4. Tipuri abstracte de date

In acest capitol, vom implementa cateva din structurile de date prezentate in Capitolul 3. Utilitatea acestor implementari este dubla. In primul rand, le vom folosi pentru a exemplifica programarea orientata pe obiect prin elaborarea unor noi tipuri abstracte. In al doilea rand, ne vor fi utile ca suport puternic si foarte flexibil pentru implementarea algoritmilor studiati in Capitolele 6-9. Utilizand tipuri abstracte pentru principalele structuri de date, ne vom putea concentra exclusiv asupra algoritmilor pe care dorim sa ii programam, fara a mai fi necesar sa ne preocupam de implementarea structurilor necesare.

Elaborarea fiecarei clase cuprinde doua etape, nu neaparat distincte. In prima, vom stabili facilitatile clasei, adica functiile si operatorii prin care se realizeaza principalele operatii asociate tipului abstract. De asemenea, vom stabili structura interna a clasei, adica datele membre si functiile nepublice. Etapa a doua cuprinde programarea, testarea si depanarea clasei, astfel incat, in final, sa avem garantia bunei sale functionari. Intregul proces de elaborare cuprinde numeroase reveniri asupra unor aspecte deja stabilite, iar fiecare modificare atrage dupa sine o intreaga serie de alte modificari. Nu vom prezenta toate aceste iteratii, desi ele au fost destul de numeroase, ci doar rezultatele finale, comentand pe larg, atat facilitatile clasei, cat si detaliile de implementare. Vom explica astfel si cateva aspecte ale programarii orientate pe obiect in limbajul C++, cum sunt clasele parametrice si mostenirea (derivarea). Dorim ca prin aceasta maniera de prezentare sa oferim posibilitatea de a intelege modul de functionare si utilizare al claselor descrise, chiar daca anumite aspecte, legate in special de implementare, nu sunt suficient aprofundate.

4.1 Tablouri

In mod surprinzator, incepem cu tabloul, structura fundamentala, predefinita in majoritatea limbajelor de programare. Necesitatea de a elabora o noua structura de acest tip provine din urmatoarele inconveniente ale tablourilor predefinite, inconveniente care nu sunt proprii numai limbajelor C si C++:

- Numarul elementelor unui tablou trebuie sa fie o expresie constanta, fixata in momentul compilarii.
- Pe parcursul executiei programului este imposibil ca un tablou sa fie marit sau micsorat dupa necesitati.

- Nu se verifica incadrarea in limitele admisibile a indicilor elementelor tablourilor.
- Tabloul si numarul elementelor lui sunt doua entitati distincte. Orice operatie cu tablouri (atribuiri, transmiteri de parametri etc) impune specificarea explicita a numarului de elemente ale fiecarui tablou.

4.1.1 Alocarea dinamica a memoriei

Diferenta fundamentala dintre tipul abstract pe care il vom elabora si tipul tablou predefinit consta in *alocarea dinamica*, in timpul executiei programului, a spatiului de memorie necesar stocarii elementelor sale. In limbajul C, alocarea dinamica se realizeaza prin diversele variante ale functiei `malloc()`, iar eliberarea zonelor alocate se face prin functia `mfree()`. Limbajul C++ a introdus alocarea dinamica in structura limbajului. Astfel, pentru alocare avem operatorul `new`. Acest operator returneaza adresa* zonei de memorie alocata, sau valoarea `0` – daca alocarea nu s-a putut face. Pentru eliberarea memoriei alocate prin intermediul operatorului `new`, se foloseste un alt operator numit `delete`. Programul urmator exemplifica detaliat functionarea acestor doi operatori.

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

int main( ) {
  // Operatorul new are ca argumente numele unui tip T
  // (predefinit sau definit de utilizator) si dimensiunea
  // zonei care va fi alocata. Valoarea returnata este de
  // tip "pointer la T". Operatorul new returneaza 0 in
  // cazul in care alocarea nu a fost posibila.

  // se aloca o zona de 2048 de intregi
  int *pi = new int [ 2048 ];

  // se aloca o zona de 64 de elemente de tip
  // intErval cu domeniul implicit
  intErval *pi_m = new intErval [ 64 ];

  // se aloca o zona de 8192 de elemente de tip float
  float *pf = new float [ 8192 ];
```

* In limbajul C++, tipul de data care contine adrese este numit *pointer*. In continuare, vom folosi termenul “pointer”, doar atunci cand ne referim la tipul de data. Termenul “adresa” va fi folosit pentru a ne referi la valoarea datelor de tip pointer.

```
    // De asemenea, operatorul new poate fi folosit pentru
    // alocarea unui singur element de un anumit tip T,
    // precizand eventual si argumentele constructorului
    // tipului respectiv.

    // se aloca un intreg initializat cu 8
    int *i = new int( 8 );

    // se aloca un element de tip intErval
    // cu domeniul admisibil -16, ..., 15
    intErval *m = new intErval( 16, -16 );

    // se aloca un numar real (float) initializat cu 32
    float *f = new float( 32 );


    // Zonele alocate pot fi eliberate oricand si in orice
    // ordine, dar numai prin intermediul pointerului
    // returnat de operatorul new.

    delete [ ] pf;
    delete [ ] pi;
    delete i;
    delete f;
    delete [ ] pi_m;
    delete m;

    return 0;
}
```

Operatorul `new` initializeaza memoria alocata prin intermediul constructorilor tipului respectiv. In cazul alocarii unui singur element, se invoca constructorul corespunzator argumentelor specificate, iar in cazul alocarii unui tablou de elemente, operatorul `new` invoca constructorul implicit pentru fiecare din elementele alocate. Operatorul `delete`, inainte de eliberarea spatiului alocat, va invoca destructorul tipului respectiv. Daca zona alocata contine un tablou de elemente si se doreste invocarea destructorului pentru fiecare element in parte, operatorul `delete` va fi invocat astfel:

```
delete [ ] pointer;
```

De exemplu, ruland programul

```
#include <iostream.h>

class X {
public:
  X( )  { cout << '*'; }
  ~X( ) { cout << '~'; }
private:
  int x;
};

int main( ) {
  cout << '\n';

  X *p =new X [ 4 ];
  delete p;

  p = new X [ 2 ];
  delete [ ] p;

  cout << '\n';
  return 0;
}
```

constatam ca, in alocarea zonei pentru cele patru elemente de tip `X`, constructorul `X()` a fost invocat de patru ori, iar apoi, la eliberare, destructorul `~X()` doar o singura data. In cazul zonei de doua elemente, atat constructorul cat si destructorul au fost invocati de cate doua ori. Pentru unele variante mai vechi de compilatoare C++, este necesar sa se specifice explicit numarul elementelor din zona ce urmeaza a fi eliberata.

In alocarea dinamica, cea mai uzuala eroare este generata de imposibilitatea alocarii memoriei. Pe langa solutia banala, dar extrem de incomoda, de testare a valorii adresei returnate de operatorul `new`, limbajul C++ ofera si posibilitatea invocarii, in caz de eroare, a unei functii definite de utilizator. Rolul acesteia este de a obtine memorie, fie de la sistemul de operare, fie prin eliberarea unor zone deja ocupate. Mai exact, atunci cand operatorul `new` nu poate aloca spatiul solicitat, el invoca functia a carei adresa este data de variabila globala `_new_handler` si apoi incearca din nou sa aloce memorie. Variabila `_new_handler` este de tip “pointer la functie de tip `void` fara nici un argument”, `void (*_new_handler)()`, valoarea ei implicita fiind `0`.

Valoarea `0` a pointerului `_new_handler` marcheaza lipsa functiei de tratare a erorii si, in aceasta situatie, operatorul `new` va returna `0` ori de cate ori nu poate aloca memoria necesara. Programatorul poate modifica valoarea acestui pointer, fie direct:

```
_new_handler = no_mem;
```

unde `no_mem` este o functie de tip `void` fara nici un argument,

```
void no_mem( ) {
  cerr << "\n\n no mem. \n\n";
  exit( 1 );
}
```

fie prin intermediul functiei de biblioteca `set_new_handler`:

```
set_new_handler( no_mem );
```

Toate declaratiile necesare pentru utilizarea pointerului `_new_handler` se gasesc in fisierul header `new.h`.

4.1.2 Clasa `tablou`

Noul tip, numit `tablou`, va avea ca date membre numarul de elemente si adresa zonei de memorie in care sunt memorate acestea. Datele membre fiind `private`, adica inaccesibile din exteriorul clasei, oferim posibilitatea obtinerii numarului elementelor tabloului prin intermediul unei functii membre publice numita `size()`. Iata definitia completa a clasei `tablou`.

```
class tablou {
public:
  // constructorii si destructorul
  tablou( int = 0 );       // constructor (numarul de elemente)
  tablou( const tablou& ); // constructor de copiere
 ~tablou( )  { delete a; } // elibereaza memoria alocata

  // operatori de atribuire si indexare
  tablou& operator =( const tablou& );
  int& operator []( int );

  // returneaza numarul elementelor
  size( ) { return d; }

private:
  int  d; // numarul elementelor (dimensiunea) tabloului
  int *a; // adresa zonei alocate

  // functie auxiliara de initializare
  void init( const tablou& );
};
```

Definitiile functiilor membre sunt date in continuare.

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

tablou::tablou( const tablou& t ) {
  // initializarea obiectului invocator cu t
  init( t );
}

tablou& tablou::operator =( const tablou& t ) {
  if ( this != &t ) { // este o atribuire inefectiva x = x?
    delete a;         // eliberarea memoriei alocate
    init( t );        // initializarea cu t
  }
  return *this;       // se returneaza obiectul invocator
}

void tablou::init( const tablou& t ) {
  a = 0; d = 0;              // valori implicite
  if ( t.d > 0 ) {           // verificarea dimensiunii
    a = new int [ d = t.d ]; // alocarea si copierea elem.
    memcpy( a, t.a, d * sizeof( int ) );
  }
}

int& tablou::operator []( int i ) {
  static int z; // "elementul" tablourilor de dimensiune zero
  return d? a[ i ]: z;
}
```

Fara indoiala ca cea mai spectaculoasa definitie este cea a operatorului de indexare `[]`. Acesta permite atat citirea unui element dintr-un `tablou`:

```
tablou x( n );
// ...
cout << x[ i ];
```

cat si modificarea valorii (scrierea) lui:

```
cin >> x[ i ];
```

Facilitatile deosebite ale operatorului de indexare `[]` se datoreaza tipului valorii returnate. Acest operator nu returneaza elementul `i`, ci o referinta la elementul `i`, referinta care permite accesul atat in scriere, cat si in citire a variabilei de la adresa respectiva.

Clasa `tablou` permite utilizarea tablourilor in care nu exista nici un element. Operatorul de indexare `[]` este cel mai afectat de aceasta posibilitate, deoarece intr-un tablou cu zero elemente va fi greu de gasit un element a carui referinta sa fie returnata. O solutie posibila consta in returnarea unui element fictiv, unic pentru toate obiectele de tip `tablou`. In cazul nostru, acest element este variabila locala `static int z`, variabila alocata static, adica pe toata durata rularii programului.

