**Programare orientată pe obiecte**

**Suport de curs redactat de lector dr. Mihail Cherciu**

**Universitatea din Bucureşti, Facultatea de Matematică şi Informatică, Catedra de Fundamentele Informaticii**

**10 noiembrie 2008**

## Cuprins

**Introducere**

**1 O trecere în revistă a tehnicilor de programare**

## 1.1 Programare nestructurată

**1.2 Programare procedurală**

**1.3 Programare modulară**

**1.4 Un exemplu cu structuri de date**

### **1.4.1 Lucrul cu liste simplu înlănţuite**

### **1.4.2 Lucrul cu mai multe liste**

**1.5 Probleme specifice programării modulare**

### **1.5.1 Creare şi distrugere explicite**

### **1.5.2 Date şi operaţii decuplate**

### **1.5.3 Lipsa precizării tipurilor**

### **1.5.4 Strategii şi reprezentare**

**1.6 Programare orientată pe obiecte**

**1.7 Exerciţii**

**2 Tipuri abstracte de date**

## 2.1 Tratarea problemelor

## 2.2 Proprietăţi ale tipurilor abstracte de date

## 2.3 Tipuri abstracte de date generice

## 2.4 Notaţii

## 2.5 Tipuri abstracte de date şi orientarea pe obiecte

## 2.6 Exerciţii

**3 Concepte ale orientării pe obiecte**

## 3.1 Implementarea tipurilor abstracte de date

## 3.2 Clase

## 3.3 Obiecte

## 3.4 Mesaje

## 3.5 Rezumat

## 3.6 Exerciţii

**4 Alte concepte ale orientării pe obiecte**

## 4.1 Relaţii

### **4.1.1 Relaţia "un fel de"**

### **4.1.2 Relatia "este un/o"**

### **4.1.3 Relaţia "parte din"**

### **4.1.4 Relaţia "are un/o"**

## 4.2 Moştenire

## 4.3 Moştenire multiplă

## 4.4 Clase abstracte

## 4.5 Exerciţii

**5 Din nou concepte ale orientării pe obiecte**

## 5.1 Tipuri generice

## 5.2 Legare statică şi dinamică

## 5.3 Polimorfism

**6 Implementarea conceptelor orientate pe obiecte în limbaje de programare**

## 6.1 Programare orientată pe obiecte în TurboPascal

## 6.2 C++, limbaj creat pentru programarea orientată pe obiecte

### **6.2.1 Limbajul de programare C**

**6.2.1.1 Tipuri de date**

**6.2.1.2 Instrucţiuni**

**6.2.1.3 Operatori şi expresii**

**6.2.1.4 Funcţii**

**6.2.1.5 Pointeri şi tablouri**

**6.2.1.6 Pointeri la funcţii**

**6.2.1.7 Un prim program**

### **6.2.2 De la C la C++**

**6.2.2.1 Incompatibilităţi între limbajele C şi C++**

**6.2.2.2 Extensii de bază**

**6.2.2.2.1 Tipuri de date**

**6.2.2.2.2 Funcţii**

**6.2.2.2.3 Alte extensii de bază**

**6.2.2.3 Extensii orientate pe obiecte**

**6.2.2.3.1 Clase şi obiecte**

**6.2.2.3.2 Constructori**

**6.2.2.3.3 Destructori**

**6.2.2.3.4 Moştenire**

**6.2.2.3.4.1 Tipuri de moştenire**

**6.2.2.3.4.2 Construcţie**

**6.2.2.3.4.3 Distrugere**

**6.2.2.3.4.4 Moştenire multiplă**

**6.2.2.3. 5 Polimorfism**

**6.2.2.3. 6 Clase abstracte**

**6.2.2.3. 7 Supraîncărcarea operatorilor**

**6.2.2.3. 8 Prieteni**

### **6.2. 3 Cum se scrie un program**

### **6.2.4 Exerciţii**

**7 Rezolvările exerciţiilor**

## 7.1 O trecere în revistă a tehnicilor de programare

## 7.2 Tipuri abstracte de date

## 7.3 Concepte ale orientării pe obiecte

## 7.4 Alte concepte ale orientării pe obiecte

## 7.5 Mai mult despre C++

**Bibliografie**

**Introducere**

Orientarea pe obiecte (OO) este o tehnologie de proiectare şi programare bazată pe un sistem de concepte informatice apropiat felului în care este percepută şi înţeleasă în mod obişnuit lumea reală. Această abordare s-a dovedit a fi deosebit de fructuoasă pentru elaborarea unor produse informatice performante şi uşor de învăţat şi utilizat. Această tehnologie este de asemenea foarte adecvată pentru elaborarea mai uşoară a versiunilor succesive şi ale diferitelor variante ale unui produs informatic, utilizarea ei având ca rezultat o creştere semnificativă a eficienţei activităţii de dezvoltare de software.

Caracteristică acestei metodologii de dezvoltare este analiza prealabilă a problemelor puse de o anumită aplicaţie, activitate urmărind elaborarea, printr-un proces de abstractizare, a unor tipuri abstracte de date, înglobând date şi operaţii asupra datelor. Ulterior aceste tipuri abstracte de date se implementează în limbaje de programare adecvate.

Capitolul 1 prezintă pe scurt evoluţia tehnicilor de programare, scopul fiind acela de a se reliefa diferenţele esenţiale dintre programarea OO şi celelalte tehnici de programare. În capitolul 2 sunt definite tipurile abstracte de date şi sunt expuse ideile de bază legate de această noţiune. Capitolele 3-5 prezintă in mod sistematic conceptele orientării pe obiecte şi problemele ridicate de implementarea acestor concepte. Capitolul 6, cel mai amplu şi cu structura cea mai complexă, este dedicat prezentării a două limbaje în care sunt implementare conceptele OO: TurboPascal şi C++. TurboPascal este tratat pe scurt doar în secţiunea 6.1, restul capitolului fiind ocupat cu prezentarea limbajului C++, creat special pentru OO.

Toate capitolele de mai sus conţin exerciţii. Rezolvările exerciţiilor sunt date în capitolul 7, ultimul al prezentului curs. Cursul conţine şi o bibliografie cu 5 titluri.

Expunerea este bazată în mare măsură pe prezentarea şi tratarea amănunţită a unor exemple. Secţiunea 6.2.5 prezintă în întregime un proiect privind lucrul cu liste simplu înlanţuite.

Cursul urmează cu unele modificări şi adaptări linia lucrării [web2], publicată pe Internet, care este un curs pe care îl găsim foarte adecvat scopului nostru. De asemenea, prezentul curs este apropiat spiritului lucrării [6]. Deşi prezintă multe elemente ale limbajelor de programare discutate, mai ales C şi C++, cursul de faţă nu poate suplini manualele pentru aceste limbaje. In consecinţă recomandăm consultarea de astfel de manuale, dintre care în bibliografie sunt trecute [1], [2], [3], [4], [5] şi [web4].

**1 O trecere în revistă a tehnicilor de programare**

Acest capitol este o scurtă trecere în revistă a tehnicilor de programare. Utilizăm un exemplu simplu pentru a ilustra proprietăţile lor particulare şi a indica ideile şi problemele cele mai importante care apar în legătură cu ele.

În linii mari, cineva care învaţă să programeze parcurge următoarele etape:

 Programare nestructurată,

 Programare procedurală,

 Programare modulară şi

 Programare orientată pe obiecte.

Prezentul capitol este organizat după cum urmează. Secţiunile 1.1, 1.2, 1.3 descriu pe scurt primele trei tehnici de programare. Prezentăm apoi un exemplu simplu de utilizare a programării modulare pentru a implementa un modul pentru liste simplu înlănţuite (secţiunea 1.4). Legat de acesta, enunţăm câteva probleme privind acest tip de tehnică în secţiunea 1.5. În final, secţiunea 1.6 descrie cea de-a patra tehnică de programare.

## 1.1 Programare nestructurată

De obicei, cei care învaţă să programeze încep cu scrierea unor programe mici şi simple constând numai dintr-un program principal. Aici, prin "program principal" se înţelege o secvenţă de comenzi, sau *instrucţiuni* care modifică date care sunt *globale* în întregul program. Putem ilustra aceasta ca în Figura 1.1.

program principal

*date*

program

Figura 1.1

După cum desigur ştiţi, această tehnică de programare are dezavantaje considerabile dacă programele devin destul de mari. De exemplu, dacă aceeaşi secvenţă de instrucţiuni este necesară în diferite locuri ale programului, secvenţa trebuie copiată. Aceasta a dus la ideea de a se extrage aceste secvenţe să li se dea nume şi să se ofere o tehnică pentru a le apela şi a se reveni din aceste *proceduri.*

**1.2 Programare procedurală**

Cu programarea procedurală putem să grupăm secvenţele de instrucţiuni folosite de mai multe ori într-un singur loc. O *apelare de procedură* este utilizată pentru a se invoca procedura.

După ce secvenţa este prelucrată, execuţia programului continuă exact după poziţia în care a fost făcută apelarea (Figura 1.2).

**Program principal**

**Procedură**

Figura 1.2

Prin introducerea *argumentelor* şi a procedurilor de proceduri ( *subproceduri* ) programele pot fi scrise mai structurat şi cu risc mai mic de eroare, deoarece, dacă o procedură este corectă, de fiecare dată când este utilizată produce rezultate corecte. Prin urmare, în caz de eroare, căutarea acesteia se poate reduce la acele locuri despre care nu se ştie dacă sunt corecte.

Un program poate să fie acum considerat ca o secvenţă de apelări de procedură. Programul principal trebuie să transfere datele la fiecare apelare, datele sunt prelucrate de proceduri şi, când programul se termină, datele care rezultă sunt furnizate. Astfel, *fluxul de date* poate să fie ilustrat ca un graf ierarhic (arbore) ca în Figura 1.3 (în care procedurile nu au subproceduri).

**program principal**

*date*

procedura1

procedura2

procedura3

program

Figura 1.3

În rezumat: Avem acum un singur program care este împărţit în părţi mai mici numite proceduri. Pentru a face posibiliă utilizarea procedurilor generale sau a grupurilor de proceduri şi în alte programe, ele trebuie să fie disponibile separat. Din acest motiv s-a conceput programarea modulară care permite gruparea procedurilor în *modul*e.

**1.3 Programare modulară**

În programarea modulară proceduri care realizează împreună o anumită funcţionalitate sunt grupate în module *separate.*

Un program nu mai constă deci dintr-o singură parte. El este constituit acum din mai multe părţi mai mici care interacţionează prin intermediul apelărilor de procedură (Figura 1.4).

program

**program principal**

*date*

modul1

*date*+*date*1

modul2

*date*+*date*2

procedura1

procedura2

procedura3

Figura 1.4

Fiecare modul poate să aibă propriile sale date şi apare cel mult o dată în întregul program. Aceasta permite fiecărui modul să gestioneze o *stare* internă care este modificată prin apelări ale procedurilor acestui modul. Pe parcursul executării programului în fiecare moment fiecare modul are numai o singură stare.

**1.4 Un exemplu cu structuri de date**

Programele utilizează structurile de date pentru a memora datele. Există mai multe structuri de date, de exemplu liste, arbori, tablouri, stive, cozi etc. Fiecare din aceste structuri de date pot să fie caracterizate prin structură şi metode de acces.

### **1.4.1 Lucrul cu liste simplu înlănţuite**

Bine cunoscute şi destul de mult utilizate sunt listele simplu înlănţuite care utilizează o structură foarte simplă, constând din elemente care sunt legate succesiv unul de altul, ca în Figura 1.5.

Figura 1.5

Listele simplu înlănţuite furnizează metode de acces pentru a *adăuga* un nou element şi pentru a *şerge* elementul din faţă. Structurile de date mai complicate pot să utilizeze unele structuri deja existente. De exemplu, o coadă poate să fie structurată ca o listă simplu înlănţuită. Cozile furnizează metode de acces pentru a *adăuga* un element la sfîrşit şi de a *obţine* primul element (comportare numită în engleză "first-in first-aut", prescurtat FIFO).

Vom prezenta acum un exemplu pe care îl utilizăm pentru expunerea unor concepte de proiectare. Deoarece acest exemplu este utilizat numai pentrua ilustra aceste concepte şi probleme, el nu este nici complet, nici optim. În capitolul 6, secţiunea 6.2.5 se va face o discuţie orientată pe obiecte completă având ca scop proiectarea structurilor de date.

Să presupunem că dorim să programăm o listă într-un limbaj de programare ca C sau Modula-2. Deoarece considerăm că lista este o structură de date frecvent utilizată, decidem să o implementăm într-un *modul* separat. În mod normal, aceasta ne obligă să scriem două fişiere: *definiţia interfeţei* şi *fişierul de implementare.* În acest capitol vom utiliza un pseudocod simplu care poate fi înţeles fără dificultate. Vom utiliza comentarii cuprinse în "/\*...\*/". Definiţia interfeţei ar putea arăta ca mai jos.

/\*

\* Definitia interfetei pentru un modul care implementeaza

\* o lista simplu inlantuita pentru a memora date de orice tip

\*/

MODULE Lista-Simplu\_Inlantuita-1

BOOL list\_initialize();

BOOL list\_append(ANY data);

BOOL list\_delete();

list\_end();

ANY list\_getFirst();

ANY list\_getNext();

BOOL list\_isEmpty();

END Lista-Simplu\_Inlantuita-1

Declarările din interfaţă descriu numai *ce* parte a informaţiei este disponibilă şi **nu** *cum* această parte devine disponibilă. Se *ascunde* informaţia privind implementarea în fişierul de implementare. Aplicăm astfel un principiu fundamental în ingineria programării care constă din *ascunderea* informaţiei despre implementarea efectivă. Aceasta ne dă posibilitatea să schimbăm implementarea, de exemplu să utilizăm un algoritm mai rapid, dar care consumă mai multă memorie, fără să apară necesitatea de a schimba alte module ale programului întrucât apelările de proceduri rămân aceleaşi. Este de asemenea posibil ca declarările din interfaţă să fie făcute într-un limbaj de programare, iar implementarea să fie făcută într-un alt limbaj de programare, cu condiţia să existe o anumită compatibilitate între cele două limbaje. De exemplu, în mediul vizual de programare Delphi interfaţa şi implementarea s-au făcut într-o extensie a limbajului Pascal, iar în mediul de programare C++Builder s-a păstrat implementarea din Delphi, dar s-a creat o interfaţă scrisă într-o extensie a limbajului C++.

Revenind la interfaţa propusă pentru liste, observăm că ideea acestei interfeţe este următoarea: înainte de utilizarea listei trebuie să se apeleze *list\_initialize()* pentru a se iniţializa variabilele locale din modul. Următoarele două proceduri implementează operaţiile de accesare şi modificare a structurii arătate mai sus: *append* (pentru adăugare) şi *delete* (pentru ştergere). Se observă că toate procedurile din interfaţă întorc valori al căror tip este specificat înaintea antetului declarării de procedură, deci aceste proceduri pot fi utilizate ca *funcţii*. Tipul BOOL întors de unele dintre dintre ele are două valori şi este introdus pentru a modela valorile de adevăr TRUE şi FALSE, iar întoarcerea unei valori de acest tip de o funcţie are semnificaţia terminării cu succes sau cu eşec a execuţiei funcţiei. Procedura *append* necesită o discuţie mai detaliată. Funcţia *list\_append()* are un argument *date* de un tip arbitrar. Acest lucru este necesar deoarece dorim să utilizăm lista în mai multe contexte diferite, deci tipul elementelor care se introduc în listă nu este cunoscut dinainte. Prin urmare, trebuie să utilizăm un tip special ANY care ne permite să îi ataşăm orice tip de date (nu orice limbaj de programare are un astfel de tip; în C acesta poate fi introdus cu *pointeri*). A treia procedură *list\_end()* trebuie să fie apelată când programul încetează, pentru a permite modulului să şteargă conţinutul variabilelor sale interne. De exemplu putem dori să eliberăm memoria alocată.

Cu următoarele proceduri *list\_getFfirst()* şi *list\_getNext()* este oferit un mecanism simplu de traversare prin listă. Traversarea poate fi făcută utilizându-se următoarea ciclare:

ANY data;

data <- list\_getFirst();

WHILE data IS VALID DO

faceCeva(data);

data <- list\_getNext();

END

Avem acum un modul pentru liste care ne permite să utilizăm o listă cu orice tip al elementelor. Ce s-ar întâmpla însă dacă am avea nevoie de mai mult de o listă într-unul din programe?

### **1.4.2 Lucrul cu mai multe liste**

Decidem să reproiectăm modulul nostru pentru liste pentru a putea să gestioneze mai mult decât o listă. Prin urmare creăm o nouă descriere a interfeţei care include acum o definiţie pentru un tip numit *reprezentantă de listă*. Această reprezentantă este utilizată în orice procedură dată pentru a identifica în mod unic lista cu care se lucrează. Definiţia interfeţei pentru acest nou modul arată aşa:

/\*

\* Un modul de liste pentru mai mult de o lista.

\*/

MODULE Lista-Simplu-Inlantuita-2

DECLARE TYPE reprez\_lista\_t;

reprez\_lista\_t list\_create();

list\_destroy(reprez\_lista\_t this);

BOOL list\_append(reprez\_lista\_t this, ANY data);

ANY list\_getFirst(reprez\_lista\_t this);

ANY list\_getNext(reprez\_lista\_t this);

BOOL list\_isEmpty(reprez\_lista\_t this);

END Lista-Simplu-Inlantuita-2;

Utilizăm *DECLARE TYPE* pentru a introduce noul tip pe care îl numim *reprez\_lista\_t.* Nu specificăm cum este reprezentat şi implementat în mod efectiv acest tip. Şi de această dată *ascundem* detaliile de implementare în fişierul de implementare. Se observă diferenţa faţă de versiunea precedentă, în care ascundeam numai funcţii, respectiv proceduri. Acum ascundem şi informaţiile pentru un tip de date definit de utilizator, numit *reprez\_lista\_t*. Utilizăm *list\_create()* pentru a obţine reprezentanta unei noi liste (vidă). Fiecare procedură conţine acum argumentul special *this* care identifică chiar lista respectivă. Toate procedurile operează acum asupra acestei reprezentante în loc să opereze asupra unei liste globale a modulului. Am putea spune acum că putem crea *obiecte listă.* Fiecare astfel de obiect poate fi identificat în mod unic prin reprezentanta sa şi sunt aplicabile numai acele *metode* care sunt definite pentru a opera asupra acestei reprezentante.

## 1.5 Probleme specifice programării modulare

Secţiunea precedentă arată că deja programăm având în minte unele concepte orientate pe obiecte. Exemplul listelor care a fost expus evidenţiază anumite probleme specifice pe are le vom prezenta acum.

**1.5.1 Creare şi distrugere explicite**

În exemplu, de câte ori dorim să utilizăm o listă, trebuie să declarăm explicit o reprezentantă şi să executăm o apelare a *list\_create()* pentru a obţine una validă. După utilizarea listei trebuie să apelăm în mod explicit *list\_destroy()* pentru reprezentanta listei pe care dorim să o distrugem. Dacă dorim să utilizăm o listă în interiorul unei proceduri, să-i spunem foo(), trebuie s-o facem în următorul cadru:

PROCEDURE foo()

BEGIN

reprez\_lista\_t lista\_mea;

lista\_mea <- list\_create();

/\* Se face ceva cu lista\_mea \*/

...

list\_destroy(lista\_mea);

END

Să comparăm lista cu alte tipuri de date, de exemplu cu un întreg. Întregii sunt declaraţi în interiorul unui domeniu particular (de exemplu, în interiorul unei proceduri). Odată ce au fost definiţi, ei pot fi utilizaţi. Dacă domeniul este părăsit (de exemplu procedura unde întregul a fost definit), întregul este pierdut. El este creat şi distrus în mod automat. Unele compilatoare chiar iniţializează întregii nou creaţi cu o anumită valoare, de obicei 0 (zero). Care este diferenţa faţă de "obiectele" listă? Timpul de viaţă al unei liste este şi el definit de domeniul său, deci ea poate fi creată de îndată ce se intră în domeniu şi poate fi distrusă de îndată ce el este părăsit. La momentul creării o listă trebuie iniţializată ca vidă. Prin urmare, am dori să putem defini o listă ca şi pe un întreg. Un cadru pentru aceasta ar putea arăta aşa:

PROCEDURE foo() BEGIN

reprez\_lista\_t lista\_mea; /\* Lista este creata si

initializata \*/

/\* Se face ceva cu lista\_mea \*/

...

END /\* lista\_mea este distrusa \*/

Avantajul este că acum compilatorul se ocupă de apelările procedurilor de iniţializare şi terminare, după cum este necesar. De exemplu, aceasta asigură că lista este ştearsă corect, resursele fiind redate programului.

**1.5.2 Date şi operaţii decuplate**

Decuplarea datelor şi operaţiilor duce în mod obişnuit la o structură bazată pe operaţii mai degrabă decât pe date: modulele grupează împreună operaţiile des folosite (cum sunt operaţiile *list\_...()* ). Utilizăm

aceste operaţii furnizându-le explicit datele asupra cărora ele trebuie să opereze. Structura rezultată pentrru module este deci orientată spre operaţii, mai degrabă decât spre datele efective. Se poate spune că operaţiile definite specifică datele care să fie utilizate. În orientarea pe obiecte, structura este caracterizată de către date. Alegem reprezentările de date care corespund cel mai bine cerinţelor noastre. În consecinţă, programele devin structurate prin date, mai degrabă decât prin operaţii. Astfel, se urmează exact cealaltă cale: datele specifică operaţiile valide. Acum modulele grupează împreună reprezentările datelor.

**1.5.3 Lipsa precizării tipurilor**

În exemplul nostru privind listele trebuie să utilizăm tipul special *ANY* pentru a permite unei liste să conţină datele pe care le dorim. Aceasta implică faptul că tipul utilizat efectiv nu este stabilit de compilator, deci corectitudinea utilzării tipului nu poate fi verificată în cursul procesului de compilare. Să considerăm următorul exemplu în care compilatorul nu poate să verifice corectitudinea:

PROCEDURE foo() BEGIN

UnTipdeDate data1;

UnAltTipdeDate data2;

reprez\_lista\_t lista\_mea;

lista\_mea <- list\_create();

list\_append(lista\_mea, data1);

list\_append(lista\_mea, data2); /\* Surpriza! \*/

...

list\_destroy(lista\_mea);

END

Este responsabilitatea noastră de a asigura că lista este utilizată în mod corect. O posibilă soluţie este să se adauge informaţii suplimentare despre tipul fiecărui element al listei. Totuşi, aceasta implică mai mult efort şi obligă programatorul să cunoasca multe detalii despre datele cu care se lucrează.

Ar fi de dorit să avem un mecanism care să ne permită să specificăm cu ce date va fi definită lista. Funcţia principală a listei este mereu aceeaşi, indiferent dacă ea conţine date privind persoane, numere, maşini, sau chiar liste. Ar fi deci convenabil să declarăm o nouă listă cam aşa:

reprez\_lista\_t<Persoana> lista1; /\* o lista de persoane \*/

reprez\_lista\_t<Masina> lista2; /\* o lista de masini \*/

Rutinele corespunzătoare pentru liste ar întoarce automat tipurile corecte de date. Compilatorul ar putea să verifice automat corectitudinea.

**1.5.4 Strategii şi reprezentare**

Exemplul cu privire la liste implică operaţii pentru traversarea listei. În mod normal, pentru acest scop este definită o variabilă numită *cursor* care indică *elementul curent.* Aceasta implică o *strategie de traversare* care defineşte ordinea în care elementele structurii de date urmează să fie vizitate.

