (la tipul float apare o diferență în ceea ce privește numărul de zecimale folosite implicit).
- soluția C++ este mai simplă, scăpând de necesitatea cunoașterii detaliilor descriptorilor de format (neplăcută în C).

La baza bibliotecii de streamuri stă **redefinirea operatorilor << și >>** care, în acest context, se numesc operatori de înserție (<<) și respectiv de extractie (>>). Această denumire sugerează, împreună cu simbolul, semnificația operatorului: variabila respectivă se inserează în respectiv se extrage din streamul respectiv.

Pentru a evita dificultățile apărute la apelul unor funcții de ieșire s-a impus folosirea unor operatori. De ce s-au ales << și >> ? Întâi, trebuia ținut cont că nu pot fi adăugați operatori noi. Apoi, se doreau operatori asemănători și totuși diferiți pentru intrare și ieșire. Semnificația de "mai mic" respectiv "mai mare" fiind profund înrădăcinată pentru operatorii < și > s-a preferat **utilizarea operatorilor << și >>**, mai puțin folosiți în mod curent (pentru tipurile standard). Aceștia sugerează **sensul transferului de date** (înspite stream respectiv de la stream), au asociativitate bună (de la stânga la dreapta) și prioritate destul de redusă încât să accepte operanzi expresii aritmetice uzuale, de exemplu:

```
cout << "x + y * z = " << x + y * z;
```

În situațiile în care operatorii implicați în expresii au prioritate mai mică decât << și >> se impune utilizarea parantezelor, de exemplu:

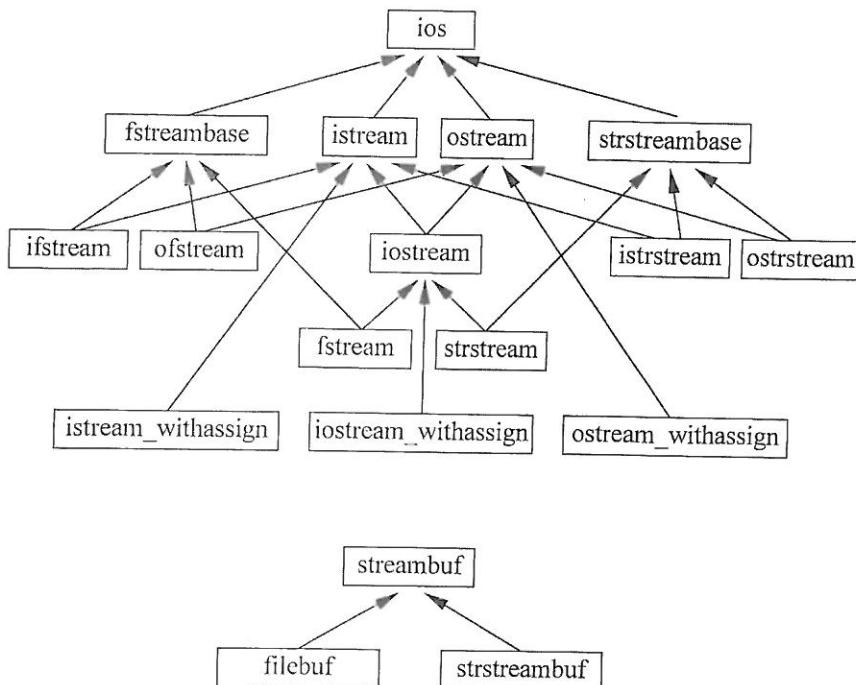


Figura 5.1 Cele două ierarhii de clase

```
cout << "x || y && z" = " << (x || y && z);
```

Deși nu vom putea prezenta o descriere completă a celor două ierarhii de clase ale bibliotecii I/O din motive de spațiu (oricum, nici nu ne propunem să ceva) prezentăm în Figura 5.1 aceste ierarhii, cu notația consacrată în UML în care sensul săgeților indică spre clasa părinte.

Cu toate că asupra unora vom mai reveni prezentăm în Tabelul 5.1 o scurtă descriere a acestor clase (atenție la numele lor, care este destul de sugestiv).

În acest moment putem aborda întrebarea: ce sunt **cin**, **cout**, **cerr** și **clog**? Răspunsul îl găsim în următorul cod existent în **<iostream.h>**:

```
extern istream_withassign cin;
extern ostream_withassign cout;
extern ostream_withassign cerr;
extern ostream_withassign clog;
```

Programare Orientată pe Obiecte - Principii

<code>ios</code>	<u>input-output stream</u>	clasa de bază a ierarhiei
<code>istream</code>	input stream	derivată <u>virtual</u> public din <code>ios</code> responsabilă cu operațiile de intrare
<code>ostream</code>	output stream	derivată <u>virtual</u> public din <code>ios</code> responsabilă cu operațiile de ieșire
<code>iostream</code>	<u>input-output stream</u>	derivată <u>multiplu</u> din <code>istream</code> și <code>ostream</code>
<code>fstreambase</code>	file stream base	
<code>ifstream</code>	<u>input file stream</u>	pentru stream-uri implementate pe fișiere
<code>ofstream</code>	<u>output file stream</u>	
<code>fstream</code>	file stream	
<code>strstreambase</code>	string stream base	
<code>istrstream</code>	<u>input string stream</u>	pentru stream-uri implementate pe siruri
<code>ostrstream</code>	<u>output string stream</u>	
<code>strstream</code>	string stream	
<code>istream_withassign</code>	input stream with assign	
<code>ostream_withassign</code>	output stream with assign	dar cu posibilitatea de a atribui (modifica) <code>streambuf</code> -ul asociat
<code>iostream_withassign</code>	input-output stream with assign	
<code>streambuf</code>	stream buffer	gestionare buffer asociat cu <code>stream</code> -ul - general și specializat pentru fișiere și siruri
<code>filebuf</code>	file buffer	
<code>strstreambuf</code>	string stream buffer	

Tabelul 5.1 Scurtă descriere a claselor

Deci `cin`, `cout`, `cerr` și `clog` sunt **obiecte**, instanțieri implicite ale claselor `istream_withassign` respectiv `ostream_withassign`. Pentru înțelegerea comportamentului acestora vom analiza clasele lor de bază din cadrul ierarhiei. Oricum, în problema bibliotecii de stream-uri, avem întotdeauna posibilitatea, foarte recomandabilă, de a analiza **fișierele antet** (`<iostream.h>` și altele asociate). În cele ce urmează vom prezenta asemenea secvențe cu mici modificări (doar simplificări, pentru a renunța la detalii care nu prezintă semnificație, dar ar îngreuna lizibilitatea, cum ar fi declarații Cdecl, FAR etc și traduceri ale unor comentarii).

5.2 Starea unui stream

Pentru a explica acest concept prezentăm o secvență de cod existentă în `<iostream.h>`:

```
class ios
{
// . . . . .
```

```

public:
    // biti de stare a stream-ului
    enum io_state {
        goodbit = 0x00, // nici un bit setat, totul e ok
        eofbit = 0x01, // la sfîrșit de fișier
        failbit = 0x02, // ultima operație I/O a esuat
        badbit = 0x04, // s-a încercat o operatie invalida
        hardfail = 0x80 // eroare nerecuperabila
    };

    // aflarea/setarea stării curente a stream-ului
    int rdstate(); // întoarce starea stream-ului
    int eof(); // non-zero la sfârșit de fișier
    int fail(); // non-zero dacă o operație a eşuat
    int bad(); // non-zero dacă a apărut o eroare
    int good(); // non-zero dacă nici un bit setat
    void clear(int = 0); // seteaza starea stream

protected:
    int state; // aici se memorează starea (pe biti)
    void setstate(int); // seteaza toți bitii de stare
// . . . .
};


```

Considerăm că explicațiile anterioare sunt suficiente pentru a înțelege metodele disponibile pentru operarea cu starea stream-ului. Mai facem doar următoarele observații:

- metoda clear nu poate opera asupra flagului hardfail iar diferența între fail() și bad() constă în accea că bad() testează biții "badbit" și "hardfail" iar fail() testează în plus și bitul "failbit".
- După cum era de așteptat, variabila state este declarată protected, pentru a fi accesibilă claselor derivate dar inaccesibilă pentru acces direct.
- Remarcăm faptul că membrii și metodele destinate gestionării stării unui stream se află în clasa ios deci, prin moștenire, în toată ierarhia de clase.

5.3 Operații de intrare-ieșire fără format

Pentru aceste operații clasele istream respectiv ostream pun la dispoziție un set de metode publice de citire respectiv scriere descrise în continuare (conform <iostream.h>):

```

class istream : virtual public ios
{
// . . . .
public:
    // extrage caractere într-un tablou
    istream& get( signed char *, int, char = '\n');

```

```

istream& get(unsigned char *, int, char = '\n');
istream& read( signed char *, int);
istream& read(unsigned char *, int);
    // extrage caractere într-un tablou până la terminator
istream& getline( signed char *, int, char = '\n');
istream& getline(unsigned char *, int, char = '\n');
    // extragere un singur caracter
istream& get(unsigned char &);
istream& get( signed char &);
int      get(); 

int      peek(); // următorul caracter fără extragerea sa
istream& putback(char); // pune înapoi caracterul

    // extrage și neglijăază, cu oprire la delimitator
istream& ignore(int = 1, int = EOF);
// . . . . .
};

class ostream : virtual public ios
{
// . . . . .
public:
    ostream& put(char);           // inseră caracterul
    ostream& write(const signed char *, int); // inseră tabl
    ostream& write(const unsigned char *, int); // - ' ' -
// . . . . .
};

```

După cum se constată transferul datelor se face fără a ține cont de semnificația acestora, metodele disponibile fiind echivalente cu cele existente în C (getc, putc, read, write), numite acolo de nivel coborât. Aceste operații sunt specifice situațiilor în care datele sunt binare (și, de cele mai multe ori, fluxul implicat este un fișier).