O atentie deosebita merita si operatorul de atribuire `=`. Dupa cum am precizat in Sectiunea 2.3, structurile pot fi atribuite intre ele, membru cu membru. Pentru clasa `tablou`, acest mod de functionare a operatorului implicit de atribuire este inacceptabil, deoarece genereaza referiri multiple la aceeasi zona de memorie. Iata un exemplu simplu de ceea ce inseamna referiri multiple la aceeasi zona de memorie.

Fie `x` si `y` doua obiecte de tip tablou. In urma atribuirii `x = y` prin operatorul predefinit `=`, ambele obiecte folosesc aceeasi zona de memorie pentru memorarea elementelor. Daca unul dintre ele inceteaza sa mai existe, atunci destructorul sau ii va elibera zona alocata. In consecinta, celalalt va lucra intr-o zona de memorie considerata libera, zona care poate fi alocata oricand altui obiect. Prin definirea unui nou operator de atribuire specific clasei `tablou`, obiectele din aceasta clasa sunt atribuite corect, fiecare avand propria zona de memorie in care sunt memorate elementele.

O alta observatie relativa la operatorul de atribuire se refera la valoarea returnata. Tipurile predefinite permit concatenarea operatorului de atribuire in expresii de forma

```
i = j = k;
// unde i, j si k sunt variabile de orice tip predefinit
```

Sa vedem ce trebuie sa facem ca, prin noul operator de atribuire definit, sa putem scrie

```
iT = jT = kT;
// iT, jT si kT sunt obiecte de tip tablou
```

Operatorul de atribuire predefinit are asociativitate de dreapta (se evalueaza de la dreapta la stanga) si aceasta caracteristica ramane neschimbata la supraincarcare. Altfel spus, `iT = jT = kT` inseamna de fapt `iT = (jT = kT)`, sau `operator =( iT, operator =( jT, kT) )`. Rezulta ca operatorul de atribuire trebuie sa returneze operandul stang, sau o referinta la acesta. In cazul nostru, operandul stang este chiar obiectul invocator. Cum in fiecare functie membra este implicit definit un pointer la obiectul invocator, pointer numit `this` (acesta),

operatorul de atribuire va returna o referinta la obiectul invocator prin instructiunea

```
return *this;
```

Astfel, sintaxa de concatenare poate fi folosita fara nici o restrictie.

In definitia clasei tablou a aparut un nou constructor, *constructorul de copiere*

```
tablou( const tablou& )
```

Este un constructor a carui implementare seamana foarte mult cu cea a operatorului de atribuire. Rolul sau este de a initializa obiecte de tip `tablou` cu obiecte de acelasi tip. O astfel de operatie, ilustrata in exemplul de mai jos, este in mare masura similara unei copieri.

```
tablou x;
// ...
tablou y = x;   // se invoca constructorul de copiere
```

In lipsa constructorului de copiere, initializarea se face implicit, adica membru cu membru. Consecintele negative care decurg de aici au fost discutate mai sus.

4.1.3 Clasa parametrica `tablou<T>`

Utilitatea clasei `tablou` este strict limitata la tablourile de intregi, desi un tablou de `float`, `char`, sau de orice alt tip `T`, se manipuleaza la fel, functiile si datele membre fiind practic identice. Pentru astfel de situatii, limbajul C++ ofera posibilitatea generarii automate de clase si functii pe baza unor *sabloane* (*template*). Aceste sabloane, numite si *clase parametrice*, respectiv *functii parametrice*, depind de unul sau mai multi parametri care, de cele mai multe ori, sunt tipuri predefinite sau definite de utilizator.

Sablonul este o declaratie prin care se specifica forma generala a unei clase sau functii. Iata un exemplul simplu: o functie care returneaza maximul a doua valori de tip `T`.

```
template <class T>
T max( T a, T b ) {
  return a > b? a: b;
}
```

Acest sablon se citeste astfel: `max()` este o functie cu doua argumente de tip `T`, care returneaza maximul celor doua argumente, adica o valoare de tip `T`. Tipul `T`

poate fi orice tip predefinit, sau definit de utilizator, cu conditia sa aiba definit operatorul de comparare `>`, fara de care functia `max()` nu poate functiona.

Compilatorul nu genereaza nici un fel de cod pentru sabloane, pana in momentul in care sunt efectiv folosite. De aceea, sabloanele se specifica in fisiere header, fisiere incluse in fiecare program sursa C++ in care se utilizeaza clasele sau functiile parametrice respective*. De exemplu, in functia

```
void f( int ia, int ib, float fa ) {
  int   m1 = max( ia, ib );
  float m2 = max( ia, fa );
}
```

se invoca functiile `int max(int, int)` si `float max(float, float)`, functii generate automat, pe baza sablonului de mai sus

Conform specificatiilor din Ellis si Stroustrup, *"The Annotated C++ Reference Manual"*, generarea sabloanelor este un proces care nu implica nici un fel de conversii. In consecinta, linia

```
  float m2 = max( ia, fa );
```

este eronata. Unele compilatoare nu semnaleaza aceasta erorare, deoarece invoca totusi conversia lui `ia` din `int` in `float`. Atunci cand compilatorul semnaleaza eroarea, putem declara explicit functia (vezi si Sectiunea 10.2.3)

```
float max( float, float );
```

declaratie care nu mai necesita referirea la sablonul functiei `max()`. Aceasta declaratie este, in general, suficienta pentru a genera functia respectiva pe baza sablonului.

Pana cand limbajul C++ va deveni suficient de matur pentru a fi standardizat, "artificiile" de programare de mai sus sunt deseori indispensabile pentru utilizarea sabloanelor.