Pentru o structură de date simplă cum este cea de listă simplu înlănţuită putem concepe doar o singură strategie de traversare. Plecând de la cel mai din stânga element se vizitează succesiv vecinii din dreapta până când se ajunge la ultimul element. Totuşi, structuri de date mai complexe, cum sunt de exemplu arborii, pot fi traversate utilizând mai multe strategii. Lucrurile pot fi chiar mai complicate, uneori strategiile de traversare depinzând de contextul particular în care o structură de date este utilizată. În consecinţă, are sens să se separe reprezentarea efectivă sau forma structurii de date de strategia sa de traversare. Vom examina acest fapt mai în detaliu în capitolul 6, secţiunea 6.2.5.

Ceea ce am arătat cu privire la strategia de traversare este valabil şi pentru alte feluri de strategii. De exemplu, inserţia poate fi făcută astfel încât să existe sau nu o ordonare a elementelor.

## 1.6 Programare orientată pe obiecte

Programarea orientată pe obiecte rezolvă unele din problemele menţionate. Spre deosebire de alte tehnici, avem acum o reţea de *obiecte* care interacţionează, fiecare având starea sa proprie. (Figura 1.6)

program

obiect1

*date*

obiect4

*date*

obiect3

*date*

obiect2

*date*

Figura 1.6

Să considerăm din nou exemplul privitor la liste. Problema pusă de programarea modulară este că trebuie să se creeze şi să se distrugă în mod explicit reprezentantele de liste. Apoi se utilizează procedurile modulului pentru modificarea fiecăreia dintre reprezentante.

Spre deosebire de aceasta, în programarea orientată pe obiecte putem avea smultan oricât de multe obiecte listă este necesar. În loc să se apeleze o procedură căreia să-i fie furnizată reprezentanta corectă a unei liste, se va trimite un *mesaj* respectivului obiect listă. În linii mari, se poate spune că fiecare obiect implementează propriul său modul care să permită, de exemplu, ca mai multe liste să coexiste.

Fiecare obiect trebuie să se creeze şi să se distrugă în mod automat. În consecinţă, nu mai este nevoie să se apeleze în mod explicit o procedură de creare şi terminare.

S-ar putea pune întrebarea: "Şi ce-i cu asta? Nu este aceasta doar o tehnică modulară de programare mai sofisticată?". Aşa pare să fie, dacă aceasta ar fi totul în legătură cu orientarea pe obiecte. Din fericire, nu este. Începând cu capitolele următoare vor fi introduse noi caracteristici şi concepte ale orientării pe obiecte, care fac ca programarea orientată pe obiecte să fie cu adevărat o nouă tehnică de programare, care să permită creşterea eficienţei activităţilor de proiectare şi programare a aplicaţiilor.

**1.7 Exerciţii**

1. Exemplul privitor la liste include tipul special *ANY* pentru a putea permite unei liste să conţină date de orice tip. Presupuneţi că doriţi să scrieţi un modul pentru o listă specializată de întregi care oferă verificarea tipului. Tot ceea ce aveţi este definiţia interfeţei modulului *Lista-Simplu-Inlantuita-2.*

(a) Cum arată definiţia interfeţei pentru un modul *Intreg-Lista* ?

(b) Discutaţi problemele care apar prin utilizarea tipului *ANY* pentru elementele listei în modulul *Lista-Simplu-Inlantuita-2.*

(c) Care sunt soluţiile posibile ale acestor probleme?

2. Care sunt principalele diferenţe conceptuale între programarea orientată pe obiecte şi alte tehnici de programare?

**2 Tipuri abstracte de date**

Unii autori prezintă proiectarea şi programarea orientată pe obiecte ca fiind proiectarea şi programarea de *tipuri abstracte de date* şi a relaţiilor dintre ele. În acest capitol prezentăm acest concept fundamental şi investigăm mai detaliat conceptele utilizate în legătură cu exemplul privitor la liste din capitolul precedent.

**2.1 Tratarea problemelor**

Prima activitate pe care trebuie să o desfăşoare o persoană atunci când scrie un program este enunţarea precisă a *problemei* pe care urmează să o rezolve. Cel mai frecvent activitatea de programare are ca scop rezolvarea problemelor din "viaţa reală". Aceste probleme apar de obicei cu aspect nebulos şi primul lucru care trebuie făcut este înţelegerea problemei pentru a se separa elementele necesare de cele inutile. Rezultatul acestui efort este o *prezentare abstractă* sau un *model* al problemei. Acest proces de modelare se numeşte *abstractizare* şi este ilustrat în Figura 2.1.

**Problemă reală**

**Model**

***Abstractizare***

Figura 2.1

Faptul că modelul reprezintă o prezentare abstractă a problemei implică definirea unor *proprietăţi* ale problemei. Aceste proprietăţi includ *date* şi *operaţii*.

Să considerăm, de exemplu, că administraţia unei instituţii doreşte un program pentru gestiunea angajaţilor instituţiei. Apar o serie de întrebări: ce informaţii sunt necesare administraţiei? Ce operaţii trebuie efectuate? Angajaţii sunt persoane reale care pot fi caracterizate de multe proprietăţi, ca de exemplu:

nume,

înălţime,

data naşterii,

sex,

număr de identificare,

numărul camerei unde lucrează,

culoarea părului,

preocupări extraprofesionale, etc.

Cu certitudine, nu toate aceste proprietăţi sunt necesare pentru rezolvarea problemei administraţiei. În consecinţă, trebuie creat un model de angajat pentru această problemă. Acest model conţine numai proprietăţile care sunt necesare pentru a satisface cerinţele administraţiei, de exemplu: nume, data naşterii sex, număr de identificare. Aceste proprietăţi sunt *datele* modelului de angajat. Astfel, o persoană reală va fi descrisă cu ajutorul unui "angajat abstract". Desigur, nu este suficientă numai descrierea. Trebuie definite anumite *operaţii* cu ajutorul cărora administraţia să poată lucra cu "angajaţii abstracţi". De exemplu, trebuie să existe o operaţie care să permită crearea unui nou "angajat abstract" atunci când o nouă persoană este angajată de instituţie. În consecinţă, trebuie identificate operaţiile care să poată fi efectuate cu un angajat abstract. De asemenea, putem decide ca accesul la datele angajaţilor să fie făcut numai prin anumite operaţii asociate. Aceasta ne permite să ne asigurăm că datele sunt în permanenţă într-o stare bună. De exemplu, putem să verificăm dacă o anumită dată este corectă.

În consecinţă, abstractizarea este procesul de structurare a unei probleme nebuloase în entităţi bine precizate prin definirea *datelor* şi *operaţiilor*. Prin urmare, aceste entităţi combină datele şi operaţiile, care nu sunt separate unele de celelalte.

**2.2 Proprietăţi ale tipurilor abstracte de date**

Exemplul din secţiunea precedentă arată că prin abstratizare se crează o entitate bine precizată care să poată fi prelucrată în mod specific. O astfel de entitate defineşte *structura de date* a elementelor unei anumite mulţimi. De exemplu, fiecare angajat are datele: nume, dată de naştere, sex, număr de identificare.

Structura de date poate fi accesată numai cu *operaţii* definite în mod expres. Acest set de operaţii este numit *interfaţă* şi este *exportat* de entitate. O entitate cu proprietăţile descrise mai sus este numită *tip abstract de date* (TAD). Figura 2.2 prezinta un TAD care constă dintr-o stuctură abstractă de date şi operaţii. Numai operaţiile sunt vizibile din exterior şi constituie interfaţa.

tip abstract de date

**structură abstractă de date**

operaţii

*interfaţă*

Figura 2.2

După ce se defineşte un TAD, se pot crea *instanţieri* ale sale prin acordarea de valori datelor din structură. De exemplu, atunci când un nou angajat este "creat", structura de date este încărcată cu valorile efective: se obţine astfel o *instanţiere* a unui angajat abstract. Se pot crea atâtea instanţieri ale unui angajat abstract câte sunt necesare pentru a descrie fiecare persoană reală angajată. Caracteristicile unei persoane reale vor fi tratate astfel într-un mod formalizat.

***Definiţie (Tip abstract de date).*** *Un* ***tip abstract de date*** *(TAD) este caracterizat de următoarele proprietăţi:*

*1. Exportă un* ***tip****.*

*2. Exportă un* ***set de operaţii.*** *Acest set este numit* ***interfaţă****.*

*3. Operaţiile din interfaţă reprezintă* ***unicul*** *mecanism de acces la stuctura de date a tipului.*

*4. Domeniul de aplicabilitate (termen prin care se desemnează atât d****omeniul valorilor*** *tipului, cât şi* ***domeniile de definiţie ale operaţiilor*** *şi* ***rezultatele operaţiilor****) este definit prin axiome şi precondiţii.*

Cu prima proprietate este posibil să se creeze mai mult de o instanţiere a unui TAD, aşa cum s-a văzut în exemplul de mai sus, prin declararea unei instanţieri a tipului. Reamintim şi exemplul anterior referitor la liste. În prima versiune am implementat o listă ca un modul şi eram capabili să utilizăm numai o singură listă la un moment dat. A doua versiune introduce "reprezentanta" ca o referinţă la un "obiect listă". Conform celor arătate acum, reprezentanta, împreună cu operaţiile definite în modulul listei defineşte un TAD *Lista.*

1. Când utilizăm reprezentanta definim variabila corespunzătoare ca fiind de tipul *Lista.*

2. Interfaţa către instanţierile tipului *Lista* este asigurată de către fişierul de definiţie a interfeţei.

3. Deoarece definiţia interfeţei nu include reprezentarea efectivă a reprezentantei, ea nu poate fi modificată direct.

4. Domeniul de aplicabilitate este definit de semnificaţia (semantica) operaţiilor introduse. Axiomele şi precondiţiile includ enunţuri ca:

"O listă vidă este o listă"

"Fie l=(d1, d2, d3,..., dN) o listă. Atunci l.append(dM) are ca rezultat l=(d1, d2, d3,..., dN, dM)."

"Primul element al unei liste poate fi şters numai dacă lista nu este vidă."

Proprietăţile tipului *Lista* sunt rezultatul felului în care noi înţelegem dorim şi putem să utilizăm listele. Este responsabilitatea noastră de a utiliza instanţierile tipului în conformitate cu regulile enunţate.

**Importanţa incapsulării structurilor de date**

Principiul de a ascunde structura de date utilizată şi de a furniza numai o interfaţă bine definită este cunoscut sub numele de *incapsulare.* De ce este atât de important ca structura de date să fie incapsulată? Pentru a răspunde la această întrebare să considerăm următorul exemplu matematic în care dorim să definim un TAD pentru numerele complexe.

În ceea ce urmează este suficient să ştim că numerele complexe constau din două părţi: *partea reală* şi *partea imaginară.* Ambele părţi sunt reprezentate prin numere reale. Pentru numerele complexe sunt definite mai multe operatii: adunarea, scăderea, îmulţirea, împărţirea sunt câteva dintre ele. Axiomele şi precondiţiile sunt definite prin definiţia numărului complex. De exemplu, există element neutru pentru adunare.

Pentru a reprenzenta un număr complex este necesar să se definească structura de date care să fie utilizată de TAD-ul său. Se pot concepe cel puţin două posibilităţi de a face acest lucru:

 Ambele părţi sunt memorate într-un tablou cu două elemente unde primul element indică partea reala si al doilea element indică partea imaginară a numărului complex *c*.Dacă *x* reprezintă partea reală şi *y* partea imaginară, putem să le accesăm prin indicarea de indici ai tabloului: *x=c[0]* şi *y=c[1].*

 Ambele părţi sunt memorate într-o structură cu două componente, reprezentate prin *nume*. Dacă numele componentei reprezentând partea reală este *r* şi cel al părţii imaginare este *i* atunci *x* şi *y* se obţin ca *x=c.r* şi *y=c.i.*

Punctul 3 al definiţiei TAD spune că pentru orice acces la structura de date trebuie să fie definit un operator. Exemplul de mai sus pare să contazică această cerinţă. Este acest lucru adevărat?

Să considerăm din nou cele două posibilităţi de a reprezenta numere complexe şi să ne ocupăm de partea reală.

În prima versiune *x* primeşte valoarea *c[0]*. În cea de a doua versiune x primeşte valoarea *c.r*. În ambele cazuri *x* primeşte ca valoare "ceva". Acest "ceva" diferă de structura efectivă de date utilizată. În ambele cazuri operaţia efectivă "primeşte valoarea" are acelaşi înţeles, şi anume de a declara că *x* este egal cu partea reală a numărului complex *c*: cele două cazuri înglobează aceeaşi semantică.

Dacă ne gândim la operaţii mai complexe impactul decuplării structurii de date de operaţii devine şi mai clar. De exemplu, adunarea a două numere complexe necesită să se efectueze o adunare pentru fiecare parte. Prin urmare este necesar să se acceseze valoarea fiecărei părţi, care este diferită pentru fiecare versiune. Dacă se realizează operaţia "adunare" se pot *incapsula* aceste detalii care rămân separate de utilizarea operaţiei şi invizibile pentru utilizator. În contextul unei aplicaţii, pur şi simplu "se adună două numere complexe", fără a se ţine seama de felul în care această operaţie este definită efectiv.

O dată creat un TAD pentru numere complexe, fie acesta *Complex*, el poate fi utilizat în acelaşi mod ca şi alte tipuri de date, bine cunoscute cum ar fi intregii. În rezumat: separarea dintre structurile de date si operaţii şi constrângerea ca structura de date să fie accesată numai printr-o interfaţă bine definită ne permite să alegem structurile de date cele mai potrivite în contextul unei aplicaţii.

**2.3 Tipuri abstracte de date generice**

TAD-urile sunt utilizate pentru a se defini un nou tip, din care pot fi create instanţieri. Aşa cum s-a arătat în exemplul cu listele, uneori aceste instanţieri pot opera şi ele cu diferite tipuri de date. De exemplu, putem să ne gândim la o listă de persoane sau la o listă de maşini sau chiar la o **listă de liste**. Definiţia semantică a listei este mereu aceeaşi. Numai tipul datelor elementare se schimbă după tipul datelor cu care lista operează.

Această informaţie adiţională ar putea fi specificată ca un **parametru generic** care este specificat în momentul creării instanţierii. Astfel, o instanţiere a unui **TAD generic** este de fapt o instanţiere a unei variante particulare a acestui TAD. O listă de persoane poate fi deci declarată, presupunându-se că s-a definit în prealabil tipul de date **Persoana** reprezentând persoanele:

Lista<Persoana> listaDePersoane;

Parantezele unghiulare cuprind aici tipul de date pentru care trebuie creată o variantă a TAD-ului generic **Lista**. **listaDePersoane** oferă interfaţa ca orice altă listă, dar operează asupra instanţierilor tipului **Persoana**.

**2.4 Notaţii**

Deoarece TAD furnizează un mod abstract de descriere a proprietăţilor unei mulţimi de entităţi, utilizarea lor este independentă de orice limbaj de programare particular. Introducem deci, următoarea notaţie. Fiecare descriere a unui TAD constă din două părţi:

**Date** : Această parte descrie structura de date utilizată în TAD într-un mod neformalizat.

**Operaţii**

Această parte descrie operaţiile valide pentru acest TAD, deci descrie interfaţa sa. Utilizăm operaţiile speciale **constructor** pentru a descrie acţiunile care urmează să fie executate când o entitate a acestui TAD este creată şi **destructor** pentru a descrie acţiunile care urmează să fie executate când o entitate este distrusă. Pentru fiecare operaţie se dau *argumente*, *precondiţii* şi *postcondiţii.*

Prezentăm ca exemplu descrierea unui TAD numit *Intreg*. Fie *k* o expresie întreagă:

**TAD** *Intreg* este

**Date**

O secvenţă de cifre precedată opţional de un semn plus sau minus.

Notăm cu *N* acest număr întreg cu semn.

**Operaţii**

**constructor**

Crează un nou întreg.

**destructor**

Distruge un întreg existent.

**sum(k)**

Crează un nou întreg care este suma lui *N* şi *k*. În consecinţă, *postcondiţia* acestei operaţii este *rez=N+k*.

Aceasta este o relaţie matematică a cărei valoare

este "adevărat" pentru valorile lui *sum*, *N* şi *k*

după ce s-a executat *sum*.

**dif(k)**

Crează un nou întreg, care este diferenţa dintre *N* şi *k*.

Deci postcondiţia acestei operaţii este *rez=N-k.*

**atr(k)**

Atribuie lui *N* valoarea *k*. Postcondiţia acestei operaţii

este *N=k.*

**...**

**Sfârşit**

Descrierea de mai sus este specificaţia pentru TAD *Intreg*. Se observă că am utilizat cuvinte pentru numele operaţiilor, în locul unor simboluri matematice, pentru a nu se produce confuzii între declaraţiile de operaţii (care reprezintă *sintaxa*) şi postcondiţii (care reprezintă *semantica*). Totuşi pentru a face cât mai uşoară citirea specificaţiilor TAD se pot folosi atât cuvinte cât şi simboluri.

Limbajele de programare reale pot folosi o altă implementare pentru un TAD. De exemplu se poate implementa operaţia *add* cu operatorul infixat "+", ceea ce conduce la o reprezentare mai intuitivă pentru adunarea întregilor.

**2.5 Tipuri abstracte de date şi orientarea pe obiecte**

TAD-urile permit crearea de instanţieri cu proprietăţi şi comportare bine definite. În proiectarea şi programarea orientate pe obiecte, TAD-urile se numesc **clase**. Prin urmare, o clasă defineşte proprietăţile **obiectelor** care sunt instanţieri într-un mediu orientat pe obiecte.

Programarea orientată pe obiecte este "programarea cu TAD-uri". Se combină funcţionalitatea a diferite TAD-uri pentru a rezolva o problemă. Prin urmare, instanţierile (obiectele) TAD-urilor (claselor) sunt create, distruse şi utilizate în mod dinamic.

**2.6 Exerciţii**

1. *TAD Intreg*

(a) De ce nu sunt precondiţii pentru operaţiile *sum* şi *dif* ? Evident, descrierea TAD-ului *Intreg* este incompletă. Adăugaţi metodele *inm* (înmulţire), *imp* (împărţire) şi încă una. Descrieţi impactul lor specificând pre- şi postcondiţii.

2. Proiectaţi un TAD *Fractie* care descrie proprietăţile fracţiilor.

(a) Ce structură de date poate fi utilizată?

(b) Cum se prezintă interfaţa?

(c) Prezentaţi câteva axiome şi precondiţii.

3. Descrieţi cu cuvintele proprii proprietăţile tipurilor abstracte de date.

4. De ce este necesar să se includă axiome şi precondiţii în descrierea unui TAD?

5. Descrieţi cu cuvintele proprii relaţiile între:

 instanţiere şi tip abstract de date,

 tip abstract de date şi tipul de date corespunzător,

 instanţieri ale unui tip abstract de date.

**3 Concepte ale orientării pe obiecte**

În continuare vom prezenta conceptele orientate pe obiecte, din care despre unele s-a amintit mai sus, într-un mod mai detaliat, arătând şi denumirile sub care sunt cunoscute în limbajele de programare orientate pe obiecte cunoscute.

**3.1 Implementarea tipurilor abstracte de date**

Programele orientate pe obiecte permit *implemetarea* TAD-urilor. În consecinţă, atunci când un TAD este implementat, avem o reprezentare a sa pe care o putem utiliza.

Să considerăm din nou TAD Intreg. Limbajele de programare ca Pascal, C, Modula-2 şi altele oferă o implementare pentru el, numită adesea *int* sau *integer*. Atunci cănd este creată o variabilă de acest tip, se pot utiliza operaţiile care sunt date pentru ea. De exemplu, se pot aduna doi întregi:

int i, j, k; /\* Se definesc trei intregi \*/

i = 1; /\* Atribuie 1 intregului i \*/

j = 2; /\* Atribuie 2 intregului j \*/

k = i + j; /\* Atribuie suma lui i si j lui k \*/

În fragmentul de cod sursă de mai sus vom evidenţia relaţia cu TAD-ul *Intreg*. Prima linie defineşte trei instanţieri *i*, *j* şi *k* ale tipului *Intreg*. În consecinţă, pentru fiecare instanţiere, operaţia specială *constructor* trebuie apelată. În exemplul nostru, acest lucru este făcut implicit de către compilator. Compilatorul rezervă memorie pentru a păstra valoarea unui întreg şi "leagă" numele corespunzător cu aceasta.Dacă ne referim la *i*, ne referim de fapt la acea zonă de memorie care a fost "construită" de definiţia lui *i* opţional, compilatoarele pot alege să iniţializeze memoria, de exemplu cu 0 (zero). Următoarea linie

i=1;

atribuie lui i valoarea 1. Deci putem să descriem această linie cu ajutorul notaţiei TAD astfel:

Execută operaţia *atr* cu argumentul *1* asupra instanţierii lui *Intreg i*. Aceasta se scrie aşa: i.atr(1)

Avem acum o reprezentare pe două nivele. Primul nivel este nivelul definirii unui TAD, în care se specifică tot ceea se face cu o instanţiere a acestuia prin invocarea operaţiilor definite. La acest nivel, pre- şi postcondiţiile sunt utilizate pentru a descrie ceea ce se efectuează.

În exemplul următor includem aceste condiţii în acolade.

{Precondiţie: *i=n* unde *n* este orice *Intreg* }

*i.atr(1)*

{Postcondiţie: *i=1*}

Reamintim că acum discutăm despre nivelul TAD, în consecinţă, condiţiile sunt formulate cu mijloace matematice.

Nivelul al doilea este nivelul de implementare, în care este aleasă pentru operaţie o *reprezentare* efectivă într-un limbaj de programare.

În limbajul C semnul "=" (egal) implementează operaţia *atr()*. În limbajul Pascal a fost următoarea reprezentare: *i:=1*; În ambele cazuri este implementată operaţia *atr*. Să considerăm linia

k=i+j;

Evident "+" a fost ales pentru a implemeta operaţia *sum*. Putem citi partea "*i+j*" ca "adună valoarea lui *j* la valoarea lui *i*", deci, la nivelul TAD se obţine:

{Precondiţie: Fie *i=n1* şi *j=n2* cu *n1, n2 Întregi* particulari }

i.sum(j)

{Postcondiţie:*i=n1* şi *j=n2* }

Postcondiţia asigură că *i* şi *j* nu îşi schimbă valorile. Reamintim specificaţia lui *sum*. Ea arată că **un nou** *Intreg* este creat, a cărui valoare este suma. Prin urmare, trebuie să dăm un mecanism pentru a accesa această nouă instanţiere. Facem aceasta cu operaţia *atr* aplicată instanţierii *k*.

{Precondiţie:Fie *k=n* unde *n* este un *Intreg* oarecare}

k.atr(i.sum(j)*)*

{Postcondiţie:*k=i+j* }

După cum se vede, unele limbaje de programare aleg o reprezentare care este aproape identică cu formularea matematică utilizată în pre- şi postcondiţii. Acest fapt face dificilă departajarea între cele două nivele.

**3.2 Clase**

O clasă este o *reprezentare* efectivă a unui TAD. Ea conţine detalii de implementare pentru structurile de date şi operaţiile utilizate. Lucrăm cu TAD-ul *Intreg* şi proiectăm propria noastră clasă pentru el:

class Intreg {

attributes:

int i

methods:

setValue(int n)

Intreg addValue(Intreg j)

}

În exemplul de mai sus, ca şi în următoarele utilizăm o notaţie care nu aparţine unui anumit limbaj de programare (este un pseudolimbaj). În această notaţie **class{...}** reprezintă definiţia unei clase. Între acolade se află două secţiuni, una fiind **attributes** şi alta **methods** care definesc implementarea structurilor de date şi ale operaţiilor respectivului TAD. Distingem din nou între cele două nivele utilizând termeni diferiţi: la nivelul de implementare vorbim despre "atribute" care sunt elementele structurii de date de la nivelul TAD. Aceasta se aplică şi la "metode" care sunt implementări ale operaţiilor TAD. În exemplul nostru, structura de date constă dintr-un singur element: o secvenţă de cifre cu semn. Atributul corespunzător este un întreg obişnuit al unui limbaj de programare. Definim numai două metode s*etValue()* şi *addValue()* reprezentând cele două operaţii *atr* şi *sum* respectiv.