În legătură cu metodele prezentate anterior mai facem următoarele remarcări:

- get și getline sunt funcții cu un parametru luând valoare implicită.
- diferența între get și getline având 3 parametri constă în aceea că, deși ambele se opresc la citirea numărului dorit de caracter (parametrul_2 minus 1) sau la atingerea terminatorului (parametrul_3), get nu extrage terminatorul din istream pe când getline îl extrage. Astfel, get va continua (la un apel ulterior) cu citirea terminatorului iar getline după acesta (tipic, cu linia următoare, dacă terminatorul are valoarea implicită).
- metodele get de extragere a unui singur caracter extrag primul caracter din istream indiferent care este acesta, chiar dacă este *caracter de tip spațiu*. Amintim că, conform funcției isspace, acestea sunt: *horizontal tab* ('\t', 9), *newline* ('\n', 10), *vertical tab*

(‘w’,11), *formfeed* (‘f’,12), *carriage return* (‘r’,13), *space* (‘ ‘,32).

- ca și în C, operațiile de intrare-iesire fără format se folosesc mai ales când avem de a face cu fișiere.

5.4 Operații de intrare-iesire cu format

După cum rezultă și din primul exemplu prezentat, pentru operațiile de intrare-iesire cu format se folosesc operatorii de **extracție** `>>` și respectiv **inserție** `<<`, definiți pentru fiecare din **tipurile de date standard** existente în limbaj. Declarațiile acestora se găsesc în `<iostream.h>` și le prezentăm în continuare:

```
class istream : virtual public ios
{
// . . . . .
public:
    // extrage un caracter
    istream& operator>> ( signed char & );
    istream& operator>> (unsigned char &);

    // extrage valori numerice din reprezentarea lor ascii
    istream& operator>> (short & );
    istream& operator>> (unsigned short & );
    istream& operator>> (int & );
    istream& operator>> (unsigned int & );
    istream& operator>> (long & );
    istream& operator>> (unsigned long & );
    istream& operator>> (float & );
    istream& operator>> (double & );
    istream& operator>> (long double & );

    // extrage un sir
    istream& operator>> ( signed char * );
    istream& operator>> (unsigned char * );
// . . . . .
};

class ostream : virtual public ios
{
// . . . . .
public:
    // inserează un caracter
    ostream& operator<< ( signed char );
    ostream& operator<< (unsigned char );

    // inserează reprezentarea ascii a valorii numerice
    ostream& operator<< (short );
```

```
ostream& operator<< (unsigned short);
ostream& operator<< (int);
ostream& operator<< (unsigned int);
ostream& operator<< (long);
ostream& operator<< (unsigned long);
ostream& operator<< (float);
ostream& operator<< (double);
ostream& operator<< (long double);

// inserează un sir
ostream& operator<< (const signed char *);
ostream& operator<< (const unsigned char *);

// inserează reprezentarea ascii a valorii pointerului
ostream& operator<< (void *);
// . . . . .
};
```

Deci, existența acestor declarații (și implementarea operatorilor în cadrul bibliotecii) este cea care ne permite să utilizăm operatorii de inserție și de extracție. Faptul că acești operatori **întorc o referință la streamul respectiv** ne dă dreptul să **cascadăm** acești operatori în secvențe de genul:

```
cout << x << y << z << t; // la fel cascadăm și pentru cin
```

Fiecare operator, începând cu al doilea, se aplică asupra a ceea ce înțoarce cel anterior (același stream). Operatorii se aplică de la stânga la dreapta, **conform asociativității** acestora.

Facem următoarea remarcă referitoare la operațiile de intrare cu format:

- de această dată la extragerea unui singur caracter se extrage primul caracter din istream diferit de **caracter de tip spațiu** (care au fost amintite anterior) – deci se săracă caracterele de tip spațiu. Faptul este explicabil deoarece acum avem de a face cu o extragere formatată.

și următoarea remarcă valabilă atât pentru cele fără formatat cât și pentru cele cu format:

- extragerea (formatată și neformatată) este definită pentru istream iar inserarea (formatată și neformatată) pentru ostream. Clasa iostream **moștenește multiplu** istream și ostream deci va avea disponibile atât extragerea cât și inserarea (dar acestea **nu vor putea fi folosite alternat** în aceeași secvență de extrageri sau inserări).

5.5 Format de inserție-extracție. Stare a formatului

Prin redefinirea inserției și extracției s-a reușit realizarea de operații de intrare-iesire cu format.

Dar, deocamdată, formatul folosit este cel implicit. Pentru ca inserția și extracția să fie cu adevărat utile trebuie ca formatul folosit să poată fi modificat după dorință (asemănător cu descriptorii de format din familia de funcții printf). Pentru a realiza acest lucru în clasa ios găsim următoarele:

```

class ios
{
// . . . . .
public:
    // fanioane de formatare
    enum {
        skipws = 0x0001, // sare spații (la intrare)
        left = 0x0002, // aliniere la stânga (la ieșire)
        right = 0x0004, // aliniere la dreapta (la ieșire)
        internal = 0x0008, // spațiu după semn sau bază
        dec = 0x0010, // conversie în / din zecimal
        oct = 0x0020, // conversie în / din octal
        hex = 0x0040, // conversie în / din hexazecimal
        showbase = 0x0080, // indică baza (la ieșire)
        showpoint = 0x0100, // forțează afișare punct zecimal
        uppercase = 0x0200, // hexazecimal cu litere mari (ieșire)
        showpos = 0x0400, // adaugă '+' la întregi pozitivi
        scientific = 0x0800, // folosește notația 1.2345E2
        fixed = 0x1000, // folosește notația 123.45
        unitbuf = 0x2000, // golește (flush) toate streamurile
                           după inserție
        stdio = 0x4000 // golește (flush) stdout și stderr
                           după inserție
    }; // sfârșit enumerare

    // citire / setare / ștergere fanioane de formatare
    long flags();
    long flags(long);
    long setf(long _setbits, long _field); // pe biți
    long setf(long); // pe biți
    long unsetf(long); // pe biți

    // constante pentru parametrul _2 din setf()
    static const long basefield; // dec | oct | hex
    static const long adjustfield; // left | right | internal
    static const long floatfield; // scientific | fixed

    // citire / setare dimensiune câmp
    int width();
    int width(int);

    // citire / setare caracter de umplere
    char fill();

```

```

char      fill(char);

    // citire / setare digiți precizie la flotant
int      precision();
int      precision(int);

protected:
    long     x_flags;      // biții fanoanelor de formatare
    int      x_width;     // dimensiune câmp la ieșire
    int      x_fill;       // caracter de umplere la ieșire
    int      x_precision; // precizia la ieșire flotantă
// . . . . .
};

```

Considerăm că explicațiile anterioare sunt suficiente pentru a înțelege metodele disponibile pentru operarea cu formatul streamului. Facem doar următoarele observații:

- după cum era de așteptat, variabilele care memorează starea sunt declarate `protected`, pentru a fi accesibile claselor derivate dar inaccesibile pentru acces direct.
- pentru membrii “dimensiune câmp”, “caracter umplere” și “digiți precizie flotant” avem metode de citire a valorii și de setarea acestora.
- pentru diferenții biți de stare a formatului avem la dispoziție metode de citire, scriere și scriere pe biți. La cele pe biți amintim:
 - `setf(long)` setează în format (`x_flags`) biții pe 1 din valoarea parametrului.
 - `unsetf(long)` sterge în format (`x_flags`) biții pe 1 din valoarea parametrului.
 - `setf(long, long)` sterge în format (`x_flags`) biții pe 1 din valoarea celui de al doilea parametru și apoi setează în format (`x_flags`) biții pe 1 din valoarea primului parametru. Se asigură astfel un mecanism comod de a evita setări eronate de biți (de exemplu aliniere atât spre stânga cât și spre dreapta). Utilitatea constantelor `basefield`, `adjustfield` și `floatfield` și modul lor de definire devine acum evident.

Prezentăm în continuare un exemplu de ieșire formatată cu formate diferite:

```

#include <iostream.h>

main()
{
double a=987.654;

cout<<a<<"\n";           // format implicit, inserția 1
cout.setf(ios::left, ios::adjustfield); // aliniere

```

```

cout.width(10);                      // dimensiune
cout.fill('_');                     // umplere
cout.precision(4);                  // precizie
cout<<a<<"\n";                   // inserția 2

cout.setf(ios::right, ios::adjustfield); // aliniere
cout.width(10);                      // dimensiune
cout.fill('>');                     // umplere
cout.precision(2);                  // precizie
cout<<a<<"\n";                   // inserția 3

cout<<a<<"\n";                   // inserția 4
cout.width(15);                      // dimensiune
cout<<a<<"\n";                   // inserția 5
}

```

Analizând și rezultatele tipărite:

```

987.654          // inserția 1
987.651          // inserția 2
>>>>>>>987.65  // inserția 3
987.65          // inserția 4
>>>>>>>987.65  // inserția 5

```

constatăm următoarele:

- inserția implicită se face pe numărul minim de caractere necesare (corespunzător valorii implicite 0 pentru `x_width`).
- toate modificările de format făcute funcționează corespunzător.
- analizând inserția 4 se pare că modificările de format sunt valabile doar pentru următoarea inserție. Inserția 5 dovedește contrariul: toate modificările de format se păstrează cu excepția dimensiunii – aceasta se pune automat pe 0 (valoarea implicită) după fiecare inserție.

5.6 Manipulatori predefiniți

După cum am văzut anterior se poate controla formatul folosit la operațiile de intrare-iesire. Soluția prezentată are însă o limitare importantă: nu se pot înlăntui inserții (sau extractii) cu formate diferite - pentru acest lucru s-au introdus manipulatori de format (pe scurt manipulatori).

Manipulatorii predefiniți se împart în două categorii:

- fără argumente
 - dec, hex, oct – setează fanioanele cu același nume din `x_flags`.

Programare Orientată pe Obiecte - Principii

- ws – elimină toate caracterele de tip spațiu din intrare.
- endl – inserează ‘\n’ apoi golește (flush) streamul.
- ends – inserează un terminator de tip NULL.
- flush – golește (flush) streamul.
- cu un argument
 - setbase – setează baza de numerație pentru întregi (8, 10 sau 16).
 - setiosflags, resetiosflags – setează respectiv șterge în x_flags biții pe 1 din argument.
 - setfill, setw, setprecision – atribuie argumentul variabilelor membru x_width, x_fill, x_precision.