Pentru sabloanele de clase, lucrurile decurg aproximativ in acelasi mod, adica generarea unei anumite clase este declansata de definitiile intalnite in program. Pentru clasa parametrica `tablou<T>` definitiile

---

* In prezent sunt utilizate doua modele generale pentru instantierea (generarea) sabloanelor, fiecare cu anumite avantaje si dezavantaje. Reprezentative pentru aceste modele sunt compilatoarele Borland C++ si translatoarele Cfront de la AT&T. Ambele modele sunt compatibile cu plasarea sabloanelor in fisiere header.

```
tablou<float> y( 16 );
tablou<int> x( 32 );
tablou<unsigned char> z( 64 );
```

provoaca generarea clasei `tablou<T>` pentru tipurile `float`, `int` si `unsigned char`. Fisierul header (`tablou.h`) al acestei clase este:

```
#ifndef __TABLOU_H
#define __TABLOU_H

#include <iostream.h>

template <class T>
class tablou {
public:
  // constructorii si destructorul
  tablou( int = 0 );       // constructor (numarul de elemente)
  tablou( const tablou& ); // constructor de copiere
 ~tablou( )  { delete [ ] a; } // elibereaza memoria alocata

  // operatori de atribuire si indexare
  tablou& operator =( const tablou& );
  T& operator []( int );

  // returneaza numarul elementelor
  size( ) { return d; }

  // activarea/dezactivarea verificarii indicilor
  void vOn ( ) { v = 1; }
  void vOff( ) { v = 0; }

protected:
  int  d; // numarul elementelor (dimensiunea) tabloului
  T *a;   // adresa zonei alocate
  char v; // indicator verificare indice

  // functie auxiliara de initializare
  void init( const tablou& );
};

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
}
```

```
template <class T>
tablou<T>::tablou( const tablou<T>& t ) {
  // initializarea obiectului invocator cu t
  init( t );
}

template <class T>
tablou<T>& tablou<T>::operator =( const tablou<T>& t ) {
  if ( this != &t ) { // este o atribuire inefectiva x = x?
    delete [ ] a;     // eliberarea memoriei alocate
    init( t );        // initializarea cu t
  }
  return *this;       // se returneaza obiectul invocator
}

template<class T>
void tablou<T>::init( const tablou<T>& t ) {
  a = 0; v = 0; d = 0;     // valori implicite
  if ( t.d > 0 ) {         // verificarea dimensiunii
    a = new T [ d = t.d ]; // alocarea si copierea elem.
    for ( int i = 0; i < d; i++ ) a[ i ] = t.a[ i ];
    v = t.v;               // duplicarea indicatorului
  }                        // pentru verificarea indicilor
}

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

  if ( d == 0 ) // tablou de dimensiune zero
    return z;

  if ( v == 0 || ( 0 <= i && i < d ) )
    // verificarea indicilor este dezactivata,
    // sau este activata si indicele este corect
    return a[ i ];

  cerr << "\n\ntablou -- " << i
       << ": indice exterior domeniului [0, "
       << ( d - 1 ) << "].\n\n";

  return z;
}
```

Intr-o prima aproximare, diferentele fata de clasa neparametrica `tablou` sunt urmatoarele:

- Nivelul de incapsulare `protected` a inlocuit nivelul `private`. Este o modificare necesara procesului de derivare al claselor, prezentat in sectiunile urmatoare.

- Eliberarea zonei alocate dinamic trebuie sa se realizeze prin invocarea destructorului tipului `T` pentru fiecare element. Deci, in loc de `delete a`, este obligatoriu sa scriem `delete [] a` atat in destructor, cat si in operatorul de atribuire. De asemenea, copierea elementelor in functia `init()` nu se mai poate face global, prin `memcpy()`, ci element cu element, pentru a invoca astfel opratorul de atribuire al tipului `T`.
- Prezenta definitiilor functiilor membre in fisierul header nu este o greseala. De fapt, este vorba de sabloanele functiilor membre.

Printre inconvenientele tablourilor predefinite am enumerat si imposibilitatea detectarii indicilor eronati. Dupa cum se observa, am completat clasa parametrica `tablou<T>` cu functiile publice `vOn()` si `vOff()`, prin care se activeaza, respectiv se dezactiveaza, verificarea indicilor. In functie de valoarea logica a variabilei private `v`, valoare stabilita prin functiile `vOn()` si `vOff()`, operatorul de indexare va verifica, sau nu va verifica, corectitudinea indicelui. Operatorul de indexare a fost modificat corespunzator.

Pentru citirea si scrierea obiectelor de tip `tablou<T>`, supraincarcam operatorii respectivi (`>>` si `<<`) ca functii nemembre. Convenim ca, in operatiile de citire/scriere, sa reprezentam tablourile in formatul

```
[dimensiune] element1 element2 ...
```

Cei doi operatori pot fi implementati astfel:

```
template <class T>
istream& operator >>( istream& is, tablou<T>& t ) {
  char c;

  // citirea dimensiunii tabloului incadrata de '[' si ']'
  is >> c;
  if ( c != '[' ) { is.clear( ios::failbit ); return is; }
  int n; is >> n; is >> c;
  if ( c != ']' ) { is.clear( ios::failbit ); return is; }

  // modificarea dimensiunii tabloului,
  // evitand copierea elementelor existente
  t.newsize( 0 ).newsize( n );

  // citirea elementelor
  for ( int i = 0; i < n; is >> t[ i++ ] );

  return is;
}
```

```
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;
}
```

Acesti operatori sunt utilizabili doar daca obiectelor de tip `T` li se pot aplica operatorii de extragere/inserare `>>`, respectiv `<<`. In caz contrar, orice incercare de a aplica obiectelor de tip `tablou<T>` operatorii mai sus definiti, va fi semnalata ca eroare la compilarea programului.

