2. Programare orientata pe obiect

Desi aceasta carte este dedicata in primul rand analizei si elaborarii algoritmilor, am considerat util sa folosim numerosii algoritmi care sunt studiati ca un pretext pentru introducerea elementelor de baza ale programarii orientate pe obiect in limbajul C++. Vom prezenta in capitolul de fata notiuni fundamentale legate de obiecte, limbajul C++ si de abstractizarea datelor in C++, urmand ca, pe baza unor exemple detaliate, sa conturam in capitolele urmatoare din ce in ce mai clar tehnica programarii orientate pe obiect. Scopul urmarit este de a surprinde acele aspecte strict necesare formarii unei impresii juste asupra programarii orientate pe obiect in limbajul C++, si nu de a substitui cartea de fata unui curs complet de C++.

2.1 Conceptul de obiect

Activitatea de programare a calculatoarelor a aparut la sfarsitul anilor ‘40. Primele programe au fost scrise in limbaj masina si de aceea depindeau in intregime de arhitectura calculatorului pentru care erau concepute. Tehnicile de programare au evoluat apoi in mod natural spre o tot mai neta separare intre conceptele manipulate de programe si reprezentarile acestor concepte in calculator.

In fata complexitatii crescande a problemelor care se cereau solutionate, structurarea programelor a devenit indispensabila. Scoala de programare Algol a propus la inceputul anilor ‘60 o abordare devenita intre timp clasica. Conform celebrei ecuatii a lui Niklaus Wirth:

algoritmi + structuri de date = programe

un program este format din doua parti total separate: un ansamblu de proceduri si un ansamblu de date asupra carora actioneaza procedurile. Procedurile sunt privite ca si cutii negre, fiecare avand de rezolvat o anumita sarcina (de facut anumite prelucrari). Aceasta modalitate de programare se numeste *programare dirijata de prelucrari*. Evolutia calculatoarelor si a problemelor de programare a facut ca in aproximativ zece ani programarea dirijata de prelucrari sa devina ineficienta. Astfel, chiar daca un limbaj ca Pascal-ul permite o buna structurare a programului in proceduri, este posibil ca o schimbare relativ minora in structura datelor sa provoace o dezorganizare majora a procedurilor.

Inconvenientele programarii dirijate de prelucrari sunt eliminate prin *incapsularea* datelor si a procedurilor care le manipuleaza intr-o singura entitate numita *obiect*. Lumea exterioara obiectului are acces la datele sau procedurile lui doar prin intermediul unor operatii care constituie *interfata* obiectului. Programatorul nu este obligat sa cunoasca reprezentarea fizica a datelor si procedurilor utilizate, motiv pentru care poate trata obiectul ca pe o cutie neagra cu un comportament bine precizat. Aceasta caracteristica permite realizarea unor *tipuri abstracte de date*. Este vorba de obiecte inzestrate cu o interfata prin care se specifica interactiunile cu exteriorul, singura modalitate de a comunica cu un astfel de obiect fiind invocarea interfetei sale. In terminologia specifica programarii orientate pe obiect, procedurile care formeaza interfata unui obiect se numesc *metode*. Obiectul este singurul responsabil de maniera in care se efectueaza operatiile asupra lui. Apelul unei metode este doar o cerere, un *mesaj* al apelantului care solicita executarea unei anumite actiuni. Obiectul poate refuza sa o execute, sau, la fel de bine, o poate transmite unui alt obiect. In acest context, programarea devine *dirijata de date*, si nu de prelucrarile care trebuie realizate.

Utilizarea consecventa a obiectelor confera programarii urmatoarele calitati:

- *Abstractizarea datelor*. Nu este nevoie de a cunoaste implementarea si reprezentarea interna a unui obiect pentru a-i adresa mesaje. Obiectul decide singur maniera de executie a operatiei cerute in functie de implementarea fizica. Este posibila supraincarcarea metodelor, in sensul ca la aceleasi mesaje, obiecte diferite raspund in mod diferit. De exemplu, este foarte comod de a desemna printr-un simbol unic, +, adunarea intregilor, concatenarea sirurilor de caractere, reuniunea multimilor etc.
- *Modularitate*. Structura programului este determinata in mare masura de obiectele utilizate. Schimbarea definitiilor unor obiecte se poate face cu un minim de implicatii asupra celorlalte obiecte utilizate in program.
- *Flexibilitate*. Un obiect este definit prin comportamentul sau gratie existentei unei interfete explicite. El poate fi foarte usor introdus intr-o biblioteca pentru a fi utilizat ca atare, sau pentru a construi noi tipuri prin *mostenire*, adica prin specializare si compunere cu obiecte existente.
- *Claritate*. Incapsularea, posibilitatea de supraincarcare si modularitatea intaresc claritatea programelor. Detaliile de implementare sunt izolate de lumea exterioara, numele metodelor pot fi alese cat mai natural posibil, iar interfetele specifica precis si detaliat modul de utilizare al obiectului.

2.2 Limbajul C++

Toate limbajele de nivel inalt, de la FORTRAN la LISP, permit adaptarea unui stil de programare orientat pe obiect, dar numai cateva ofera mecanismele pentru

utilizarea directa a obiectelor. Din acest punct de vedere, mentionam doua mari categorii de limbaje:

- Limbaje care ofera doar facilitati de abstractizarea datelor si incapsulare, cum sunt Ada si Modula-2. De exemplu, in Ada, datele si procedurile care le manipuleaza pot fi grupate intr-un pachet (package).
- Limbaje orientate pe obiect, care adauga abstractizarii datelor notiunea de mostenire.

