10. Derivare publica, functii virtuale

Derivarea publica si functiile virtuale sunt mecanismele esentiale pentru programarea orientata pe obiect in limbajul C++. Cele doua exemple prezentate in acest capitol au fost alese pentru a ilustra nu numai eleganta deosebita a programarii orientate pe obiect, ci si problemele care pot apare atunci cand se fac greseli de proiectare.

10.1 Ciurul lui Eratostene

Acest exemplu este construit pornind de la un cunoscut algoritm pentru determinarea numerelor prime. Geograful si astronomul grec Eratostene din Cirena (sec. III a.Ch.) a avut ideea de a transforma proprietatea numerelor prime de a nu fi multiplii nici unuia din numerele mai mici decat ele, intr-un criteriu de selectie (cernere): din sirul primelor primelor *n* numere naturale se elimina pe rand multiplii lui 2, 3, 5 etc, elementele ramase fiind cele prime. Astfel, 2 fiind prim, din sir se elimina multiplii lui 2 adica 4, 6, 8 etc. Urmatorul numar ramas este 3, deci 3 este prim si se vor elimina numerele 9, 15, 21 etc, multiplii pari ai lui 3 fiind deja eliminati. Si asa mai departe, pana la determinarea tuturor numerelor prime mai mici decat un numar dat.

Implementarea ciurului lui Eratostene prezentata in continuare* nu este foarte eficienta ca timp de executie si memorie utilizata. In schimb, este atat de “orientata pe obiect”, incat merita sa fie prezentata ca una din cele mai tipice aplicatii C++. Ciurul este construit dintr-un sir de *site* si un generator de numere numit *contor*. Fiecare sita corespunde unui numar prim. Ea solicita valori (numere) de cernut de la sita urmatoare si lasa sa treaca, returnand sitei anterioare, doar acele valori care nu sunt multipli ai numarului prim corespunzator sitei. Ultimul element in aceasta structura de site este contorul, care nu face decat sa genereze numere, rand pe rand. Primul element este ciurul propriu-zis, din care vor iesi doar numere prime. In plus, ciurul mai are sarcina de a crea sita corespunzatoare numarului prim tocmai determinat.

La inceput, avem doar ciurul si contorul, acesta din urma initializat cu valoarea 2 (Figura 10.1). Prima valoare extrasa din ciur este si prima valoare returnata de

---

* Implementarea este preluata din R. Sethi, “*Programming Languages. Concepts and Constructs*”, Sectiunea 6.7.

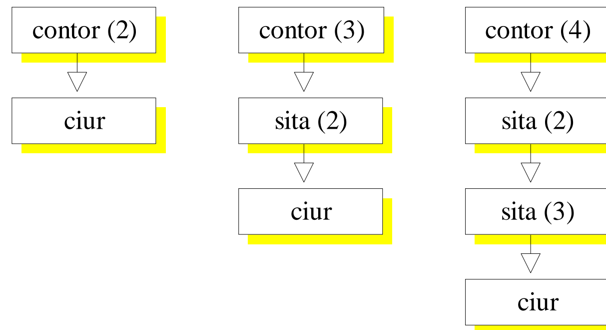

Figura 10.1 Ciurul lui Eratostene.

contor, si anume 2. Dupa aceasta prima iteratie, contorul va avea valoarea 3, iar intre ciur si contor se va insera prima sita, sita corespunzatoare lui 2. Ciurul va solicita o noua valoare sitei 2 care, la randul ei, va solicita o noua valoare contorului. Contorul emite 3, schimbandu-si valoarea la 4, 3 va trece prin sita 2 si va ajunge la ciur. Imediat, sita 3 se insereaza in lista existenta.

Contorul, la solicitarea ciurului, solicitare transmisa sitei 3, apoi sitei 2, va returna in continuare 4. Valoarea 4 nu trece de sita 2, dar sita 2 insista, caci trebuie sa raspunda solicitarii primite, astfel incat va primi un 5. Aceasta valoare trece de toate sitele si lista are un nou element. Continuand procesul, constatam ca 6 se blocheaza la sita 2, 7 trece prin toate sitele (5, 3, 2), iar valorile 8 si 9 sunt blocate de sitele 2, respectiv 3. La un moment dat, contorul va avea valoarea *n*, iar in lista vor fi sitele corespunzatoare tuturor numerelor prime mai mici decat *n*.