***Definiţie (Clasă)****. O* ***clasă*** *este implementarea unui tip abstract de date (TAD). Ea defineşte* ***atributele*** *şi* ***metodele*** *care implementează structurile de date şi operaţiile din TAD, respectiv.*

Instanţierile claselor se numesc *obiecte*. În consecinţă, clasele definesc proprietăţile şi comportarea unei mulţimi de obiecte.

Conceperea de TAD-uri şi declararea, într-un limbaj de programare, a claselor ce le sunt asociate formează obiectul **proiectării**, iar conceperea algoritmilor pentru realizarea operaţiilor şi implementarea acestora prin definirea metodelor în limbajul de programare formează obiectul **programării** orientate pe obiecte.

O trăsătură caracteristică a orientării pe obiecte este ponderea mare pe care o are faza de proiectare în realizarea unei aplicaţii.

**3.3 Obiecte**

Reamintim exemplul cu angajaţii de la secţiunea 2.1. Am vorbit acolo despre *instanţieri* ale unor *angajaţi abstracţi.* Aceste instanţieri sunt exemple efective de angajat abstract deci ele conţin valori efective care reprezintă un anume angajat. Numim aceste instanţieri *obiecte*.

Obiectele sunt identificabile în mod unic printr-un *nume*. Prin urmare, putem avea două obiecte distincte cu acelaşi set de valori. Acest fapt este întâlnit în limbajele de programare "tradiţionale" unde putem avea, de exemplu, doi întregi *i* şi *j* cu aceeaşi valoare "*2*".

Observăm utilizarea lui "*i*" şi "*j*" în ultima propoziţie ca nume pentru doi întregi. Vom numi totalitatea valorilor la un moment dat ale atributelor ataşate unui obiect *starea* obiectului.

***Definiţie (Obiect).*** *Un* ***obiect*** *este o instanţiere a unei clase. El poate fi identificat în mod unic prin* ***numele*** *său şi în fiecare moment are o* ***stare*** *care reprezintă valorile atributelor sale în acel moment*.

Starea unui obiect se schimbă în funcţie de metodele care sunt aplicate asupra sa. Numim această posibilă secvenţă de schimbari ale stării *comportarea* *obiectului*.

***Definiţie (Comportare). Comportarea*** *unui obiect este definită de mulţimea metodelor care pot fi aplicate asupra sa.*

Am introdus până acum două concepte principale ale orientării pe obiecte: clasă si obiect. Programarea orientată pe obiecte (P00) este deci implementarea tipurilor abstracte de date sau, mai simplu, scrierea de clase. În momentul execuţiei instanţierile acestor clase, obiectele, realizează scopul programului schimbându-şi stările. În consecinţă, se poate gândi la un program în curs de execuţie ca la o colecţie de obiecte. Apare întrebarea: cum *reacţionează* aceste obiecte? Apare necesitatea conceptului de *mesaj* care este prezentat în continuare.

**3.4 Mesaje**

Un program aflat în execuţie este o colecţie de obiecte în care obiectele sunt create, distruse şi *interacţionează* între ele. Această interacţiune este bazată pe *mesaje* care sunt trimise de la un obiect la altul, prin care obiectul emiţător cere obiectului receptor să aplice o metodă asupra sa. Pentru a înţelege această comunicare, vom considera din nou clasa *Intreg* prezentată mai sus. În pseudo-limbajul de programare pe care îl utilizăm putem să creăm noi obiecte şi să aplicăm metode asupra lor. De exemplu, putem folosi

Intreg i; /\* Defineste un nou obiect intreg \*/

i.setValue(1); /\* Ii atribuie valoarea 1 \*/

pentru a stabili faptul că obiectul *i* îşi atribuie valoarea 1. Aceasta este mesajul "Aplică metoda *setValue* cu argumentul *1* asupra ta " trimis obiectului *i*. Am notat trimiterea unui mesaj cu ".". Această notaţie este utilizată în C++; alte limbaje orientate pe obiecte ar putea utiliza alte notaţii, de exemplu "–>".

Trimiterea unui mesaj care cere unui obiect să aplice o metodă este similară cu apelul unei proceduri în limbajele"tradiţionale" de programare. Totuşi, în orientarea pe obiecte se au în vedere obiecte autonome care comunică unele cu altele schimbând mesaje. Obiectele reacţionează atunci când primesc mesaje aplicând metode asupra lor înşile. Ele pot de asemenea să refuze executarea unei metode, de exemplu dacă obiectul apelant nu este autorizat să execute metoda cerută.

În exemplul nostru, mesajul şi metoda care trebuie aplicată atunci când mesajul este primit au acelaşi nume. Am trimis "*setValue* cu argumentul *1*" obiectului *i* care aplică "*setValue(1)*". În orientarea pe obiecte aplicarea sau executarea unei metode se numeşte *invocare*.

***Definiţie (Mesaj).*** *Un* ***mesaj*** *este o cerere către un obiect de a invoca una din metodele sale.*

Un mesaj conţine

 *numele* metodei şi

 *argumentele* metodei.

În consecinţă, invocarea unei metode este o reacţie cauzată de recepţionarea unui mesaj.

***Definiţie******(Metodă)****. O* ***metodă*** *este asociată cu o clasă. Un obiect invocă o metodă ca o reacţie la recepţionarea unui mesaj*.

**3.5 Rezumat**

Considerarea unui program ca o colecţie de obiecte care interacţionează este un principiu fundamental în programarea orientată pe obiecte. Obiectele din această colecţie reacţionează la primirea unor mesaje, schimbându-şi starea în funcţie de invocarea unor metode care poate, la rândul ei, să cauzeze trimiterea altor mesaje către alte obiecte. Acest fapt este ilustrat în Figura 3.1.

obiect2

Program

obiect4

obiect3

obiect2

Figura 3.1

În această figură programul constă din numai patru obiecte. Aceste obiecte îşi trimit mesaje unul altuia, după cum indică săgeţile. Se observă că obiectul al treilea îşi trimite lui însuşi un mesaj.

Cum ne poate ajuta această concepţie să dezvoltăm software? Pentru a răspunde la această întrebare vom revedea felul cum s-a dezvoltat software cu limbajele procedurale. Primul pas constă în a diviza problema în părţi mai mici care să poată fi tratate cu uşurinţă separat. Aceste părţi erau concepute ca orientate spre proceduri, care, în rezolvarea problemei, aveau un rol esenţial, în timp ce datele ocupau o poziţie secundară. Să considerăm, de exemplu, felul în care un caracter apare pe ecranul unui calculator atunci când este apăsată o tastă. Într-un mediu procedural urmează să fie scrişi următorii paşi necesari pentru a trimite un caracter pe ecran:

1. aşteaptă până la apăsarea unei taste

2. citeşte valoarea tastei

3. scrie valoarea tastei la poziţia curentă a cursorului.

Nu se pot distinge obiectele ca entităţi cu proprietăţi şi comportare bine definite. Întru-un mediu orientat pe obiecte se pot distinge obiectele *tastă* şi *ecran* aflate în interacţiune. Atunci când o tastă primeşte mesajul că trebuie să îşi schimbe starea şi să fie apăsată, obiectul său corespunzător trimite un mesaj către obiectul ecran să afişeze valoarea asociată tastei.

**3.6 Exerciţii**

1. Clasă.

(a) Prin ce se deosebeşe o clasă de un TAD?

(b) Proiectaţi o clasă pentru TAD-ul *Complex.* Ce reprezentanţi alegeţi pentru operaţiile din TAD? Justificaţi alegerea.

2. Obiecte in interacţiune.

Alegeţi din viaţa dumneavoastră cotidiană o activitate care nu are prea mulţi paşi (de exemplu: privitul la TV, gătirea unei mâncări, etc). Descrieţi această activitate în formă procedurală şi în formă orientată pe obiecte. Încercaţi să vedeţi lumea ca fiind alcătuită din obiecte. Ce dificultăţi întâmpinaţi?

3. Mesaje.

(a) De ce vorbim despre "mesaje" în loc de "apelări de proceduri" ?

(b) Arătaţi câteva mesaje care au sens în mediul Internet (Trebuie deci să identificaţi obiectele).

(c) De ce termenul "mesaje" este mai potrivit în contextul ultimului exerciţiu decât termenul "apelare de procedură"?

**4 Alte concepte orientate pe obiecte**

Mai sus s-au prezentat conceptele fundamentale ale POO. În continuare se vor prezenta mai multe detalii.

**4.1 Relaţii**

În exerciţiile 2.6.5 aţi investigat deja relaţiile dintre tipurile abstracte de date şi instanţieri şi le-aţi descris cu propriile cuvinte. Aici vor fi tratate mai detaliat.

**4.1.1 Relaţia "un fel de"**

Să presupunem că trebuie să scriem un program pentru a desena. Acest program va permite desenarea a diferite *obiecte*: puncte, cercuri, dreptunghiuri, triunghiuri, etc. Pentru fiecare obiect dăm o definiţie de *clasă.* De exemplu, clasa *Punct* defineşte un punct prin coordonatele sale:

class Punct {

attributes:

int x, y

methods:

setX(int nouX)

getX()

setY(int nouY)

getY()

}

Continuăm să definim clasele programului nostru de desen cu o clasă care descrie cercuri. Un cerc este definit prin centru şi rază:

class Cerc {

attributes:

int x, y,

raza

methods:

setX(int nouX)

getX()

setY(int nouY)

getY()

setRaza(int nouaRaza)

getRaza()

}

Comparând cele două definiţii de clase observăm următoarele:

 Ambele clase au două date membre *x* şi *y*. În clasa *Punct* aceste elemente descriu poziţia punctului, iar în clasa *Cerc* ele descriu centrul cercului. Astfel, *x* şi *y* au acelaşi înteles: în ambele clase descriu poziţia obiectelor asociate prin definirea unui punct.

 Ambele clase oferă acelaşi set de metode pentru a obţine şi a atribui valoarea celor două date membre *x* şi *y*.

 Clasa *Cerc* "adaugă" o nouă dată membră *raza* şi metode corespunzătoare de acces.

Cunoscând proprietăţile clasei *Punct* putem descrie cercul ca un punct plus o rază şi metodele de a o accesa. Deci, un cerc este "un fel de" punct. Totuşi, un cerc este ceva mai "specializat". Ilustrăm acest fapt în Figura 4.1.

**Cerc**

**Punct**

**un fel de**

Figura 4.1

În aceasta şi în următoarele figuri clasele sunt reprezentate prin dreptunghiuri. Numele lor începe totdeauna cu majuscule. Săgeata indică direcţia unei relaţii, deci se va citi "Cerc este un fel de Punct".

**4.1.2 Relatia "este un/o"**

Relaţia precedentă este utilizată la nivelul clasei pentru a descrie relaţiile dintre două clase similare. Dacă sunt create obiecte a două astfel de clase, ne referim la relaţia lor ca la o relaţie "este un/o". Figura 4.2 ilustrează această relaţie. În această figură şi în următoarele, obiectele sunt reprezentate ca dreptunghiuri cu colţurile rotunjite. Numele lor conţin numai litere mici.

**cerc**

**punct**

**este un**

Figura 4.2

**4.1.3 Relaţia "parte din"**

Deseori avem nevoie să putem construi obiecte prin combinarea altor obiecte. Cunoaştem aceasta deja de la programarea procedurală, unde se construiesc structuri sau înegistrări punând împreună date de mai multe tipuri. Să revenim la programul nostru de desen. Să pesupunem că deja au fost create mai multe clase pentru diferite figuri. Dacă dorim să avem o figură specială, pe care o vom numi *logo*, formată dintr-un cerc şi un triunghi (presupunând că avem deja definită o clasă *Triunghi*) atunci această figură poate fi reprezentată de o nouă clasă *Logo* care constă din două părţi*,* *Cerc*, şi *Triunghi*. Spunem deci despre fiecare din aceste două clase că este "parte din" noua clasă:

class Logo {

attributes:

Cerc cerc

Triunghi triunghi

methods:

set(Punct loc)

}

Ilustrăm acest fapt în Figura 4.3.

**Cerc**

**Logo**

**parte din**

**Triunghi**

**parte din**

Figura 4.3

**4.1.4 Relaţia "are un/o"**

Această relaţie este inversa relaţiei "parte a". Putem deci să adăugăm această relaţie la figura care ilustrează relaţia "parte a" prin adăugarea unor săgeţi cu sensul schimbat (Figura 4.4).

**Cerc**

**Logo**

**parte din**

**Triunghi**

**parte din**

**are un**

**are un**

Figura 4.4

**4.2 Moştenire**

Moştenirea ne permite să utilizăm relaţiile "un fel de" şi "este un/o". Aşa cum se arată aici, clasele care sunt "un fel de" altă clasă au proprietăţile comune cu acesta. Exemplul nostru privitor la punct şi cerc poate fi rescris specificând că un cerc *moşteneşte de la* (în engleză *inherits from* ) punct:

class Cerc inherits from Punct {

attributes:

int raza

methods:

setRaza(int nouaRaza)

getRaza()

}

Clasa *Cerc* moşteneşte toate datele membre şi metodele de la *Punct.* Nu este nevoie ca acestea să fie definite de două ori; se utilizează numai date şi metode deja definite şi bine cunoscute. La nivelul obiectelor putem să utilizăm un cerc aşa cum am utiliza un punct, deoarece un cerc "este un" punct. De exemplu, putem defini un obiect cerc şi să dăm valori coordonatelor centrului său

Cerc uncerc

uncerc.setX(1) /\* Mostenit de la Punct \*/

uncerc.setY(2)

uncerc.setRaza(3) /\* Adaugat de Cerc \*/

"Este un" implică de asemenea că se poate utiliza un cerc oriunde este utilizat un punct. De exemplu, se poate scrie o funcţie sau o metodă, de exemplu *move(),* care să mişte un punct în direcţia *x* :

move(Punct unpunct, int deltax) {

unpunct.setX(unpunct.getX() + deltax)

}

Deoarece cercul moşteneşte de la punct, putem utiliza această funcţie având ca argument un cerc pentru a muta centrul său, deci întregul cerc:

Cerc uncerc

...

move(uncerc, 10) /\* deplaseaza cercul prin \*/

/\* deplasarea centrului sau \*/

Să formalizăm termenul "moştenire":

***Definiţie (Moştenire). Moştenirea*** *este mecanismul care permite unei clase A să primească proprietăţi ale unei clase B. Spunem "A moşteneşte de la B". Obiectele clasei A au astfel acces la atributele şi metodele clasei B fără a fi nevoie ca ele să fie redefinite.*

Următoarea definiţie introduce doi termeni cu care ne putem referi la clasele care participă la relaţia de moştenire.

***Definiţie (Supraclasă/Subclasă).*** *Dacă o clasă A moşteneşte de la o clasă B, atunci B este numită* ***supraclasă*** *a clasei A. A se numeşte* ***subclasă*** *a clasei B.*

Obiectele unei subclase pot fi utilizate acolo unde pot fi utilizate şi obiectele supraclasei corespunzătoare. Aceasta se datoreşte faptului că obiectele subclasei au o comportare comună cu obiectele supraclasei. În literatură se întâlnesc şi alţi termeni pentru "supraclasă" şi "subclasă". Supraclasele sunt numite şi *clase părinte* (în engleză *parent classes* ). Subclasele se numesc şi *clase copil* (în engleză *child classes* ) sau *clase derivate* (în engleză *derived classes* ). Bineînţeles, se poate moşteni şi de la o subclasă făcându-se din aceasta supraclasă unei noi subclase. Aceasta conduce la o ierarhie a relaţiei supraclasă/subclasă. Dacă se desenează această ierarhie, se obţine un *graf al moştenirii.* Un mod obişnuit de a desena astfel de scheme este utilizarea săgeţilor pentru a indica moştenirea între două clase sau obiecte, săgeata pornind de la subclasă la supraclasă. A se vedea Figura 4.5.

**Punct**

**Cerc**

**moşteneşte de la**

Figura 4.5

În literatură se întâlnesc şi reprezentări grafice în care săgeţile au celălalt sens, după cum diverşi autori înteleg respectiva relaţie. În continuarea prezentei lucrări săgeţile reprezentând moştenirea nu vor mai fi etichetate.

**4.3 Moştenire multiplă**

Un mecanism important orientat pe obiecte este moştenirea multiplă. Moştenire multiplă **nu** înseamnă că mai multe subclase au în comun aceeaşi supraclasă. De asemenea, **nu** înseamnă că o subclasă moşteneşte de la o clasă care este la rândul său o subclasă a altei clase. Prin moştenire multiplă se înţelege faptul că o subclasă are *mai mult decât o singură* supraclasă. Aceasta permite subclasei să moştenească proprietăţile a mai mult decât o supraclasă şi să "combine" proprietaţile supraclaselor sale. Să considerăm de exemplu din nou programul de desenat. Să presupunem că avem o clasă *Text* care permite manipularea convenabilă a textelor. De exemplu, putem avea o metodă de a *adăuga* text. În programul nostru am putea să ne propunem să utilizăm această clasă pentru a adăuga text la obiectele care reprezintă desene. Ar fi convenabil de asemenea să utlizăm rutine deja existente, cum este *move()*, pentru a deplasa textul adăugat. În consecinţă, are sens ca textul desenabil să aibă un punct care defineşte poziţia sa în interiorul zonei desenate. Prin urmare, se deriveaza o nouă clasă *TextDesenabil* care moşteneşte proprietăţi de la *Punct* şi *Text*, aşa cum se arată în Figura 4.6.

**Punct**

**Text**

**TextDesenabil**

Figura 4.6

În pseudo-limbajul nostru scriem aceasta separând supraclasele aceleiaşi subclase prin virgule:

class TextDesenabil inherits from Punct, Text {

attributes:

/\* Toate mostenite de la supraclase \*/

methods:

/\* Toate mostenite de la supraclase \*/

}

Putem utiliza obiectele clasei *TextDesenabil* atât ca puncte, cât şi ca texte. Deoarece un *TextDesenabil* "este un" *Punct*, putem să îl deplasăm

TextDesenabil textd

...

textd.move(10)

...

Deoarece "este un" *Text*, putem să-i adăugăm alt text:

textd.append("Vulpea cea roscata...")

Acum putem defini moştenirea multiplă:

***Definiţie (Moştenire multiplă).*** *Faptul că o clasă A moşteneşte de la mai mult de o clasă, adică A moşteneşte de la clasele B1, B2,..., Bn, cu n>1 se numeşte* ***moştenire multiplă****. Acest fapt poate introduce conflicte de nume în A dacă cel puţin două din supraclasele sale definesc propietăţi cu acelaşi nume.*

Definiţia de mai sus introduce *conflictele de nume* care au loc dacă mai mult o supraclasă a unei subclase utilizează acelaşi nume pentru atribute sau metode. De exemplu, să presupunem că avem o clasă *Text* care defineşte o metodă *setX()* care atribuie ca valoare textului o secvenţă de caractere "X". Se pune întrebarea: ce va moşteni *TextDesenabil* ? Versiunea din *Punct*, cea din *Text*, sau niciuna dintre ele?

Aceste conflicte pot fi soluţionate in cel puţin două moduri:

 Supraclasele se dau într-o anumită ordine şi aceasta defineşte care proprietate va fi accesibilă. Celelalte vor fi "ascunse".

 Subclasa trebuie să rezolve conflictul definind explicit cum vor fi utilizate proprietăţile cu acelaşi nume din supraclasele sale.

Prima soluţie nu este prea convenabilă deoarece ea introduce consecinţe implicite în legătură cu ordinea în care clasele moştenesc una de la alta. Pentru cazul al doilea, subclasele trebuie să definească în mod explicit proprietăţile care sunt implicate într-un conflict de nume.

Un tip special de conflict de nume este introdus dacă o clasă *D* moşteneşte multiplu de la supraclasele *B* şi *C* care sunt şi ele derivate dintr-o supraclasă A. Aceasta conduce la graful de moştenire arătat în Figura 4.7.

**B**

**C**

**D**

**A**

Figura 4.7

Se pune întrebarea: ce proprietăţi moşteneşte efectiv clasa *D* de la supraclasele sale *B* şi *C* ? Unele din limbajele de programare existente rezolvă acest graf de moştenire derivând *D* cu:

 proprietăţile lui *A* plus

 proprietăţile lui *B* şi *C* **fără** proprietăţile pe care aceste clase le-au moştenit de la *A*.

În consecinţă, *D* nu poate să introducă conflicte de nume cu nume din clasa *A*. Totuşi, dacă *B* şi *C* au proprietăţi cu acelaşi nume, *D* ajunge într-un conflict de nume.

O altă posibilă soluţie este ca *D* să moştenească din ambele ramuri de moştenire. În această soluţie, *D* posedă **două** copii ale proprietăţilor din *A* : una este moştenită de *B* şi cealaltă de *C*.

Deşi moştenirea multiplă este un mecanism puternic orientat pe obiecte, problemele introduse de conflictele de nume au determinat pe unii autori să îl "condamne". Deoarece rezultatele moştenirii multiple pot fi întotdeauna obţinute utilizând moştenirea (simplă), unele limbaje de programare orientate pe obiecte nu permit utilizarea sa. Totuşi, utilizată cu atenţie, în anumite condiţii, moştenirea multiplă reprezintă un instrument elegant şi eficient de proiectare orientată pe obiecte.

**4.4 Clase abstracte**

Utilizând moştenirea putem determina o subclasă să ofere aceleaşi proprietăţi ca şi supraclasele sale. În consecinţă, obiectele unei subclase se comportă ca şi obiectele supraclaselor sale.

Uneori are sens ca numai să se descrie proprietăţile unei mulţimi de obiecte fără să li se cunoască dinainte comportarea efectivă. În exemplul nostru cu programul de desenare, fiecare obiect dă o metodă de a se desena pe sine. Totuşi, paşii necesari pentru a desena un obiect depind de forma sa. De exemplu, rutina de desenare a unui cerc diferă de rutina de desenare a unui dreptunghi.

Să denumim metoda de desenare *print()*. Pentru a forţa fiecare obiect desenabil să includă o astfel de metodă, definim o clasă *ObiectDesenabil* de la care orice altă clasă din exemplul nostru moşteneşte proprietăţile generale ale obiectelor desenabile:

abstract class ObiectDesenabil {

attributes:

methods:

print()

}

Aici introducem noul cuvânt cheie **abstract**. El este utilizat pentru a exprima faptul că proprietăţile trebuie să fie "redefinite" de clasele derivate pentru a realiza funcţionalitatea dorită. Astfel, din punctul de vedere al claselor abstracte, proprietăţile sunt numai *specficate* şi nu complet *definite*. Definiţia completă, incluzând semantica proprietăţilor, trebuie să fie dată de clasele derivate.

În continuare, fiecare clasă din programul nostru de desen trebuie să moştenească proptietăţi de la clasa generală pentru obiectele desenabile. Astfel clasa *Punct* trebuie schimbată şi devine:

class Punct inherits from ObiectDesenabil {

attributes:

int x, y

methods:

setX(int nouX)

getX()

setY(int nouY)

getY()

print() /\* Redefinire pentru Punct \*/

}

Putem acum să forţăm fiecare obiect desenabil să aibă o metodă *print* care să dea funcţionaliatea desenării obiectului în interiorul zonei de desenare. Supraclasa tuturor obiectelor desenabile, clasa *ObiectDesenabil,* nu oferă funcţionaliatea pentru desenare. Această clasă nu este destinată pentru a se crea obiecte din ea. Ea specifică mai degrabă proprietăţi care trebuie definite de către fiecare clasă derivată. Numim acest tip special de clasă, *clasă abstractă.*

***Definiţie (Clasă abstractă).*** *O clasă A se numeşte* ***clasă abstractă*** *dacă ea este utilizată numai ca supraclasă pentru alte clase. Clasa A doar specifică proprietăţi. Ea nu este utilizată pentru a se crea obiecte. Clasele derivate trebuie să definească proprietăţile clasei A.*

Clasele abstracte ne permit să structurăm graful de moştenire. Totuşi nu dorim să creăm obiecte din ele; dorim numai să exprimăm caracteristicile comune ale unei mulţimi de clase.