Prezentăm în continuare un exemplu de ieșire formatată cu manipulatori:

```
#include <iostream.h>
#include <iomanip.h> // pentru manipulatorii cu parametri

main()
{
int x=127;
double a=987.654;

cout<<x<<endl<<hex<<x<<endl<<oct<<x<<endl<<endl;

cout<<a<<endl // de fapt totul e o singura instructiune
    <<setiosflags(ios::left)<<setw(10)
    <<setfill('_')<<setprecision(4)
    <<a<<endl
    <<setiosflags(ios::right)<<setw(15)
    <<setfill('>')<<setprecision(2)
    <<a<<endl
    <<a<<endl<<setw(15)
    <<a<<endl;
}
```

Analizând și rezultatele tipărite:

```
127                // implicit
7f                // in hexazecimal
177                // in octal

987.654           // implicit
987.654          // cu un format (1)
>>>>>>>987.65  // cu alt format (2)
987.65            // cu formatul (2) dar lățime minimă (0)
```

```
>>>>>>>987.65 // cu formatul (2)
```

constatăm următoarele:

- s-a reușit inserarea cu formate diferite (sursa s-a "rupt" pe mai multe linii doar din motive de lizibilitate)
- pentru folosirea manipulatorilor cu parametri trebuie inclus <iomanip.h>.
- chiar și în cazul formatării cu manipulatori formatul se păstrează de la o inserție la alta dar `x_width` se resetează după fiecare inserție.

Acum, după ce am văzut că mecanismul manipulatorilor funcționează, să vedem și de ce funcționează ? (ce construcții ale limbajului au în spate ?). Pentru acest lucru trebuie tratați separat cei fără parametri și cei cu un parametru deoarece mecanismul este diferit.

Manipulatori fără parametri – în <iostream.h> găsim următoarele definiții:

```
class istream : virtual public ios
{
// . . . .
public:
    // manipulatori
    istream& operator>> (istream & (*_f)(istream &));
    istream& operator>> (ios & (*_f)(ios &));
// . . . .
};

class ostream : virtual public ios
{
// . . . .
public:
    // manipulatori
    ostream& operator<< (ostream& (*_f)(ostream&));
    ostream& operator<< (ios& (*_f)(ios&));
// . . . .
};
// . . . .
// manipulatori
ios&     dec(ios &);      // bază conversie zecimală
ios&     hex(ios &);      // bază conversie hexadecimală
ios&     oct(ios &);      // bază conversie octală
istream& ws(istream &);   // extrage caracterele de tip spațiu
ostream& endl(ostream &); // inserează '\n' și golește (flush)
ostream& ends(ostream &); // inserează terminator sir (null)
ostream& flush(ostream &); // golește (flush)
```

Constatăm că, pentru a permite existența manipulatorilor, clasele `istream` și `ostream` **redefinesc**

operatorii de inserție și extracție pentru tipul pointer la funcție cu prototipurile descrise anterior și se definesc 7 asemenea funcții manipulator. În momentul inserției unui asemenea pointer metoda operator apelează funcția respectivă cu streamul curent (indicat de *this*) ca și parametru, funcția putând deci face inserări, extracții sau modificări de format în / din streamul primit parametru.

Apariția unui manipulator fără parametri într-o secvență de inserții / extracții echivalează deci cu **inserția / extracția unui pointer la funcție**, pentru care există redefiniții operatorii de inserție / extracție.

Manipulatori cu parametri – în <iomanip.h> (atenție, trebuie inclus pentru a putea folosi manipulatorii cu parametri) găsim următoarele definiții:

```
class smanip_int {
    ios& (*_fn)(ios &, int); // membru pointer la functie
    int _ag; // membru intreg
public:
    smanip_int(ios& (*_f)(ios&, int), int _a):_fn(_f),_ag(_a) {}
    friend istream& operator>>(istream& _s,const smanip_int& _f)
        { (*_f._fn)(_s, _f._ag); return _s; }
    friend ostream& operator<<(ostream& _s,const smanip_int& _f)
        { (*_f._fn)(_s, _f._ag); return _s; }
};

// asemănător se definește smanip_long (cu long în loc de int)

// baza de conversie 8, 10 sau 16
smanip_int      setbase(int _b);

// sterge în x_flags biții pe 1 din argument
smanip_long      resetiosflags(long _b);

// setează în x_flags biții pe 1 din argument
smanip_long      setiosflags(long _b);

// setează caracterul de umplere
smanip_int      setfill(int _f);

// setează digitii precizie la flotant
smanip_int      setprecision(int _n);

// setează dimensiune câmp
smanip_int      setw(int _n);
```

În acest caz, datorită priorității superioare a operatorului () de apel de funcție față de prioritatea

operatorilor de inserție / extracție, se execută întâi un simplu apel de funcție (manipulatorul), care întoarce, prin valoare, un obiect de tipul smanip_int sau smanip_long pentru care există redefiniți operatorii de inserție / extracție. În acest fel în / din stream se inserează / extrage obiectul întors de funcția manipulator. După cum constatăm, toți manipulatorii cu un parametru predefiniți (descriși mai sus) se conformează acestei reguli și întorc un obiect de tipul smanip_int sau smanip_long. Vom discuta în continuare doar cazul obiectelor smanip_int, pentru smanip_long lucrurile petrecându-se asemănător (în fapt, ambele clase sunt definite simultan, printr-un mecanism de tip #define sau folosind clase template, în funcție de implementare).

Analizând clasa smanip_int constatăm existența unui constructor public cu 2 parametri – un pointer la o funcție și o valoare întreagă – care doar salvează valorile parametrilor în variabile membru. Mai găsim redifiniți (ca și funcții friend) operatorii de inserție și extracție pentru această clasă, operatori care apeleză funcția indicată de pointerul la funcție având ca și parametri streamul și valoarea întreagă. Prin acest mecanism al obiectului temporar de tip smanip_int se reușește “pasarea” valorii parametrului manipulatorului spre funcția care execută efectiv operația, păstrând eleganța înlăturării operatorilor de inserție / extracție. Detalii suplimentare se vor găsi la tratarea manipulatorilor cu parametri definiți de utilizator.

5.7 Manipulatori definiți de utilizator

Pe lângă manipulatorii predefiniți existenți programatorul poate defini și manipulatori noi, care pot opera schimbări complexe asupra formatului folosit în continuare. Deși exemplele care vor fi prezentate nu indică acest lucru, amintim că dimensiunea câmpului (parametrul x_width din format) este resetată după fiecare inserție. Deoarece mecanismul manipulatorilor este diferit pentru cei fără parametri și pentru cei cu parametri, vom prezenta în cele ce urmează câte un exemplu de definire de manipulator propriu pentru fiecare tip.

Definire manipulator fără parametri:

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

ostream& format1(ostream& o_str)      // primul manipulator
{
    o_str<<endl<<"format1"<<endl;
    o_str<<setiosflags(ios::left);
    o_str<<setw(10);
    o_str<<setfill('_');
    o_str<<setprecision(4);
```

```
return o_str;
}

ostream& format2(ostream& o_str)      // al doilea manipulator
{
    o_str<<endl<<"format2"<<endl;
    o_str<<setiosflags(ios::right);
    o_str<<setw(15);
    o_str<<setfill('>');
    o_str<<setprecision(2);
    return o_str;
}

main()
{
double a=987.654;

    cout<<a<<endl<<format1<<a<<endl<<format2<<a<<endl;
}
```

care va produce la ieșire:

```
987.654
format1
987.654_____
format2
>>>>>>>987.65
```

Analizând exemplul constatăm:

- manipulatorii fără parametri se definesc simplu ca funcții având un prototip impus: primesc și întorc referință la un stream (istream sau ostream după cum dorim să îi folosim la extracție sau inserție) sau la un obiect de tipul ios.
- pentru folosire se inserează / extrage pointerul la funcția respectivă (numele funcției).
- apelul lor se face (de către operatorul de inserție / extracție redefinit de ostream / istream) cu streamul curent ca și parametru.

Remarcăm mecanismul simplu de apel al manipulatorilor fără parametri, bazat doar pe redefinirea, în clasele ostream respectiv istream, a inserției / extracției pentru tipul pointer la funcție.

Definire manipulator cu parametri:

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

ios& my_manip_f(ios& i, int nr) // funcție auxiliară
{
    if(nr==1){                                // formatul 1
        i.setf(ios::left, ios::adjustfield);
        i.width(10);
        i.fill('_');
        i.precision(4);
    }
    if(nr==2){                                // formatul 2
        i.setf(ios::right, ios::adjustfield);
        i.width(15);
        i.fill('>');
        i.precision(4);
    }
    return i;
}

smanip_int my_manip(int n) // manipulator propriu-zis
{
    return smanip_int(my_manip_f, n); // prin valoare
}

main()
{
double a=987.654;

    cout<<a<<endl<<my_manip(1)<<a<<endl<<my_manip(2)<<a<<endl;
}
```

care va produce la ieșire:

```
987.654
987.654
>>>>>>987.654
```

Analizând exemplul constatăm:

- manipulatorii cu parametri (în exemplul nostru `my_manip`) se definesc ceva mai complicați, ca și funcții având un prototip impus: primesc un parametru întreg și întorc (**prin valoare**) un obiect de tipul `smanip_int` (sau `smanip_long`) construit pe baza unui constructor având ca și parametri: un pointer la o funcție auxiliară asociată (cu prototip impus) și valoarea parametrului manipulatorului.
- În funcția auxiliară, având prototipul impus și primind parametru o referință la `ios`, nu putem folosi operatorii de inserție / extracție și manipulatorii predefiniți deoarece

aceştia sunt definiţi doar în clasele derivate istream şi ostream şi suntem obligaţi să facem formataările cu metodele lui ios.