Desi definitiile de mai sus restrang mult multimea limbajelor calificabile ca "orientate pe obiect", aceste limbaje raman totusi foarte diverse, atat din punct de vedere al conceptelor folosite, cat si datorita modului de implementare. S-au conturat trei mari familii, fiecare accentuand un anumit aspect al notiunii de obiect: limbaje de clase, limbaje de cadre (frames) si limbaje de tip actor.

Limbajul C++* apartine familiei limbajelor de clase. O *clasa* este un tip de date care descrie un ansamblu de obiecte cu aceeasi structura si acelasi comportament. Clasele pot fi imbogatite si completate pentru a defini alte familii de obiecte. In acest mod se obtin ierarhii de clase din ce in ce mai specializate, care *mostenesc* datele si metodele claselor din care au fost create. Din punct de vedere istoric primele limbaje de clase au fost Simula (1973) si Smalltalk-80 (1983). Limbajul Simula a servit ca model pentru o intrega linie de limbaje caracterizate printr-o organizare statica a tipurilor de date.

Sa vedem acum care sunt principalele deosebiri dintre limbajele C si C++, precum si modul in care s-au implementat intrarile/iesirile in limbajul C++.

2.2.1 Diferentele dintre limbajele C si C++

Limbajul C, foarte lejer in privinta verificarii tipurilor de date, lasa programatorului o libertate deplina. Aceasta libertate este o sursa permanenta de erori si de efecte colaterale foarte dificil de depanat. Limbajul C++ a introdus o verificare foarte stricta a tipurilor de date. In particular, apelul oricarei functii trebuie precedat de *declararea* functiei respective. Pe baza declaratiilor, prin care se specifica numarul si tipul parametrilor formali, parametrii efectivi poat fi verificati in momentul compilarii apelului. In cazul unor nepotriviri de tipuri, compilatorul incearca realizarea corespondentei (*matching*) prin invocarea unor conversii, semnaland eroare doar daca nu gaseste nici o posibilitate.

```
float maxim( float, float );
float x = maxim( 3, 2.5 );
```

* Limbaj dezvoltat de Bjarne Stroustrup la inceputul anilor '80, in cadrul laboratoarelor Bell de la AT&T, ca o extindere orientata pe obiect a limbajului C.

In acest exemplu, functia `maxim()` este declarata ca o functie de tip `float` cu doi parametri tot de tip `float`, motiv pentru care constanta intreaga `3` este convertita in momentul apelului la tipul `float`. Declaratia unei functii consta in *prototipul functiei*, care contine tipul valorii returnate, numele functiei, numarul si tipul parametrilor. Diferenta dintre *definitie* si declaratie – notiuni valabile si pentru variabile – consta in faptul ca definitia este o declaratie care provoaca si rezervare de spatiu sau generare de cod. Declararea unei variabile se face prin precedarea obligatorie a definitiei de cuvantul cheie `extern`. Si o declaratie de functie poate fi precedata de cuvantul cheie `extern`, accentuand astfel ca functia este definita altundeva.

Definirea unor functii foarte mici, pentru care procedura de apel tinde sa dureze mai mult decat executarea propriu-zisa, se realizeaza in limbajul C++ prin functiile `inline`.

```
inline float maxim( float x, float y ) {
  putchar( 'r' ); return x > y? x: y;
}
```

Specificarea `inline` este doar orientativa si indica compilatorului ca este preferabil de a inlocui fiecare apel cu corpul functiei apelate. Expandarea unei functii `inline` nu este o simpla substitutie de text in progamul sursa, deoarece se realizeaza prin pastrarea semanticii apelului, deci inclusiv a verificarii corespondentei tipurilor parametrilor efectivi.

Mecanismul de verificare a tipului lucreaza intr-un mod foarte flexibil, permitand atat existenta functiilor cu un numar variabil de argumente, cat si a celor *supraincarcate*. Supraincarcarea permite existenta mai multor functii cu acelasi nume, dar cu paremetri diferiti. Eliminarea ambiguitatii care apare in momentul apelului se rezolva pe baza numarului si tipului parametrilor efectivi. Iata, de exemplu, o alta functie `maxim()`:

```
inline int maxim( int x, int y ) {
  putchar( 'i' ); return x > y? x: y;
}
```

(Prin apelarea functiei `putchar()`, putem afla care din cele doua functii `maxim()` este efectiv invocata).

In limbajul C++ nu este obligatorie definirea variabilelor locale strict la inceputul blocului de instructiuni. In exemplul de mai jos, tabloul `buf` si intregul `i` pot fi utilizate din momentul definirii si pana la sfarsitul blocului in care au fost definite.

```
#define DIM 5

void f( ) {
  int buf[ DIM ];

  for ( int i = 0; i < DIM; )
    buf[ i++ ] = maxim( i, DIM - i );
  while ( --i )
    printf( "%3d ", buf[ i ] );
}
```

In legatura cu acest exemplu, sa mai notam si faptul ca instructiunea `for` permite chiar definirea unor variabile (variabila `i` in cazul nostru). Variabilele definite in instructiunea `for` pot fi utilizate la nivelul blocului acestei instructiuni si dupa terminarea executarii ei.