Operatorul de extragere (citire) `>>` prezinta o anumita particularitate fata de celelalte functii care opereaza asupra tablourilor: trebuie sa modifice chiar dimensiunea tabloului. Doua variante de a realiza aceasta operatie, dintre care una prin intermediul functiei `newsize( )`, sunt discutate in Exercitiile 4.2 si 4.3.

Marcarea erorilor la citire se realizeaza prin modificarea corespunzatoare a starii `istream`-ului prin

```
is.clear( ios::failbit );
```

Dupa cum am precizat in Sectiunea 2.3.2, starea unui `istream` se poate testa printr-un simplu `if ( cin >> ... )`. Odata ce un `istream` a ajuns intr-o stare de eroare, nu mai raspunde la operatorii respectivi, decat dupa ce este readus la starea normala de utilizare prin instructiunea

```
is.clear();
```

4.2 Stive, cozi, heap-uri

Stivele, cozile si heap-urile sunt, in esenta, tablouri manipulate altfel decat prin operatorul de indexare. Acesata afirmatie contrazice aparent definitiile date in Capitolul 3. Aici se precizeaza ca stivele si cozile sunt liste liniare in care inserarile/extragerile se fac conform unor algoritmi particulari, iar heap-urile sunt arbori binari completi. Tot in Capitolul 3 am aratat ca reprezentarea cea mai comoda pentru toate aceste structuri este cea secventiala, bazata pe tablouri.

In terminologia specifica programarii orientate pe obiect, spunem ca tipurile `stiva<T>`, `coada<T>` si `heap<T>` sunt *derivate* din tipul `tablou<T>`, sau ca *mostenesc* tipul `tablou<T>`. Tipul `tablou<T>` se numeste *tip de baza* pentru

tipurile `stiva<T>`, `coada<T>` si `heap<T>`. Prin mostenire, limbajul C++ permite atat crearea unor subtipuri ale tipului de baza, cat si crearea unor tipuri noi, diferite de tipul de baza. Stivele, cozile si heap-urile vor fi tipuri noi, diferite de tipul de baza tablou. Posibilitatea de a crea subtipuri prin derivare, o facilitate deosebit de puternica a programarii orientate pe obiect si a limbajului C++, va fi exemplificata in Sectiunile 11.1 si 10.2.

4.2.1 Clasele `stiva<T>` si `coada<T>`

Clasa `stiva<T>` este un tip nou, derivat din clasa `tablou<T>`. In limbajul C++, derivarea se indica prin specificarea claselor de baza (pot fi mai multe!), imediat dupa numele clasei.

```
template <class T>
class stiva: private tablou<T> {
// ....
};
```

Fiecare clasa de baza este precedata de atributul `public` sau `private`, prin care se specifica modalitatea de mostenire. O clasa derivata `public` este un subtip al clasei de baza, iar una derivata `private` este un tip nou, distinct fata de tipul de baza.

Clasa derivata mosteneste toti membrii clasei de baza, cu exceptia constructorilor si destructorilor, dar nu are acces la membrii `private` ai clasei de baza. Atunci cand este necesar, acest incovenient poate fi evitat prin utilizarea in clasa de baza a nivelului de acces `protected` in locul celui `private`. Membrii `protected` sunt membri privati, dar accesibili claselor derivate. Nivelul de acces al membrilor mosteniti se modifica prin derivare astfel:

- Membrii neprivati dintr-o clasa de baza publica isi pastreaza nivelele de acces si in clasa derivata.
- Membrii neprivati dintr-o clasa de baza privata devin membri `private` in clasa derivata.

Revenind la clasa `stiva<T>`, putem spune ca mosteneste de la clasa de baza `tablou<T>` membrii

```
int d;
T  *a;
```

ca membri `private`, precum si cei doi operatori (publici in clasa `tablou<T>`)

```
tablou& operator =( const tablou& );
T& operator []( int );
```

tot ca membri `private`.

Pe baza celor de mai sus, se justifica foarte simplu faptul ca prin derivarea privata se obtin tipuri noi, total distincte fata de tipul de baza. Astfel, nu este disponibila nici una din facilitatile clasei de baza `tablou<T>` in exteriorul clasei `stiva<T>`, existenta clasei de baza fiind total ascunsa utilizatorului. In schimb, pentru implementarea propriilor facilitati, clasa `stiva<T>` poate folosi din plin toti membrii clasei `tablou<T>`. Prin derivarea `private`, realizam deci o *reutilizare* a clasei de baza.

Definirea unei stive derivata din tablou se realizeaza astfel (fisierul `stiva.h`):

```
#ifndef __STIVA_H
#define __STIVA_H

#include <iostream.h>
#include "tablou.h"

template <class T>
class stiva: private tablou<T> {
public:
  stiva( int d ):  tablou<T>( d ) { s = -1; }

  push( const T& );
  pop (       T& );

private:
  int s;  // indicele ultimului element inserat
};

template <class T>
stiva<T>::push( const T& v ) {
  if ( s >= d - 1 ) return 0;
  a[ ++s ] = v;     return 1;
}

template <class T>
stiva<T>::pop( T& v ) {
  if ( s < 0 )  return 0;
  v = a[ s-- ]; return 1;
}

#endif
```

Inainte de a discuta detaliile de implementare, sa remarcam o anumita inconsecventa aparuta in definitia functiei `pop()` din Sectiunea 3.1.1. Aceasta

functie returneaza fie elementul din varful stivei, fie un mesaj de eroare (atunci cand stiva este vida). Desigur ca nu este un detaliu deranjant atat timp cat ne intereseaza doar algoritmul. Dar, cum implementam efectiv aceasta functie, astfel incat sa cuprindem ambele situatii? Intrebarea poate fi formulata in contextul mult mai general al tratarii exceptiilor. Rezolvarea unor cazuri particulare, a exceptiilor de la anumite reguli, problema care nu este strict de domeniul programarii, poate da mai putine dureri de cap prin aplicarea unor principii foarte simple. Iata, de exemplu, un astfel de principiu formulat de Winston Churchill: “Nu ma intrerupeti in timp ce intrerup”.