- efectiv se va inseră / extrage în / din stream acest obiectul smanip_int întors prin valoare de către funcţia manipulator
- funcţia auxiliară asociată se apelează (de către operatorul de inserție / extracție redefinit de smanip_int) având ca și parametri streamul curent și parametrul întreg.

Remarcăm mecanismul laborios de apel al manipulatorilor cu parametri, mecanism necesar pentru:

- a putea "cascada" asemenea manipulatori împreună cu inserția / extracția altor tipuri.
- a transmite parametrul manipulatorului spre o funcție operator de inserție / extracție care să aibă acces la streamul implicat.

5.8 Intrare – ieșire pentru tipuri definite de utilizator

O altă facilitate a bibliotecii de streamuri este că permite (pe lângă inserția / extracția tipurilor predefinite) în mod unitar inserția / extracția tipurilor definite de utilizator.

Deși mecanismul nu este prea complicat, având la bază simpla redefinire a operatorilor de inserție și / sau extracție pentru tipul definit de utilizator, eleganța acestui mecanism face ca această facilitate să fie una foarte răspândită.

Exemplificăm în cele ce urmează acest mecanism pentru o clasă proprie PUNCT (simplificată):

```
#include <iostream.h>

class PUNCT
{
    int x,y;
public:
    friend istream& operator>>(istream& i, PUNCT& ob)
    { i>>ob.x; i>>ob.y; return i; }
    friend ostream& operator<<(ostream& o, PUNCT& ob)
    { o<<"x="<<ob.x<<" y="<<ob.y; return o; }
};

PUNCT un_punct;

main()
{
    cout<<"introduceti x si y "; cin>>un_punct;
    cout<<"coordonatele sunt "; cout<<un_punct;
}
```

care va produce la rulare:

```
introduceti x si y 1 2      // acum un_punct.x=1, un_punct.y=2
coordonatele sunt x=1 y=2
```

Facem următoarele observații:

- valorile 1 și 2 cu bold și italic au fost introduse de la tastatură (nu afișate direct de către program).
- operatorii de inserție și extracție s-au redefinit ca și funcții friend pentru clasa PUNCT.
- citirea respectiv scrierea obiectului un_punct s-a realizat simplu și elegant. Avantajul este și mai vizibil pentru clase mai complexe (cum ar fi tablouri, liste, arbori) și pentru streamuri de tip fstream.
- deși nu apare vizibil în acest exemplu, inserția / extracția tipurilor predefinite și a celor definite de utilizator poate să fie "amestecat", în orice succesiune.

5.9 Legarea streamurilor ("Tying of streams")

O problemă specifică care apare într-o secvență de genul:

```
int x;
cout<<"introduceti x:" // inserție
cin>>x;               // extracție
```

este următoarea: cum avem garanția că textul inserției apare pe ecran înainte de execuția extracției (având în vedere că streamurile sunt "buffer-ate", după cum se va descrie mai amănunțit ulterior)? Soluția acestei probleme se găsește în clasa ios în următorul cod:

```
class ios
{
// . . . . .
public:
    // citirea / scrierea ostream-ului legat de acest stream
    ostream* tie();           // citește x_tie
    ostream* tie(ostream*);   // atribuie x_tie și întoarce
                             // vechea valoare
protected:
    ostream* x_tie;          // ostream-ul legat, dacă există
// . . . . .
};
```

Există deci un pointer (evident protected) spre ostream-ul asociat stream-ului curent și metode de citire și scriere a acestui pointer. Acum, orice extracție (>>) forțează golirea ostream-ului

legat ("tied stream", dacă el există), aşa încât suntem siguri că inserțiile (<<) anterioare au ajuns efectiv "la destinație".

O secvență de genul:

```
cout<<"introduceti x:" // inserție
cin>>x;             // extractie
```

este deci echivalentă cu:

```
cout<<"introduceti x:" // inserție
cout.flush();          // golire (flush) a ostream-ului legat
cin>>x;              // extractie (precedată de golirea
                      //         ostream-ului legat)
```

deoarece cout este legat de cin printr-o secvență `cin.tie(&cout)`. Un apel de genul `is.tie(0)` anulează legătura anterioară a streamului.

5.10 Buffer-area streamurilor

Operațiile de intrare – ieșire sunt specificate fără mențiunea tipului streamurilor, dar nu toate dispozitivele pot fi tratate uniform în ceea ce privește strategia de buffer-are. Soluția este aceea de a furniza la inițializare diferite tipuri de buffere. Asupra acestora există însă un set comun de operații, aşa încât istream și ostream nu cunosc detaliile implementării. Evident, pentru această problemă există **soluția**, extrem de potrivită, a **mecanismului metodelor virtuale** - metodele underflow și overflow sunt declarate virtuale. Aceste metode sunt suficiente pentru a actualiza, după dorință, strategia de buffer-are și sunt un exemplu foarte bun de utilizare a mechanismului metodelor virtuale pentru a permite tratarea uniformă a unor facilități logice echivalente, având însă implementări diferite. În <iostream.h> găsim:

```
class streambuf {
// . . . .
public:
    // constructori și destructori
    streambuf();           // streambuf gol
    streambuf(char*, int); // streambuf cu un tablou dat
    virtual ~streambuf();

    // extragere caractere din buffer
    int sgetc();            // extrage 1 caracter
    int sgetn(char*, int);  // extrage n caractere
    virtual int underflow(); // umple bufferul gol
```

```

        // punere caractere in buffer
        int sputc(int);           // pune 1 caracter
        int sputn(const char*, int); // pune n caractere
    virtual int overflow(int = EOF); // golește (flush) buffer

protected:
    // o mulțime de metode de accesat membrii privați de către
    // clasele derivate

private:
    short alloc_; // nenul dacă buffer trebuie eliberat în final
    short unbuf_; // nenul dacă nu este buffer-at
    char* base_; // începutul zonei buffer
    char* ebuf_; // sfîrșit+1 al zonei buffer
    char* pptr_; // următoarea locație pentru pus (inserat)
    char* qptr_; // următoarea locație pentru luat (extras)
// . . . . .
};

// . . . . .

class ios
{
// . . . . .

public:
    streambuf* rdbuf(); // intoarce streambuf asociat

protected:
    streambuf* bp; // streambuf-ul asociat
// . . . . .
};


```

Clasa streambuf are metode de citire (extragere) din buffer a unuia sau mai multor caractere și de scriere (inserție) în buffer a unuia sau mai multor caractere. Pentru tratarea cazurilor când bufferul devine plin respectiv gol există metodele virtuale **overflow** respectiv **underflow**.

Clasele strstreambuf și filebuf vor moșteni streambuf și vor adăuga constructori mai deosebiți, metode specifice și vor redefini metodele **virtuale** **overflow** și **underflow** astfel încât clasa ios (și cele derivate) să utilizeze aceste clase buffer în mod transparent.

Clasa ios (clasa de bază a ierarhiei) are un streambuf asociat pe care îl va folosi în operațiile de intrare-iesire.

5.11 Streamuri și siruri

Pentru lucrul cu streamuri reprezentate pe siruri există, în cadrul celor două ierarhii, clasele strstreambase, istrstream, ostrstream și strstream și respectiv strstreambuf. Pentru utilizarea

acestora este necesară includerea antetului <strstrea.h>. Acesta, la rândul său, include implicit <iostream.h> care nu mai trebuie inclus explicit (dar clasele descrise în el sunt vizibile).

În <strstrea.h> găsim:

```
class strstreambase : public virtual ios {  
public:  
    strstreambuf* rdbuf(); // implementata {return &this->buf;}  
private:  
    strstreambuf buf;      // strstreambuf-ul asociat  
protected:  
    // 2 constructori și destrutor  
};                                // declararea clasei e prezentata complet
```

Mai merită amintite metodele str(), existente în ostrstream și strstream, care întorc adresa șirului pe care este implementat bufferul asociat (și deci și streamul). Clasa ostrstream mai are în plus doar o alta metodă (pe care nu o mai descriem); în rest avem doar constructori și destrutori (și moștenirile corespunzătoare ierarhiei).

Un exemplu de utilizare a unui stream reprezentat pe șiruri este dat în continuare:

```
#include <strstrea.h> // pentru ostrstream  
#include <stdio.h>     // pentru printf  
  
main()  
{  
    char un_sir[100],*p;  
    int x=1,y=2,z=3;  
  
    ostrstream my_str(un_sir,100); // instantiez un ostrstream  
  
    my_str<<"x=<<x<<" y=<<y<<" z=<<z<<ends; // scriu in el  
    p=my_str.rdbuf()->str(); // iau adresa sirului  
    printf("sirul = '%s'",p); // o folosesc la ceva (tiparire)  
}
```

care tipărește

```
sirul = 'x=1 y=2 z=3'
```

Constatăm astfel că programul a făcut inserții în mod clasic, fără a conta că este un stream pe șir. Șirul astfel rezultat (formatat) poate fi apoi folosit la alte operații. Mecanismul este oarecum corespunzător funcției sprintf din C standard doar că acum este transparent pentru cel care face inserția care este destinația (stream pe ecran, pe șir sau pe fișier).

5.12 Streamuri și fișiere

Pentru lucrul cu streamuri reprezentate pe fișiere există, în cadrul celor două ierarhii, clasele `fstreambase`, `ifstream`, `ofstream` și `fstream` și respectiv `filebuf`. Pentru utilizarea acestora este necesară includerea antetului `<fstream.h>`. Acesta, la rândul său, include implicit `<iostream.h>` care nu mai trebuie inclus explicit (dar clasele descrise în el sunt vizibile).

În `<iostream.h>` găsim

```
class ios
{
public:
    // moduri de deschidere a unui stream
    enum open_mode {
        in      = 0x01,    // pentru citire
        out     = 0x02,    // pentru scriere
        ate     = 0x04,    // poziționare la sfârșit
        app     = 0x08,    // în append
        trunc   = 0x10,    // trunchiază dacă există
        nocreate = 0x20,   // eșuare dacă fișierul nu există
        noreplace = 0x40,  // eșuare dacă fișierul există
        binary   = 0x80    // fișier binar (nu text)
    };

    // pentru poziționare
    enum seek_dir { beg=0, cur=1, end=2 };
// . . . . .
};
```

iar în `<fstream.h>` găsim:

```
class fstreambase : virtual public ios {
public:
    // 4 constructori și destructor

    void open(const signed char*, int, int = filebuf::openprot);
    void attach(int);
    void close();
    filebuf* rdbuf(); // implementată { return &buf; }

private:
    filebuf buf;      // filebuf-ul asociat
};
```

Clasele `ifstream`, `ofstream` și `fstream` au doar constructori, destructori și metodele `rdbuf` și `open` cu aceleași prototipuri, dar uneori având valori implicate pentru parametri.