**4.5 Exerciţii**

1. Moştenire.

Să considerăm programul de desenare.

(a) Definiţi clasa *Dreptunghi* prin moştenire de la clasa *Punct*. Punctul va indica colţul din stânga sus al dreptunghiului. Care sunt atributele acestei clase? Ce metode adiţionale introduceţi?

(b) Toate exemplele de mai sus sunt bazate pe viziunea bidimensională. Dorim acum să introducem obiecte tridimensionale cum ar fi: sfere, cuburi, paralelipipede dreptunghice, etc. Proiectaţi clasa *Sfera* utilizând o clasă *Punct-3D* ale cărei obiecte sun punctele în spaţiul tridimensional. Specificaţi rolul punctului în sferă. Ce relaţie utilizaţi între clasele *Sfera* şi *Punct-3D*?

(c) Ce funcţionalitate dă *move()* pentru obiectele tridimensionale? Daţi o exprimare cât mai precisă.

(d) Desenaţi graful de moştenire care să includaă urmatoarele clase: *ObiectDesenabil*, *Punct*, *Cerc*, *Dreptunghi*, *Punct-3D* şi *Sfera*.

(e) Priviţi graful de moştenire din Figura 4.8.

**Punct**

**Cerc**

**Sferă**

Figura 4.8

O definiţie corespunzătoare ar putea fi:

class Sfera inherits from Cerc {

attributes:

int z /\* Adauga a treia dimensiune \*/

methods:

setZ(int nouZ)

getZ()

}

Motivaţi avantajele/dezavantajele acestei alternative.

2. Moştenire multiplă. Comparaţi graful de moştenire din Figura 4.9. Aici noi arătăm că *B* şi *C* au fiecare o copie a clasei *A*.

**B**

**C**

**D**

**A**

**A**

Figura 4.9

Ce conflicte de nume pot să apară? Prezentaţi cazurile prin exemple simple de clase.

**5 Din nou concepte ale orientării pe obiecte**

Continuăm prezentarea conceptelor orientate pe obiecte cu o scurtă introducere privind comparaţia dintre legarea statică şi dinamică. Introducem polimorfismul ca un mecanism care permite obiectelor să arate ceea ce urmează să se facă în momentul execuţiei. Mai întâi, dăm o scurtă prezentare a tipurilor generice.

**5.1 Tipuri generice**

Cunoaştem deja tipurile generice din capitolul 2, unde am discutat despre tipurile abstracte de date generice. Atunci când definim o clasă, definim de fapt un *tip definit de utilizator.* Unele din aceste tipuri pot opera cu alte tipuri. De exemplu, pot fi liste de persoane, liste de maşini, liste de numere complexe, sau chiar liste de liste.

Atunci când scriem definiţia unei clase, trebuie să putem preciza dacă această clasă defineşte un tip generic. Totuşi, nu ştim cu care tipuri va fi utilizată clasa. În consecinţă, trebuie să putem defini clasa cu ajutorul unui "locţiitor" la care ne referim ca şi când ar fi tipul asupra căruia operează clasa. Astfel, definiţia clasei ne este dată ca un *model* sau *şablon* (traducerea termenului englez *template*) al unei clase efective. Definiţia clasei este creată de fapt atunci când declarăm un obiect particular. Vom ilustra aceasta cu următorul exemplu. Să presupunem că dorim să definim o clasă de liste de persoane, maşini, sau de alt tip.

template class Lista for T {

attributes:

... /\* Structura de date necesara pentru \*/

/\* implementarea listei \*/

methods:

append(T element)

T getFirst()

T getNext()

bool more()

}

Modelul de clasă *Lista* de mai sus arată ca orice altă definiţie de clasă. Totuşi, prima linie declară *Lista* ca fiind un model pentru diferite tipuri. Identificatorul *T* este utilizat ca locţiitor pentru un tip real. De exemplu, *append()* are un element ca argument. Tipul acestui element trebuie să fie tipul de date cu care o listă efectivă de obiecte este creată. De exemplu, putem să declarăm un obiect listă de persoane dacă există o definiţie a tipului *Persoana*:

Lista for Persoana persoanaLista

Persoana oPersoana,

altaPersoana

persoanaLista.append(altaPersoana)

persoanaLista.append(oPersoana)

Prima linie declară *persoanaLista* ca fiind o listă de persoane. În acest moment, compilatorul utilizează definiţia modelului, înlocuieşte fiecare apariţie a lui *T* cu *Persoana* şi crează o definiţie efectivă de clasă pentru ea. Aceasta conduce la o definiţie de clasă efectivă similară cu următoarea:

class Lista {

attributes:

... /\* Structura de date necesara pentru \*/

/\* implementarea listei \*/

methods:

append(Persoana element)

Persoana getFirst()

Persoana getNext()

bool more()

}

Aceasta nu este chiar ceea ce generază compilatorul. Compilatorul trebuie să asigure că putem crea mai multe liste pentru diferite tipuri în orice moment. De exemplu, dacă avem nevoie de o altă listă, de exemplu pentru maşini, putem scrie:

Lista for Persoana persoanaLista

Lista for Masina masinaLista

...

În ambele cazuri compilatorul generează o definiţie efectivă de clasă. Cele două definiţii nu se află în conflict deoarece compilatorul generează nume diferite. Totuşi, deoarece acest lucru nu este vizibil pentru noi, nu vom prezenta aici mai multe detalii. În orice caz, dacă declarăm o altă listă de persoane, compilatorul poate să stabilească dacă există deja o definiţie efectivă de clasă şi să o utilizeze sau, dacă este necesar, să o creeze. Astfel,

Lista for Persoana oLista

Lista for Persoana altaLista

va crea definiţia efectivă de clasă pentru *oLista* şi o va reutiliza pentru *altaLista*. Prin urmare, ambele sunt de acelaşi tip. Rezumăm aceasta în următoarea:

***Definiţie (Model de clasă).*** *Dacă o clasă A este parametrizată cu un tip de clase B, A se numeşte* ***model de clasă****. Atunci când un obiect din A este creat, B este înlocuit cu un* ***tip efectiv****. Aceasta permite definiţia unei* ***clase efective*** *bazată pe modelul specificat pentru A si tipul efectiv.*

Putem defini modele de clase cu mai mult decât um parametru. De exemplu *cataloagele* sunt colecţii de obiecte în care fiecare obiect este referit printr-o *cheie*. Bineînţeles, un catalog ar trebui să fie capabil să conţină orice tip de obiect. De asemenea, sunt diferite posibilităţi pentru chei. De exemplu, ele pot fi şiruri de numere. Prin urmare vom defini un model de clasă *Catalog* care este bazat pe doi parametri-tipuri, unul pentru chei şi unul pentru obiectele conţinute.

**5.2 Legare statică şi dinamică**

În limbajele de programare cu reguli stricte de stabilire şi de utilizare a tipurilor, tipurile trebuie *declarate* înaintea utilizării lor. Aceasta implică *definirea* variabilei, care produce rezervarea spaţiului de către compilator. De exemplu, în Pascal o expresie ca

var i : integer;

declară variabila *i* ca fiind de tipul *integer*. În plus, ea defineşte un spaţiu de memorie suficient pentru a păstra o variabilă de tipul respectiv.

Cu declaraţia se *leagă* numele *i* de tipul *integer*. Această legătura este adevărată în conformitate cu scopul în care *i* este creată. Aceasta permite compilatorului să verifice în timpul compilării compatibilitatea tipurilor. De exemplu, următoarea atribuire va produce o eroare de incompatibilitate a tipurilor atunci când este compilată:

var i : integer;

...

i := 'sir';

Numim acest tip particular de legătură "static" deoarece este fixat în momentul compilării.

***Definiţie (Legare statică).*** *Dacă tipul T al unei variabile este asociat în mod explicit cu numele său N prin declaraţie, spunem că N este* ***legată static*** *de T. Procesul de asociere se numeste* ***legare statică****.*

Există limbaje de programare care nu utilizează variabile cu un tip asociat în mod explicit. De exemplu, anumite limbaje permit să fie introduse variabile atunci când sunt utilizate:

... /\* Nicio aparitie a lui i \*/

i := 123 /\* Crearea lui i ca intreg \*/

Tipul lui *i* este cunoscut atunci când variabila primeşte o valoare. În acest caz *i* este de tipul *integer* deoarece i s-a atribuit un număr întreg. Astfel, deoarece *conţinutul* lui *i* este un număr întreg, *tipul* lui *i* este *integer*.

***Definiţie (Legare dinamică).*** *Dacă tipul T al unei variabile cu numele N este implicit asociat prin conţinutul său, spunem că N este* ***legată dinamic*** *de T. Procesul de asociere se numeşte* ***legare dinamică****.*

Cele două definiţii diferă prin momentul când tipul este legat de variabilă. Să considerăm următorul exemplu (scris în limbajul de programare JavaScript, care permite numai legarea dinamică a variabilelor):

if (conditie() == TRUE)

n = 123;

else

n = 'abc';

Tipul lui *n* după instrucţiunea *if* depinde de evaluarea lui *conditie()*. Dacă aceasta este TRUE, *n* este de tip întreg, iar în celălalt caz este de tip şir de caractere.

**5.3 Polimorfism**

Polimorfismul (termen de origine greacă poli=mai multe morphe=formă) permite unei entităţi (de exemplu: variabilă, funcţie sau obiect) să aibă diferite reprezentări (adică sub acelaşi nume se prezintă diferite conţinuturi). Aici vom pune în evidenţă mai multe tipuri de polimorfism.

Primul tip este similar cu conceptul de legare dinamică. În acest caz, tipul unei variable depinde de conţinutul său într-un anumit moment:

v := 123 /\* v este intreg \*/

... /\* utilizeaza v ca intreg \*/

v := 'abc' /\* v "comuta" la un sir de caractere \*/

... /\* utilizeaza v ca sir de caractere \*/

***Definiţie (Polimorfism (1)).*** *Conceptul de legare dinamică permite unei variabile să primească diferite tipuri în funcţie de conţinutul său într-un anume moment. Această particularitate a unei variabile este numită* ***polimorfism****.*

Un alt tip de polimorfism poate să fie definit pentru funcţii. De exemplu să presupunem că dorim să definim o funcţie *isNull()* care întoarce rezultatul TRUE dacă argumentul său este 0(zero) şi FALSE altfel. Pentru numere întregi este uşor:

boolean isNull(int i) {

if (i == 0) then

return TRUE

else

return FALSE

endif

}

Totuşi, dacă dorim să facem verificarea pentru numere reale, va trebui să utilizăm altă comparaţie datorită problemelor de precizie:

boolean isNull(real r) {

if (r < 0.01 and r > -0.99) then

return TRUE

else

return FALSE

endif

}

În ambele cazuri dorim ca funcţia să aibă numele *isNull.* În limbajele de programare fără polimorfism pentru funcţii nu putem să declarăm aceste două funcţii deoarece numele *isNull* ar fi definit de două ori. Fără polimorfism pentru funcţii, dubla definire a numelor ar duce la ambiguitate. Totuşi, dacă limbajul ar ţine cont de *argumentele* funcţiilor, ar fi posibil. Astfel funcţiile (sau metodele) sunt unic definite prin:

 *numele* funcţiei (sau metodei) şi

 *tipurile* din *lista de argumente.*

Deoarece listele de argumente ale celor doua funcţii *isNull* diferă, compilatorul poate să construiască corect apelul de funcţie utilizând tipurile efective ale argumentelor:

var i : integer

var r : real

i = 0

r = 0.0

...

if (isNull(i)) then... /\* Utilizeaza isNull(int) \*/

...

if (isNull(r)) then... /\* Utilizeaza isNull(real) \*/

***Definiţie (Polimorfism (2)).*** *Dacă o funcţie (sau metodă) este definită prin combinarea dintre*

* numele său şi*

* lista tipurilor argumentelor sale*

*spunem că avem* ***polimorfism****.*

Acest tip de polimorfism ne permite să reutilizăm acelaşi nume pentru funcţii (sau metode) dacă listele de argumente sunt diferite. Uneori acest tip de polimorfism este numit *supraîncărcare, supraacoperire* sau *supradefinire.*

Ultimul tip de polimorfism permite unui obiect să-şi aleagă metodele corecte. Să considerăm din nou funcţia *move()*, care are un obiect al clasei *Punct* ca argument. Am utilizat această funcţie ca orice obiect al claselor derivate, deoarece are loc relaţia "este un/o".

Să considerăm o funcţie *display()* care să fie utilizată pentru a afişa obiecte desenabile. Declaraţia acestei funcţii ar putea fi:

display(ObiectDesenabil o) {

...

o.print()

...

}

Dorim să utilizăm această funcţie cu obiecte ale claselor derivate din *ObiectDesenabil*:

Cerc uncerc

Punct unpunct

Dreptunghi undreptunghi

display(unpunct) /\* Trebuie invocata unpunct.print() \*/

display(uncerc) /\* Trebuie invocata uncerc.print() \*/

display(undreptunghi)/\* Trebuie invocata undreptunghi.print() \*/

Metoda care va fi utilizată efectiv va fi definită de obiectul *o* al funcţiei *display()*. Deoarece acest lucru este cam complicat, dăm un exemplu mai abstract, dar mai simplu:

class Baza {

attributes:

methods:

virtual foo()

bar()

}

class Derivata inherits from Baza {

attributes:

methods:

virtual foo()

bar()

}

demo(Baza o) {

o.foo()

o.bar()

}

Baza obaza

Derivata oderivata

demo(obaza)

demo(oderivata)

În acest exemplu definim două clase: *Baza* şi *Derivata*. Fiecare clasă defineşte două metode: *foo()* şi *bar()*. Prima metodă este definită ca virtuală (prin cuvântul cheie **virtual**). Aceasta înseamnă că dacă această metodă este invocată, definiţia sa va fi evaluată prin conţinutul obiectului.

Definim apoi o funcţie *demo()* care are un obiect din *Baza* ca argument. Prin urmare, putem să utilizăm această funcţie cu obiecte ale clasei *Derivata* ca şi cum ar fi valabilă relaţia "este un/o". Apelăm această funcţie cu un obiect din *Baza* şi cu un obiect din clasa *Derivata*, respectiv să presupunem că funcţiile *foo()* şi *bar()* sunt definite astfel încât singurul lucru pe care îl fac este să-şi scrie numele şi clasa în care sunt definite. Atunci ieşirea este:

foo() din Baza apelata.

bar() din Baza apelata.

foo() din Derivata apelata.

bar() din Baza apelata.

De ce? Să vedem ce se execută. Prima apelare a funcţiei *demo()* utilizează un obiect din *Baza*. Astfel, argumentul funcţiei este "încărcat " cu un obiect al clasei *Baza*. Atunci când se invocă metoda *foo()* funcţionalitatea ei reală bazata pe conţinutul curent al obiectului corespunzător *o*. De această dată, el este un obiect din *Baza*. În consecinţă, este apelată metoda *foo()* aşa cum este definită în clasa *Baza*. Apelarea metodei *bar()* nu este afectată de conţinutul obiectului. Ea **nu** este declarată ca **virtuală**. În consecinţă, metoda *bar()* este apelată în domeniul clasei *Baza*. A doua apelare a funcţiei *demo()* are un obiect din *Derivata* ca argument. Astfel, argumentul *o* este încărcat cu un obiect din *Derivata*. Totuşi, *o* însuşi nu reprezintă decât partea *Baza* a obiectului furnizat *oderivata*.

În sfârşit, apelarea metodei *foo()* este evaluată prin examinarea conţinutului lui *o*, deci este apelată în interiorul domeniului clasei *Derivata*. Pe de altă parte, *bar()* este evaluată tot în domeniul clasei *Baza*.

***Definiţie (Polimorfism (3)).*** *Obiectele supraclaselor pot să fie încărcate cu obiecte ale subclaselor lor. Operatorii şi metodele subclaselor pot să fie definiţi pentru a fi evaluaţi în două contexte:*

*1. Pe baza tipului* ***declarat*** *al obiectului (numit şi* ***tip static****), ceea ce conduce la o evaluare în interiorul domeniului supraclasei.*

*2. Pe baza tipului* ***conţinutului*** *obiectului (numit şi* ***tip dinamic****), ceea ce conduce la o evaluare în interiorul domeniului subclasei care le conţine.*

*Al doilea tip este numit* ***polimorfism****.*

**6 Implementarea conceptelor orientate pe obiecte în limbaje de programare**

În acest capitol prezentăm felul cum sunt tratate chestiunile privind programarea orientată pe obiecte în două limbaje de programare care, datorită calităţilor lor, sunt printre cele mai cunoscute şi utilizate: TurboPascal, pentru care vom face o scurtă prezentare, şi C++, pentru care prezentarea va fi mult mai amplă.

**6.1 Programare orientată pe obiecte în TurboPascal**

Se presupune cunoscut limbajul TurboPascal.

În TurboPascal corespondentul clasei este tipul declarat de utilizator **object**, a cărui declarare are sintaxa asemănătoare cu a tipului **record**. Componentele unui tip **object** pot fi atât date (reprezentând *atribute*), cât şi proceduri sau funcţii (reprezentând *metode*). Declararea unui tip **object** este permisă numai în partea de declarare a modulului (deci **nu** în corpul funcţiilor şi al procedurilor). Definiţiile metodelor asociate trebuie plasate în acelaşi domeniu după declararea tipului **object**. Referirea la componentele unei variabile având un tip **object** se face cu sintaxa *nume\_variabilă.nume\_componenta[(lista\_de\_ argumente\_efective)]*. Exemplul de mai jos prezintă un program în care se defineşte şi se utilizează un tip **object** pentru numere complexe.

type complex=object

{date}

re, im:real;

{declarari de metode}

procedure comp(a, b:real);

procedure atr(v:complex);

function sum\_re(v:complex):real;

function sum\_im(v:complex):real;

procedure sum(v:complex; var s:complex);

procedure citeste;

procedure scrie;

end;

{implementarea metodelor}

procedure complex.comp(a, b:real);

begin

re:=a;

im:=b

end;

procedure complex.atr(v:complex);

begin

comp(v.re, v.im)

end;

function complex.sum\_re(v:complex):real;

begin

sum\_re:=re+v.re

end;

function complex.sum\_im(v:complex):real;

begin

sum\_im:=im+v.im

end;

procedure complex.sum(v:complex; var s:complex);

begin

s.re:=re+v.re;

s.im:=im+v.im

end;

procedure complex.citeste;

begin

read(re, im); readln

end;

procedure complex.scrie;

begin

write('re=', re, ' im=', im); writeln

end;

{program principal}

var c1, c2:complex;

begin

c1.citeste;

c2.atr(c1);

c2.sum(c1, c1);

c1.scrie;

c2.scrie;

readln

end.

Dăm mai jos o scurtă prezentare a celorlalte elemente de programare orientată pe obiecte pe care TurboPascal le oferă.

 Posibilitatea ca un tip **object** să **moştenească** un alt tip **object**, declaraţia începând cu o specificare a tipurilor cu sintaxa:

*tip\_obiect* object *tip\_obiect\_mostenit*

urmată de declararea datelor şi metodelor.

 Posibilitatea de a se **supraîncărca** metodele moştenite prin definirea în obiectul moştenitor unor metode cu acelaşi nume, dar cu un corp diferit şi, eventual, cu altă listă de argumente.

 Posibilitatea ca unele din metodele unui tip obiect, să fie declarate *virtuale* prin adăugarea cuvântului cheie **virtual** la declaraţia metodei, ca de exemplu:

procedure *nume*(*argumente*); virtual;

ceea ce obligă la declararea ca virtuale a metodelor cu acelaşi nume ale tipurilor **object** moştenitoare. Efectul acestor declararaţii este rezolvarea tuturor apelurilor procedurilor virtuale în momentul **execuţiei** şi nu al compilării, adică rezolvarea apelurilor este dinamică. Acest mecanism de apel trebuie să fie pregătit în timpul execuţiei prin apelarea înainte de apelarea oricărei metode virtuale a unei metode declarată cu cuvântul cheie **constructor** (în loc de **procedure** sau **function**).

Metodele care nu sunt virtuale se numesc *statice*. Este interzisă supraîncărcarea metodelor virtuale cu metode statice.

Pentru mai multe informaţii asupra limbajului TurboPascal, ca şi asupra programării orientate pe obiecte în acest limbaj, recomandăm să se consulte [1].

Din scurta prezentare de mai sus se observă absenţa implementării unor concepte importante ale orientării pe obiecte cum sunt: clase abstracte, modele de clase, moştenire multiplă. Principiul incapsulării datelor şi metodelor este de asemenea incomplet implementat, lipsind elmentele care să permită limitarea şi diferenţierea accesului la componentele unui tip **object**.

Aceste lipsuri se explică prin faptul că limbajul TurboPascal a fost realizat de firma Borland prin extensia limbajului standard Pascal, care este un limbaj procedural tipic, astfel încât să fie înglobate şi celelalte tehnici de programare: programarea modulară şi programarea orientată pe obiecte. Conceptele orientării pe obiecte s-au implementat însă incomplet, probabil pentru că limbajul de bază standard Pascal are caracteristici care făceau foarte dificilă o abordare mai amplă: ar fi trebuit creat un limbaj cu totul nou, mult îndepărtat de limbajul de bază.

Situaţia este substanţial diferită în cazul limbajului C++, cu toate că şi acest limbaj a fost realizat prin extensia "limbajului de bază" C. Flexibilitatea deosebită a limbajului C a permis ca adăugarea facilităţilor de programare orientată pe obiecte, care este principala raţiune a creării limbajului C++, să se facă într-un mod mult mai complet şi eficient, fără să se piardă posibilitatea ca limbajul să fie utilizat doar ca un limbaj procedural obişnuit.

**6.2 C++ - limbaj creat pentru programarea orientată pe obiecte**

Această secţiune este o introducerie în C++. Mai întâi facem o scurtă prezentare a limbajului de programare C, apoi arătăm cum C++ extinde limbajul C prin lărgirea sistemului de tipuri, prin alte câteva facilităţi şi, cel mai important, prin **concepte orientate pe obiecte**. Prezentăm, de asemenea, înainte de extensiile aduse de C++, şi incompatibilităţile între C++ şi C.

**6.2.1 Limbajul de programare C**

Dezvoltat spre sfârşitul anilor '70, C a obţinut un succes uriaş datorită dezvoltării sistemului de operare UNIX, care a fost aproape în întregime scris în acest limbaj. Spre deosebire de alte limbaje de nivel înalt, C a fost scris de programatori pentru programatori. Astfel, el permite uneori formulări care, în alte limbaje, cum ar fi Pascal, sunt interzise din cauza influenţei lor nefaste asupra stilului de programare. Oricum, dacă este utlizat cu o anumită rigoare, C este un limbaj la fel de bun ca şi multe alte limbaje de nivel înalt. Comentariile în C sunt cuprinse între /\*...\*/.

**6.2.1.1 Tipuri de date**

Tabelul 6.1 descrie tipurile de date ale limbajului C în implementarea Turbo C++ şi Borland C++. *Lungimea* specificată este măsurată în octeţi. *Domeniul* este o consecinţă a lungimii. Se pot obţine informaţii asupra lungimii unui tip cu operatorul **sizeof**.