Desi transmiterea parametrilor in limbajul C se face numai prin valoare, limbajul C++ autorizeaza in egala masura si transmiterea prin *referinta*. Referintele, indicate prin caracterul `&`, permit accesarea in scriere a parametrilor efectivi, fara transmiterea lor prin adrese. Iata un exemplu in care o procedura interschimba (swap) valorile argumentelor.

```
void swap( float& a, float& b ) {
  float tmp = a; a = b; b = tmp;
}
```

Referintele evita duplicarea provocata de transmiterea parametrilor prin valoare si sunt utile mai ales in cazul transmiterii unor structuri. De exemplu, presupunand existenta unei structuri de tip `struct punct`,

```
struct punct {
  float x;     /* coordonatele unui */
  float y;     /* punct din plan    */
};
```

urmatoarea functie transforma un punct in simetricul lui fata de cea de a doua bisectoare.

```
void sim2( struct punct& p ) {
  swap( p.x, p.y ); // p.x si p.y se transmit prin
                    // referinta si nu prin valoare
  p.x = -p.x; p.y = -p.y;
}
```

Parametrii de tip referinta pot fi protejati de modificari accidentale prin declararea lor `const`.

```
void print( const struct punct& p ) {
  // compilatorul interzice orice tentativa
  // de a modifica variabila p
  printf( "(%4.1f, %4.1f) ", p.x, p.y );
}
```

Caracterele `//` indica faptul ca restul liniei curente este un comentariu. Pe langa aceasta modalitate noua de a introduce comentarii, limbajul C++ a preluat din limbajul C si posibiliatea incadrarii lor intre `/*` si `*/`.

Atributul `const` poate fi asociat nu numai parametrilor formali, ci si unor definitii de variabile, a caror valoare este specificata in momentul compilarii. Aceste variabile sunt variabile read-only (constante), deoarece nu mai pot fi modificate ulterior. In limbajul C, constantele pot fi definite doar prin intermediul directivei `#define`, care este o sursa foarte puternica de erori. Astfel, in exemplul de mai jos, constanta intreaga `dim` este o variabila propriu-zisa accesibila doar in functia `g()`. Daca ar fi fost definita prin `#define` (vezi simbolul `DIM` utilizat in functia `f()` de mai sus) atunci orice identificator `dim`, care apare dupa directiva de definire si pana la sfarsitul fisierului sursa, este inlocuit cu valoarea respectiva, fara nici un fel de verificari sintactice.

```
void g( ) {
  const int dim = 5;
  struct punct buf[ dim ];

  for ( int i = 0; i < dim; i++ ) {
    buf[ i ].x = i;
    buf[ i ].y = dim / 2. - i;

    sim2(  buf[ i ] );
    print( buf[ i ] );
  }
}
```

Pentru a obtine un prim program in C++, nu avem decat sa adaugam obisnuitul

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

precum si functia `main()`

```
int main( ) {
  puts( "\n main." );

  puts( "\n  f( )" ); f( );
  puts( "\n  g( )" ); g( );
```

```
  puts( "\n ---\n" );
  return 0;
}
```

Rezultatele obtinute in urma rularii acestui program:

```
r
 main.

  f( )
iiiii  4   3   3   4
  g( )
(-2.5, -0.0) (-1.5, -1.0) (-0.5, -2.0)
( 0.5, -3.0) ( 1.5, -4.0)
 ---
```

suprind prin faptul ca functia `float maxim( float, float )` este invocata inaintea functiei `main()`. Acest lucru este normal, deoarece variabila `x` trebuie initializata inaintea lansarii in executie a functiei `main()`.

2.2.2 Intrari/iesiri in limbajul C++

Limbajul C++ permite definirea tipurilor abstracte de date prin intermediul claselor. Clasele nu sunt altceva decat generalizari ale structurilor din limbajul C. Ele contin date membre, adica variabile de tipuri predefinite sau definite de utilizator prin intermediul altor clase, precum si functii membre, reprezentand metodele clasei.

Cele mai utilizate clase C++ sunt cele prin care se realizeaza intrarile si iesirile. Reamintim ca in limbajul C, intrarile si iesirile se fac prin intermediul unor functii de biblioteca cum sunt `scanf()` si `printf()`, functii care permit citirea sau scrierea numai a datelor (variabilelor) de tipuri predefinite (`char`, `int`, `float` etc.). Biblioteca standard asociata oricarui compilator C++, contine ca suport pentru operatiile de intrare si iesire nu simple functii, ci un set de clase adaptabile chiar si unor tipuri noi, definite de utilizator. Aceasta biblioteca este un exemplu tipic pentru avantajele oferite de programarea orientata pe obiect. Pentru fixarea ideilor, vom folosi un program care determina termenul de rang *n* al sirului lui Fibonacci prin algoritmul *fib*2 din Sectiunea 1.6.4.

```
#include <iostream.h>

long fib2( int n ) {
  long i = 1, j = 0;

  for ( int k = 0; k++ < n; j = i + j, i = j - i );
  return j;
}

int main( ) {
  cout << "\nTermenul sirului lui Fibonacci de rang ... ";

  int n;
  cin >> n;

  cout << " este " << fib2( n );
  cout << '\n';

  return 0;
}
```

Biblioteca standard C++ contine definitiile unor clase care reprezinta diferite tipuri de *fluxuri de comunicatie* (`stream`-uri). Fiecare flux poate fi *de intrare*, *de iesire*, sau de *intrare/iesire*. Operatia primara pentru fluxul de iesire este *inserarea* de date, iar pentru cel de iesire este *extragerea* de date. Fisierul prefix (header) `iostream.h` contine declaratiile fluxului de intrare (clasa `istream`), ale fluxului de iesire (clasa `ostream`), precum si declaratiile obiectelor `cin` si `cout`:

```
extern istream cin;
extern ostream cout;
```

Operatiile de inserare si extragere sunt realizate prin functiile membre ale claselor `ostream` si `istream`. Deoarece limbajul C++ permite existenta unor functii care supraincarca o parte din operatorii predefiniti, s-a convenit ca inserarea sa se faca prin supraincarcarea operatorului de decalare la stanga `<<`, iar extragerea prin supraincarcarea celui de decalare la dreapta `>>`. Semnificatia secventei de instructiuni

```
cin >> n;
cout << " este " << fib2( n );
```

este deci urmatoarea: se citeste valoarea lui `n`, apoi se afiseaza sirul `" este "` urmat de valoarea returnata de functia `fib2()`.