Programare Orientată pe Obiecte - Principii

Prezentăm în continuare un exemplu de inserarea unui obiect de tip definit de programator (o matrice rudimentară) într-un stream de tip fișier.

```
#include <fstream.h>

class M // o clasa "matrice"
{
    int m[3][3];
    friend istream& operator>>(istream& i, M& a);
    friend ostream& operator<<(ostream& o, M& a);
public:
    M();
}a; // si o instanta de matrice

M::M() // constructor
{
    for(int y=0;y<3;y++)
        for(int x=0;x<3;x++)
            m[y][x]=10*y+x; // o initializare carecore
}

ostream& operator<<(ostream& o, M& m)
{
    for(int y=0;y<3;y++,o<<endl) // atentie la endl
        for(int x=0;x<3;x++)
            o<<m.m[y][x]<<" "; // insereaza un element
    return o;
}

istream& operator>>(istream& i, M& m)
{
    for(int y=0;y<3;y++)
        for(int x=0;x<3;x++)
            i>>m.m[y][x]; // extrage un element
    return i;
}

main()
{
    // insereaza matricea in stream (fisier)
    ofstream ofs("matrice.txt"); // instantiere un ofstream
    ofs<<a; // inserare a pe stream
    ofs.close(); // inchidere stream

    // extrage matricea din stream (fisier)
    ifstream ifs("matrice.txt"); // instantiere un ifstream
    ifs>>a; // extragere a de pe stream
    ifs.close(); // inchidere stream
```

}

În urma rulării matricea va avea (după extracție) tot valorile cu care a fost inițializată în constructor. Vom avea însă fișierul matrice.txt cu următorul conținut:

```
0 1 2  
10 11 12  
20 21 22
```

Atragem atenția că, în mod normal (deși nu obligatoriu), inserția și extracția sunt gândite în pereche, pentru a putea ulterior extrage de pe stream obiectul inserat.

5.13 Concluzii

După cum am amintit la începutul capitolului, putem privi biblioteca de streamuri ca și un exemplu de aplicare a noțiunilor întâlnite în capitolele anterioare. Astfel avem:

- **încapsulare** (de exemplu: starea streamului și a formatului sunt memorate în variabile protected, pointerii din streambuf sunt privați, metodele sunt în majoritate publice).
- **instanță încuibărită** (membrul buf din clasa strstreambase).
- **moștenire** (a se vedea ambele ierarhii din Figura 5. 1).
- **moștenire multiplă virtuală** (în clasa ostream, de aceea starea streamului și starea formatului sunt unice atât pentru operațiile de intrare cât și pentru cele de ieșire).
- **polimorfism** (metodele underflow și overflow de la clasa streambuf sunt **virtuale** pentru a putea fi înlocuite).
- **redefinire de operatori** (utilizarea << și >> se bazează pe redefinirea, în clasele ostream și istream, a acestor operatori).

Rezumat - de ținut minte :-)

- **inserție (<<)** și **extracție (>>)**
- ierarhile de clase, ce sunt cin și cout
- starea unui stream
- operații de intrare-ieșire fără **format** și cu **format, cascadare**
- **format** de inserție-extracție, **stare** a formatului
- manipulatori fără parametrii și cu parametrii: utilizare, definire, **mecanism**
- legarea și buffer-area streamurilor
- streamuri și **clase proprii** - redefinirea inserției și extracției ca și **funcții friend**
- streamuri pe **șir** și streamuri pe **fișier**
- **exemplu real** de scriere a unei ierarhii de clase

Cap. 6 - Tratarea exceptiilor

“Exceptie” este unul din acele cuvinte care au semnificație diferită pentru persoane diferite. Mecanismul C++ de tratare a **exceptiilor** este proiectat pentru a permite tratarea **erorilor**, mai ales în programe ce folosesc componente soft scrise independent. Exceptiile sunt privite în acest context ca și o **generalizare a noțiunii de eroare**, putând avea și exceptii care nu sunt erori.

Mecanismul de tratare a exceptiilor este destinat doar tratării exceptiilor sincrone. El este de fapt o structură de control ne-locală (“non-local control structure”) care poate fi privită ca un mecanism de întoarcere. Totuși, principalul scop al mecanismului C++ de tratare a exceptiilor este tratarea erorilor.

6.1 Originea problemei

Problema apare la interfața între un **modul “client”**, care folosește niște servicii, și un **modul “nucleu”**, care le oferă. În acest context, prin modul nucleu înțelegem orice modul care furnizează servicii, fie el modul de bibliotecă (și, eventual, strâns legat de nucleul sistemului de operare) sau modul al propriei aplicații, iar prin modul client înțelegem orice modul care apelează (invocă, utilizează) servicii ale modulului nucleu.

Problema își are originea în următoarea **contradicție**:

- **Modulul nucleu** - poate detecta o excepție dar nu știe cum dorește clientul să o trateze (poate detecta, nu poate trata).
- **Modulul client** – știe cum trebuie tratată excepția nu poate detecta apariția ei (poate trata, nu poate detecta).

6.2 Soluția tradițională de tratare a erorilor și exceptiilor

În mod tradițional pentru tratarea erorilor aveam de ales între următoarele variante:

- **terminarea aplicației** – acest lucru se întâmplă implicit când o excepție nu este “prinsă” de nimeni. De cele mai multe ori însă putem (și trebuie) să facem mai mult.
- **întoarcerea unei valori reprezentând “eroare”** – aceasta este o soluție des întâlnită, cu toții am testat vreodată dacă un apel de funcție de bibliotecă nu a întors cumva eroare. Uneori însă, când toate valorile întoarse sunt semnificative, se impune alegerea pentru

variabila întoarsă a unui tip “mai mare” pentru a putea discerne codul erorii. Necesitatea testării permanente a valorii întoarse strică însă eleganța codului.

- **poziționarea unui indicator global** – este și ea des întâlnită (împreună cu soluția precedentă sau singură). Necesitatea testării acestui indicator global devine repede supărătoare astfel încât eleganța codului are din nou mult de suferit. În prezența concurenței (a procesării paralele) soluția este evident inaceptabilă.
- **ignorarea excepției** – este amintită mai mult ca și posibilitate, decât ca soluție propriu-zisă. Din momentul ignorării unei excepții nu mai avem nici o garanție asupra predictibilității funcționării ulterioare a programului.
- **invocarea unui handler specializat** – este o soluție corectă, când se generează un apel spre o funcție de tratare a excepției. Principala problemă este aceea că mecanismul nu este general, fiind valabil doar pentru un număr foarte restrâns de excepții.

6.3 Soluția C++ de tratare a excepțiilor

Deoarece toate soluțiile amintite anterior prezintă **dezavantaje majore**, în C++ s-a introdus un **mecanism nou** de tratare a excepțiilor, care are la bază 3 **cuvinte cheie**:

- **try** – delimită un bloc la nivelul clientului care conține instrucțiuni susceptibile de a genera excepții (semnificația fiind “încearcă să execute acest cod”).
- **throw** – se execută de către modulul nucleu când se detectează o situație excepțională pentru a semnaliza modulului client acest lucru (semnificația fiind de “aruncă excepția modulului client”).
- **catch** – delimită un bloc la nivelul clientului care se va executa în momentul intercepțării unei excepții (semnificația fiind “prinde acest tip de excepție”).

În sarcina modulului client intră:

- punerea codului “vulnerable” într-un bloc **try**.
- tratarea în blocuri **catch** a diferitelor excepții ce ar putea să apară.

În sarcina modulului nucleu intră (uneori tot noi scriem și acest modul):

- detectarea situației deosebite și generarea excepției corespunzătoare cu **throw**.

Un exemplu tipic de utilizare este prezentat în Figura 6.1.

6.4 Discriminarea excepțiilor

În continuare prezentăm un exemplu care dorește să evidențieze majoritatea situațiilor care apar în practică:

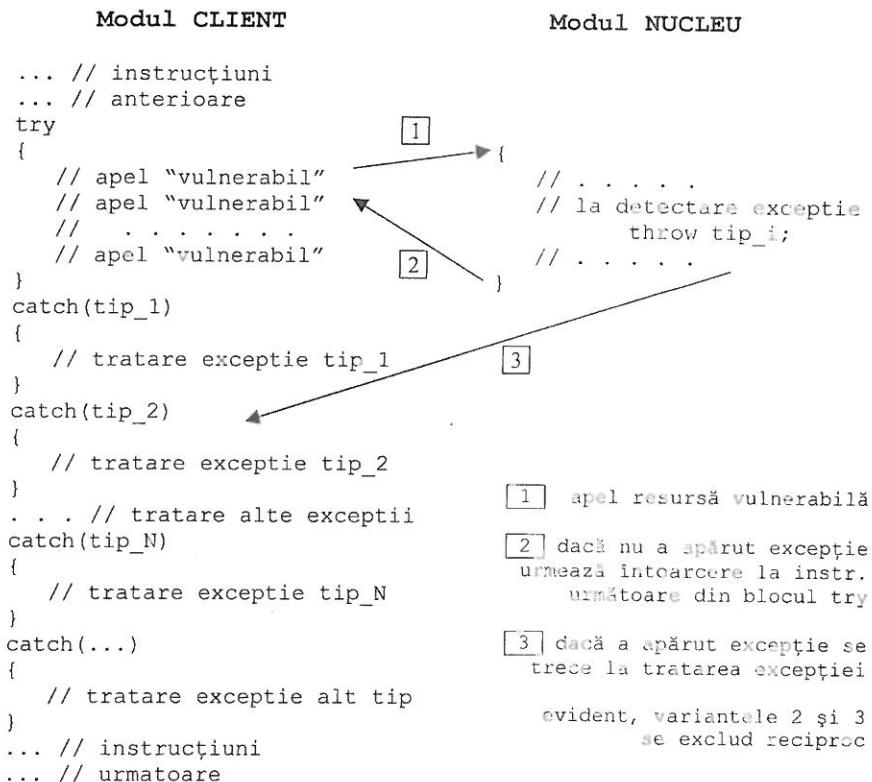


Figura 6.1 Exemplificarea mecanismului de tratare a excepțiilor

```

#include <stdio.h>
#include <conio.h>

class CError1 {};
class CError2 {};
class CError3
{
public: int err;
        CError3(int ecode) {err=ecode;}
};
class CError4 {};

void functie(int ch)          // pe post de modul nucleu
{
    printf("\nIntrare functie '%c'", ch );
}