Tabelul 6.1

|  |  |  |  |
| --- | --- | --- | --- |
| **Tip** | **Descriere** | **Lungime** | **Domeniu/Precizie** |
|  |  |  |  |
| <signed> char | Octet cu semn | 1 | 128..127 |
| double | Număr în dublă precizie | 8 | ca (1, 7X10308..1, 7X10308)  precizie 15 cifre |
| <signed> <short>int |  <signed> short | Întreg cu semn | 2 | 215.. 215  1 |
| float | Număr în virgulă mobilă | 4 | ca (3, 4X1038..3, 4X1038)  precizie 7 cifre |
| <signed> long <int> | Întreg lung cu semn | 4 | 231.. 231  1 |
| long double | Număr în cea mai mare precizie | 10 | ca (3, 4X104932..3, 4X104932)  precizie 19 cifre |
| unsigned char | Octet fără semn | 1 | 0..255 |
| unsigned <short> <int> | Întreg fără semn | 2 | 0.. 216  1 |
| unsigned long <int> | Întreg lung fără semn | 4 | 0.. 232  1 |

Implementarea Unix are următoarele deosebiri faţă de tabelul de mai sus:

* tipul **short** <**int>** are lungimea 2, domeniul 215.. 215  1 şi coincide cu <**signed> short <int>**;
* tipul **unsigned short** <**int>** are lungimea 2 şi domeniul 0.. 216  1;
* tipul <**signed> int** are lungimea 4, domeniul 231.. 231  1 şi coincide cu <**signed> long <int>**;
* tipul **unsigned**  <**int>** are lungimea 4, domeniul 0.. 232  1 şi coincide cu tipul **unsigned long <int>**;
* există modificatorul **long long** şi tipul <**signed> long long <int>** cu lungimea 8 şi domeniul 263.. 263  1;
* există tipul **unsigned long long <int>** cu lungimea 8 şi domeniul 0.. 264  1;
* nu există tipul **long double**.

Variabilele acestor tipuri sunt definite simplu prin specificarea tipului înaintea numelui:

int un\_int;

float un\_float;

long long un\_intreg\_foarte\_lung;

Cu **struct** se pot combina mai multe tipuri definite. În alte limbaje acest tip structurat este numit **record**:

struct data\_s {

int ziua, luna, an;

} oData;

*Definiţia* de mai sus pentru *oData* este de asemenea *declaraţia* unei structuri numită *data\_s*. Putem defini alte variabile de acest tip referindu-ne la structură prin nume:

struct data\_s altaData;

Nu este necesar să denumim structurile. Dacă le omitem numele, nu putem să le reutilizam. Totuşi, dacă denumim o structură, putem doar să o *declarăm,* fără a defini o variabilă:

struct timp\_s {

int ora, minut, secunda;

};

Putem să utilizăm această structură aşa cum s-a arătat pentru *altaData*. Acest lucru este foarte asemănător cu definiţiile de tipuri din alte limbaje, unde un tip este **declarat** înaintea **definirii** unei variabile de acest tip. Variabilele trebuie definite înaintea utilizării lor. Aceste definiţii trebuie să apară înaintea oricărei instrucţiuni, deci ele formează partea superioară a unui *bloc de instrucţiuni.*

**6.2.1.2 Instrucţiuni**

C conţine toate tipurile uzuale de instrucţiuni. Instrucţiunile sunt terminate prin ";". Putem să grupăm mai multe instrucţiuni în blocuri prin plasarea lor între acolade. În interiorul fiecărui bloc putem să definim noi variabile:

{ /\* Incepe blocul exterior \*/

int i; /\* Defineste un i \*/

i = 1; /\* Atribuie lui i valoarea 1 \*/

{ /\* Incepe un nou bloc \*/

int i; /\* Defineste un nou i \*/

i = 2; /\* care primeste valoarea 2 \*/

} /\* Inchide blocul \*/

/\* Aici i este 1 din nou din blocul exterior \*/

}

Tabelul 6.2 listează toate instrucţiunile

Tabelul 6.2

|  |  |
| --- | --- |
| **Instrucţiune** | **Descriere** |
|  |  |
| ; | Instrucţiunea de efect nul. |
| *expr*; | Instrucţiunea expresie. Efectuează expresia *expr*. |
| break; | Înceteză execuţia instrucţiunii repetitive sau sau switch. |
| continue; | Utilizată numai în instrucţiuni repetitive. Produce trecerea la ciclul următor. |
| do  *instr*  while(*expr*) | Execută *instr* cât timp *expr* are valoarea diferită de 0 (orice valaore nenulă este interpretată ca valoare de adevăr TRUE). |
| for ([*expr1*]; [*expr2*]; [*expr3*];)  *instr* | Prescurtare pentru o instrucţiune while în care *expr1* este iniţializare, *expr2* este condiţie, iar *expr3* se efectuează după fiecare ciclu. Lipsa *expr3* se interpretează ca valoare de adevăr TRUE. |
| goto *eticheta*; | Transfer la poziţia din aceeaşi definiţie de funcţie indicată de *eticheta*. Destinaţia este *eticheta* urmată de ":". |
| if (*expr*) *instr1* [else *instr2*]; | Instrucţiune condiţională. Dacă valoarea de adevăr a expr este TRUE se execută *instr1*, altfel se execută *instr2* (dacă există). |
| return [*expr*]; | Revenire din funcţie. Dacă funcţia nu întoarce valoare, *expr* nu este obligatorie şi nu se utilizează, altfel este obligatorie şi valoarea sa este valoarea întoarsă. |
| switch (*expr*) {  case *const-expr 1*: *lista\_de\_instr1*  *…*  case *const-expr*: *lista\_de\_instr*  [default: *lista\_de\_instr*]  } | Valoarea *expr* este comparată cu vlorile *const-expr i* (pentru *i* de la 1 la n) şi se execută secvenţa de instrucţiuni *lista\_ de\_ instr i*, continuându-se cu executarea următoarelor secvenţe de instrucţiuni, până la execuţia unei instrucţiuni break. Dacă nicio clauză case nu corespunde, se execută secvenţa *lista\_ de\_ instr* de la default (dacă există). |
| while (*expr*) *instr* | Execută *instr* cât timp *expr* are valoarea de adevăr TRUE. |

Instrucţiunea **for** este singura care diferă esenţial de instrucţiunile **for** cunoscute în alte limbaje. O formă specială are şi instrucţiunea **switch**, care execută lista de instrucţiuni începând cu "eticheta" **case** corespunzătoare valorii expresiei testate până la execuţia unei instrucţiuni **break**. De asemenea, posibilitatea de a se termina execuţia instrucţiunilor repetitive cu instrucţiunea **break** şi posibilitatea de a se întrerupe ciclul curent cu instrucţiunea **continue** sunt elemente care pot contribui la construirea unor programe eficiente. Toate celelalte instrucţiuni diferă de cele din alte limbaje mai mult sau mai puţin prin sintaxă. Mai jos arătăm două blocuri care au absolut aceeaşi funcţionalitate. Unul utilizează bucla **while**, iar celălalt instrucţiunea **for** :

{

int ix, suma;

suma = 0;

ix = 0; /\* initializare \*/

while (ix < 10) { /\* conditie \*/

suma = suma + 1;

ix = ix + 1; /\* pas \*/

}

}

{

int ix, suma;

suma = 0;

for (ix = 0; ix < 10; ix = ix + 1)

suma = suma + 1;

}

Cele de mai sus pot fi bine înţelese dacă se ţine seama de încă o proprietate caracteristică a limbajului C: **instrucţiunea de atribuire este în acelaşi timp o expresie**.

**6.2.1.3 Operatori şi expresii**

În C aproape orice este o expresie. De exemplu, instrucţiunea de atribuire "=" întoarce valoarea operandului din dreapta. Ca un "efect colateral" atribuie valoarea operatorului din stânga. Astfel,

ix = 12;

atribuie lui *ix* valoarea *12* (presupunând că *ix* are un tip compatibil). Deoarece atribuirea este de asemenea o expresie, putem combina mai multe atribuiri; de exemplu:

kx = jx = ix = 12;

S-a efectuat ceea ce, în stilul comun celor mai multe limbaje de programare s-ar fi efectuat prin

ix = 12;

jx = 12;

kx = 12;

În C valoarea 0 (zero) reprezintă valoarea de adevăr FALSE. Orice altă valoare reprezintă TRUE. De exemplu, funcţia standard *strcmp()*, cu două şiruri ca argumente, întoarce *-1* dacă primul este mai mic decât al doilea, *0* dacă sunt egale şi *1* dacă primul este mai mare decât al doilea. Pentru a testa dacă două şiruri *str1* şi *str2* sunt egale, se utilizează adesea instrucţiunea condiţională:

if (!strcmp(str1, str2)) {

/\* str1 este egal cu str2 \*/

}

else {

/\* str1 nu este egal cu str2 \*/

}

Semnul de exclamare reprezintă operatorul boolean NOT (negatie). Astfel, expresia este evaluată TRUE numai dacă *strcmp()* întoarce 0. Expresiile se construiesc prin combinarea termenilor (sau *operanzilor*) şi a *operatorilor.* Termenii pot fi constante, variabile, sau expresii. Operatorii limbajului C sunt cei cunoscuţi din alte limbaje, la care se adaugă câţiva care reprezintă prescurtări ale unor combinaţii ale altor operatori. Tabelul 6.3 listează toţi operatorii disponibili. Coloana a doua arată prioritatea lor, numerele mai mici indicând o prioritate mai înaltă şi numerele egale, aceeaşi prioritate. Coloana a treia conţine aritatea (adică numărul de operanzi). Aritatea primului operator (apelul de funcţie) este varabilă, depinzând de felul în care este declarat primul operand (funcţie sau pointer la funcţie) printr-un prototip. Ultima coloană listează ordinea de evaluare.

Tabelul 6.3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Pre-ceden-ţă** | **Ope-rator** | **Descriere** | **Aritate** | **Ordine** |
|  |  |  |  |  |
| 1 | () | Operatorul apelare de funcţie | variabilă | din stânga |
| 1 | [] | Operatorul de referire la element de tablou prin indice | 2 | din stânga |
| 1 | -> | Selecţie indirectă | 2 | din stânga |
| 1 | . | Selecţie directă | 2 | din stânga |
| 2 | ! | Negaţie (NOT) logică | 1 | din dreapta |
| 2 | ~ | Negaţie (NOT) la nivel de bit | 1 | din dreapta |
| 2 | + | Plus unar | 1 | din dreapta |
| 2 | - | Minus unar | 1 | din dreapta |
| 2 | ++ | Preincrementare sau postincrementare | 1 | din dreapta |
| 2 | -- | Predecrementare sau postdecrementare | 1 | din dreapta |
| 2 | & | Adresă | 1 | din dreapta |
| 2 | \* | Indirectare (dereferire) | 1 | din dreapta |
| 2 | sizeof | (lungimea operandului în octeţi) | 1 | din dreapta |
| 3 | \* | Înmulţire | 2 | din stânga |
| 3 | / | Împărţire | 2 | din stânga |
| 3 | % | Restul împărţirii întregi (modulo) | 2 | din stânga |
| 4 | + | Adunare | 2 | din stânga |
| 4 | - | Scădere | 2 | din stânga |
| 5 | << | Deplasare la stânga pe biţi | 2 | din stânga |
| 5 | >> | Deplasare la dreapta pe biţi | 2 | din stânga |
| 6 | < | Operatorul "mai mic" | 2 | din stânga |
| 6 | <= | Operatorul "mai mic sau egal" | 2 | din stânga |
| 6 | > | Operatorul "mai mare" | 2 | din stânga |
| 6 | >= | Operatorul "mai mare sau egal" | 2 | din stânga |
| 7 | == | Operatorul "egal" | 2 | din stânga |
| 7 | != | Operatorul "diferit" | 2 | din stânga |
| 8 | & | AND (ŞI; conjuncţie) pe biţi | 2 | din stânga |
| 9 | ^ | XOR (SAU exclusiv) pe biţi | 2 | din stânga |
| 10 | | | OR (SAU; disjuncţie) pe biţi | 2 | din stânga |
| 11 | && | AND (ŞI; conjuncţie) logic | 2 | din stânga |
| 12 | || | OR (SAU; disjuncţie) logic | 2 | din stânga |
| 13 | ?: | Operator condiţional (ternar: a ? x : y întoarce x dacă a are valoarea TRUE şi y altfel) | 3 | din dreapta |
| 14 | = | Atribuire simplă | 2 | din dreapta |
| 14 | op= | Atribuire combinată cu un operator binar op, unde op poate fi unul din operatorii  \* / % + - & ^ | << >> | 2 | din dreapta |
| 15 | , | Operatorul "virgulă" (evaluare succesivă) | 2 | din stânga |

Probabil că cele mai multe din aceste operaţii vă sunt deja cunoscute. Să observăm mai întâi că operatorii booleeni *&,* |, ^ au prioritate inferioară operatorilor *==* şi !*=*$. În consecinţă, dacă dorim să scriem expresii care conţin aceşti operatori, trebuie, aşa ca în exemplul de mai jos, în care se face verificarea unei succesiuni de biţi cu ajutorul unei "măşti":

if ((sir\_de\_biti & MASCA) == MASCA) {

...

}

să se închidă operaţiile booleene între paranteze. Operatorii de incrementare/ decrementare ++ şi -- pot fi explicaţi prin următorul exemplu. Dacă avem următoarea secvenţă de instrucţiuni

a = a + 1;

b = a;

putem utiliza operatorul de *preincrementare* b = ++a;

De asemenea, dacă avem instrucţiunile în ordinea inversă

b = a;

a = a + 1;

putem utiliza operatorul de *postincrementare*

b = a++;

Astfel operatorul de preincrementare mai întâi incrementează variabila asociată, apoi întoarce noua valoare, în timp ce operatorul de postincrementare întoarce mai întâi valoarea, apoi incrementerază variabila. Aceleaşi reguli se aplică operatorilor de pre- şi postdecrementare --.

Apelările de funcţii, atribuirile imbricate şi operatorii de incrementare/ decrementare produc efecte colaterale în momentul aplicării. Aceasta poate introduce dependenţe de compilator deoarece ordinea de evaluare în anumite situaţii este dependentă de compilator. Să considerăm următorul exemplu care demonstrează acest lucru:

a[i] = i++;

Depinde de ordinea în care compilatorul este prevăzut să evalueze expresiile din stânga şi din dreapta operatorului de atribuire dacă vechea sau noua valoare a lui *i* este utilizată ca indice în tabloul *a*.

Operatorul condiţional ?: este o prescurtare pentru o instrucţiune **if** utilizată în mod obişnuit. De exemplu, pentru a atribui variabilei *max* cea mai mare dintre valorile variabilelor *a*, *b*, se poate utiliza:

if (a > b)

max = a;

else

max = b;

Aceasta se poate scrie mai scurt:

max = (a > b) ? a : b;

Următorul operator neobişnuit este cel de atribuire combinată cu alt operator. Utilizăm adesea atribuiri de următoarea formă:

expr1 = (expr1) op (expr2)

de exemplu

i = i \* (j + 1);

În aceste atribuiri valoarea din stânga apare şi în partea dreaptă, imediat lângă operatorul de atribuire. C permite ca aceste tipuri de atribuire să fie prescurtate ca:

i\*= j + 1;

Putem face asta cu aproape toţi operatorii binari. De reţinut că operatorul de mai sus implementează forma lungă, deşi "*j+1*" nu este în paranteze. Următorul operator neobişnuit este operatorul ", ". El poate fi explicat cel mai bine printr-un exemplu:

i = 0;

j = (i += 1, i += 2, i + 3);

Acest operator îşi evaluează argumentele de la stânga la dreapta şi întoarce valoare expresiei din dreapta. Astfel, în exemplul de mai sus operatorul evalueză mai întâi "*i+=1*" care, ca un efct colateral, incrementează valoarea lui *i*; apoi este evaluată expresia "*i+=2*" care adaugă *2* la *3,* deci *i* primeşte valoarea *3*. A treia expresie este evaluată şi valoarea sa este întoarsă ca rezultat al operatorului. Astfel, *j* primeşte valoarea *b*.

Operatorul virgulă introduce o anumită "capcană" atunci când se utilizează tablouri *n*-dimensionale cu *n>1*. O eroare frecventă este utilizarea unei liste de indici separaţi prin virgule:

int matrice[10][5]; // matrice bidimensionala

int i;

...

i = matrice[1, 2]; // NU FUNCTIONEAZA!!

i = matrice[1][2]; // BINE

În realitate, la prima atribuire, lista de indici separaţi prin virgule este interpretată ca operator virgulă, ceea ce conduce la o atribuire a adresei celei de a treia secvenţe de cinci elemente a matricii! Limbajul C ignoră valorile neutilizate ale expresiilor. De exemplu, se poate scrie secvenţa:

ix = 1;

4711;

jx = 2;

dar linia a doua, deşi întoarece o valoare (*4711*), nu are nici un efect. Trecând peste aceste lucruri oarecum stranii, vom lua în considerare lucrurile cu adevărat utile. În continuare vom discuta despre *funcţii.*

**6.2.1.4 Funcţii**

Deoarece C este un limbaj procedural, el permite definirea *funcţiilor.* Procedurile sunt simulate prin funcţii care nu întorc valori, mai exact, întorc un tip special numit **void**. Funcţiile sunt declarate ca şi variabile, dar trebuie să aibă argumente cuprinse între paranteze (chiar dacă nu există argumente, parantezele trebuie specificate):

int sum(int to);/\* Declaratia functiei sum cu un argument \*/

int bar(); /\* Declaratia functiei bar fara argumente \*/

void foo(int ix, int jx)/\* Declaratia functiei foo cu doua

argumente \*/

Pentru a defini efectiv o funcţie, se adaugă corpul său: C permite numai transmiterea argumentelor prin valori. În consecinţă nu se poate schimba valoarea unui argument al unei funcţii. Dacă doriţi să transmiteţi un argument prin referinţă, trebuie să transmiteţi ca valoare a argumentului corespunzător al funcţiei, *adresa* variabilei a cărei valoare doriţi să fie schimbată, deci trebuie să utilizaţi *pointeri*.

**6.2.1.5 Pointeri şi tablouri**

Una din cele mai obişnuite probleme ale programării in C (şi uneori în C++) este înţelegerea pointerilor şi tablourilor. În C (C++) aceste concepte sunt foarte înrudite, cu unele mici (dar esenţiale) diferenţe.

Se declară un pointer punând un asterisc între tipul de date şi numele variabilei sau a funcţiei:

char \*strp; /\* strp este 'pointer la char' \*/

Accesăm conţinutul unui pointer utilizând din nou un asterisc:

\*strp = 'a'; /\* Un singur caracter \*/

Ca şi în alte limbaje, trebuie să furnizăm un anumit spaţiu pentru valoarea la care trimite un pointer. Un pointer la caractere poate fi utilizat pentru a indica un şir de caractere. Şirurile de caractere în C se termină prin caracterul special NULL (*0* sau constanta de tip *char* *'0'*). Astfel, putem avea şiruri de orice lungime. Şirurile sunt închise între ghilimele:

strp = "salut";

În acest caz compilatorul adaugă automat caracterul NULL la sfârşit. Astfel, *strp* trimite la un şir de 6 caractere: primul este 's', al doilea 'a', etc. Putem accesa caracterele cu un indice în *strp*:

strp[0] /\* s \*/

strp[1] /\* a \*/

strp[2] /\* l \*/

strp[3] /\* u \*/

strp[4] /\* t \*/

strp[5] /\* \0 \*/

Primul caracter este egal şi cu *\*strp*, ceea ce poate fi scris şi ca *\*(strp+0).* Aceasta reprezintă ceea ce se numeste *aritmetica pointerilor* şi este una din cele mai puternice caracteristici ale limbajului C. Astfel, avem urmatoarele egalităţi:

\*strp == \*(strp + 0) == strp[0]

\*(strp + 1) == strp[1]

\*(strp + 2) == strp[2]

...

De reţinut că aceste egalităţi sunt adevarate pentru orice tip de date. Adunarea **nu** este orientată spre octeţi, ci este orientată spre lungimea respectivului tip de pointer!

Pointerul *strp* poate să trimită şi la alte locaţii de memorie. Destinaţia sa este *schimbătoare.* Spre deosebire de aceasta, tablourile sunt pointeri *ficşi.* Ei trimit spre o zona predefinită de memorie care este specificată între paranteze drepte:

char str[6];

Putem să considerăm *str* ca o constantă pointer care trimite la o zonă cu 6 caractere. **Nu** este permis să fie utilizat astfel:

str = "salut"; /\* EROARE \*/

deoarece aceasta ar însemna schimbarea pointerului pentru a trimite la 's'. Trebuie să **copiem** şirul în zona de memorie pentru date. Putem utiliza o funcţie din biblioteca C standard, numită *strcpy().*

strcpy(str, "salut"); /\* E bine \*/

De reţinut că, exceptând atribuirea, se poate folosi *str* oriunde poate fi folosit un pointer de caracter, deoarece el este un pointer (fix).

**6.2.1.6 Pointeri la funcţii**

O caracteristică a limbajului C este posibilitatea de a se defini variabile **pointer la funcţie**. O astfel de variabilă poate primi ca valoare adresa la care este plasat în memorie în timpul execuţiei programului codul unei funcţii care are un anumit prototip (constând, după cum se ştie, din tipul rezultatului şi tipurile argumentelor funcţiei) specificat prin declaraţia de variabilă. Sintaxa unei astfel de declaraţii este:

*tip* (\**nume*) (*lista\_de\_tipuri*)

unde *tip* este un tip oarecare sau **void**, reprezentând tipul de rezultat întors funcţiile a căror adresă poate fi valoare a pointerului declarat,  *nume* identificatorul care desemnează pointerul, iar *lista\_de\_tipuri* este vidă sau conţine un singur tip sau este o succesiune de tipuri separate de virgule, aşa cum se utilizează într-o declaraţie (prototip) de funcţie.

Iată câteva exemple de astfel de declaraţii:

int (\*pf-int1)(float, int); // este intors tipul int

void (\*pointf)(int, int, \*char); // nu exista rezultat

double (\*pf-d)(int); // este intors tipul double

char \*(\*pf-int1)(); // este intors pointer la char

Daca se declară o funcţie, de exemplu

int f-int(float, int);

un pointer declarat conform cu prototipul acestei funcţii poate primi ca valoare, cu ajutorul operatorului de atribuire, adresa din memorie a funcţiei, astfel:

pf-int1=f-int;

Se observă că numele unei funcţii este un pointer constant care are ca valoare adresa funcţiei.

Cu un pointer la funcţie se poate construi, utilizându-se operatorul de apel de funcţie, o expresie al cărei efect este apelarea funcţiei a cărei adresă a fost în prealabil atribuită ca valoare pointerului. Astfel, expresia

pf-int1(1.5, -2)

poate fi utilizată, după executarea atribuirii de mai sus, în locul expresiei

f-int(float, int).

Pointerii la funcţie permit elaborarea unor programe deosebit de flexibile, ei permiţând ca un program să stabilească în mod dinamic (adică în timpul execuţei) dacă va fi apelată o funcţie sau alta într-un anumit punct al său. De asemenea, ei lărgesc considerabil posibilităţile de programare deoarece pot fi utilizaţi ca argumente ale unor funcţii sau ca elemente ale unor structuri.

Se pot defini tablouri de pointeri la funcţii şi poiteri la pointeri la funcţii, aşa cum ilustrăm prin exemplele de mai jos:

int (\*tab-pf[5]) (char \*, int); // tablou cu 5 elemente pointer la

// functii

float (\*\*ppf) (double, int, int); // pointer la pointer la functie

ceea ce poate adăuga elemente importante de flexibilitate programelor.

**6.2.1.7 Un prim program**

Introducem aici primul program, destul de des folosit: un program care afişează "Salut, lume!" pe ecran:

#include <stdio.h>

/\* Variabilele globale trebuie sa fie aici \*/

/\* Definitiile de functii trebuie sa fie aici \*/

int main() {

puts("Salut, lume!");

return 0;

} /\* main \*/

Prima linie arată oarecum straniu. Explicarea ei necesită anumite informaţii despre cum sunt tratate programele C (şi C++) de către compilator. În linii mari pasul de compilare este împărţit în doi paşi. Primul pas este numit "preprocesare" şi este utilizat pentru pregătirea unui cod C pur. Acest pas tratează prima linie ca pe o comanda de a include un fişier numit *stdio.h* în sursă. Parantezele unghiulare indică faptul că fişierul urmează să fie căutat pe calea standard de căutare configurată pentru calculator. Fişierul însuşi furnizează anumite declaraţii sau definiţii pentru intrările/ieşirile standard. De exemplu, declară o funcţie numită *put().* În pasul de preprocesare sunt, de asemenea, şterse comentariile.