Fluxurile de comunicatie `cin` si `cout` lucreaza in mod implicit cu terminalul utilizatorului. Ca si pentru programele scrise in C, este posibila redirectarea lor spre alte dispozitive sau in diferite fisiere, in functie de dorinta utilizatorului.

Pentru sistemele de operare UNIX si DOS, redirectarile se indica adaugand comenzii de lansare in executie a programului, argumente de forma `>nume-fisier-iesire`, sau `<nume-fisier-intrare`. In `iostream.h` mai este definit inca un flux de iesire numit `cerr`, utilizabil pentru semnalarea unor conditii de exceptie. Fluxul `cerr` este legat de terminalul utilizatorului si nu poate fi redirectat.

Operatorii de inserare (`<<`) si extragere (`>>`) sunt, la randul lor, supraincarcati astfel incat operandul drept sa poata fi de orice tip predefinit. De exemplu, in instructiunea

```
cout << " este " << fib2( n );
```

se va apela operatorul de inserare cu argumentul drept de tip `char*`. Acest operator, ca si toti operatorii de inserare si extragere, returneaza operandul stang, adica `stream-`ul. Astfel, invocarea a doua oara a operatorului de inserare are sens, de acesata data alegandu-se cel cu argumentul drept de tip `long`. In prezent, biblioteca standard de intrare/iesire are in jur de 4000 de linii de cod, si contine 15 alternative pentru fiecare din operatorii `<<` si `>>`. Programatorul poate supraincarca in continuare acesti operatori pentru propriile tipuri.

2.3 Clase in limbajul C++

Ruland programul pentru determinarea termenilor din sirul lui Fibonacci cu valori din ce in ce mai mari ale lui `n`, se observa ca rezultatele nu mai pot fi reprezentate intr-un `int`, `long` sau `unsigned long`. Solutia care se impune este de a limita rangul `n` la valori rezonabile reprezentarii alese. Cu alte cuvinte, `n` nu mai este de tip `int`, ci de un tip care limiteaza valorile intregi la un anumit interval. Vom elabora o clasa corespunzatoare acestui tip de intregi, clasa utila multor programe in care se cere mentinerea unei valori intre anumite limite.

Clasa se numeste `intErval`, si va fi implementata in doua variante. Prima varianta este realizata in limbajul C. Nu este o clasa propriu-zisa, ci o structura care confirma faptul ca orice limbaj permite adaptarea unui stil de programare orientat pe obiect si scoate in evidenta inconvenientele generate de lipsa mecanismelor de manipulare a obiectelor. A doua varianta este scrisa in limbajul C++. Este un adevarat tip abstract ale carui calitati sunt si mai bine conturate prin comparatia cu (pseudo) tipul elaborat in C.

2.3.1 Tipul `intErval` in limbajul C

Reprezentarea interna a tipului contine trei membri de tip intreg: marginile intervalului si valoarea propriu-zisa. Le vom grupa intr-o structura care, prin intermediul instructiunii `typedef`, devine sinonima cu `intErval`.

```
typedef struct {
  int min; /* marginea inferioara a intervalului */
  int max; /* marginea superioara a intervalului */
  int v;   /* valoarea, min <= v, v < max        */
} intErval;
```

Variabilele (obiectele) de tip `intErval` se definesc folosind sintaxa uzuala din limbajul C.

```
intErval numar = { 80, 32, 64 };
intErval indice, limita;
```

Efectul acestor definitii consta in rezervarea de spatiu pentru fiecare din datele membre ale obiectelor `numar`, `indice` si `limita`. In plus, datele membre din `numar` sunt initializate cu valorile 80 (`min`), 32 (`max`) si 64 (`v`). Initializarea, desi corecta din punct de vedere sintactic, face imposibla functionarea tipului `intErval`, deoarece marginea inferioara nu este mai mica decat cea superioara. Deocamdata nu avem nici un mecanism pentru a evita astfel de situatii.

Pentru manipularea obiectelor de tip `intErval`, putem folosi atribuiri la nivel de structura:

```
limita = numar;
```

Astfel de atribuiri se numesc *atribuiri membru cu membru*, deoarece sunt realizate intre datele membre corespunzatoare celor doua obiecte implicate in atribuire.

O alta posibilitate este accesul direct la membri:

```
indice.min = 32; indice.max = 64;
indice.v = numar.v + 1;
```

Selectarea directa a membrilor incalca proprietatile fundamentale ale obiectelor. Reamintim ca un obiect este manipulat exclusiv prin interfata sa, structura lui interna fiind in general inaccesibila.

Comportamentul obiectelor este realizat printr-un set de metode implementate in limbajul C ca functii. Pentru `intErval`, acestea trebuie sa permita in primul rand selectarea, atat in scriere cat si in citire, a valorii propriu-zise date de membrul `v`.

Functia de scriere `atr()` verifica incadrarea noii valori in domeniul admisibil, iar functia de citire `val()` pur si simplu returneaza valoarea `v`. Practic, aceste doua functii implementeaza o forma de incapsulare, izoland reprezentarea interna a obiectului de restul programului.

```
int atr( intErval *pn, int i ) {
  return pn->v = verDom( *pn, i );
}

int val( intErval n ) {
  return n.v;
}
```

Functia `verDom()` verifica incadrarea in domeniul admisibil:

```
int verDom( intErval n, int i ) {
  if ( i < n.min || i >= n.max ) {
    fputs( "\n\nintErval -- valoare exterioara.\n\n", stderr);
    exit( 1 );
  }
  return i;
}
```

Utilizand consecvent cele doua metode ale tipului `intErval`, obtinem obiecte ale caror valori sunt cu certitudine intre limitele admisibile. De exemplu, utilizand metodele `atr()` si `val()`, instructiunea

```
indice.v = numar.v + 1;
```

devine

```
atr( &indice, val( numar ) + 1 );
```

Deoarece `numar` are valoarea 64, iar domeniul `indice`-lui este 32, ..., 64, instructiunea de mai sus semnaleaza depasirea domeniului variabilei `indice` si provoaca terminarea executarii programului.