```

```

switch(ch)
{
    case 'A': throw CError1();           // se genereaza
    case 'B': throw CError2();           // diferite
    case 'C': throw CError3(7);          // exceptii
    case 'D': throw CError4();
}
printf("\nIesire functie '%c'", ch );
}

void main()
{
    printf("\nINCEPUT");
    try
    {
        printf("\nApelEZ");
        functie(getch()); // aici trebuie apasata o tasta
        printf("\nSucces");
    }
    catch(CError1)                  // urmeaza discriminarea
    {                                // exceptiilor
        printf("\nCatch CError1");
    }
    catch(CError2 e)
    {
        printf("\nCatch CError2");
    }
    catch(CError3& e)
    {
        printf("\nCatch CError3 %d", e.err);
    }
    catch(...)                      // ramura "default"
    {
        printf("\nCatch general");
    }
    printf("\nSFIRSIT");           // de aici se continua dupa
}                                // constructia try-catch

```

Programul se va comporta diferit în funcție de tasta pe care o apăsăm în momentul rulării. Vom considera pe rând toate situațiile semnificative, vom indica ce tipărește programul și vom încerca să explicăm acest lucru:

Se apasă tasta E

```

INCEPUT
ApelEZ
Intrare functie 'E'

```

```
Iesire functie 'E'  
Sucses  
SFIRSAT
```

Totul se desfășoară normal, mecanismul de excepții nu intervine în nici un fel.

Se apasă tastă A

```
INCEPUT  
Apelez  
Intrare functie 'A'  
Catch CError1  
SFIRSAT
```

În switch se generează o excepție de tipul CError1, se abandonează corpul funcției și blocul try, se continuă cu blocul catch cu potrivire de tip și apoi cu instrucțiunile de după blocul try-catch.

Se apasă tastă B

```
INCEPUT  
Apelez  
Intrare functie 'B'  
Catch CError2  
SFIRSAT
```

Lucrurile se petrec asemănător cu cazul anterior, singura diferență este apariția variabilei e în catch-ul corespunzător.

Se apasă tastă C

```
INCEPUT  
Apelez  
Intrare functie 'C'  
Catch CError3 7  
SFIRSAT
```

Lucrurile se petrec asemănător cu cazul anterior, dar variabila e apare prin referință. În plus, avem accesibilă și valoarea 7 transmisă din corpul funcției (de la throw).

Se apasă tastă D

```
INCEPUT  
Apelez  
Intrare functie 'D'
```

Catch general
SFÎRSIT

În situația în care nu avem nici un bloc catch pentru tipul respectiv, se intră la catch(...).

Din exemplele anterioare reies și următoarele observații generale:

- La apariția oricărei exceptii se abandonează **corful funcției și corful blocului try**. Dacă ultima instrucțiune din blocul try ajunge să se execute, avem garanția că întreg blocul try s-a executat cu succes (aici am putea da un mesaj de genul “operația a fost executată cu succes”).
- La generarea unei exceptii (de un anumit tip) se continuă cu blocul catch **cu potrivire de tip**. Această discriminare a exceptiilor se poate face doar după tip (cazul CError1) sau poate fi utilizat și un obiect exceptie (cazul CError2 și CError3).
- Situația seamănă cu un apel de funcție (apelul blocului catch) și transmitere de parametru – prin valoare pentru exemplul CError2 și prin referință pentru exemplul CError3.
- În situația CError3 s-a realizat ceea ce se numește **“tratare de exceptii cu schimb de informații”**. Deși această facilitate este deosebit de utilă (modulul nucleu poate transmite informații modulului client cu detalii despre cauza exceptiei) ea are la bază o idee destul de simplă, anume aceea că obiectul exceptie poate avea membri (date și metode). Modul în care se va genera obiectul de exceptie și se va transmite el blocului catch va fi tratat ulterior.
- Tratarea exceptiilor cu schimb de informații se poate face atât în cazul variabilei exceptie transferată prin referință (ca în exemplul CError3) cât și în cel cu transfer prin valoare (de exemplu, s-ar fi putut face și pentru CError2) dar, evident, nu se poate face când discriminarea se face doar după tip (cazul CError1).
- Pentru un anumit tip discriminarea se poate face:
 - prin tip (de exemplu CError1)
 - prin tip și variabilă transmisă prin valoare (de exemplu CError2)
 - prin tip și variabilă transmisă prin referință (de exemplu CError3)

dar **numai prin una din aceste posibilități**. În cazul existenței mai multor blocuri catch pentru același tip, se generează eroare la compilare.

- Dacă nu există nici un bloc catch cu potrivire de tip se intră pe la catch(...). Dacă acesta nu există, exceptia va fi tratată de către modulul client ierarhic superior (apelantul acestui cod).
- La apariția unei exceptii se execută un singur bloc catch. După terminarea corpului acestuia se continuă cu instrucțiunile de după construcția try-catch.

6.5 Detalii despre implementarea mecanismului de tratare a excepțiilor

Acum, după ce am văzut cum se utilizează mecanismul de tratare a excepțiilor, să încercăm să înțelegem și ceea ce se întâmplă “intern”. Pentru aceasta am considerat exemplul următor:

```
#include <stdio.h>

class X           // clasa oarecare
{
public: int x;
    X(int x0){x=x0; printf("\nCXE %d", x);}
    ~X()       {printf("\nDX  %d", x);}
};

class E           // clasa excepție
{
public: int e;
    E(int e0) {e=e0; printf("\nCEE %d", e);}
    E(E& ee) {e=ee.e; printf("\nCEC %d", e);}
    ~E()       {printf("\nDE  %d", e);}
};

void functie()      // pe post de modul "nucleu"
{
    X a(1);
    throw E(11);
}

X b(2);

void main()
{
    X c(3);

    printf("\nINCEPUT");
    try
    {
        X d(4);
        printf("\nApelez");
        functie();           // apelare modul "nucleu"
        printf("\nSucces");
    }
    catch(E e)
    {
        X f(5);
        printf("\ncatch E %d", e.e);
    }
    catch(...)
}
```

```

{
    X g(E);
    printf("\ncatch ...");
}
printf("\nSFIRSAT");
}

```

La rularea programului se va tipări:

```

CXE 2      // instantiere b
CXE 3      // instantiere c
INCEPUT
CXE 4      // instantiere d
Apelez
CXE 1      // instantiere a
CEE 11     // instanțiere obiect exceptie "local"
CEC 11     // instanțiere (copiere) obiect exceptie "global"
DE 11      // distrugere obiect exceptie "local"
DX 1       // distrugere a - parasire fortata a functiei
DX 4       // distrugere d - parasire fortata bloc try
CEC 11     // instantiere e      (***)
CXE 5      // instantiere f
catch E 11 // utilizare e (e.x=11)
DX 5       // distrugere f
DE 11      // distrugere e      (***)
DE 11      // distrugere obiect exceptie "global"
SFIRSAT
DX 3       // distrugere c
DX 2       // distrugere b

```

Din analiza rezultatelor observăm:

- Se apeleză constructorii instanțelor b, c, d și a în mod normal.
- La generarea unei excepții - throw E(11) - **se instanțiază un obiect de clasă excepție (E)** folosind constructorul cu parametri (deoarece am considerat un caz de excepție cu transfer de informații). Dacă am fi avut un caz de excepție fără transfer de informații, s-ar fi instanțiat obiectul excepție folosind constructorul fără parametri (dar, totuși, trebuiau puse parantezele după throw). Remarcăm **asemănarea** sintaxei cu cea a definirii unui obiect, dar cu **throw** în loc de **new** (și deosebirea în ceea ce privește parantezele).
- La părăsirea corpului funcției **se face o copie**, cu constructorul de copiere, obiectului excepție instanțiat la throw (mechanism asemănător întoarcerii de obiect prin valoare).
- După generarea excepției se părăscă forțat corpul funcției și corpul blocului try și se **distrug toate obiectele locale ale acestor blocuri** – cel de excepție "local", apoi a și d.
- La intrarea în blocul catch **se instanțiază obiectul e**, folosind constructorul de copiere,

prin copiere din obiectul de excepție "global". (***)

- Se instanțiază f și se utilizează obiectul e (cu valoarea 11 transmisă în momentul generării excepției și memorată în obiectele excepție).
- La părăsirea corpului blocului catch se distrugе f.
- La părăsirea corpului blocului catch se distrugе e. (***)
- La părăsirea construcției try-catch se distrugе obiectul excepție "global".
- Se continuă cu instrucțiunile următoare construcției try-catch și cu distrugerea instanțelor c și b.

Constatăm că în modulul nucleu se instanțiază **efectiv** un obiect excepție, care apoi se **aruncă** (pasează) spre modulul client, care îl **prinde** și îl poate folosi (accesa).

Dacă **discriminarea** excepției s-ar fi făcut **prin referință** - adică am fi avut catch(E& e) – atunci nu s-ar mai fi copiat obiectul excepție "global" în e ci s-ar fi folosit direct acesta (referit prin numele e), iar **liniile notate cu (***) nu ar mai fi apărut**.