Pentru implementarea acestui comportament, avem nevoie de o lista inlantuita, in care fiecare element este sursa de valori pentru predecesor si isi cunoaste propria sursa (elementul succesor). Altfel spus, fiecare element are cel putin doi membri: adresa sursei si functia care cerne valorile.

```
class element {
public:
  element( element *src ) { sursa = src; }
  virtual int cerne( ) { return 0; }

protected:
  element *sursa;
};
```

Acest `element` este un prototip, deoarece lista contine trei tipuri diferite de elemente, diferentiate prin functia de cernere:

- Ciurul, care creeaza site.
- Sitele, care cern valorile.
- Contorul, care doar genereaza valori.

Cele trei tipuri fiind particularizari ale tipului element, le vom deriva public din acesta, creand astfel trei subtipuri.

```
class contor: public element {
public:
  contor( int v ): element( 0 ) { valoare = v; }
  int cerne( ) { return valoare++; };

private:
  int valoare;
};

class ciur: public element {
public:
  ciur( element *src ): element( src ) { }
  int cerne( );
};

class sita: public element {
public:
  sita( element *src, int f ): element(src) { factor = f; }
  int cerne( );

private:
  int factor;
};
```

Clasa contor este definita complet. Primul element generat ("cernut") este stabilit prin v, parametrul constructorului. Pentru clasa sita, functia de cernere este mult mai selectiva. Ea solicita de la sursa valori, pana cand primeste o valoare care nu este multiplu al propriei valori.

```
int sita::cerne( ) {
  while ( 1 ) {
    int n = sursa->cerne( );
    if ( n % factor ) return n;
  }
}
```

Pentru ciur-ul propriu-zis, functia de cernere nu mai are nimic de cernut. Valoarea primita de la sita sursa este in mod sigur un numar prim, motiv pentru care ciur-ul va crea sita corespunzatoare.

```
int ciur::cerne( ) {
 int n = sursa->cerne( );
 sursa = new sita( sursa, n );
 return n;
}
```

Se observa ca noua sita este creata si inserata in ciur printr-o singura instructiune:

```
 sursa = new sita( sursa, n );
```

al carei efect poate fi exprimat astfel: sursa nodului ciur este o noua sita (cu valoarea n) a carei sursa va fi sursa actuala a ciurului. O astfel de operatie de inserare este una dintre cele mai uzuale in lucrul cu liste.

Prin programul urmator, ciurul descris poate fi pus in functiune pentru determinarea numerelor prime mai mici decat o anumita valoare.

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

// definitiile claselor element, contor, ciur, sita

void no_mem( ) { cerr << "no_mem"; exit( 1 ); }

int main( ) {
  _new_handler = no_mem;

  int max;
  cout << "max ... "; cin >> max;

  ciur s( &contor( 2 ) );
  int prim;

  do {
    prim = s.cerne( );
    cout << prim << ' ';
  } while ( prim < max );
  cout << '\n';

  return 0;
}
```

Inainte de a introduce valori max prea mari, este bine sa va asigurati ca stiva programului este suficient de mare si ca aveti suficient de multa memorie libera pentru a fi alocata dinamic.

Folosind cunostintele expuse pana acum, aceasta ciudata implementare a algoritmului lui Eratostene nu are cum sa functioneze. Iata cel putin doua motive:

- In instructiunea `int n = sursa->cerne( )` din clasele `sita` si `ciur`, membrul `sursa`, mostenit de la clasa de baza `element`, este de tip pointer la `element`, deci functia `cerne()` invocata este cea definita in clasa `element`. In consecinta, ceea ce se obtine ar trebui sa fie un sir infinit de `0`-uri (desigur, in limita memoriei disponibile).
- Exista o nepotrivire intre argumentele formale (de tip pointer la `element`) ale constructorilor claselor `ciur`, `sita` si argumentele actuale cu care sunt invocati acesti constructori. Astfel, constructorul clasei `ciur` este invocat cu un pointer la `contor`, iar constructorul clasei `sita` este invocat prima data cu un pointer la `contor` si apoi cu pointeri la `sita`.

Elementele esentiale in elucidarea acestor aspecte sunt derivarile publice si definitia din clasa `element`:

```
virtual int cerne( ) { return 0; }
```

Prin derivarea `public`, tipurile `ciur`, `sita` si `contor` sunt subtipuri ale tipului de baza `element`. Conversia de la un subtip (tip derivat public) la tipul de baza este o conversie sigura, bine definita. Membrii tipului de baza vor fi totdeauna corect initializati cu valorile membrilor respectivi din subtip, iar valorile membrilor din subtip, care nu se regasesc in tipul de baza, se vor pierde. Din aceleasi motive, conversia subtip-tip de baza se extinde si asupra pointerilor sau referintelor. Altfel spus, in orice situatie, un obiect, un pointer sau o referinta la un obiect dintr-un tip de baza, poate fi inlocuit cu un obiect, un pointer sau o referinta la un obiect dintr-un tip derivat public.

Declaratia `virtual` din functia `cerne()` permite implementarea *legaturilor dinamice*. Prin redefinirea functiei `cerne()` in fiecare din subtipurile derivate din `element`, se permite invocarea diferentiata a functiilor `cerne()` printr-o sintaxa unica:

```
sursa->cerne( )
```

Daca `sursa` este de tip pointer la `element`, atunci, dupa cum am precizat mai sus, oricand este posibil ca obiectul `sursa` sa fie dintr-un tip derivat din `element`. Functia `cerne()` fiind virtuala in tipul de baza, functia efectiv invocata nu va fi cea din tipul de baza, ci cea din tipul actual al obiectului invocator. Acest mecanism implica stabilirea unei legaturi dinamice, in timpul executiei programului, intre tipul actual al obiectului invocator si functia virtuala.

Datorita faptului ca definitia din clasa de baza a functiei `cerne()`, practic nu are importanta, este posibil sa o lasam nedefinita:

```
virtual int cerne( ) = 0;
```

Consecinta acestei actiuni este ca nu mai putem defini obiecte de tip `element`, clasa putand servi doar ca baza pentru derivarea unor subtipuri. Astfel de functii se numesc *virtuale pure*, iar clasele respective sunt numite *clase abstracte*.

10.2 Tablouri initializate virtual

Tabloul, una din cele mai simple structuri de date, are drept caracteristica principala adresarea elementelor sale in timp constant. Regasirea sau modificarea valorii unui element de tablou sunt operatii care necesita un timp constant, independent de factori cum ar fi pozitia (indicele) elementului, sau faptul ca este neinitializat, initializat sau chiar modificat de mai multe ori.

Adresarea elementelor pentru citirea sau modificarea valorilor lor, operatie numita *indexare*, este considerata operatia fundamentala asociata structurii de tablou. Chiar daca este usor de implementat, cu un timp de executie constant (vezi operatorul de indexare din clasa `tablou<T>`, Sectiunea 4.1.3), uneori indexarea se dovedeste a fi costisitoare. Algoritmul *nim* ne-a pus in fata unei astfel de situatii: tabloul *init* este util doar daca poate fi nu numai adresat, ci si initializat in timp constant, timp imposibil de atins in conditiile initializarii fiecarui element prin operatorul de indexare. Aparent, tabloul nu se preteaza la o astfel de initializare si deci ar fi bine sa lucram cu o alta structura. De exemplu, cu o lista a elementelor modificate. Valoarea oricarui element este cea memorata in lista, sau este o valoare implicita, daca el nu apare aici. Initializarea nu mai depinde, in aceasta situatie, de numarul elementelor, dar nici adresarea nu mai este o operatie cu timp constant de executie!

Initializarea unui tablou in timp constant, impreuna cu accesarea elementelor tot in timp constant, sunt doua cerinte aparent contradictorii pentru structura de tablou. Eliminarea contradictiei, in caz ca este posibila (si este), impune completarea tabloului cu o noua operatie elementara, *initializarea*, precum si modificarea corespunzatoare a operatorului de indexare. Obtinem un alt tip de tablou, in care elementele nu mai sunt initializate efectiv, fiecare in parte, ci virtual, printr-un operator de initializare globala.

In continuare, vom prezenta o structura de *tablou initializat virtual*[*], precum si implementarea corespunzatoare in limbajul C++.

---

[*] Ideea acestei structuri este sugerata in A. V. Aho, J. E. Hopcroft si J. D. Ullman, *"The Design and Analysis of Computer Algorithms"*.

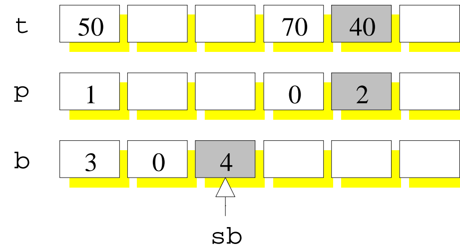

Figura 10.2 Structura de tablou initializat virtual.

10.2.1 Structura

Intreaga constructie se bazeaza pe o idee similara cu mai sus mentionata lista a elementelor modificate. Este o idee cat se poate de naturala, completata cu o ingenioasa modalitate de evitare a parcurgerii secventiale a listei.

Tabloului initializat virtual `a` i se asociaza un tablou `b`, in care sunt memorate pozitiile elementelor modificate. De exemplu, daca s-au modificat elementele `a[3]`, `a[0]` si `a[4]`, atunci primele trei pozitii din `b` au valorile `3`, `0` si `4`. Se observa ca `b` este o lista implementata secvential, sub forma unui tablou.

Daca nici o pozitie din `b` nu a fost ocupata, atunci orice element din `a` are valoarea implicita (stabilita apriori). Daca `b` este complet ocupat, atunci toate elementele din `a` au fost modificate, deci valorile lor pot fi obtinute direct din locatiile corespunzatoare din `a`. Pentru evidenta locatiilor ocupate din lista `b` (si implicit a celor libere), vom folosi variabila `sb`, variabila initializata cu `-1`.

Cum procedam in cazul general, cand doar o parte din elemente au fost modificate? Nu vom parcurge locatiile ocupate din `b`, ci va trebui sa decidem in timp constant daca un anumit element, de exemplu `a[1]`, a fost sau nu a fost modificat. Pentru a atinge aceasta “performanta”, vom completa structura cu un alt tablou, tabloul `p`, in care sunt memorate pozitiile din `b` ale fiecarui element modificat (Figura 10.2).

Tabloul `p`, tablou paralel cu tabloul `a`, se initializeaza pe masura ce elementele din `a` sunt modificate. In exemplul nostru, `p[3]` este `0` deoarece elementul `a[3]` a fost modificat primul, ocupand astfel prima pozitie in `b`. Apoi, `p[0]` este `1` pentru ca `a[0]` este al doilea element modificat, iar `p[4]` are valoarea `2` deoarece al treilea element modificat a fost `a[4]`. Valoarea lui `a[1]`, valoare nemodificata, se obtine prin intermediul tablourilor `p` si `b` astfel:

- Daca p[1] nu este o pozitie valida in b, adica 0 > p[1] sau p[1] > sb, atunci a[1] nu a fost modificat si are valoarea implicita.
- Daca p[1] este o pozitie valida in b, atunci, pentru ca a[1] sa nu aiba valoarea implicita, b[p[1]] trebuie sa fie 1. Deoarece a[1] nu a fost modificat, nici o pozitie ocupata din b nu are insa valoarea 1. Deci, a[1] are valoarea implicita.

Sa vedem acum ce se intampla pentru un element deja modificat, cum este a[0]. Valoarea lui p[0] corespunde unei pozitii ocupate din b, iar b[p[0]] este 0, deci a[0] nu are valoarea implicita.

10.2.2 Implementarea (o varianta de nota sase)

Tabloul initializat virtual cu elemente de tip T, tablouVI<T> (se poate citi chiar “tablou sase”), este o clasa cu o functionalitate suficient de bine precizata pentru a nu pune probleme deosebite la implementare. Vom vedea ulterior ca apar totusi anumite probleme, din pacate majore si imposibil de ocolit. Pana atunci, sa stabilim insa structura clasei tablouVI<T>. Fiind, in esenta, vorba tot de un tablou, folosim clasa tablou<T> ca tip public de baza. Altfel spus, tablouVI<T> este un subtip al tipului tablou<T> si poate fi folosit oricand in locul acestuia. Alaturi de datele mostenite de la tipul de baza, noua clasa are nevoie de:

- Cele doua tablouri auxiliare p si b.
- Intregul sb, contorul locatiilor ocupate din b.
- Elementul vi, in care vom memora valoarea implicita a elementelor tabloului.

In privinta functiilor membre avem nevoie de:

- Un constructor (constructorii nu se mostenesc), pentru a dimensiona tablourile si a fixa valoarea implicita.
- O functie (operator) de initializare virtuala, prin care, in orice moment, sa “initializam” tabloul.
- Un operator de indexare.

In mare, structura clasei tablouVI<T> este urmatoarea:

```
template <class T>
class tablouVI: public tablou<T> {
public:
  tablouVI( int, T );

  tablouVI& operator =( T );
  T& operator []( int );
```

```
private:
  T vi;            // valoarea implicita

  tablou<int> p, b; // tablourile auxiliare p si b
  int sb;           // pointer in b
};
```

unde operatorul de atribuire este cel care realizeaza intializarea virtuala a tabloului.

Indexarea este operatia cea mai dificil de implementat. Dificultatea provine din necesitatea de a-i conferi acestui operator o functionalitate similara celui din clasa `tablou<T>`, in sensul de a putea fi folosit atat pentru returnarea valorii elementelor, cat si pentru modificarea lor. Pentru a nu complica in mod inutil implementarea, convenim ca primul acces la fiecare element sa implice si initializarea elementului respectiv cu valoarea implicita. In consecinta, modificarea valorii unui element se realizeaza prin simpla returnare a referintei elementului. Operatorul de indexare este implementat astfel:

```
template<class T>
T& tablouVI<T>::operator []( int i) {
  static T z;    // elementul returnat in caz de eroare

  // verificarea indicelui i
  if ( i < 0 || i >= d ) {
    cerr << "\n\ntablouIV -- " << d
         << ": indice eronat: " << i << ".\n\n";
    return z;
  }

// returnarea valorii elementului i
  int k = p[ i ];
  if ( 0 <= k && k <= sb && b[ k ] == i )
    // element deja initializat
    return a[ i ];
  else
    // elementul se initializeaza cu valoarea implicita
    return a[ b[ p[ i ] = ++sb ] = i ] = vi;
}
```

Operatorul de atribuire implementat mai jos poate fi oricand invocat pentru initializarea virtuala. Argumentul lui este valoarea implicita asociata tuturor elementelor tabloului:

```
template<class T>
tablouVI<T>& tablouVI<T>::operator =( T v ) {
  vi = v; sb = -1;
  return *this;
}
```

De asemenea, putem realiza initializarea virtuala si prin intermediul constructorului clasei tablouVI<T>:

```
template<class T>
tablouVI<T>::tablouVI( int n, T v ):
            tablou<T>( n ), vi( v ), p( n ), b( n ), sb( -1 ) {
}
```

Operatiile de initializare a obiectelor prin constructori constituie una din cele mai bine fundamentate parti ale limbajului C++. Pentru clasa tablou<T>, initializarea elementelor tabloului este ascunsa in constructorul