Aceasta implementare este departe de a fi completa si comod de utilizat. Nu ne referim acum la aspecte cum ar fi citirea (sau scrierea) obiectelor de tip `intErval`, operatie rezolvabila printr-o functie de genul

```
void cit( intErval *pn ) {
  int i;
  scanf( "%d", &i );
  atr( pn, i );
}
```

ci la altele, mult mai delicate, cum ar fi:

I_1 Evitarea unor initializari eronate din punct de vedere semantic si interzicerea utilizarii obiectelor neinitializate:

```
intErval numar = {80,32,64}; // obiect incorect initializat
intErval indice, limita;     // obiecte neinitializate
```

I_2 Interzicerea modificarii necontrolate a datelor membre:

```
indice.v = numar.v + 1;
```

I_3 Sintaxa foarte incarcata, diferita de sintaxa obisnuita in manipularea tipurilor intregi predefinite.

In concluzie, aceasta implementare, in loc sa ne simplifice activitatea de programare, mai mult a complicat-o. Cauza nu este insa conceperea gresita a tipului `intErval`, ci lipsa facilitatilor de manipulare a obiectelor din limbajul C.

2.3.2 Tipul `intErval` in limbajul C++

Clasele se obtin prin completarea structurilor uzuale din limbajul C cu setul de functii necesar implementarii interfetei obiectului. In plus, pentru realizarea izolarii reprezentarii interne de restul programului, fiecarui membru i se asociaza nivelul de incapsulare `public` sau `private`. Un membru `public` corespunde, din punct de vedere al nivelului de accesibilitate, membrilor structurilor din limbajul C. Membrii `private` sunt accesibili doar in *domeniul clasei*, adica in clasa propriu-zisa si in toate functiile membre. In clasa `intErval`, membrii publici sunt doar functiile `atr()` si `val()`, iar membrii `verDom()`, `min`, `max` si `v` sunt privati.

```
class intErval {
public:
  int atr( int );
  int val( ) { return v; }

private:
  int verDom( int );

  int min, max;
  int v;
};
```

Obiectele de tip `intErval` se definesc ca si in limbajul C.

```
intErval numar;
intErval indice, limita;
```

Aceste obiecte pot fi atribuite intre ele (fiind structuri atribuirea se va face membru cu membru):

```
limita = numar;
```

si pot fi initializate (tot membru cu membru) cu un obiect de acelasi tip:

```
intErval cod = numar;
```

Selectarea membrilor se face prin notatiile utilizate pentru structuri. De exemplu, dupa executarea instructiunii

```
indice.atr( numar.val( ) + 1 );
```

valoarea obiectului `indice` va fi valoarea obiectului `numar`, incrementata cu 1. Aceasta operatie poate fi descrisa si prin intructiunea

```
indice.v = numar.v + 1;
```

care, desi corecta din punct de vedere sintactic, este incorecta semantic, deoarece `v` este un membru `private`, deci inaccesibil prin intermediul obiectelor `indice` si `numar`.

Dupa cum se observa, au disparut argumentele de tip `intErval*` si `intErval` ale functiilor `atr()`, respectiv `val()`. Cauza este faptul ca functiile membre au un argument implicit, concretizat in *obiectul invocator*, adica obiectul care selecteaza functia. Este o conventie care intareste si mai mult atributul de functie membra (metoda) deoarece permite invocarea unei astfel de functii numai prin obiectul respectiv.

Definirea functiilor membre se poate face fie in corpul clasei, fie in exteriorul acestuia. Functiile definite in corpul clasei sunt considerate implicit `inline`, iar pentru cele definite in exteriorul corpului se impune precizarea statutului de functie membra. Inainte de a defini functiile `atr()` si `verDom()`, sa observam ca functia `val()`, definita in corpul clasei `intErval`, incalca de doua ori cele precizate pana aici. In primul rand, nu selecteaza membrul `v` prin intermediul unui obiect, iar in al doilea rand, `v` este privat! Daca functia `val()` ar fi fost o functie obisnuita, atunci observatia ar fi fost cat se poate de corecta. Dar `val()` este functie membra si atunci:

- Nu poate fi apelata decat prin intermediul unui obiect invocator si toti membrii utilizati sunt membrii obiectului invocator.

- Incapsularea unui membru functioneaza doar in exteriorul domeniului clasei. Functiile membre fac parte din acest domeniu si au acces la toti membrii, indiferent de nivelul lor de incapsulare.

Specificarea atributului de functie membra se face precedand numele functiei de operatorul domeniu `::` si de numele domeniului, care este chiar numele clasei. Pentru asigurarea consistentei clasei, functiile membre definite in exterior trebuie obligatoriu declarate in corpul clasei.

```
int intErval::verDom( int i ) {
  if ( i < min || i >= max ) {
    cerr << "\n\nintErval -- " << i
         << ": valoare exterioara domeniului [ "
         << min << ", " << (max - 1) << " ].\n\n";
    exit( 1 );
  }
  return i;
}

int intErval::atr( int i ) {
  return v = verDom( i );
  // verDom(), fiind membru ca si v, se va invoca pentru
  // obiectul invocator al functiei atr()
}
```

Din cele trei inconveniente mentionate in finalul Sectiunii 2.3.1 am rezolvat, pana in acest moment, doar inconvenientul I_2, cel care se refera la incapsularea datelor. In continuare ne vom ocupa de I_3, adica de simplificarea sintaxei.