Tratarea exceptiilor devine o chestiune foarte complicata, mai ales in cazul utilizarii unor functii sau obiecte dintr-o biblioteca. Autorul unei biblioteci de functii (obiecte) poate detecta exceptiile din timpul executiei dar, in general, nu are nici o idee cum sa le trateze. Pe de alta parte, utilizatorul bibliotecii stie ce sa faca in cazul aparitiei unor exceptii, dar nu le poate detecta. Notiunea de exceptie, notiune acceptata de Comitetul de standardizare ANSI C++, introduce un mecanism consistent de rezolvare a unor astfel de situatii. Ideea este ca, in momentul cand o functie detecteaza o situatie pe care nu o poate rezolva, sa semnaleze (`throw`) o exceptie, cu speranta ca una din functiile (direct sau indirect) invocatoare va rezolva apoi problema. O functie care este pregatita pentru acest tip de evenimente isi va anunta in prealabil disponibilitatea de a trata (`catch`) exceptii.

Mecanismul schitat mai sus este o alternativa la tehnicile traditionale, atunci cand acestea se dovedesc a fi inadecvate. El ofera o cale de separare explicita a secventelor pentru tratarea erorilor de codul propriu-zis, programul devenind astfel mai clar si mult mai usor de intretinut. Din pacate, la nivelul anului 1994, foarte putine compilatoare C++ implementeaza complet mecanismul `throw`–`catch`. Revenim de aceea la “stilul clasic”, stil independent de limbajul de programare folosit. Uzual, la intalnirea unor erori se actioneaza in unul din urmatoarele moduri:

- Se termina programul.
- Se returneaza o valoare reprezentand “eroare”.
- Se returneaza o valoare legala, programul fiind lasat intr-o stare ilegala.
- Se invoca o functie special construita de programator pentru a fi apelata in caz de eroare.

Terminarea programului se realizeaza prin revenirea din functia `main()`, sau prin invocarea unei functii de biblioteca numita `exit()`. Valoarea returnata de `main()`, precum si argumentul intreg al functiei `exit()`, este interpretat de sistemul de operare ca un *cod de retur* al programului. Un cod de retur nul (zero) semnifica executarea corecta a programului.

Pana in prezent, am utilizat tratarea exceptiilor prin terminarea programului in clasa `intErval`. Un alt exemplu de tratare a exceptiilor se poate remarca la

operatorul de indexare din clasa `tablou<T>`. Aici am utilizat penultima alternativa din cele patru enuntate mai sus: valoarea returnata este legala, dar programul nu a avut posibilitatea de a trata eroarea.

Pentru stiva si, de fapt, pentru multe din structurile implementate aici si susceptibile la situatii de exceptie, am ales varianta a doua: returnarea unei valori reprezentand “eroare”. Pentru a putea distinge cat mai simplu situatiile normale de cazurile de exceptie, am convenit ca functia `pop()` sa transmita elementul din varful stivei prin intermediul unui argument de tip referinta, valoarea returnata efectiv de functie indicand existenta sau inexistenta acestui element. Astfel, secventa

```
while( s.pop( v ) ) {
  // ...
}
```

se executa atat timp cat in stiva `s` mai sunt elemente, variabila `v` avand de fiecare data valoarea elementului din varful stivei. Functia `push()` are un comportament asemanator, secventa

```
while( s.push( v ) ) {
  // ...
}
```

executandu-se atata timp cat in stiva se mai pot insera elemente.

In continuare, ne propunem sa analizam mai amanuntit contributia clasei de baza `tablou<T>` in functionarea clasei `stiva<T>`. Sa remarcam mai intai invocarea constructorului tipului de baza pentru initializarea datelor membre mostenite, invocare realizata prin *lista de initializare a membrilor*:

```
stiva( int d ): tablou<T>( d ) { s = -1; }
```

Utilizarea acestei sintaxe speciale se datoreaza faptului ca executia oricarui constructor se face in doua etape. Intr-o prima etapa, *etapa de initializare*, se invoca constructorii datelor membre mostenite de la clasele de baza, conform listei de initializare a membrilor. In a doua etapa, numita *etapa de atribuire*, se executa corpul propriu-zis al constructorului. Necesitatea unei astfel de etapizari se justifica prin faptul ca initializarea membrilor mosteniti trebuie rezolvata in mod unitar de constructorii proprii, si nu de cel al clasei derivate. Daca lista de initializare a membrilor este incompleta, atunci, pentru membrii ramasi neinitializati, se invoca constructorii impliciti. De asemenea, tot in etapa de initializare se vor invoca constructorii datelor membre de tip clasa si se vor initializa datele membre de tip `const` sau referinta.