În cel de al doilea pas, codul C generat în primul pas este compilat şi se crează cod executabil. Fiecare cod executabil trebuie să conţină o funcţie numită *main().* Această funcţie este apelată atunci când programul este pornit (adică lansat în execuţie). Acestă funcţie întoarce un întreg care reprezintă starea de ieşire a programului.

Funcţia *main()* poate să aibă argumente care reprezintă argumentele din linia de comandă. Îi arătăm aici, dar nu vom explica felul cum pot fi utilizaţi. Pentru astfel de explicaţii se pot consulta [2], [3]:

#include <stdio.h>

int main(int argc, char \*argv[]) {

int ix;

for (ix = 0; ix < argc; ix++)

printf("Argumentul meu nr. %d este %s\n", ix, argv[ix]);

return 0;

} /\* main \*/

Primul argument, *argc* întoarce numărul de argumente din linia de comandă. Argumentul al doilea, *argv* este un tablou de şiruri de caractere (reamintim că şirurile sunt reprezentate prin pointeri la caractere. Astfel, *argv* este un tablou de pointeri la caractere).

**6.2.2 De la C la C++**

Această secţiune prezintă mai întâi unele incompatibilităţi între limbajele C şi C++, apoi extensiile limbajului C care au fost introduse de C++. De asemenea sunt arătate conceptele orientate pe obiecte şi implementarea lor.

**6.2.2.1 Incompatibilităţi între limbajele C şi C++**

Deşi C++ a fost conceput ca extensie a limbajului C, există construcţii în C care nu sunt admise sau sunt interpretate diferit în C++. Numim aceste diferenţe între cele două limbaje **incompatibilităţi.** Prezentăm mai jos pe scurt pe cele mai importante dintre ele.

1. Constantele caracter (caracter cuprins între apostrofuri) au în C tipul **int**, iar în C++ au tipul **char** şi sunt reprezentate pe un octet. Prin urmare,

sizeof('a')

are în C valoarea 2, în implementarea Borland C++, respectiv valoarea 4 în implementarea Unix, iar în C++ are valoarea 1 în ambele implementări.

1. În C++ un salt cu instrucţiunea goto nu poate trece peste o declaraţie cu iniţializare decât dacă trece peste întregul bloc al declaraţiei. De exemplu, programul

/\* In C este permis, dar in C++ nu este permis saltul peste o

declaratie cu initializare daca nu se face peste intregul bloc

care o contine.\*/

#include <stdio.h>

void main(void)

{ int a;

printf("a="); scanf("%d", &a);

if (a>10) goto et; // eticheta din blocul interior

{ int b=10;

printf("b="); scanf("%d", &b); a+= b;

et: printf("a=%d b=%d\n", a, b);

}

}

este corect în C, dar nu este corect în C++, deoarece instrucţiunea **goto** trece peste declaraţia cu iniţializare din blocul interior, fără a trece peste tot acest bloc.

1. C++ admite numai conversia de tip de la un pointer de un tip oarecare la tipul **void, nu** şi conversia în sens invers, spre deosebire de C. În C++ conversia în sens invers se poate face numai cu operatorul de conversie explicită (notat **(***tip***)** ). Dăm câteva exemple:

void\* pv;

int\* pi;

pv=pi; /\*corect in C si in C++\*/

pi=pv; /\*corect in C, incorect in C++ \*/

pi=(int\*)pv; /\* corect in C si in C++ \*/

1. În C++ este oblgatorie existenţa prototipului sau a definiţiei unei funcţii înaintea apelării sale şi se acceptă numai prototipuri de funcţii în care se specifică lista argumentelor. Prin urmare, inexistenţa listei de argumente în prototip este interpretată în C++ ca o declaraţie de funcţie fără argumente, nefiind necesar ca lista să fie înlocuită cu cuvântul cheie **void**. În C se pot specifica argumente efective la apelarea unei funcţii al cărei prototip nu specifică argumente, fiind necesar, pentru ca lista argumentelor să fie considerată vidă, să se specifice **void** explicit în prototip.
2. Limbajul C permite ca funcţia *main()* să fie apelată de o altă funcţie şi chiar de ea însăşi, dar în C++ acest lucru este interzis. Dăm mai jos două exemple de programe: unul în care funcţia *main()* este apelată de altă funcţie

/\* In C functia main() poate fi apelata de alta functie (apel recursiv indirect). In C++ acest lucru este interzis. \*/

#include<stdio.h>

#include<conio.h>

#include<process.h>

#include<stdlib.h>

int a=10;

void main(void);

void f(void) { a--; main();}

void main(void)

{

if (a) {printf("\na=%d ", a); f();} else exit(getch());

}

şi unul în care este apelată de ea însăşi

/\* In C functia main() se poate apela singura.

In C++ acest lucru este interzis. \*/

#include<stdio.h>

#include<conio.h>

#include<process.h>

#include<stdlib.h>

int a=10;

void main(void)

{

if (a) {printf("\na=%d ", a); a--; main();} else exit(getch());

}

Dacă sunt compilate cu un compilator de C, compilarea se încheie cu succes şi, la execuţie fiecare din programe va afişa

10 9 8 7 6 5 4 3 2 1

şi se va opri după apăsarea unei taste. Un compilator de C++ nu va accepta nici unul din aceste programe.

1. După cum se va vedea la 6.2.3 (Paşii compilării), programele pot fi alcătuite din mai multe fişiere sursă. Un identificator declarat (şi, eventual iniţializat) într-un fişier poate avea aceeaşi semnificaţie în întregul program (ceea ce înseamnă că va ocupa o anumită zonă de memorie în programul executabil) sau va fi recunoscut doar la nivelul fişierului în care este declarat (deci, dacă va apărea o nouă declarare a sa în alt fişier, acestei noi declarări îi va corespunde altă zonă de memorie în programul executabil). Se spune că identificatorii din prima categorie sunt cu **legătură externă**, iar cei din a doua categorie sunt cu **legătură internă**. Atribuirea de memorie în funcţie de tipul de legătură se face prin acţiunea compilatorului şi a editorului de legături. Pentru variabilele globale care sunt declarate cu modificatorul **const** (adică nu îşi pot schimba valoarea) legătura implicită este în C *externă*, iar în C++ *internă*. Legătura internă pentru o variabilă globală se declară cu modificatorul **static**, iar cea externă cu modificatorul **extern**. În concluzie, există anumite echivalenţe între declaraţiile de variabile globale cu modificatorul **const**, pe care le arătăm mai jos:

/\* C \*/ /\* C++ \*/

static const int n; const int n;

const int n; extern const int n;

**6.2.2.2 Extensii de bază**

Aici vom prezenta extensii ale conceptelor deja introduse de C. C++ prezintă un nou comentariu care este introdus de două slash-uri (//) şi care ţine până la sfârşitul liniei. Putem utiliza amândouă stilurile de comentariu, ca în exemplul de mai jos:

/\* comentariul C poate sa includa // si poate sa se

intinda peste mai multe linii. \*/

// /\* Acesta este stilul de comentariu C++ \*/ pana la sfarsitul liniei

În C variabilele trebuie definite la începutul unui bloc sau în afara oricărui bloc. C++ ne permite să definim variabilele oriunde în interiorul unui bloc. Astfel, variabilele şi obiectele pot fi definite acolo unde sunt utilizate.

**6.2.2.2.1 Tipuri de date**

C++ introduce un nou tip de date numit *referinţă.* Putem gândi că variabilele de acest tip sunt "pseudonime" ale unor variabile sau obiecte "reale". Aşa cum un pseudonim nu poate exista fără corespondentul său real, nu pot fi definite referinţe singure. Caracterul & (ampersand) este utilizat pentru definirea referinţelor. De exemplu:

int ix; /\* ix este variabla "reala" \*/

int &rx = ix; /\* rx este "pseudonim" pentru ix \*/

ix = 1; /\* de asemenea rx == 1 \*/

rx = 2; /\* de asemenea ix == 2 \*/

Referinţele pot fi utilizate ca argumente de funcţii şi valori de întoarcere. Aceasta permite să se transmită argumentele prin referinţă sau să se întoarcă o modalitate de a accesa variabilele sau obiectele calculate.

Tabelul 6.4 ne oferă o trecere în revistă a declaraţiilor posibile. Ea nu este completă, prin faptul că nu conţine toate combinaţiile posibile. Totuşi acestea sunt probabil cele mai frecvent utilizate.

Tabelul 6.4

|  |  |  |
| --- | --- | --- |
| **Declaraţie** | nume **este …** | **Exemplu** |
| ***tip*** nume; | de tipul ***tip*** | **int num;** |
| ***tip*** nume[]; | tablou (deschis) de ***tip*** | **int num[];** |
| ***tip*** nume[*n*]; | tablou cu *n* elemente de tipul ***tip***  (nume[*0*], nume[*1*], …, nume[*n-1*]) | **int num[5];** |
| ***tip*** \*nume; | pointer la tipul ***tip*** | **int \*num;** |
| ***tip*** \*nume[]; | tablou (deschis) de pointeri la tipul ***tip*** | **int \*num[];** |
| ***tip*** (\*nume[]); | tablou (deschis) de pointeri la tipul ***tip*** | **int \*(num[]);** |
| ***tip*** (\*nume)[]; | pointer la tablou (deschis) de tipul ***tip*** | **int (\*num)[];** |
| ***tip*** &nume; | referinţă la tipul ***tip*** | **int &num;** |
| ***tip*** nume( ); | funcţie care întoarce tipul ***tip*** | **int num();** |
| ***tip*** \*nume( ); | funcţie care întoarce pointer la tipul ***tip*** | **int \*num();** |
| ***tip*** \*(nume( )); | funcţie care întoarce pointer la tipul ***tip*** | **int \*(num());** |
| ***tip*** (\*nume)( ); | pointer la funcţie care întoarce tipul ***tip*** | **int (\*num)();** |
| ***tip*** &nume( ); | funcţie care întoarce referinţă la tipul tip | **int &num();** |

În C şi C++ se poate utiliza modificatorul **const** pentru a se declara anumite aspecte ale unei variabile (sau obiect) ca fiind constante. Următorul tabel 6.5 listează posibilele combinaţii şi descrie semnificaţia lor.

Tabelul 6.5

|  |  |
| --- | --- |
| **Declaraţie** | nume **este …** |
| **const** ***tip*** nume = ***valoare***; | constantă de tipul ***tip*** |
| ***tip*** **\*const** nume **= *valoare***; | pointer constant la tipul ***tip*** |
| **const** ***tip*** \*nume = ***valoar****e*; | pointer (variabil) la constantă de tipul ***tip*** |
| **const** ***tip*** \***const** nume = ***valoare***; | pointer constant la constantă de tipul ***tip*** |

Vom investiga unele exemple de variabile declarate cu modificatorul **const** împreună cu utilizarea lor. Să considerăm următoarele declaraţii:

int i; // un intreg obisnuit

int \*ip; // pointer neinitializat la

// intreg

int \* const cp = &i; // pointer constant la intreg

const int ci = 7; // intreg constant

const int \*cip; // pointer la intreg constant

const int \* const cicp = &ci; // pointer constant la constanta

// intreaga

Următoarele atribuiri sunt **corecte** :

i = ci; // atribuie o constanta intreaga unui intreg

\*cp = ci; // atribuie o constanta intreaga unei variabile

// care este referita de un pointer constant

cip = &ci; // schimba valoarea unui pointer la constanta

// intreaga

cip = cicp; // unui pointer la constanta intreaga i se

// atribuie valoarea unui pointer constant la un

// intreg constant

Următoarele atribuiri sunt **incorecte**:

ci = 8; // nu se poate schimba valoarea unui intreg

// constant

\*cip = 7; // nu se poate schimba un intreg constant referit

// de un pointer

cp = &ci; // nu se poate schimba valoarea unui pointer

// constant

ip = cip; // aceasta ar permite sa se schimbe valoarea

// intregului constant \*cip cu \*ip

Când se utilizează referinţele, trebuie avute în vedere anumite particularităţi. Să vedem următorul exemplu de program:

#include <stdio.h>

int main() {

const int ci = 1;

const int &cr = ci;

int &r = ci; // creaza un intreg temporar pentru referinta

// cr = 7; // nu se poate atribui valoare unei referinte

// constante

r = 3; // schimba valoarea intregului temporar

printf("ci == %d, r == %d, cr==%d\n", ci, r, cr);

return 0;

}

Dacă se compilează cu GNU C++, compilatorul afişează următorul avertisment:

conversion from 'const int' to 'int &' discards const

Ceea ce se întâmplă de fapt este crearea de către compilator a unei variabile întregi temporare cu valoarea lui *ci* cu care este iniţializată referinţa lui *r*. Prin urmare, când se schimbă *r*, valoarea variabilei temporare este cea care se schimbă, şi nu *ci*. Această variabilă temporară există atât timp cât există referinţa *r*.

Alte compilatoare (de exemplu TURBO C++, BORLAND C++) fac acelaşi lucru, dar nu afişează nici un avertisment.

Referinţa *cr* este definită ca *read-only* (referinţă constantă). Aceasta interzice utilizarea ei în partea stângă a atribuirilor. Dacă am şterge marcajul comentariu (//) din faţa liniei care conţine o astfel de atribuire interzisă, compilatorul ar da un mesaj de eroare.

**6.2.2.2.2 Funcţii**

C++ permite supraîncărcarea aşa cum s-a definit la sectiunea 5.3. De exemplu, putem să definim două funcţii *max()*, din care una întoarce cel mai mare dintre doi întregi, iar cealaltă întoarce cel mai mare dintre două şiruri de caractere.

#include <stdio.h>

int max(int a, int b) {

if (a > b) return a;

return b;

}

char \*max(char \*a, char \* b) {

if (strcmp(a, b) > 0) return a;

return b;

}

int main() {

printf("max(19, 69) = %d\n", max(19, 69));

printf("max(abc, def) = %s\n", max("abc", "def"));

return 0;

}

Programul din exemplul de mai sus defineşte aceste două funcţii care diferă prin listele lor de argumente, deci sunt definite două funcţii diferite. Prima apelare a funcţiei *printf()* din funcţia *main()* produce o apelare a primei versiuni a *max()*, deoarece are ca argumente doi întregi. La fel, a doua apelare a funcţiei *printf()* produce o apelare a celei de a doua versiuni a *max()*.

Se pot utiliza referinţe pentru a se obţine o funcţie cu un pseudonim (alias) al unui argument efectiv de la apelarea unei funcţii. Acest fapt permite să se schimbe valoarea argumentului efectiv şi este cunoscut şi la alte limbaje cu transfer prin referinţă a argumentelor:

void foo(int prinValoare, int &prinReferinta) {

prinValoare = 42;

prinReferinta = 42;

}

void bar() {

int ix, jx;

ix = jx = 1;

foo(ix, jx);

/\* ix == 1, jx == 42 \*/

}

Utilizarea referinţelor la întoarcerea valorilor funcţiilor se specifică prin plasarea simbolului & după tipul valorii întoarse. În acest caz orice operaţie efectuată cu valoarea întoarsă de funcţie va utiliza referinţa la variabila plasată după instrucţiunea return care a produs oprirea execuţiei funcţiei şi întoarcerea valorii. Aceasta permite ca astfel de apleuri de funcţii să fie utilizate şi în membrul stâng al unei atribuiri.

Exemplu:

#include <iostream.h>

int a;

int & schimba(int b)

{

a+=b;

cout<<"in schimba a="<<a<<endl;

return a; // se intoarce referinta la variabila globala a schimbata de

// functie

}

int main()

{

schimba(2)+=3; // se schimba valoarea lui a de catre functie si de

// catre atribuire

cout<<"in main a="<<a<<endl;

}

Întoarcerea referinţelor la variabile locale (sau temporare, de exemplu rezultatul unor expresii) din blocul funcţiei este incorectă (semnalată de obicei de compilator cu un avertisment) deoarece în acest caz variabila locală este eliminată imediat după execuţia funcţiei, deci este eliminată şi referinţa, iar valoarea nu mai este întoarsă în funcţia apelantă. De notat că, dacă nu s-ar fi folosit referinţă ca valoare de întoarcere, varabila locală sau temporară respectivă ar fi fost eliminată după transmiterea rezultatului în funcţia apelantă.

**6.2.2.2.3 Alte extensii de bază**

Pe lângă extensiile de bază prezentate mai sus, C++ oferă şi altele, dintre care prezentăm mai jos pe cele mai frecvent folosite, având în vedere şi utilitatea acestora în scrierea de programe mai eficiente şi mai clare.

**Utilizarea identificatorilor declaraţi const în iniţializări şi dimensiuni de tablouri**

Expresiile care conţin identificatori declaraţi const pot fi folosite în C++ pentru iniţializări şi dimensionări de tablouri, fapt interzis în C. Exemple:

const int n=20, m=90;

int a=2\*n;

float vect[n+1]; mat[n+2][3\*m]

**Argumente cu valori implicite**

În C++ există posibilitatea declarării funcţiilor cu valori implicite ale argumentelor. La apelarea unei astfel de funcţii se poate omite specificarea argumentelor efective pentru acei argumente formale care au declarate valori implicite şi se transferă automat valorile efective:

#include<iostream.h>

void func (int, int=10);

void main()

{ cout<<"func(5, 99)"

func(5, 99); //apel normal

cout<<"func(5), ";

func(5); //apel cu un singur argument

}

void func(inti, intj)

{ cout <<"argumente:i="<<i;

cout<<"j="<<j<<"\n";

}

Programul afişează:

func(5, 99), argumente: i=5j=99

func(5), argumente:i=5j=10

Un apel fără argumente (*func()*; ) este eronat deoarece argumentul i nu are asociată o valoare implicită.

**Alocarea/eliberarea dinamică a memoriei folosind operatorii new şi delete**

Alocarea şi eliberarea dinamică a memoriei se face în C cu anumite funcţii de bibliotecă (*malloc()*, *free()* şi altele). Pe lângă acestea, în C++ se introduc operatorii unari **new** (pentru alocare) şi **delete** (pentru eliberare). Operatorul **new** poate fi utilizat în următoarele forme:

*pt\_la\_tip* = new *tip*;

*pt\_la\_tip* = new *tip*(*val\_init*);

*pt\_la\_tip* = new *tip*[*n*];

unde:

- *tip* = tipul variabilei dinamice, care poate fi un tip de date oarecare;

* *pt\_la\_tip* = o variabilă pointer de tip *tip*;
* *val\_init* = expresie cu a cărei valoare se iniţializează variabila dinamică.

Varianta a treia se utilizează pentru alocarea memoriei pentru *n* elemente de tipul *tip* (un tablou cu *n* elemente) şi *n* este o expresie întreagă. Se pot aloca şi tablouri multidimensionale, specificând toate dimensiunile. Iniţializarea tablourilor nu este posibilă. Operatorul **new** alocă spaţiul necesar, corespunzător tipului, şi oferă ca rezultat:

-dacă alocarea a reuşit un pointer de tipul **(***tip* **\*)** conţinînd adresa zonei de memorie alocate; -în caz contrar (memorie insuficientă sau fragmentată), un pointer cu valoarea NULL (=0).

Exemplul următor ilustrează sintaxa pentru toate variantele:

int n=15, \*pi1, \*pi2, \*pit1, \*pit2;

pi1=new int;

//variabila intreaga initializata

pi2=new int(23)

//variabila intraga initializata cu 23

pit1=new int[330];

//tablou unidimensional int

pit2=new int[n][3\*n];

//tablou bidimensional int

Operatorul **delete** are sintaxa

delete *pt\_la\_tip*;

unde *pt\_la\_tip* este adresa obţinută în urma unei alocări cu operatorul **new**, utilizarea altei valori fiind ilegală, comportarea programului într-o astfel de situaţie fiind nedefinită.

Ca şi în cazul utilizării funcţiilor de alocare şi eliberare dinamică a memoriei, este necesar ca, după ce unei variabile pointer i s-a atribuit o adresă cu operatorul **new**, să nu i se atribuie altă adresă şi să nu se iasă din blocul în care pointerul este declarat decât după eliberarea vechii zone de memorie cu operatorul **delete**. În caz contrar, vechea zonă de memorie va rămâne ocupată, făra a mai putea fi utilizată, ceea ce poate duce, mai ales prin repetarea în cursul execuţiei a operaţiilor de mai sus la mari consumuri inutile de memorie. De reţinut că grşelile de mai sus nu sunt semnalate în cursul comiplării sau al execuţiei.

**Funcţii "inline"**

Directiva preprocesor **#define** oferă posibilitatea utilizării macrodefiniţiilor cu argumente. Pentru fiecare apelare preprocesorul inserează în textul programului, în locul apelului, textul macrodefiniţiei în care se substituie numele argumentelor formale cu textele care reprezintă argumentele efective (de reţinut că este o simplă înlocuire de texte, fără nicio verificare de corectitudine sintactică). Textul rezultat este preluat apoi de compilatorul propiu\_zis. Princialul avantaj al macrodefiniţiei faţa de funcţie este obţinerea unui timp de execuţie mai scurt, deoarece codul obiect este inserat efectiv în program şi se elimină operaţiile asociate apelului unei funcţii (salvări şi restabiliri de registre ale unei microprocesorului, transfer de argumente prin stivă, etc), iar principalul dezavantaj este creşterea dimensiunii codului obiect generat de compilator, deoarece pentru fiecare apel se inserează codul sursă corespunzător instrucţiunilor din macrodefiniţie. Procedeul este avantajos atunci când codul obiect rezultat din compilarea macrodefiniţiei nu este prea lung şi se fac puţine inserări de macrodefiniţii, dar, în momentul execuţiei, se fac multe execuţii ale operaţiilor apărute ca urmare a utilizării macrodefiniţiilor, ceea ce se poate întâmpla dacă substituirile de macrodefiniţii se fac în interiorul blocurilor unor instrucţiuni repetitive.

Un dezavantaj special este constituit de faptul că "expresiile" din macrodefiniţii sunt de fapt simple texte care nu sunt supuse nici verificării corectitudinii, nici nu sunt evaluate înainte de substituire, ceea ce poate duce la efecte nedorite.

Un exemplu simplu îl reprezintă macrodefiniţia:

#define patrat(x) x\*x

...

int i, j, rez;

rez=patrat(i); //cirect:rez=i\*i;

rez=patrat(i+1); //eroare:rez=i+1\*i+1, deci

//rez=2\*i+1

care, după cum se vede, nu reuşeşte să definească o funcţie care să calculeze pătratul unei expresii. Eroarea de mai sus poate fi înlăturată definind *patrat(x)* astfel:

#define patrat(x) (x)\*(x)

...

rez=patrat(i+1); //corect:rez=(i+1)\*(i+1);

Nici acum însă nu se elimină toate posibilităţile de eroare, ceea ce se vede mai jos:

rez=patrat(++i); //eroare :rez=(++i)\*(++i),

//dubla incrementare

C++ oferă posibilitatea declarării funcţiilor "**inline**" care combină avantajele funcţiilor propriu-zise cu cele ale macrodefiniţilor. Definiţiile acestora sunt compilate ca şi cele ale funcţiilor obişnuite, dar, dar, spre deosebire de funcţiile obişnuite, la fiecare apelare codul obiect al funcţiei **inline** este inserat în codul programului de către compilator. În acest fel, ca şi în cazul macrodefiniţiilor se elimină operaţiile aferente apelului. Funcţiile **inline** păstrează însă toate proprietăţiile funcţiilor în privinţa verificării validităţii apelurilor, modului de calcul şi transfer al argumentelor, etc.

Se păstrează avantajul creşterii vitezei, da şi dezavantajul creşterii dimensiunii codului. Nu mai este posibilă compilarea separată a unei funcţii **inline** sau utilizarea unui pointer către o asemenea funcţie.