Limbajul C++ permite nu numai supraincarcarea functiilor, ci si a majoritatii operatorilor predefiniti. In general, sunt posibile doua modalitati de supraincarcare:

- Ca functii membre, caz in care operandul stang este implicit obiect invocator.
- Ca functii nemembre, dar cu conditia ca cel putin un argument (operand) sa fie de tip clasa.

Pentru clasa `intErval`, ne intereseaza in primul rand operatorul de atribuire (implementat deocamdata prin functia `atr()`) si un operator care sa corespunda functiei `val()`. Desi pare surprinzator, functia `val()` nu face altceva decat sa converteasca tipul `intErval` la tipul `int`. In consecinta, vom implementa aceasta functie ca operator de conversie la `int`. In noua sa forma, clasa `intErval` arata astfel:

```
class intErval {
public:
  // operatorul de atribuire corespunzator functiei atr()
  int operator =( int i ) { return v = verDom( i ); }

  // operatorul de conversie corespunzator functiei val()
  operator int( ) { return v; }

private:
  int verDom( int );

  int min, max;
  int v;
};
```

Revenind la obiectele `indice` si `numar`, putem scrie acum

```
indice = (int)numar + 1;
```

sau direct

```
indice = numar + 1;
```

conversia `numar`-ului la `int` fiind invocata automat de catre compilator. Nu este nimic miraculos in aceasta invocare “automata”, deoarece operatorul `+` nu este definit pentru argumente de tip `intErval` si `int`, dar este definit pentru `int` si `int`. Altfel spus, expresia `numar + 1` poate fi evaluata printr-o simpla conversie a primului operand de la `intErval` la `int`.

O alta functie utila tipului `intErval` este cea de citire a valorii `v`, functie denumita in paragraful precedent `cit()`. Ne propunem sa o inlocuim cu operatorul de extragere `>>`, pentru a putea scrie direct `cin >> numar`. Supraincarcarea operatorului `>>` ca functie membra nu este posibila, deoarece argumentul stang este obiectul invocator si atunci ar trebui sa scriem `n >> cin`.

Operatorul de extragere necesar pentru citirea valorii obiectelor de tip `intErval` se poate defini astfel:

```
istream& operator >>( istream& is, intErval& n ) {
  int i;
  if ( is >> i )  // se citeste valoarea
    n = i;        // se invoca operatorul de atribuire
  return is;
}
```

Sunt doua intrebari la care trebuie sa raspundem referitor la functia de mai sus:

- Care este semnificatia testului `if ( is >> i )`?
- De ce se returneaza `istream`-ul?

In testul `if ( is >> i )` se invoca de fapt operatorul de conversie de la `istream` la `int`, rezultatul fiind valoarea logica *true* (valoare diferita de zero) sau *false* (valoarea zero), dupa cum operatia a decurs normal sau nu.

Returnarea `istream`-ului este o modalitate de a aplica operatorului `>>` sintaxa de concatenare, sintaxa utilizata in expresii de forma `i = j = 0`. De exemplu, obiectele `numar` si `indice` de tip `intErval`, pot fi citite printr-o singura instructiune

```
cin >> numar >> indice;
```

De asemenea, remarcam si utilizarea absolut justificata a argumentelor de tip referinta. In lipsa lor, obiectul `numar` ar fi putut sa fie modificat doar daca i-am fi transmis adresa. In plus, utilizarea sintaxei de concatenare provoaca, in lipsa referintelor, multiplicarea argumentului de tip `istream` de doua ori pentru fiecare apel: prima data ca argument efectiv, iar a doua oara ca valoare returnata.

Clasa `intErval` a devenit o clasa comod de utilizat, foarte bine incapsulata si cu un comportament similar intregilor. Incapsularea este insa atat de buna, incat, practic, nu avem nici o modalitate de a initializa limitele superioara si inferioara ale domeniului admisibil. De fapt, am revenit la inconvenientul I_1 mentionat in finalul Sectiunii 2.3.1. Problema initializarii datelor membre in momentul definirii obiectelor nu este specifica doar clasei `intErval`. Pentru rezolvarea ei, limbajul C++ ofera o categorie speciala de functii membre, numite *constructori*. Constructorii nu au tip, au numele identic cu numele clasei si sunt invocati automat de catre compilator, dupa rezervarea spatiului pentru datele obiectului definit.

Constructorul necesar clasei `intErval` are ca argumente limitele domeniului admisibil. Transmiterea lor se poate face implicit, prin notatia

```
intErval numar( 80, 32 );
```

sau explicit, prin specificarea constructorului

```
intErval numar = intErval( 80, 32 );
```

Definitia acestui constructor este

```
intErval::intErval( int sup, int inf ) {
  if ( inf >= sup ) {
    cerr << "\n\nintErval -- domeniu incorect specificat [ "
         << inf << ", " << (sup - 1) << " ].\n\n";
    exit( 1 );
  }
  min = v = inf;
  max =     sup;
}
```

Datorita lipsei unui constructor fara argumente, compilatorul va interzice orice declaratii in care nu se specifica domeniul. De exemplu,

```
intErval indice;
```

este o definitie incompleta, semnalata la compilare. Mai mult, definitiile incorecte semantic cum este

```
intErval limita( 32, 80 );
```

sunt si ele detectate, dar nu de catre compilator, ci de catre constructor. Acesta, dupa cum se observa, verifica daca limita inferioara a domeniului este mai mica decat cea superioara, semnaland corespunzator domeniile incorect specificate.