Continuand analiza contributiei tipului de baza `tablou<T>`, sa remarcam ca in clasa `stiva<T>` nu s-au definit constructorul de copiere, operatorul de atribuire si destructorul. Initializarea si atribuirea obiectelor de tip stiva cu obiecte de acelasi tip, precum si distrugerea acestora, se realizeaza totusi corect, datele membre mostenite de la `tablou<T>` fiind manipulate de functiile membre ale acestui tip. In functia

```
void f( ) {
  stiva<int> x( 16 );
  stiva<int> y = x;
  x = y;
}
```

initializarea lui `y` cu `x` se face membru cu membru, pentru datele proprii clasei `stiva<T>` (intregul `top`), si prin invocarea constructorului de copiere al clasei `tablou<T>`, pentru initializarea datelor membre mostenite (intregul `d` si adresa `a`). Atribuirea `x = y` se efectueaza membru cu membru, pentru datele proprii, iar pentru cele mostenite, prin invocarea operatorului de atribuire al clasei `tablou<T>`. La terminarea functiei, obiectele `x` si `y` vor fi distruse prin invocarea destructorilor in ordinea inversa a invocarii constructorilor, adica destructorul clasei `stiva<T>` (care nu a fost precizat pentru ca nu are de facut nimic) si apoi destructorul clasei de baza `tablou<T>`.

Implementarea clasei `coada<T>` se face pe baza precizarilor din Sectiunea 3.1.2, direct prin modificarea definitiei clasei `stiva<T>`. In locul indicelui `top`, vom avea doua date membre, si anume indicii `head` si `tail`, iar functiile membre `push()` si `pop()` vor fi inlocuite cu `ins_q()`, respectiv `del_q()`. Ca exercitiu, va propunem sa realizati implementarea efectiva a acestei clase.

4.2.2 Clasa heap<T>

Vom utiliza structura de heap descrisa in Sectiunea 3.4 pentru implementarea unei clase definita prin operatiile de inserare a unei valori si de extragere a maximului. Clasa parametrica `heap<T>` seamana foarte mult cu clasele `stiva<T>` si `coada<T>`. Diferentele apar doar la implementarea operatiilor de inserare in heap si de extragere a maximului. Definitia clasei `heap<T>` este:

```
#ifndef __HEAP_H
#define __HEAP_H

#include <iostream.h>
#include <stdlib.h>
#include "tablou.h"
```

```
template <class T>
class heap: private tablou<T> {
public:
  heap( int d ): tablou<T>( d ) { h = -1; }
  heap( const tablou<T>& t ): tablou<T>( t )
    { h = t.size( ) - 1; make_heap( ); }

  insert    ( const T& );
  delete_max(       T& );

protected:
  int h;      // indicele ultimului element din heap

  void percolate( int );
  void sift_down( int );
  void make_heap( );
};

template <class T>
heap<T>::insert( const T& v ) {
  if ( h >= d - 1 ) return 0;
  a[ ++h ] = v; percolate( h );
  return 1;
}

template <class T>
heap<T>::delete_max( T& v ) {
  if ( h < 0 ) return 0;
  v = a[ 0 ];
  a[ 0 ] = a[ h-- ]; sift_down( 0 );
  return 1;
}

template <class T>
void heap<T>::make_heap( ) {
  for ( int i = (h + 1) / 2; i >= 1; sift_down( --i ) );
}

template <class T>
void heap<T>::percolate( int i ) {
  T *A = a - 1;  // a[ 0 ]     este A[ 1 ], ...,
                 // a[ i - 1 ] este A[ i ]
  int k = i + 1, j;

  do {
    j = k;
    if ( j > 1 && A[ k ] > A[ j/2 ] ) k = j/2;
    T tmp = A[ j ]; A[ j ] = A[ k ]; A[ k ] = tmp;
  } while ( j != k );
}
```

```
template <class T>
void heap<T>::sift_down( int i ) {
  T *A = a - 1;  // a[ 0 ]     este A[ 1 ], ...,
                 // a[ n - 1 ] este A[ n ]
  int n = h + 1, k = i + 1, j;

  do {
    j = k;
    if ( 2*j <= n && A[ 2*j   ] > A[ k ] ) k = 2*j;
    if ( 2*j <  n && A[ 2*j+1 ] > A[ k ] ) k = 2*j+1;
    T tmp = A[ j ]; A[ j ] = A[ k ]; A[ k ] = tmp;
  } while ( j != k );
}

#endif
```

Procedurile `insert()` si `delete_max()` au fost adaptate stilului de tratare a exceptiilor prezentat in sectiunea precedenta: ele returneaza valorile logice *true* sau *false*, dupa cum operatiile respective sunt, sau nu sunt posibile.

Clasa `heap<T>` permite crearea unor heap-uri cu elemente de cele mai diverse tipuri: `int`, `float`, `long`, `char` etc. Dar incercarea de a defini un heap pentru un tip nou `T`, definit de utilizator, poate fi respinsa chiar in momentul compilarii, daca acest tip nu are definit operatorul de comparare `>`. Acest operator, a carui definire ramane in sarcina proiectantului clasei `T`, trebuie sa returneze *true* (o valoare diferita de `0`) daca argumentele sale sunt in relatia `>` si *false* (adica `0`) in caz contrar. Pentru a nu fi necesara si definirea operatorului `<`, in implementarea clasei `heap<T>` am folosit numai operatorul `>`.