Deoarece constatăm că obiectul excepție instanțiat la throw ajunge în blocul catch după copieri succesive, **trebuie avută în vedere eventuala redefinire a constructorului de copiere** pentru clasa excepție (pentru a evita problemele care pot să apară prin copierea membru cu membru realizată implicit și analizată într-un capitol anterior).

După cum am văzut, înainte de intrarea în blocul catch se părăsesc forțat funcția și blocul try. De aceea **trebuie obligatoriu eliminată orice dependență a obiectului excepție de variabile locale ale acestor blocuri** (care nu mai există în momentul utilizării obiectului excepție).

6.6 Tratare polimorfică a excepțiilor

În situația în care avem nevoie de multe clase de excepție iar acestea au multe în comun, putem pune ceea ce este comun într-o clasă de bază și folosi apoi moștenirea. Acest lucru simplifică declararea claselor de excepție dar impune **atenție suplimentară** în ceea ce privește **ordinea sevențelor catch**. Dacă considerăm clasele următoare:

```
class CB {};           // clasa de baza
class CD1 : public CB {}; // clasa derivata 1
class CD2 : public CB {}; // clasa derivata 2
```

atunci, pentru o discriminare corectă a excepțiilor, trebuie puse întâi blocurile catch ale claselor derivate și abia apoi blocul catch al clasei de bază:

```
try{/*...*/}
catch(CD1){}
catch(CD2){}
catch(CB) {} // aici se discrimineaza doar exceptia CD1
              // aici se discrimineaza doar exceptie CD2
              // aici se discrimineaza doar exceptia CB
```

În cazul în care ordinea este cea a declarării claselor, funcționarea ar fi următoarea:

```
try{/*...*/}
catch(CB) {} // aici se discrimineaza exceptiile CB,CD1,CD2
catch(CD1){}
catch(CD2) {} // aici nu se mai discrimineaza nici o exceptie
               // aici nu se mai discrimineaza nici o exceptie
```

probabil diferită de ceea ce ne-am dorit. Cauza acestui comportament este **conversia** pe care o face compilatorul dinspre clasele derivate CD1 și CD2 înspre clasa de bază (conversie aici nedorită).

Această discriminare a disponibilelor clase într-o unică secvență catch poate fi și foarte **folositoare** dacă aranjăm ca tratarea dorită să se realizeze printr-o metodă virtuală a obiectului exceptie. În acest fel vom avea o singură secvență catch iar tratarea va fi corespunzătoare tipului efectiv al exceptiei (la baza soluției fiind **polimorfismul**, prin legarea dinamică oferită de metodele virtuale).

```
class CB      { virtual void rezolva(); };
class CD1 : public CB { void rezolva(); };
class CD2 : public CB { void rezolva(); };

try{/*...*/}
catch(CB& e)
{
    e.rezolva(); // rezolva corect toate exceptiile (prin
                  //          polimorfism)
```

Prin această soluție putem rezolva una din cele mai mari probleme ale mecanismului C++ de tratare a exceptiilor: pe măsură ce numărul secvențelor catch crește, se compromite eleganța modulelor client. Putem avea acum un singur bloc catch indiferent de numărul exceptiilor discriminate.

6.7 Achiziție și eliberare de resurse

O situație care poate pune probleme deosebite este aceea a achiziționării și eliberării de resurse. Deoarece soluția este identică oricare ar fi resursa implicată, exemplificăm în cele ce urmează pe cazul în care resursa este un fișier. O situație tipică este următoarea:

```
void FolosesteFisier(char *nume)
{
FILE* fp = fopen(nume, "r");

    // se foloseste fp
    fclose(fp);           // inchidem fisierul
}
```

Dacă însă, pe parcursul utilizării fișierului, se generează o excepție se părăsește funcția (după cum am văzut anterior) și nu se mai face închiderea fișierului. Pentru a evita această problemă (se "uită" eliberarea resursei, ceea ce poate duce la epuizarea resursei) se impune utilizarea unui bloc try-catch astfel:

```
void FolosesteFisier(char *nume)
{
FILE* fp = fopen(nume, "r");

    try
    {
        // se foloseste fp
    }
    catch(...)
    {
        fclose(fp); // inchidere in situatia de exceptie
        throw;      // pasam exceptia "mai sus"
    }
    fclose(fp); // inchidere in situatia normala
}
```

Soluția este corectă dar neelegantă, mai ales în situația în care folosim mai multe resurse în corpul funcției. Există însă o soluție mult mai elegantă, aplicabilă în situații de genul:

```
void achizitie()
{
    // achizitie resursa 1
    // ...
    // achizitie resursa n

    // utilizare resurse

    // eliberare resursa n
    // ...
    // eliberare resursa 1
}
```

în care este esențial faptul că resursele se eliberează în ordine inversă alocării. Situația amintește foarte mult de apelul constructorilor și destrutorilor pentru obiectele locale ale

funcției. De aceea putem aborda această problemă prin folosirea de obiecte ale unei clase care are constructori și destructori.

Pentru exemplul anterior cu fișiere, definim clasa FilePtr care se comportă ca și FILE*:

```
class FilePtr
{
    FILE* fp;
public:
    FilePtr(char* n, char *a){fp=fopen(n,a);}
    FilePtr(FILE* p0){fp=p0;}
    ~FilePtr(){fclose(fp);}
    operator FILE*(){return fp;}
};
```

În acest moment utilizarea devine cât se poate de simplă:

```
void FolosesteFisier(char *nume)
{
    FilePtr fis(nume, "r");
    // se foloseste fis
}
```

În acest caz închiderea fișierului (eliberarea resursei) se face automat, pe destructorul obiectului, la părăsirea funcției (normală sau prin apariția unei excepții) fără a mai trebui să “ne batem capul” cu această problemă.

Remarcăm **redefinirea operatorului “cast”** (conversie forțată de tip) pentru a ne permite utilizarea elegantă a obiectului ca și cum ar fi FILE*. În acest fel este validă o expresie de genul

```
int c=fgetc(fis);
```

deoarece compilatorul face conversia de la FilePtr (care este furnizat la apel) la FILE* (cum este așteptat de funcția fgetc) folosind operatorul cast redefinit (care întoarce valoarea fp din obiect).

Tehnica prezentată poartă numele de **“resource acquisition is initialization”**. Ea este o tehnică generală, care se bazează, după cum am văzut, pe proprietățile constructorilor și destructorilor și pe interacțiunea acestora cu mecanismul de tratare a excepțiilor. Pe scurt, se echivalează **alocarea de resurse cu inițializarea de obiecte** bazându-ne apoi pe faptul că se vor distruge automat toate obiectele care au fost deja inițializate (construite).

Un obiect nu se consideră complet construit atât timp cât constructorul său nu s-a executat în întregime. Acest fapt trebuie accentuat deoarece apelul destructorului se face doar pentru obiectele construite complet. (Afirmațiile anterioare sunt valabile și pentru obiecte "încuibărite" sau "încuibaritoare").

Dacă este necesar ca într-un constructor să se aloce două (sau mai multe) resurse, este posibil să apară o excepție la alocarea celei de a doua și obiectul să se afle într-o stare "construit pe jumătate" periculoasă deoarece, obiectul nefiind construit complet, nu îl se va apela destructorul. Situația se rezolvă aplicând tehnica "resource acquisition is initialization" membrilor, ca în exemplul următor:

```
class X
{
    FilePtr f1;          // resursa 1, de exemplu fisier
    FilePtr f2;          // resursa 2, de exemplu fisier
    X(char *n1, char *n2):
        f1(n1),           // achizitie resursa 1
        f2(n2)            // achizitie resursa 2
    {
    }                   // corp propriu-zis
    // . .
};
```

În această situație, chiar dacă se generează excepție la alocarea celei de a două resurse, se va apela destructorul primei instanțe încuibărante, deoarece aceasta fusese construită complet.

O altă situație la fel de periculoasă este aceea în care în corpul constructorului, după alocarea unei singure resurse, mai trebuie făcute și alte operații (fără legătură cu resursele). Dacă aceste operații duc la generarea unei excepții, obiectul nu se consideră construit și nu se apelează destructorul, deci se pierde resursa. Soluția este tot aceea de a aplica tehnica "resource acquisition is initialization" și de a face alocarea resursei într-un obiect încuibărit.

Se recomandă **aplicarea acestei tehnici pentru fiecare resursă** (decic, alocarea / eliberarea fiecărei resurse este trecută în corpul constructorului / destructorului unui obiect) astfel ca în restul codului să nu mai avem alocări / eliberări de resurse ci doar cod fără legătură cu alocarea / eliberarea acestora. În acest fel orice excepție care apare determină eliberarea corespunzătoare a tuturor resurselor pentru care s-a reușit deja alocarea.

6.8 Opțiuni la tratarea unei excepții

În cele ce urmează prezentăm situațiile care pot fi întâlnite la tratarea unei excepții:

```

int f(int arg)
{
    try{
        g(arg);
    }
    catch(x1){
        // repara ceva si reia (daca stim sigur ca va fi OK)
        g(arg);
    }
    catch(x2){
        // calculeaza si intoarce un rezultat
        return 2;
    }
    catch(x3){
        // arunca mai sus exceptia
        throw;
    }
    catch(x4){
        // converteste x4 in alta exceptie si arunc-o mai sus
        throw xxii;
    }
    catch(x5){
        // rezolva definitiv si treci la linia urmatoare
    }
    catch(...){
        // renunta
        terminate();      // terminare anormala a programului
    }                      // ("Abnormal program termination")
// . . .
}

```

Atragem atenția doar asupra următoarelor situații:

- throw simplu re-aruncă aceeași excepție (același obiect excepție) "mai sus", excepția urmând a fi tratată de blocul try ierarhic superior (al apelantului).
- din punct de vedere al limbajului o excepție se consideră tratată în momentul intrării în blocul catch corespunzător, deci orice excepție care apare în blocurile catch este tratată de blocul try ierarhic superior (al apelantului).