Funcţiile **inline** se definesc şi se urilizează similar cu cele obişnuite, cu excepţia adăugării specificatorului **inline**. Problema din exemplul de mai sus se poate rezolva complet cu o funcţie **inline**, după cum urmează:

inline int patrat(int x){ return x\*x; }

...

rez=patrat(++i);

**Operatorul de rezoluţie**

În C++ este definit operatorul de rezoluţie (**: :** ) care permite accesul la un identificator global, dintr-un bloc în care acesta nu este vizibil datorită unei redeclarări. Se mai numeşte operator de acces. De exemplu:

char sir[20]="Sir global";

void func()

{

char \*sir; //variabila locala

sir="Sir local";

puts(::sir); //afiseaza sirul global

puts(sir); //afiseaza sirul local

}

**6.2.2.3 Extensii orientate pe obiecte**

În această secţiune prezentăm felul cum conceptele orientate pe obiecte din capitolul 3 sunt utilizate la C++.

**6.2.2.3.1 Clase şi obiecte**

C++ permite declarări şi definiri de clase. Instanţierile claselor se numesc *obiecte*. Reamintim programul de desenare dat ca exemplu în capitolul 4. Acolo am construit o clasă *Punct*. În C++ aceasta arată aşa:

class Punct {

int \_x, \_y; // coordonatele punctului

public: // inceputul sectiunii interfata

void setX(const int val);

void setY(const int val);

int getX() { return \_x; }

int getY() { return \_y; }

};

Punct unpunct;

Se declară o clasă *Punct* şi se declară un obiect *unpunct*. Putem să gândim o definiţie de clasă ca pe o definiţie de structură cu funcţii (sau "metode"). În plus, se pot specifica mai detaliat *"drepturi de acces".* De exemplu *\_x* şi *\_y* sunt *private,* deoarece datele membre ale claselor sunt private. Prin urmare, trebuie să se "comute" explicit drepturile de acces pentru a se declara urmatoarele ca *publice*. Aceasta se face prin utilizarea cuvântului cheie *public* urmat de ":". Orice membru care urmează acest cuvânt cheie este accesibil din afara clasei.

Putem să comutăm iarăşi la drepturi de acces private începând o secţiune privată cu cuvântul cheie *private.* Acest lucru este posibil de câte ori este necesar:

class Foo {

// private implicit...

public:

// ceea ce urmeaza este public pana...

private:

//...aici, unde se revine la private...

public:

//... si iarasi la public.

};

Amintim că o structură *struct* este o combinaţie de diferite date membre care sunt accesibile din afară. Putem acum să exprimăm o structură cu ajutorul unei clase, în care toţi membrii sunt declarate ca publice:

class Struct {

public: // Membrii structurii sunt implicit publici

// date, metode

};

C++ face exact acelaşi lucru cu *struct*. Structurile sunt tratate ca şi clasele. În timp ce membrii claselor (definite cu *class*) sunt implicit privaţi, membrii structurilor (definite cu *struct*) sunt publici. Totuşi, putem să utilizăm de asemenea *private* : pentru a comuta la o secţiune privată în structuri.

Să revenim la clasa noastră *Punct*. Interfaţa ei începe cu secţiunea publică în care definim patru metode, câte două pentru fiecare coordonată pentru a atribui şi, respectiv, a furniza valoarea sa. Metodele pentru a atribui valori sunt numai declarate. Funcţionalitatea lor efectivă urmează să fie definită. Metodele pentru furnizarea valorii au un corp de funcţie. Ele sunt definite în *interiorul clasei,* sau cu un termen englez uzual, sunt *metode inline.*

Acest tip de definire a metodelor este util pentru corpuri de funcţie mici şi simple. El de asemenea îmbunatăţeşte performanţa, deoarece corpurile metodelor inline sunt "copiate" în cod oriunde apare o apelare a unei astfel de metode.

Dimpotrivă, apelările metodelor de obţinere a valorii vor avea ca rezultat o apelare de funcţie "adevarată". Definim aceste metode în afara declaraţiei de clasă. Aceasta face necesar să se declare cărei clase aparţine o anumită definiţie de metodă. De exemplu, o altă clasă ar putea să definească o metodă *setX()* care este diferită de aceea din *Punct*. Trebuie să putem defini *domeniul* de definiţie; utilizăm deci operatorul de rezoluţie "**::**":

void Punct::setX(const int val) {

\_x = val;

}

void Punct::setY(const int val) {

\_y = val;

}

Aici definim metoda *setX()* (*setY()*) în interiorul domeniului clasei *Punct*. Obiectul *unpunct* poate să utilizeze aceste metode pentru a a-şi atribui şi a furniza informaţii despre el însuşi:

Punct unpunct;

unpunct.setX(1); // Initializare

unpunct.setY(1);

//

// x este necesar incepand de aici, deci il definim aici si

// il initializam cu coordonata x a lui unpunct

//

int x = unpunct.getX();

Apare întrebarea: cum "ştiu" metodele din ce obiect sunt invocate? Aceasta se face prin transferul implicit al unui pointer la obiectul care invocă metoda. Putem accesa acest pointer în înteriorul metodei prin cuvântul cheie **this**. Definiţiile metodelor *setX()* şi *setY()* utilizează membrii clasei *\_x* şi *\_y* respectiv. Dacă sunt invocaţi de un obiect, aceşti membri sunt "automat" atribuiţi obiectului corect. Putem să utilizăm *this* pentru a ilustra ce se întamplă de fapt:

void Punct::setX(const int val) {

this->\_x = val; // Aceasta se utilizeaza pentru a referi

// obiectul invocant

}

void Punct::setY(const int val) {

this->\_y = val;

}

Aici utilizăm explicit pointerul *this* pentru a dereferi explicit obiectul invocant. Din fericire, compilatorul inserează automat aceste dereferiri ale membrilor clasei, deci putem să utilizăm efectiv primele definiţii ale metodelor *setX()* şi *setY()*. Totuşi, uneori are sens să se ştie că există un pointer *this* disponibil care indică obiectul invocant.

În mod curent avem nevoie să apelăm metodele de atribuire pentru a iniţializa un obiect punct. Totuşi, am putea dori să iniţializăm punctul atunci când îl definim. Utilizăm, prin urmare, metode speciale numite *constructori*.

**6.2.2.3.2 Constructori**

Constructorii sunt metode pentru a iniţializa un obiect în momentul definirii sale. Lărgim clasa noastră *Punct* astfel încât să iniţializeze un punct cu coordonatele *(0, 0)*:

class Punct {

int \_x, \_y;

public:

Punct() {

\_x = \_y = 0;

}

void setX(const int val);

void setY(const int val);

int getX() { return \_x; }

int getY() { return \_y; }

};

Constructorii au acelaşi nume cu clasa (astfel sunt recunoscuţi ca fiind constructori). Ei nu au valoare de întoarcere. Ca şi alte metode, ei pot avea argumente. De exemplu, putem dori să iniţializăm un punct cu alte coordonate decât *(0, 0)*. Definim deci un al doilea constructor având două argumente întregi în interiorul clasei:

class Punct {

int \_x, \_y;

public:

Punct() {

\_x = \_y = 0;

}

Punct(const int x, const int y) {

\_x = x;

\_y = y;

}

void setX(const int val);

void setY(const int val);

int getX() { return \_x; }

int getY() { return \_y; }

};

Constructorii sunt apelaţi implicit atunci când se definesc obiecte ale claselor lor

Punct apunct; // Punct::Punct()

Punct bpunct(12, 34); // Punct::Punct(const int, const int)

Cu constructorii putem să iniţializăm obiectele în momentul definirii aşa cum am cerut în capitolul 1 pentru lista simplu înlănţuită. Putem acum să iniţializăm o clasă *Lista* în care constructorii să iniţializeze corect obiectele lor.

Dacă dorim să creăm un punct din alt punct, copiind deci proprietăţile unui obiect în unul nou creat, uneori trebuie să programăm procesul de copiere. De exemplu, să considerăm clasa *Lista* care alocă dinamic memorie pentru elementele sale. Dacă dorim să creăm o a doua listă care este o copie a primei, trebuie să alocăm memorie şi să copiem elementele individuale. În clasa noastră *Punct* adăugăm deci un al treilea constructor care se ocupă de copierea corectă a valorilor de la un obiect la cel nou creat:

class Punct {

int \_x, \_y;

public:

Punct() {

\_x = \_y = 0;

}

Punct(const int x, const int y) {

\_x = x;

\_y = y;

}

Punct(const Punct &din) {

\_x = din.\_x;

\_y = din.\_y;

}

void setX(const int val);

void setY(const int val);

int getX() { return \_x; }

int getY() { return \_y; }

};

Al treilea constructor are o referinţa constantă la un obiect al clasei *Punct* ca argument şi atribuie lui *\_x* şi *\_y* valorile corespunzătoare ale obiectului furnizat.

Acest tip de constructor este atât de important încât are propriul său nume: *constructor de copiere.* Este foarte recomandabil să se dea pentru fiecare clasă un astfel de constructor, chiar dacă nu este la fel de simplu ca în exemplul nostru. Constructorul de copiere este apelat în următoarele cazuri:

Punct apunct; // Punct::Punct()

Punct bpunct(apunct); // Punct::Punct(const Punct &)

Punct cpunct = apunct; // Punct::Punct(const Punct &)

Cu ajutorul constructorilor am satisfăcut una din cerinţele noastre privind implementarea tipurilor de date abstracte: iniţializarea în momentul definirii. Avem nevoie încă de un mecanism care să "distrugă" automat un obiect atunci când devine non-valid (de exemplu, din cauza părăsirii domeniului său. Prin urmare, clasele pot să definească *destructori*.

**6.2.2.3.3 Destructori**

Să considerăm o clasă *Lista*. Elementele listei sunt adăugate şi şterse în mod dinamic. Constructorul ne ajută să creăm o listă iniţială vidă. Totuşi, atunci când ieşim din domeniul definiţiei unui obiect listă, trebuie să ne asigurăm că memoria alocată este eliberată. Definim prin urmare o metodă specială numită *destructor* care este apelată o dată pentru fiecare obiect în momentul distrugerii sale:

void foo() {

Lista olista; // Lista::Lista() este initializata

// cu lista vida.

... // adauga/sterge elemente

} // Apelarea destructorului!

Distrugerea unui obiect are loc automat atunci când obiectul îşi părăseşte domeniul de definiţie sau este distrus în mod explicit. Ultima acţiune are loc atunci când se alocă dinamic un obiect şi îl eliberăm dacă nu mai este necesar.

Destructorii sunt declaraţi ca şi constructorii. Astfel, şi ei utilizează numele clasei, dar prefixat cu simbolul

~ *(tilda):*

class Punct {

int \_x, \_y;

public:

Punct() {

\_x = \_y = 0;

}

Punct(const int x, const int y) {

\_x = xval;

\_y = yval;

}

Punct(const Punct &din) {

\_x = din.\_x;

\_y = din.\_y;

}

~Punct() { /\* Nu se face nimic! \*/ }

void setX(const int val);

void setY(const int val);

int getX() { return \_x; }

int getY() { return \_y; }

};

Destructorii nu au argumente. Este chiar eronat să se definească argumente, deoarece destructorii sunt apelaţi implicit în momentul distrugerii: nu există deci şansa să se specifice argumente efective.

**6.2.2.3.4 Moştenire**

În pseudo-limbajul nostru, am formulat moştenirea prin "inherits from". În C++ aceste cuvinte sunt înlocuite prin caracterul ":". De exemplu, să proiectăm o clasă pentru puncte tridimensionale. Dorim, desigur, să reutilizăm clasa noastră deja existentă *Punct*. Începem proiectarea clasei noastre după cum urmează:

class Punct3D : public Punct {

int \_z;

public:

Punct3D() {

setX(0);

setY(0);

\_z = 0;

}

Punct3D(const int x, const int y, const int z) {

setX(x);

setY(y);

\_z = z;

}

~Punct3D() { /\* Nimic de facut \*/ }

int getZ() { return \_z; }

void setZ(const int val) { \_z = val; }

};

**6.2.2.3.4.1 Tipuri de moştenire**

Se observă din nou cuvântul cheie **public** utilizat în prima linie a definiţiei clasei (*signatura* sa). Acesta este necesar deoarece în C++ se disting două tipuri de moştenire: **publică** şi **privată**. Implicit clasele derivă în mod privat una din cealaltă. În consecinţă, trebuie să se informeze explicit compilatorul pentru a utiliza moştenirea publică.

Tipul de moştenire influenţează drepturile de acces la membrii unor supraclase. Orice este declarat *private* într-o supraclasă va fi inaccesibil în subclasă. Utilizând moştenirea publică, orice este *public* rămâne *public*. Atunci când se utilizează moştenirea privată, orice nu este privat în supraclasă devine privat în subclasă. Efectul cumulat al dreptului de acces din supraclasă şi al modului de moştenire asupra dreptului de acces din subclasă ai membrilor supraclasei este complet arătat în tabelul Tabelul 6.6

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Tip de moştenire** | | |
| **private** | **protected** | **public** |
|  |  |  |  |
| **private** | **inaccesibil** | **inaccesibil** | **inaccesibil** |
| **protected** | **private** | **protected** | **protected** |
| **public** | **private** | **protected** | **public** |

Tabelul 6.6

Coloana din stânga listează drepturi posibile de acces pentru membrii supraclaselor. Un rol special are un al treilea mod de acces: *protected*. Acesta este utilizat pentru membri care să fie direct utilizabili în subclase dar care să nu fie accesibili din afară. Astfel, s-ar putea spune că membrii de acest tip sunt între *private* şi *public* prin aceea ca ei pot fi utilizaţi în ierarhia de clase având drept rădăcină clasa corespunzătoare, utilizându-se şi tipul de moştenire *protected*.

Următoarele coloane arată drepturile de acces care rezultă pentru membrii unei supraclase în funcţie de tipul de moştenire.

**6.2.2.3.4.2 Construcţie**

Atunci când creăm o instanţiere a clasei *Punct3D* constructorul său este apelat. Deoarece *Punct3D* este derivată din *Punct*, constructorul clasei *Punct* este apelat şi el. Totuşi, acest constructor este apelat *înaintea* executării corpului constructorului clasei *Punct3D*. În general, înaintea execuţiei corpului unui anumit constructor, constructorii fiecărei supraclase sunt apelaţi pentru a iniţializa partea din obiectul creat. Când creăm un obiect cu

Punct3D punct(1, 2, 3);

al doilea constructor al clasei *Punct3D* este invocat, dar, înaintea executării corpului constructorului, constructorul *Punct()* este invocat pentru a iniţializa partea *Punct* a obiectului *punct*. Este bine că am definit un constructor fără argumente. Acest constructor iniţializează coordonatele bidimensionale *\_x* şi *\_y* cu *0* (zero). Deoarece *Punct3D* este derivată numai din *Punct* nu există altă apelare de constructor şi corpul lui *Punct3D(const int, const int, const int)* este executat. Aici se invocă metodele *setX()* şi *setY()* pentru a se iniţializa explicit coordonatele bidimensionale. În plus, primeşte valoare şi a treia coordonată *\_z*. Acest fapt este foarte nesatisfăcător, deoarece am definit un constructor *punct()* care are două argumente pentru iniţializa cu ele coordonatele. Astfel, trebuie numai să putem declara că, în loc de utilizarea *constructorului implicit Punct()*, se va utiliza constructorul parametrizat *Punct (const int, const int).* Putem face aceasta specificând constructorul dorit după un singur caracter "**:**" chiar înaintea corpului constructorului *Punct3D*:

class Punct3D : public Punct {

...

public:

Punct3D() {... }

Punct3D(

const int x,

const int y,

const int z) : Punct(x, y) {

\_z = z;

}

...

};

Dacă am avea mai multe supraclase, am da apelările lor de constructori ca pe nişte liste cu elementele separate prin virgule. De exemplu, să presupunem că o clasă *Parte* defineşte numai un constructor cu un argument. Atunci, pentru a crea un obiect al clasei *Compusa* trebuie să invocăm *Parte()* cu argumentul său:

class Compusa {

Parte parte;

...

public:

Compusa(const int parteParametru) : parte(parteParametru) {

...

}

...

};

Această iniţializare dinamică poate să fie de asemenea utilizată cu tipuri de clase predefinite. De exemplu, constructorii clasei *Punct* ar putea fi definiţi ca:

Punct() : \_x(0), \_y(0) {}

Punct(const int x, const int y) : \_x(x), \_y(y) {}

Este bine să utilizăm această metodă de iniţializare cât mai des posibil, deoarece ea permite compilatorului să creeze variabile şi obiecte iniţializate corect în loc să le creeze cu o valoare implicită şi să utilizeze o atribuire suplimentară, (sau alt mecanism) pentru a da valori.

**6.2.2.3.4.3 Distrugere**

Dacă un obiect este distrus, de exemplu prin părăsirea domeniului său de definiţie, destructorul clasei corespunzătoare este invocat. Dacă această clasă este derivată din alte clase, destructorii lor sunt şi ei apelaţi, ajungându-se la un lanţ recursiv de apelări.

**6.2.2.3.4.4 Moştenire multiplă**

C++ permite ca o clasă să fie derivată din mai mult de o supraclasă, aşa cum am menţionat pe scurt mai înainte. Se poate cu uşurinţă deriva din mai mult de o clasă prin specificarea supraclaselor, separate prin virgule:

class SirDesenabil : public Punct, public ObiectDesenabil {

...

public:

SirDesenabil(...) :

Punct(...),

ObiectDesenabil(...) {

...

}

~SirDesenabil() {... }

...

};

Conflictele de nume se rezolvă prin utilizarea operatorului de rezoluţie (::).

Nu vom utiliza acest tip de moştenire în cele ce urmează. Pentru mai multe detalii se pot consulta [2], [3].

**6.2.2.3. 5 Polimorfism**

În pseudo-limbajul nostru puteam să decalarăm metode ale unor clase ca fiind **virtuale**, forţând ca valoarea lor să fie bazată pe conţinut în loc de tipul obiectelor. Putem să utilizăm aceasta şi în C++:

class ObiectDesenabil {

public:

virtual void print();

};

Clasa *ObiectDesenabil* defineşte o metodă *print()* care este virtuală. Putem să derivăm din această clasă alte clase:

class Punct : public ObiectDesenabil {

...

public:

...

void print() {... }

};

Din nou *print()* este o metodă virtuală, deoarece moşteneşte această proprietate de la *ObiectDesenabil*. Funcţia *display()* care este capabilă să afişeze orice fel de obiect desenabil, poate fi definită ca:

void display(const ObiectDesenabil &ob) {

// pregateste tot ce este necesar

ob.print();

}

Atunci când se utilizează metode virtuale unele compilatoare avertizează dacă destructorul clasei corespunzătoare nu este declarat şi el virtual. Acest lucru este necesar dacă se utilizează pointeri la subclase (virtuale) atunci când aceştia sunt distruşi. Deoarece pointerul este declarat ca supraclasă, în mod normal destructorul său ar fi apelat. Dacă destructorul este virtual, destructorul obiectului efectiv referit este apelat (şi atunci, recursiv, toţi destructorii supraclaselor sale). Iată un exemplu:

class Culoare {

public:

virtual ~Culoare();

};

class Rosu : public Culoare {

public:

~Rosu(); // Mostenit virtual de la Culoare

};

class RosuDeschis : public Rosu {

public:

~RosuDeschis();

};

Utilizând aceste clase definim o *paletă* după cum urmează:

Culoare \*paleta[3];

paleta[0] = new Rosu; // Creaza dinamic un nou obiect Rosu

paleta[1] = new RosuDeschis;

paleta[2] = new Culoare;

Operatorul nou introdus **new** creează un nou obiect de tipul specificat în memoria dinamică şi întoarce un pointer la el. Astfel, primul **new** întoarce un pointer la un obiect alocat al clasei *Rosu* şi îl atribuie primului element al tabloului *paleta*. Elementele lui *paleta* sunt pointerii la *Culoare* şi, deoarece *Rosu* "este o" *Culoare*, atribuirea este corectă.

Operatorul opus lui **new** este **delete** care distruge un obiect referit de pointerul furnizat. Dacă aplicăm **delete** elementelor lui *paleta*, se produc următoarele apelări de destructori:

delete paleta[0];

// Apeleaza destructorul ~Rosu() urmat de ~Culoare()

delete paleta[1];

// Apeleaza ~RosuDeschis(), ~Rosu() si ~Culoare()

delete paleta[2];

// Apeleaza ~Culoare()

Diferitele apelări de destructori apar numai din cauză că sunt utilizaţi destructori virtuali. Dacă ei nu ar fi fost declaraţi virtuali, fiecare **delete** ar fi apelat numai *~Culoare()* (deoarece *paleta[i]* este de tipul pointer la *Culoare*).

**6.2.2.3. 6 Clase abstracte**

Clasele abstracte se definesc exact la fel ca şi clasele obişnuite. Totuşi, unele din metodele lor sunt specificate ca fiind în mod necesar definite de subclase. Se specifică *prototipul* lor care conţine tipul pe care îl întorc, numele şi argumentele, dar **nu** şi definiţia. Se poate spune că se omite corpul metodei, sau, altfel spus, se specifică "nimic". Aceasta se exprimă adăugând "**=0**" după prototipul metodei:

class ObiectDesenabil {

...

public:

...

virtual void print() = 0;

};

Această definiţie de clasă va forţa fiecare clasă derivată de la care urmează să se creeze obiecte să definească o metodă *print()*. Aceste declaraţii de metode sunt numite şi *metode pure.*

Metodele pure trebuie de asemenea să fie declarate **virtuale**, deoarece se doreşte să se utilizeze obiecte numai de la clasele derivate. Clasele care definesc metode pure se numesc *clase abstracte.*

**6.2.2.3. 7 Supraîncărcarea operatorilor**

Reamintindu-se tipul abstract de date pentru numere complexe, *Complex*, putem crea o clasă în C++, după cum urmează:

class Complex {

double \_real,

\_imag;

public:

Complex() : \_real(0.0), \_imag(0.0) {}

Complex(const double real, const double imag) :

\_real(real), \_imag(imag) {}

Complex sum(const Complex op);

Complex inm(const Complex op);

...

};

Vom putea atunci să utilizăm numere complexe şi să "calculăm" cu ele:

Complex a(1.0, 2.0), b(3.5, 1.2), c;

c = a.sum(b);

Aici atribuim lui *c* suma numerelor *a* şi *b*. Deşi absolut corect, acesta nu este un mod foarte convenabil de exprimare. Este mai natural să se utilizeze bine-cunoscutul "+" pentru a se exprima adunarea a două numere complexe. Din fericire, C++ ne permite să supraîncărcăm aproape toţi operatorii săi cu tipuri nou create. De exemplu, am putea defini un operator "+" pentru clasa noastră *Complex*:

class Complex {

...

public:

...

Complex operator +(const Complex &op) {

double real = \_real + op.\_real,

imag = \_imag + op.\_imag;

return(Complex(real, imag));

}

...

};

În acest caz, am făcut operatorul *+* membru al clasei *Complex*. O expresie de forma:

c = a + b;

este tradusă într-un apel de metodă:

c = a.operator +(b);

Astfel, operatorul + necesită numai un argument. Primul argument este implicit furnizat de obiectul invocat (în acest caz *a*).

Totuşi, o apelare a operatorului poate fi interpretată şi ca o apelare obişnuită de funcţie, ca în:

c = operator +(a, b);

În acest caz operatorul supraîncărcat **nu** este membru al unei clase. El este definit în exterior ca o funcţie supraîncărcată normală. De exemplu, am putea să definim operatorul + astfel:

class Complex {

...

public:

...

double real() { return \_real; }

double imag() { return \_imag; }

// Nu este nevoie sa se defineasca operatorul aici!