In declaratiile functiilor, limbajul C++ permite specificarea valorilor implicite ale argumentelor, valori utilizabile in situatiile in care nu se specifica toti parametrii efectivi. Aceasta facilitate este utila si in cazul constructorului clasei `intErval`. Prin declaratia

```
intErval( int = 1, int = 0 );
```

definitia

```
intErval indice;
```

nu va mai fi respinsa, ci va provoca invocarea constructorului cu argumentele implicite `1` si `0`. Corespondenta dintre argumentele actuale si cele formale se realizeaza pozitional, ceea ce inseamna ca primul argument este asociat limitei superioare, iar cel de-al doilea celei inferioare. Frecvent, limita inferioara are valoarea implicita zero. Deci la transmiterea argumentelor constructorului, ne putem limita doar la precizarea limitei superioare.

Constructorul apelabil fara nici un argument se numeste *constructor implicit*. Altfel spus, constructorul implicit este constructorul care, fie nu are argumente, fie are toate argumentele implicite. Limbajul C++ nu impune prezenta unui

constructor implicit in fiecare clasa, dar sunt anumite situatii in care acest constructor este absolut necesar.

Dupa aceste ultime precizari, definitia clasei `intErval` este:

```
class intErval {
public:
  intErval( int = 1, int = 0 );
  ~intErval( ) { }

  int operator =( int i ) { return v = verDom( i ); }
  operator int( ) { return v; }

private:
  int verDom( int );

  int min, max;
  int v;
};
```

Se observa aparitia unei noi functii membre, numita `~intErval()`, al carui corp este vid. Ea se numeste *destructor*, nu are tip si nici argumente, iar numele ei este obtinut prin precedarea numelui clasei de caracterul `~`. Rolul destructorului este opus celui al constructorului, in sensul ca realizeaza operatiile necesare distrugerii corecte a obiectului. Destructorul este invocat automat, inainte de a elibera spatiul alocat datelor membre ale obiectului care inceteaza sa mai existe. Un obiect inceteaza sa mai existe in urmatoarele situatii:

- Obiectele definite intr-o functie sau bloc de instructiuni (obiecte cu *existenta locala*) inceteaza sa mai existe la terminarea executarii functiei sau blocului respectiv.
- Obiectele definite global, in exteriorul oricarei functii, sau cele definite `static` (obiecte cu *existenta statica*) inceteaza sa mai existe la terminarea programului.
- Obiectele alocate dinamic prin operatorul `new` (obiecte cu *existenta dinamica*) inceteaza sa mai existe la invocarea operatorului `delete`.

Ca si in cazul constructorilor, prezenta destructorului intr-o clasa este optionala, fiind lasata la latitudinea proiectantului clasei.

Pentru a putea fi inclusa in toate fisierele sursa in care este utilizata, definitia unei clase se introduce intr-un fisier header (prefix). In scopul evitarii includerii de mai multe ori a aceluiasi fisier (*includeri multiple*), se recomanda ca fisierele header sa aiba structura

```
#ifndef simbol
#define simbol

// continutul fisierului

#endif
```

unde `simbol` este un identificator unic in program. Daca fisierul a fost deja inclus, atunci identificatorul `simbol` este deja definit, si deci, toate liniile situate intre `#ifndef` si `#endif` vor fi ignorate. De exemplu, in fisierul `intErval.h`, care contine definitia clasei `intErval`, identificatorul `simbol` ar putea fi `__INTeRVAL_H`. Iata continutul acestui fisier:

```
#ifndef __INTeRVAL_H
#define __INTeRVAL_H

#include <iostream.h>

class intErval {
public:
  intErval( int = 1, int = 0 );
  ~intErval( ) { }

  int operator =( int i ) { return v = verDom( i ); }
  operator int( ) { return v; }

private:
  int verDom( int );

  int min, max;
  int v;
};

istream& operator >>( istream&, intErval& );

#endif
```

Functiile membre se introduc intr-un fisier sursa obisnuit, care este legat dupa compilare de programul executabil. Pentru clasa `intErval`, acest fisier este:

```
#include "intErval.h"
#include <stdlib.h>

intErval::intErval( int sup, int inf ) {
  if ( inf >= sup ) {
    cerr << "\n\nintErval -- domeniu incorect specificat [ "
         << inf << ", " << (sup - 1) << " ].\n\n";
```

```
    exit( 1 );
  }
  min = v = inf;
  max =     sup;
}

int intErval::verDom( int i ) {
  if ( i < min || i >= max ) {
    cerr << "\n\nintErval -- "
         << i << ": valoare exterioara domeniului [ "
         << min << ", " << (max - 1) << " ].\n\n";
    exit( 1 );
  }
  return i;
}

istream& operator >>( istream& is, intErval& n ) {
  int i;
  if ( is >> i )  // se citeste valoarea
    n = i;        // se invoca operatorul de atribuire
  return is;
}
```

Adaptarea programului pentru determinarea termenilor sirului lui Fibonacci necesita doar includerea fisierului `intErval.h`, precum si schimbarea definitiei rangului `n` din `int` in `intErval`.