```
template<class T>
tablou<T>::tablou( int dim ) {
  a = 0; v = 0; d = 0;      // valori implicite
  if ( dim > 0 )            // verificarea dimensiunii
    a = new T [ d = dim ];  // alocarea memoriei
}
```

constructor invocat prin lista de intializare a membrilor din constructorul clasei tablouVI<T>. Mai exact, expresia new T[d] are ca efect invocarea constructorului implicit al tipului T, pentru fiecare din cele d elemente alocate dinamic. Acest comportament (absolut justificat) al operatorului new este total inadecvat unei initializari in timp constant. Ne intrebam daca putem doar aloca spatiul, fara a-l si intializa. Daca tipul T nu are nici un constructor, atunci initializarea spatiului alocat este inutila, deoarece acest tip admite obiecte neinitializate. Dar, daca tipul T are cel putin un constructor, atunci inseamna ca obiectele de tip T nu pot fi neinitializate si, in consecinta, este necesar un constructor implicit (apelabil fara nici un argument) pentru a initializa spatiul alocat prin new. Astfel, am ajuns la primul motiv pentru care aceasta implementare este doar de nota sase: tabloul tablouVI<T> este (virtual) initializat in timp constant, numai daca tipul T nu are constructor, altfel spus, daca permite lucrul cu obiecte neinitializate.

Problema pe care ne-o punem acum este in ce masura responsabilitatea verificarii acestei conditii poate fi preluata de compilator sau de proiectantul clasei tablouVI<T>. Compilatorul poate semnala, in cel mai bun caz, absenta constructorului implicit. Proiectantul nu este nici el intr-o situatie mai buna, deoarece:

- Nu poate modifica comportamentul operatorului `new` astfel incat sa nu mai invoce constructorul implicit.
- Prezenta (sau absenta) constructorilor clasei `T` nu poate fi verificata in timpul rularii programului.

Solutia este reproiectarea clasei, pentru a se obtine o varianta mai putin naiva. De exemplu, in tabloul propiu-zis, se pot memora adresele elementelor, si nu elementele.

Obiectele de tip `tablouVI<T>` genereaza necazuri si in momentul in care inceteaza sa mai existe. Stim ca, in aceste situatii, se vor invoca destructorii datelor membre si cel al clasei de baza (in aceasta ordine). Ajungem din nou la clasa `tablou<T>` si la destructorul acesteia:

```
~tablou( ) { delete [ ] a; }
```

care invoca destructorul clasei `T` (in caz ca `T` are destructor) pentru fiecare obiect alocat la adresa `a`. Efectele destructorului asupra obiectelor care nu au fost niciodata initializate sunt greu de prevazut. Rezulta ca prezenta destructorului in clasa `T` este chiar periculoasa, spre deosebire de prezenta constructorului care va genera doar pierderea timpului constant de initializare.

Continuand analiza deficientelor clasei `tablouIV<T>`, ajungem la banala operatie de atribuire `a[ i ] = vi` din operatorul de indexare. Daca tipul `T` are un operator de atribuire, atunci acest operator considera obiectul invocator (cel din membrul drept) deja initializat si va incerca sa-l distruga in aceeasi maniera in care procedeaza si destructorul. In cazul nostru, premisa este contrara: `a[ i ]` nu este initializat, deci nu ne trebuie o operatie de atribuire, ci una de initializare a obiectului din locatia `a[ i ]` cu valoarea `vi`. Iata un nou argument in favoarea utilizarii unui tablou de adrese si nu a unui tablou de obiecte.

Fara a mai conta la nota acordata, sa observam ca operatiile de initializare si de atribuire intre obiecte de tip `tablouVI<T>` sunt nu numai generatoare de surprize (neplacute), ci si foarte ineficiente. Surprizele sunt datorate constructorilor si destructorilor clasei `T` si au fost analizate mai sus. Ineficienta se datoreaza faptului ca nu este necesara parcurgerea in intregime a tablourilor implicate in transfer, ci doar parcurgerea elementelor purtatoare de informatie (initializate). Cauza acestor probleme este operarea membru cu membru in clasa `tablouVI<T>`, prin intermediul constructorului de copiere si al operatorului de atribuire din clasa `tablou<T>`.

Concluzia este ca tabloul initializat virtual genereaza o multime de probleme. Aceasta, deoarece procesul de initializare si cel opus, de distrugere, sunt tocmai elementele imposibil de ocolit in semantica structurilor de tip clasa din limbajul C++. Implementarea prezentata, chiar daca este doar de nota sase, poate fi

utilizata cu succes pentru tipuri de date predefinite, sau care nu necesita constructori si destructori. De asemenea, se vor evita operatiile de copiere (initializari, atribuiri, transmiteri de parametri prin valoare) intre obiectele de acest tip.

10.2.3 `tablouVI<T>` ca subtip al tipului `tablou<T>`

Derivarea publica instituie o relatie speciala intre tipul de baza si cel derivat. Tipul derivat este un subtip al celui de baza, putand fi astfel folosit oriunde este folosit si tipul de baza. Aceasta flexibilitate se bazeaza pe o conversie standard a limbajului C++, si anume conversia de la tipul derivat public catre tipul de baza.

Prin functionalitatea lui, tabloul initializat virtual este o particularizare a tabloului. Decizia de a construi tipul `tablouVI<T>` ca subtip al tipului `tablou<T>` este deci justificata. Simpla derivare publica nu este suficienta pentru a crea o veritabila relatie tip-subtip. De exemplu, sa consideram urmatorul program pentru testarea clasei `tablouVI<T>`.

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

// declaratie necesara pentru a evita
// referirea la sablon - vezi Sectiunea 4.1.3
ostream& operator <<( ostream&, tablou<int>& );

main( ) {
  cout << "\nTablou (de intregi) initializat virtual."
       << "\nNumarul elementelor, valoarea implicita ... ";
  int n, v;  cin >> n >> v;
  tablouVI<int> x6( n, v );

  cout << "\nIndicele, valoarea (prin indicele -1 se\n"
       << " modifica valoarea implicita) <EOF>:\n...";
  while( cin >> n >> v ) {
    if ( n == -1 ) x6 = v; else x6[ n ] = v;
    cout << "...";
  }
  cin.clear( );

  cout << '\n' << x6 << '\n';
  return 1;
}
```

Acest program este corect, dar valorile afisate nu sunt cele care ar trebui sa fie. Cauza este operatorul de indexare `[]` din `tablou<T>`, operator invocat in functia

```
template <class T>
ostream& operator <<( ostream& os, tablou<T>& t ) {
  int n = t.size( );

  os << " [" << n << "]: ";
  for ( int i = 0; i < n; os << t[ i++ ] << ' ' );
  return os;
}
```

prin intermediul argumentului t. Noi dorim ca, atunci cand t este de tip tablouVI<T>, operatorul de indexare [] invocat sa fie cel din clasa tablouVI<T>. De fapt, ceea ce urmarim este o legare (selectare) dinamica, in timpul rularii programului, a operatorului [] de tipul actual al obiectului invocator. Putem obtine acest lucru declarand virtual operatorul de indexare din clasa tablou<T>:

```
template <class T>
class tablou {
  // ..
public:
  // ...
  virtual T& operator []( int );
  // ...
};
```

O functie declarata virtual intr-o clasa de baza este o functie a carei implementare depinde de tip, in sensul ca va fi reimplementata pentru unele din tipurile derivate. Atunci cand se invoca printr-o referinta sau pointer la clasa de baza, functia virtuala permite selectarea variantei sale redefinite pentru tipul actual al obiectului invocator. In cazul nostru, operatorii de indexare au fost redefiniti in clasa derivata tablouVI<T>. Deci, prin declararea ca functii virtuale in clasa de baza, se realizeaza legarea lor dinamica de tipul actual al obiectului invocator.