};

Complex operator +(Complex &op1, Complex &op2) {

double real = op1.real() + op2.real(),

imag = op1.imag() + op2.imag();

return(Complex(real, imag));

}

În acest caz trebuie să definim metode de acces pentru părţile reală şi imaginară deoarece operatorul este definit în afara domeniului clasei. Totuşi, operatorul este atât de strâns legat de clasă, încât ar putea avea sens să se permită operatorului accesul la membrii privaţi. Aceasta se poate face prin declararea sa ca prieten al clasei *Complex*.

Obsevație. Denumirea C++ a limbajului de programare înseamnă: extensie a limbajului C prin adăugare orientării pe obiecte (primul +) și a supraîncărcării operatorilor (al doilea +).

**6.2.2.3. 8 Prieteni**

Putem defini funcţii sau clase ca fiind prieteni ai unei clase pentru a le permite accesul la datele lor membre private (utilizând cuvâtul cheie *friend*). De exemplu, în secţiunea precedentă am putea să dorim ca funcţia pentru operatorul *+* să aibă acces la datele membre private *\_real* şi *\_imag* ale clasei *Complex*. Prin urmare, declarăm operatorul *+* ca prieten al clasei *Complex*:

class Complex {

...

public:

...

friend Complex operator +(

const Complex &,

const Complex &

);

};

Complex operator +(const Complex &op1, const Complex &op2) {

double real = op1.\_real + op2.\_real,

imag = op1.\_imag + op2.\_imag;

return(Complex(real, imag));

}

Nu trebuie să folosim prieteni foarte des din cauză că aceştia reprezintă o încălcare a principiului incapsulării datelor. Dacă trebuie să folosim prietenii foarte des, este deja un semn că este timpul să ne restructurăm graful de moştenire.

**6.2. 3 Cum se scrie un program**

Până acum am prezentat numai părţi din programe foarte mici care ar putea cu uşurinţă să fie incluse într-un singur fişier. Totuşi, marile proiecte, de exemplu, un program pentru calendar, ar trebui să fie divizată în părţi uşor de manipulat, denumite adesea *module*. Modulele sunt implementate în fişiere separate. Acum vom discuta pe scurt cum se face modularizarea în C şi C++. Această discuţie este bazată pe UNIX şi compilatorul C++, GNU. Dacă utilizaţi alt software de bază, ceea ce urmează poate căpăta alt aspect. Acest lucru este în mod special valabil pentru cei care utilizează medii de dezvoltare integrate, de exemlu Borland C++.

În linii mari, modulele constau din două tipuri de fişiere: *descriere de interfaţă* şi *fişiere de implementare.* Pentru a deosebi aceste tipuri, sunt utilizate mai multe sufixe ale numelor (numite şi extensii) atunci când se compilează programe C şi C++. Tabelul 6.7 arată unele dintre ele.

Tabelul 6.7

|  |  |  |
| --- | --- | --- |
| **Tipuri de fişier** | **Implementare** | **Extensii** |
| Descriere de iterfaţă (fişiere "antet" sau "de inclus") | Borland | .h.H.hpp.HPP.hxx.HXX |
| UNIX | .h.hpp.hxx |
| Fişier de implementare (sursă) în C | Borland | .c.C |
| UNIX | .c |
| Fişier de implementare (sursă) în C++ | Borland | .cpp.CPP |
| UNIX | .cpp.c++.C.cc.cxx |
| Descriere de interfaţă (fişier "model" sau "şablon") | Borland | .tpl.TPL |
| UNIX | .tpl |

În acest suport de curs utilizăm .h pentru fişiere C++ şi .tpl pentru fişiere care conţin definiţii de modele (în engleză *template*). Chiar dacă scriem numai cod C, se poate utiliza.cc pentru a determina compilatorul să îl trateze ca C++. Acesta face mai simple combinaţiile celor două limbaje, deşi mecanismele interne de aranjare a numelor în program ale compilatorului diferă de la un limbaj la altul.

**Paşii compilării**

Procesul de compilare preia fişierele.cc, le preprocesează (înlăturând comentariile şi incluzând fişierele antet) şi le translatează în *fişiere obiect.* Sufixele tipice pentru aceste fişiere sunt.o.obj.

După terminarea cu succes a compilării, mulţimea de fişiere obiect rezultată este prelucrată de un *editor de legături* (în engleză *linker*). Acest program combină fişierele şi bibliotecile necesare şi crează un program executabil. Sub UNIX acest fişier este denumit *a.out* dacă nu se specifică altfel. Aceşti paşi sunt ilustraţi în Figura 6.1.

.cc

***compilator***

.cc

***editor de***

***legături***

.out

.h,.tpl

biblioteci

Figura 6.1

Compilatoarele moderne permit combinarea celor doi paşi. De exemplu programele mici date ca exemplemai sus pot fi compilate şi legate cu compilatorul GNU C++ după cum urmează ("exemplu.cc"este numai numele unui exemplu, desigur ):

gcc exemplu.cc

**Despre stil**

Fişierele antet sunt utilizate pentru a descrie interfaţa fişierelor de implementare. În consecinţă, sunt incluse în fiecare fişier de implementare care utilizează interfaţa unui fişier de implementare particular. Aşa cum s-a menţionat mai sus această includereeste efectuată prin copierea conţinutului fişierelui antet la fiecare apariţie a directivei de preprocesor # include ajungându-se la un fişier C++ pur uriaş. Pentru a evita includerea unor copii multiple cauzată de dependenţele multiple utilizăm *cod condiţional.* Preprocesorul defineşte directive condiţionale pentru verificarea diferitelor aspecte ale prelucrării sale. De exemplu putem verifica dacă un macro este deja definit:

#ifndef MACRO

#define MACRO /\* defineste MACRO \*/

...

#endif

Liniile dintre **#ifndef** şi **#endif** sunt incluse numai dacă MACRO nu este încă definit. Putem utiliza acest mecanism pentru a împedica efectuarea unor copii multiple ale unui fisier antet:

/\*

\*\* Exemplu de fisier antet care 'verifica' daca el este

\*\* deja inclus. Presupunem ca numele fisierului antet

\*\* este 'antetulmeu.h'

\*/

#ifndef \_\_ANTETULMEU\_H

#define \_\_ANTETULMEU\_H

/\*

\*\* Aici sunt declaratiile din interfata

\*/

#endif /\* \_\_ANTETULMEU\_H \*/

\_\_ANTETULMEU\_H este un nume unic asociat fiecărui fişier antet. Prima dată când fişierul este inclus, \_\_ANTETULMEU\_H nu este definit, astfel încât fiecare linie este inclusă şi prelucrată.Prima linie defineşte un macro numit \_\_ANTETULMEU\_H. Dacă accidental fişierul ar urma să fie inclus a doua oară (în timpul prelucrării aceluiaşi fişier de intrare), \_\_ANTETULMEU\_H este definit, deci tot ceea ce urmează până la #endif este sărit.

**6.2.4 Exerciţii**

1. *Polimorfism.*

Explicaţi de ce

void display(const ObiectDesenabil ob);

nu produce ieşirea dorită.

**7 Rezolvările exerciţiilor**

Această secţiune prezintă exemple de rezolvări pentru exerciţiile de mai sus.

**7.1 O trecere în revistă a tehnicilor de programare**

1. Discutarea modulului

*Lista-Simplu-Inlantuita-2*

(a) Definiţia interfeţei modulului *Intreg-Lista*

MODULE Intreg-Lista

DECLARE TYPE int\_reprez\_lista\_t;

int\_reprez\_lista\_t int\_lista\_create();

BOOL int\_lista\_append(int\_reprez\_lista\_t this,

int date);

INTEGER int\_lista\_getFirst(int\_reprez\_lista\_t this);

INTEGER int\_lista\_getNext(int\_reprez\_lista\_t this);

BOOL int\_lista\_isEmpty(int\_reprez\_lista\_t this);

END Intreg-Lista;

Această reprezentare introduce probleme suplimentare care sunt cauzate de nesepararea traversării de structura de date. După cum vă amintiţi, pentru a itera asupra elementelor listei am utilizat o instrucţiune de ciclare cu următoarea condiţie:

WHILE date IS VALID DO

*date* a fost iniţializat printr-o apelare a *list\_getFirst()*. Procedura de tip listă de întregi *int\_list\_getFirst()* întoarce un întreg, prin urmare nu există ceva cum ar fi "întreg invalid" care să poată fi utilizat pentru verificarea terminării ciclului.

2. Diferenţele între programarea orientată pe obiecte şi alte tehnici. În programarea orientată pe obiecte *obiectele* schimbă *mesaje* unul cu altul. În celelalte tehnici de programare, *datele* sunt schimbate între proceduri sub controlul unui program principal. Pot coexista obiecte de acelaşi fel, dar fiecare cu starea sa proprie. Acest fapt este în contrast cu abordarea modulară, în care fiecare modul are numai o stare globală.

**7.2 Tipuri abstracte de date**

1. TAD *Intreg*.

(a) Ambele operaţii *sum* şi *dif* pot să fie aplicate pentru orice valoare a lui *N*. Astfel, aceste operaţii pot fi aplicate oricând. Nu sunt restricţii pentru utilizarea lor. Totuşi, se poate descrie acest lucru cu o precondiţie care are valoarea **true**.

(b) Definim trei noi operaţii: *inm*, *imp* şi *abs*. Ultima va întoarce valoarea absolută a unui întreg. Operaţiile sunt definite după cum urmează:

inm(k)

imp(k)

abs()

Operaţia *inm* nu cere nici o precondiţie, la fel ca *sum* şi *dif.* Postcondiţia este, desigur *rez=n\*k*. Următoarea operaţie *imp* necesită să avem *k* diferit de *0* (zero). Prin urmare, definim precondiţia: *k* diferit de *0*. Ultima operaţie *abs* întoarce valoarea lui *N* dacă *N* este pozitiv sau *0* sau *-N* dacă *N* este negativ. Din nou nu contează ce valoare are *N* când această operaţie este aplicată. Postcondiţia este:

dacă *N>=0* atunci

*abs=N*

altfel

*abs=-N*.

2. *TAD Fracţie.*

(a) O fracţie simplă constă din numărător şi numitor. Ambele sunt numere întregi. Acest fapt este similar cu exemplul referitor la numărul complex prezentat în secţiune. Am putea alege celpuţin două stucturi de date pentru a păstra valorile: un *tablou* sau un *record.*

(b) Prezentarea interfeţei. Amintim că interfaţa este chiar setul de operaţii vizibile din exterior. Putem descrie o interfaţaă a unei fracţii într-o manieră verbală, în consecinţă, avem nevoie de operaţii:

 pentru a *obţine* valoarea numărătorului/ numitorului;

 pentru a *atribui* o valoare numărătorului/ numitorului;

 pentru a *aduna* o fracţie, întorcând suma;

 pentru a *scădea* o fracţie, întorcând diferenţa;

.**..**

(c) Dăm mai jos unele axiome şi precondiţii pentru fiecare fracţie care sunt de asemenea satisfăcute pentru TAD:

 Numitorul trebuie să fie diferit de 0 (zero), altfel valoarea fracţiei nu este definită.

 Dacă numărătorul este 0 (zero), valoarea fracţiei este 0 pentru orice valoare a numitorului.

 Orice număr întreg poate fi reprezentat printr-o fracţie al cărei numărător este numărul, iar numitorul este 1.

3. TAD-urile definesc proprietăţile unui set de instanţieri. Acestea furnizează o viziune abstractă asupra acestor proprietăţi furnizând un set de operaţii care pot fi aplicate asupra instanţierilor. Acest set de operaţii, adică *interfaţa,* este cel care defineşte proprietăţile instanţierilor.

Utilizarea unui TAD este restricţionată de axiome şi precondiţii. Ele definesc condiţii şi proprietăţi ale mediului în care pot fi utilizate instanţieri ale TAD-ului.

4. Avem nevoie să enunţăm axiome şi să definim precondiţii care să asigure utilizarea corectă a instanţierilorTAD-urilor. De exemplu, dacă nu declarăm 0 (zero) ca fiind elementul neutru al adunării întregilor, ar putea exista un TAD *Intreg* care să facă ceva prestabilit adunând *0* la *N*. Aceasta nu este ceeeace se aşteaptă de la un întreg. Astfel, axiomele şi precondiţiilene dau un mijloc de a ne asigura că TAD-urile "funcţionează"aşa cum dorim să o facă.

5. Descrierea relaţiilor.

(a) O instanţiere este un reprezentant efectiv TAD. (Este astfel un"exemplu" al său. Acolo unde TAD-ul declară că utilizează un"număr întreg cu semn" ca strctură de date, o instanţiere pastrează efectiv o valoare, să spunem, "-5".

(b) TAD-urile generice definesc aceleaşi proprietăţi ale TAD-urilorlor corespunzătoare. Totuşi, ele sunt destinate altui tip particular.De exemplu, TAD *Lista* defineşte proprietăţi ale listelor. Astfel, am putea avea o operaţie *adauga(elem)* care adaugăun nou element *elem* la listă. Nu spunem de ce tip este *elem*, ci numai că el va fiultimul element al listei după această operaţie. Dacă utilizăm un TAD generic *Lista* tipul acestui element este cunoscut: el este dat de parametrul generic.

(c) instanţierile aceluiaşi TAD generic ar putea fi considerate ca "rude". Ele ar putea fi "veri" a instanţierilor unui alt TAD generic dacă cele două TAD-uri generice au în comun acelaşi TAD.

**7.3 Concepte ale orientării pe obiecte**

1. Clasă.

(a) O *clasă* este implementarea efectivă a unui TAD. De exemplu, un TAD pentru întregi ar putea să includă o operaţie *atr* pentru a atribui valori instanţierilor sale. Această operaţie este implementată diferit în limbaje ca C sau Pascal. În C, semnul egal "=" defineşte operaţia de atribuire pentru întregi, în timpce în Pascal este utilizat şirul de caractere ":=". În consecinţă, clasele implementează operaţii furnizând metode. De asemenea, structura de date a TAD-ului este implementată de atributele clasei.

(b) Clasa *Complex*.

class Complex {

attributes:

Real real,

imaginar

methods:

:=(Complex c) /\* Atribuie valoarea lui c \*/

Real parteReala()

Real parteImaginara()

Complex +(Complex c)

Complex -(Complex c)

Complex /(Complex c)

Complex \*(Complex c)

}

Alegem simbolurile bine-cunoscute "*+*" pentru adunare, "*-*" pentru scădere, "*/*" pentru împărţire şi "**\***" pentru înmulţire pentru a implementa operaţiile corespunzătoare ale TAD *Complex*. Astfel, obiectele clasei *Complex* pot fi utilizate ca mai jos:

Complex c1, c2, c3

c3 := c1 + c2

Se observă că se poate scrie o instrucţiune de adunare ca mai jos:

c3 := c1.+(c2)

Aţi putea dori să înlocuiţi "*+*" cu "*sum*" pentrua ajunge la o reprezentare pe care am utilizat-o până acum. De fapt, se vede că "*+*" nu este altceva decâtun alt nume pentru "*sum*".

2. Obiecte care interacţionează.

3.Mesaje.

(a) Obiectele sunt entităţi autonome care doar furnizează o interfaţă bine definită. Apare ca natural să se vorbească despre obiecte ca şi cum ele ar fi entităţi active. De exemplu obiectele "sunt răspunzătoare" pentru ele însele, "ele" pot respinge invocarea unei metode, etc. Aceasta deosebeşte un obiect de un modul, care este pasiv. De aceea nu se vorbeşte despre apelări de proceduri. Se vorbeşte despre mesaje, prin care unui obiect i se "cere" să invoce una din metodele sale.

(a) Internet-ul furnizează mai multe obiecte. Două dintre cele mai bine cunoscute sunt "client" si "server". De exemplu, se utilizează un client FTP (obiect) pentru a se accesadatele memorate pe un server FTP (obiect). Acest fapt poate fi privit astfel: clientul "trimite un mesaj" server-ului prin care cere să i se furnizeze datele memorate acolo.

(a) În mediul client/server avem în realitate două entităţi care actioneazăla distanţă: procesele client şi server. În mod normal, aceste două entităţi schimbă date între ele sub forma unor mesaje Internet.

**7.4 Alte concepte orientate pe obiecte**

1. Moştenire

a) Definiţia clasei *Dreptunghi* :

class Dreptunghi inherits from Punct {

attributes:

int \_lungime, // Lungimea dreptunghiului

\_latime // Latimea dreptunghiului

methods:

setLungime(int nouaLungime)

getLungime()

setLatime(int nouaLatime)

getLatime()

}

În acest exemplu definim un dreptunghi prin colţul său din stânga sus (coordonatele sunt moştenitede la *Punct* ) şi dimensiunile sale. S-ar fi putut defini şi prin colţurile din stânga sus şi dreapta jos (deci prin două puncte).

Urmează să se adauge metode de acces pentru lungimea şi lăţimea dreptunghiului.

(b)Obiecte tridimensionale. O sferă este definită printr-un centru şi o rază. Centrul este un punct în spaţiul tridimensional, astfel, putem defini clasa *Sfera*

ca:

class Sfera inherits from Punct-3D {

attributes:

int \_raza;

methods:

setRaza(int nouaRaza)

getRaza()

}

Aceasta este similară clasei *Cerc* în spaţiul bidimensional (plan). Aici *Punct-3D* este un *Punct* cu o dimensiune suplimentară.

class Punct-3D inherits from Punct {

attributes:

int \_z;

methods:

setZ(int nouZ);

getZ();

}

În consecinţă, *Punct-3D* şi *Punct* sunt legate printr-o relaţie de tipul "este-un".

(c)Funcţionalitatea lui *move().* *move(),* aşa cum este definită în secţiunea 4.2, permite obiectelor tridimensionale să se deplaseze după axa X, deci numai într-o singură dimensiune. Ea face aceasta doar prin modificarea părţii bidimensionale a obiectelor tridimensionale. Această parte bidimensională este definită de clasa *Punct* şi moştenită direct sau indirect de obiectele tridimensionale.

1. Graful de moştenire (vezi Figura 7.1)

**ObiectDesenabil**

**Punct**

**Cerc**

**Punct-3D**

**Dreptunghi**

**Sfera**

Figura 7.1

(e) Alt graf de moştenire. În acest exemplu, clasa *Sfera* moşteneşte de la *Cerc* şi adaugă simplu a treia coordonată. Acesta are avantajul că o sferă poate fi tratată ca un cerc (de exemplu raza sa poate să fie uşor modificată cu metodele/funcţiile care trasează cercul). El are dezavantajul că"distribuie" tratarea obiectului (centrul în spaţiul tridimensional) prin ierarhia de moştenire: de la *Punct* prin *Cerc* la *Sfera.* Astfel, această tratare nu este accesibilă ca un tot.

2. Moştenire multiplă. Graful de moştenire din Figura 4.9 introduce evident conflicte de nume cu proprietăţile clasei *A*.

Totuşi, aceste proprietăţi sunt determinate unic prin urmărirea căii de la *D* la *A*. Astfel, *D* poate schimba proprietăţile lui *A* moştenite de la *B* urmând calea de moştenire prin *B*. La fel, *D* poate schimba proprietăţile lui *A* moştenite de *C* urmând calea de moştenire prin *C*. Prin urmare, acest conflict de nume nu duce în mod necesar la eroare, cât timp căile sunt indicate.

**7.5 Mai mult despre C++**

1. Polimorfism. Se utilizează signatura

void display(const ObiectDesenabil obj);

Să observăm întâi că în C++ argumentele funcţiilor şi metodelor sunt transmise prin valoare. Prin urmare, *obj* va fi o *copie* a argumentului efectiv dat la apelarea funcţiei. Aceasta înseamnă că *ObiectDesenabil* trebuie să fie o clasă din care să se poată crea obiecte. **Nu** ar fi acesta cazul, dacă *ObiectDesenabil* Ar fi o clasă abstractă (aşa cum ar fi dacă *print()* ar fi definit ca o metodă pură).

Dacă există o metodă virtuală *print()* care este definită de clasa *ObiectDesenabil,* atunci (deoarece *obj* este numai o copie a argumentului efectiv) această metodă este invocată. Ea **nu** este metoda definită de clasa argumentului efectiv (din cauză că acesta nu mai joacă un rol semnificativ!).

2. Adăugarea metodei de ştergere. Nu dăm un program, dar dăm algoritmul. Metoda *sterge()* tebuie să itereze aupra listei până când întâlneşte un nod cu elementul de date cerut, să şteargă acel nod şi să întoarcă 1. Dacă lista este vidă sau elementul de date nu poate fi găsit, întoarce 0(zero).

În timpul iterării, *sterge()* trebuie să compare succesiv elementul de date dat cu cele din listă. Prin urmare, poate exista o comparaţie ca:

if (date == curent->date()) {

// element gasit

}

Aici utilizăm operatorul de egalitate "**==**" pentru a compara cele două elemente. Cum aceste elemente pot fi de orice tip, ele pot fi mai ales obiecte ale claselor definite de utilizator. A pune întrebarea: cum este "egalitatea" definită pentru aceste noi tipuri? În consecinţă, pentru a se permite metodei *sterge()* să lucreze corect, lista va fi utilizată numai pentru tipuri pentru care se definesc operatorii "**==**" şi "**!=**" în mod corespunzător. Altfel sunt utilizate comparări implicite, ceea ce ar putea duce la rezultate ciudate.

3. Clasa *ListaNumărată.* O listă numărată este o listă care are asociat şi numărul elementelor sale. Astfel, când un element de date este adăugat, numărul este mărit cu unu, iar când un element este şters, este micşorat cu unu. Din nou, nu dăm implementarea completă, arătăm doar o metodă ( *adauga()* ) şi felul cum implementarea este modificată:

class ListaNumarata : public Lista {

int \_numar; // Numarul de elemente

...

public:

...

virtual void adauga(const T date) {

\_numar++; // Incrementeaza si...

Lista::adauga(date); //... adauga la lista

}

...

}

Nu orice metodă poate fi implementată astfel. La unele metode trebuie să se verifice dacă *\_numar* necesită să fie modificat sau nu. Totuşi, ideea principală este că fiecare metodă de liste este expandată (sau *specializată* ) pentru lista numărată.

4. Problema iteratorului. Pentru a rezolva problema iteratorului, ne-am putea gândi la o soluţie în care iteratorul conţine o referinţă la lista lui corespunzătoare.

În momentul creării iteratorului această referinţă este iniţializată pentru a indica lista dată. Metodele iteratorului trebuie modificate pentru a utiliza această referinţă în locul pointer-ului *\_pornire.*

**Bibliografie**

[1] T. Bălanescu, S. Gavrilă, H. Georgescu, M. Gheorghe, L. Sofonea, I. Văduva

"Pascal şi Turbo Pascal", vol 2, Ed. Tehnică, Bucureşti, 1992

[2] K. Jamsa, L. Klander "Totul despre C şi C++", Editura Teora, Bucureşti, 2002

(traducere din limba engleză)

[3] O. Catrina, I. Cojocaru "Turbo C++", Editura Teora, Bucureşti, 1993

[4] H. Schildt: "C++ manual complet", Editura Teora, Bucureşti, 1998

[5] Bjarne Stroustrup: "The C++ Programming Language", Adisson-Wesley, 3nd edition, 1997

[6] C. Spircu, I. Lopătaru "POO: analiza, proiectarea şi programarea orientată pe obiecte ", Editura Teora, Bucureşti, 1995

[7] Walter Savitch, Kenrick Mock "Absolute C++", Pearson, 6th edition, 2015

[8] Marius Băncilă "Modern C++ Programming Cookbook", Packt Publishing, 3rd edition, 2024

**Resurse on-line**

[web1] https://www.cs.montana.edu/courses/fall2003/current/210/lectures/OO/Analysis/tutorial/

(Peter Muller: Introduction to Object-Oriented Programming Using C++)

[web2] <https://cplusplus.com/doc/tutorial/> (C++ Language)

[web3] <https://www.w3schools.com/cpp/> (C++ Tutorial)

[web4] <https://cplusplus.com/files/tutorial.pdf> (C++ Language Tutorial)

[web5] <https://www.tutorialspoint.com/cplusplus/index.htm> (C++ Tutorial)