6.9 Observații privind mecanismul de tratare a excepțiilor

După cum am mai amintit, acest mecanism nu este altceva decât un **mecanism de salt** (de întoarcere). Deși a fost proiectat în principal pentru tratarea erorilor, el poate fi folosit și în situații când nu avem efectiv erori. În fapt, în toate exemplele prezentate anterior nu am avut erori ci am generat excepții care nu aveau semnificație de erori. În bibliotecile reale însă generarea excepțiilor nu ne mai revine nouă, iar semnificațiile excepțiilor sunt de cele mai multe ori apariții de erori.

Spre deosebire de excepțiile hardware la care se asigură implicit restartarea instrucțiunii care a generat excepția, mecanismul C++ de tratare a excepțiilor prezentat în acest capitol permite **doar discriminarea erorii și tratarea ei** în blocul catch corespunzător, **nu și restartarea instrucțiunii**. Dacă dorim restartarea instrucțiunii care a provocat excepția trebuie să facem acest lucru explicit.

Deși mecanismul de tratare a excepțiilor **nu este strict obiectual** (tratarea nu se face la nivel de clasă) și nu are eleganța celorlalte construcții ale limbajului C++, el este totuși în strânsă legătură cu caracterul obiectual al limbajului, mecanismul de instanțiere și transfer al obiectelor excepție fiind **tipic obiectual**.

Rezumat - de ținut minte :-)

- **Modul nucleu** (nu știe trata), **modul client** (nu poate detecta)
Soluția C++: **try, throw, catch**
- **Discriminarea excepțiilor după tip**
- **Implementarea mecanismului, obiect excepție**
- Tratare **polimorfică** a excepțiilor
- “**Resource acquisition is initialization**”, achiziție în constructor
- **Opțiuni** la tratarea unei excepții
- Se asigură doar discriminarea excepției nu și restartarea ei

Anexă

Se consideră programul următor:

```

/* 1 */      #include <stdio.h>
/* 2 */      #include <iostream.h>
/* 3 */      class CA
/* 4 */
/* 5 */      {
/* 6 */          public: int x;
/* 7 */          CA()           {x=1; printf("\nCA1 %d",x);}
/* 8 */          CA(int x0)    {x=x0; printf("\nCAE %d",x);}
/* 9 */          CA(CA &p)     {x=p.x;printf("\nCAC %d",x);}
/* 10 */         void operator=(CA &p) {x=p.x;printf("\nAOP= %d",x);}
/* 11 */         ~CA()          {printf("\nDA %d",x);}
/* 12 */         void met1(int r){printf("\nMET1A %d %d",x,r);met2(r);met3(r);}
/* 13 */         void met2(int r){printf("\nMET2A %d %d",x,r);}
/* 14 */         virtual void met3(int r){printf("\nMET3A %d %d",x,r);}
/* 15 */
/* 16 */         CA Incrementeaza(CA x, CA &y, CA z, CA &t)
/* 17 */         {
/* 18 */             CA r;  printf("\n+++++");
/* 19 */             x.x++; y.x++; z.x++; t.x++; r.x = x.x + y.x + z.x + t.x;
/* 20 */             return r;
/* 21 */
/* 22 */
/* 23 */         class CB:public CA
/* 24 */
/* 25 */         {
/* 26 */             public: int y;
/* 27 */             CB() {y=2; printf("\nCBI %d %d",x,y);}
/* 28 */             ~CB() {printf("\nDB %d %d",x,y);}
/* 29 */             CB(int x0,int y0=3);
/* 30 */             int operator+(int p) {printf("\nBOP+I");return(x+p+1);}
/* 31 */             int operator+(CA &p) {printf("\nBOP+A");return(x+p.x);}
/* 32 */             friend int operator+(CB &ol,CB &ol2) {printf("\nBOP+B");return(ol.x+ol2.x);}
/* 33 */             void met1(int r){y=p.x;printf("\nBOP= %d %d",x,y);}
/* 34 */             void met2(int r){printf("\nMET2B %d %d",x,r);}
/* 35 */             virtual void met3(int r){printf("\nMET3B %d %d",x,r);}
/* 36 */
/* 37 */             struct CC:public CA
/* 38 */
/* 39 */             {
/* 40 */                 CA a;
/* 41 */                 CC(int x0) {x=x0;printf("\nCCE %d %d",x,a.x);}
/* 42 */                 ~CC()   {printf("\nDC %d %d",x,a.x);}
/* 43 */
/* 44 */                 CB::CB(int x0,int y0):CA(x0+4)
/* 45 */                 {
/* 46 */                     y=y0+5;printf("\nCBE %d %d",x,y);
/* 47 */                     CA a, b(10); CB c, *i, d(11,12);
/* 48 */
/* 49 */                     void main()
/* 50 */
/* 51 */                     {
/* 52 */                         CA e(13), *f, g=d; CB h(14); CC *x;
/* 53 */                         printf("\nSTART");
/* 54 */                         f=new CA(20);i=new CB(21,22);
/* 55 */                         b.met1(31); d.met1(32);
/* 56 */                         printf("\nOP %d %d %d %d",d+b,h+d,h.x+2,d+1);
/* 57 */                         printf("\nXYZT %d %d %d %d",a.x,b.x,e.x,g.x);
/* 58 */                         g=Incrementeaza(a,b,e,g);
/* 59 */                         printf("\nXYZT %d %d %d %d",a.x,b.x,e.x,g.x);
/* 60 */                         delete f; delete i;
/* 61 */                         d=h;printf("\nd=h %d %d %d %d",d.x,d.y,h.x,h.y);
/* 62 */                         (cout<<endl<<c.x)<<<<c.y<<"<<(c.x<<2)<<c.y;
/* 63 */                         x=new CC(22); delete x;
/* 64 */                         printf("\nSTOP");
/* 65 */
}

```

Programare Orientată pe Obiecte - Principii

Prezentăm în continuare **rezultatul rulării sale**.

Se va urmări (de câte ori este nevoie ☺) până se înțelege ...

```

CAI 1          // pentru obiectul global a           a.x=1
CAE 10         // pentru obiectul global b           b.x=10
CAI 1          // pentru obiectul global c           c.x=1
CBI 1 2         // pentru obiectul global c           c.y=2
CAE 15         // pentru obiectul global d           d.x=15
CBE 15 17       // pentru obiectul global d           d.y=17
CAE 13         // pentru obiectul local e           e.x=13
CAC 15         // pentru obiectul local g           g.x(=d.x)=15
CAE 18         // pentru obiectul local h           h.x=18
CBE 18 8         // pentru obiectul local h           h.y=8
START          // abia aici incepe codul functiei main
CAE 20         // pentru obiectul alocat dinamic f       f->x=20
CAE 25         // pentru obiectul alocat dinamic i       i->x=25
CBE 25 27       // pentru obiectul alocat dinamic i       i->y=27
MET1A 10 31     // b.x, r
MET2A 10 31     // b.x, r
MET3A 10 31     // b.x, r
MET1A 15 32     // d.x, r
MET2A 15 32     // d.x, r met3 nu este virtuala, deci este legata static
MET3B 15 32     // d.x, r met3 este virtuala, deci este legata dinamic
BOP+I          // pentru expresia d+1 (CB+int) d.x + 1 + 1 => 17
BOP+B          // pentru expresia h+d (CB+CB)   h.x + d.x => 33
BOP+A          // pentru expresia d+b (CB+CA)   d.x + b.x => 25
OP 25 33 20 17 // ordinea evaluarii este data de conventia de apel c
XYZT 1 10 13 15 //
CAC 13         // copiere obiect e in obiect (parametru) z      z.x(=e.x)=13
CAC 1          // copiere obiect a in obiect (parametru) x      x.x(=a.x)=1
CAI 1          // pentru obiectul local r           r.x=1
++++          // corp functie, x.x=2, y.x(b.x)=11, z.x=14, t.x(g.x)=16, r.x=43
CAC 43         // copiere obiect r in obiect temp           temp.x(=r.x)=43
DA 43          // pentru obiectul r (local)
DA 14          // pentru obiectul z (parametru)
DA 2           // pentru obiectul x (parametru)
AOP= 43        // copiere obiect temp in obiect g           g.x(=temp.x)=43
DA 43          // pentru obiectul temp
XYZT 1 11 13 43 // atentie cine s-a modificar si cine nu
DA 20          // pentru obiectul f
DB 25 27       // pentru obiectul i
DA 25          // pentru obiectul i
BOP= 15 18     // pentru d=h                           d.y(=h.x)=18
d=h 15 18 18 8 //
122<<42       // atentie la tipurile implicate si la ordinea operatorilor
CAI 1          // pentru obiectul x           x->x=1
CAI 1          // pentru obiectul incuibarit x->a      x->a.x=1
CCE 22 1        // pentru obiectul x           x->x=22
DC 22 1        // pentru obiectul x
DA 1           // pentru obiectul incuibarit x->a
DA 22          // pentru obiectul x
STOP           // aici se termina codul functiei main, tiparirea insa nu
DB 18 8         // pentru obiectul local h
DA 18          // pentru obiectul local h
DA 43          // pentru obiectul local g
DA 13          // pentru obiectul local e
DB 15 18       // pentru obiectul global d
DA 15          // pentru obiectul global d
DB 1 2         // pentru obiectul global c
DA 1           // pentru obiectul global c
DA 11          // pentru obiectul global b
DA 1           // pentru obiectul global a

```

Bibliografie

1. Brian W. Kernighan, Dennis M. Ritchie - "*The C programming language*", Second Edition, ISBN 0-13-110362-8, Prentice Hall, 1988
2. Ionuț Mușlea - "*Inițiere în C++. Programare orientată pe obiecte*", Editura Microinformatica, ISBN 973-95718-2-4, Cluj-Napoca, 1992
3. Ioan Jurca - "*Programarea orientată pe obiecte în limbajul C++*", Editura Eurobit, Timișoara, 1992
4. Bjarne Stroustrup - "*The C++ programming language*", Second Edition, ISBN 0-201-53992-6, Addison-Wesley, 1993
5. Dan Roman - "*Ingineria programării obiectuale*", Editura Albastră, ISBN 973-9215-30-0, Cluj-Napoca, 1996
6. *** - Documentație compilatoare C++