Mostenirea si legaturile dinamice sunt atributele necesare programarii orientate pe obiect. Limbajul C++ suporta aceste facilitati prin mecanismul de derivare al claselor si prin functiile virtuale. Un alt element util programarii orientate pe obiect este obtinerea de informatii asupra claselor in timpul rularii programului (*RTTI* sau **R**un-**T**ime **T**ype **I**nformation). Iata o situatie simpla, in care avem nevoie de *RTTI*. Fie urmatoarea functie pentru interschimbarea a doua tablouri:

```
template <class T>
void swap( tablou<T>& a, tablou<T>& b ) {
  tablou<T> tmp = a;
  a = b;
  b = tmp;
}
```

Pentru a o invoca, putem utiliza orice argumente de tip `tablou<T>` sau `tablouVI<T>`. Nu este insa logic sa interschimbam un `tablouVI<T>` cu un `tablou<T>`. Detectarea acestei situatii (corecta din punct de vedere sintactic) se poate face numai in momentul rularii programului, prin *RTTI*. Limbajul C++ nu are facilitati proprii pentru *RTTI*, dar permite implementarea lor prin mecanismul functiilor virtuale. Multe din bibliotecile C++ profesionale ofera facilitati sofisticate de *RTTI*. Pentru exemplul de mai sus, am implementat o varianta primitiva de *RTTI*. Este vorba de introducerea functiilor virtuale `tip()` in clasele `tablou<T>` si `tablouVI<T>`, functii care returneaza codurile de identificare ale claselor respective.

```
template <class T>
class tablou {
  // ...
public:
  // ...
  virtual char tip( ) const { return 'T'; }
  // ...
};

template <class T>
class tablouVI: public tablou<T> {
public:
  // ...
  char tip( ) const { return 'V'; }
  // ...
};
```

Deci, vom introduce in functia `swap( tablou<T>&, tablou<T>& )` secventa de test a tipurilor implicate:

```
template <class T>
void swap( tablou<T>& a, tablou<T>& b ) {
  if ( a.tip( ) != b.tip( ) )
    cerr << "\n\nswap -- tablouri de tipuri diferite.\n\n";
  else {
    tablou<T> tmp = a; a = b; b = tmp;
  }
}
```

Am reusit, astfel, sa prevenim anumite operatii corecte sintactic, dar imposibil de aplicat obiectelor din tipurile derivate.

Mecanismul *RTTI* trebuie folosit cu mult discernamant. Este mai bine sa prevenim situatii ca cea de mai sus, decat sa le solutionam prin “artificii” (de tip *RTTI*) care pot duce la pierderea generalitatii functiilor sau claselor respective.

10.3 Exercitii

10.1 Daca toate elementele unui tablou initializat virtual au fost modificate, atunci testarea starii fiecarui element prin tablourile `p` si `b` este inutila. Mai mult, spatiul alocat tablourilor `p` si `b` poate fi eliberat.

Modificati operatorul de indexare al clasei `tablouVI<T>` astfel incat sa trateze si situatia de mai sus.

10.2 Operatorul de indexare al clasei `tablouVI<T>` initializeaza fiecare element cu valoarea implicita, chiar la primul acces al elementului respectiv. Procedeul este oarecum ineficient, deoarece memorarea valorii unui element are sens doar daca aceasta valoare este diferita de valoarea implicita.

Completati clasa `tablouVI<T>` astfel incat sa fie memorate efectiv doar valorile diferite de valoarea implicita.

10.3 Ceea ce diferentiaza operatorul de indexare din clasa `tablouVI<T>` fata de cel din clasa `tablou<T>` este, in cele din urma, verificarea indicelui:

- in clasa `tablou<T>` este o vorba de o simpla incadrare intre `0` si `d`
- in clasa `tablouVI<T>` este un algoritm care necesita verificarea corelatiilor dintre tablourile `p` si `b`.

Implementati precedura virtuala `check( int )` pentru verificarea indicelui in cele doua clase si modificati operatorul de indexare din clasa `tablou<T>` astfel incat operatorul de indexare din clasa `tablouVI<T>` sa nu mai fie necesar.

10.4 Implementati constructorul de copiere, operatorul de atribuire si functia de redimensionare pentru obiecte de tip `tablouVI<T>`.