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

long fib2( int n ) {
  long i = 1, j = 0;

  for ( int k = 0; k++ < n; j = i + j, i = j - i );
  return j;
}

int main( ) {
  cout << "\nTermenul sirului lui Fibonacci de rang ... ";

  intErval n = 47; cin >> n;
  cout << " este " << fib2( n );
  cout << '\n';

  return 0;
}
```

Desigur ca, la programul executabil, se va lega si fisierul rezultat in urma compilarii definitiilor functiilor membre din clasa `intErval`.

Neconcordanta dintre argumentul formal de tip `int` din `fib2()` si argumentul efectiv (actual) de tip `intErval` se rezolva, de catre compilator, prin invocarea operatorului de conversie de la `intErval` la `int`.

Programarea orientata pe obiect este deosebit de avantajoasa in cazul aplicatiilor mari, dezvoltate de echipe intregi de programatori pe parcursul catorva luni, sau chiar ani. Aplicatia prezentata aici este mult prea mica pentru a putea fi folosita ca un argument in favoarea acestei tehnici de programare. Cu toate acestea, comparand cele doua implementari ale clasei `intErval` (in limbajele C, respectiv C++), sunt deja evidente doua avantaje ale programarii orientate pe obiect:

- In primul rand, este posibilitatea dezvoltarii unor tipuri noi, definite exclusiv prin comportament si nu prin structura. Codul sursa este mai compact, dar in nici un caz mai rapid decat in situatia in care nu am fi folosit obiecte. Sa retinem ca programarea orientata pe obiect nu este o modalitate de a micsora timpul de executie, ci de a spori eficienta activitatii de programare.
- In al doilea rand, se remarca posibilitatile de a supraincarca operatori, inclusiv pe cei de conversie. Efectul este foarte spectaculos, deoarece utilizarea noilor tipuri este la fel de comoda ca si utilizarea tipurilor predefinite. Pentru tipul `intErval`, aceste avantaje se concretizeaza in faptul ca obiectele de tip `intErval` se comporta exact ca si cele de tip `int`, incadrarea lor in limitele domeniului admisibil fiind absolut garantata.

2.4 Exercitii *

2.1 Scrieti un program care determina termenul de rang *n* al sirului lui Fibonacci prin algoritmii *fib*1 si *fib*3.

2.2 Care sunt valorile maxime ale lui *n* pentru care algoritmii *fib*1, *fib*2 si *fib*3 returneaza valori corecte? Cum pot fi marite aceste valori?

Solutie: Presupunand ca un `long` este reprezentat pe 4 octeti, atunci cel mai mare numar Fibonacci reprezentabil pe `long` este cel cu rangul 46. Lucrand pe `unsigned long`, se poate ajunge pana la termenul de rang 47. Pentru aceste ranguri, timpii de executie ai algoritmului *fib*1 difera semnificativ de cei ai algoritmilor *fib*2 si *fib*3.

2.3 Introduceti in clasa `intErval` inca doua date membre prin care sa contorizati numarul de apeluri ale celor doi operatori definiti. Completati

* Chiar daca nu se precizeaza explicit, toate implementarile se vor realiza in limbajul C++.

constructorul si destructorul astfel incat sa initializeze, respectiv sa afiseze, aceste valori.

2.4 Implementati testul de primalitate al lui Wilson prezentat in Sectiunea 1.4.

2.5 Scrieti un program pentru calculul recursiv al coeficientilor binomiali dupa formula data de triunghiul lui Pascal:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{pentru } 0 < k < n \\ 1 & \text{altfel} \end{cases}$$

Analizati avantajele si dezavantajele acestui program in raport cu programul care calculeaza coeficientul conform definitiei:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

Solutie: Utilizarea definitiei pentru calculul combinarilor este o idee total neinspirata, nu numai in ceea ce priveste eficienta, ci si pentru faptul ca nu poate fi aplicata decat pentru valori foarte mici ale lui *n*. De exemplu, intr-un long de 4 octeti, valoarea 13! nu mai poate fi calculata. Functia recursiva este simpla:

```
int C( int n, int m) {
  return m == 0 ||
         m == n?   1: C( n - 1, m - 1 ) + C( n - 1, m );
}
```

dar si ineficienta, deoarece numarul apelurilor recursive este foarte mare (vezi Exercitiul 8.1). Programul complet este:

```
#include <iostream.h>

const int N = 16, M = 17;

int r[N][M];  // contorizeaza numarul de apeluri ale
              //   functiei C( int, int ) separat,
              //   pentru toate valorile argumentelor

long tr;      // numarul total de apeluri ale
              //   functiei C( int, int )
```

```
int C( int n, int m ) {
  r[n][m]++; tr++;
  return m == 0 || m == n?
    1: C( n - 1, m - 1 ) + C( n - 1, m );
}

void main( ) {
  int n, m;
  for ( n = 0; n < N; n++ )
    for ( m = 0; m < M; m++ ) r[n][m] = 0;
  tr = 0;

  cout << "\nCombinari de (maxim " << N << ") ... ";
  cin >> n;
  cout << "             luate cate ... ";
  cin >> m;
  cout << "sunt " << C( n, m ) << '\n';

  cout << "\n\nC( int, int ) a fost invocata de "
       << tr << " ori astfel:\n";
  for ( int i = 1; i <= n; i++, cout << '\n' )
    for ( int j = 0; j <= i; j++ ) {
      cout.width( 4 );
      cout << r[i][j] << ' ';
    }
}
```

Rezultatele obtinute in urma rularii sunt urmatoarele:

```
Combinari de (maxim 16) ...12
             luate cate ...7
sunt 792


C( int, int ) a fost invocata de 1583 ori astfel:
 210  210
  84  210  126
  28   84  126   70
   7   28   56   70   35
   1    7   21   35   35   15
   0    1    6   15   20   15    5
   0    0    1    5   10   10    5    1
   0    0    0    1    4    6    4    1    0
   0    0    0    0    1    3    3    1    0    0
   0    0    0    0    0    1    2    1    0    0    0
   0    0    0    0    0    0    1    1    0    0    0    0
   0    0    0    0    0    0    0    1    0    0    0  ...
```

Se observa ca `C(1,1)` a fost invocata de 210 ori, iar `C(2,2)` de 126 de ori!