Vom exemplifica utilizarea clasei `heap<T>` cu un operator `>` diferit de cel predefinit prin intermediul clasei `intErval`. Desi clasa `intErval` nu are definit operatorul `>`, programul urmator “trece” de compilare si se executa (aparent) corect.

```
#include "intErval.h"
#include "heap.h"

// dimensiunea heap-ului, margine superioara in intErval
const SIZE = 128;

int main( ) {
  heap<intErval> hi( SIZE );
  intErval v( SIZE );
```

```
  cout << "Inserare in heap (^Z/#" << (SIZE - 1) << ")\n... ";
  while ( cin >> v ) {
    hi.insert( v );
    cout << "... ";
  }
  cin.clear( );

  cout << "Extragere din heap\n";
    while ( hi.delete_max( v ) ) cout << v << '\n';

  return 0;
}
```

Justificarea corectitudinii sintactice a programului de mai sus consta in existenta operatorului de conversie de la `intErval` la `int`. Prin aceasta conversie, compilatorul rezolva compararea a doua valori de tip `intErval` (pentru operatorul `>`), sau a unei valori `intErval` cu valoarea `0` (pentru operatorul `!=`) folosind operatorii predefiniti pentru argumente de tip intreg. Utilizand acelasi operator de conversie de la `intErval` la `int`, putem defini foarte comod un operator `>`, prin care heap-ul sa devina un min-heap. Noul operator `>` este practic negarea relatiei uzuale `>`:

```
// Operatorul > pentru min-heap
int operator >( const intErval&  a, const intErval& b ) {
  return a < b;
}
```

La compilarea programului de mai sus, probabil ca ati observat un mesaj relativ la invocarea functiei “non-`const`” `intErval::operator int()` pentru un obiect `const` in functia `heap<T>::insert()`. Iata despre ce este vorba. Urmatorul program genereaza exact acelasi mesaj:

```
#include "intErval.h"

int main( ) {
  intErval       x1;
  const intErval x2( 20, 10 );

  x1 = x2;
  return 0;
}
```

Desi nu este invocat explicit, operatorul de conversie la `int` este aplicat variabilei constante `x2`. Inainte de a discuta motivul acestei invocari, sa ne oprim putin asupra manipularii obiectelor constante. Pentru acest tip de variabile (variabile constante!), asa cum este `x2`, se invoca doar functiile membre declarate explicit

`const`, functii care nu modifica obiectul invocator. O astfel de functie fiind si operatorul de conversie `intErval::operator int()`, va trebui sa-i completam definitia din clasa `intErval` cu atributul `const`:

```
operator int( ) const { return v; }
```

Acelasi efect il are si definirea non-`const` a obiectului `x2`, dar scopul nu este de a elimina mesajul, ci de a intelege (si de a elimina) cauza lui.

Atribuirea `x1 = x2` ar trebui rezolvata de operatorul de atribuire generat automat de compilator, pentru fiecare clasa. In cazul nostru, acest operator nu se invoca, deoarece atribuirea poate fi rezolvata numai prin intermediul functiilor membre explicit definite:

- `x2` este convertit la `int` prin `operator int( )`, conversie care genereaza si mesajul discutat mai sus
- Rezultatul conversiei este atribuit lui `x1` prin `operator =(int)`.

Din pacate, rezultatul atribuirii este incorect. In loc ca `x2` sa fie copiat in `x1`, va fi actualizata doar valoarea `v` a lui `x1` cu valoarea `v` lui `x2`. Evident ca, in exemplul de mai sus, `x1` va semnala depasirea domeniului sau.

Solutia pentru eliminarea acestei aparente anomalii, generate de interferenta dintre `operator int( )` si `operator =(int)`, consta in definirea explicita a operatorului de atribuire pentru obiecte de tip `intErval`:

```
intErval& intErval::operator =( const intErval& s ) {
  min = s.min; v = s.v; max = s.max;
  return *this;
}
```

Dupa ce am clarificat particularitatile obiectelor constante, este momentul sa adaptam corespunzator si clasa `tablou<T>`. Orice clasa frecvent utilizata – si `tablou<T>` este una din ele – trebuie sa fie proiectata cu grija, astfel incat sa suporte inclusiv lucrul cu obiecte constante. Vom adauga in acest scop atributul `const` functiei membre `size()`:

```
  size( ) const { return d; }
```

In plus, mai adaugam si un nou operator de indexare:

```
const T& operator []( int ) const;
```

Particularitatea acestuia consta doar in tipul valorii returnate, `const T&`, valoare imposibil de modificat. Consistenta declaratiei `const`, asociata operatorului de

indexare, este data de catre proiectantul clasei si nu poate fi verificata semantic de catre compilator. O astfel de declaratie poate fi atasata chiar si operatorului de indexare obisnuit (cel non-`const`), caci el nu modifica nici una din datele membre ale clasei `tablou<T>`. Ar fi insa absurd, deoarece tabloul se modifica de fapt prin modificarea elementelor sale.

4.3 Clasa `lista<E>`

Structurile prezentate pana acum sunt de fapt liste implementate secvential, diferentiate prin particularitatile operatiilor de inserare si extragere. In cele ce urmeaza, ne vom concentra asupra unei implementari inlantuite a listelor, prin alocarea dinamica a memoriei.

Ordinea nodurilor unei liste se realizeza prin completarea informatiei propriu-zise din fiecare nod, cu informatii privind localizarea nodului urmator si eventual a celui precedent. Informatiile de localizare, numite legaturi sau adrese, pot fi, in functie de modul de implementare ales (vezi Sectiunea 3.1), indici intr-un tablou, sau adrese de memorie. In cele ce urmeaza, fiecare nod va fi alocat dinamic prin operatorul `new`, legaturile fiind deci adrese.

Informatia din fiecare nod poate fi de orice tip, de la un numar intreg sau real la o structura oricat de complexa. De exemplu, pentru reprezentarea unui graf prin lista muchiilor, fiecare nod contine cele doua extremitati ale muchiei si lungimea (ponderea) ei. Limbajul C++ permite implementarea structurii de nod prin intermediul claselor parametrice astfel:

```
template <class E>
class nod {
// ...
  E       val;  // informatia propriu-zisa
  nod<E> *next; // adresa nodului urmator
};
```

Operatiile elementare, cum sunt parcurgerile, inserarile sau stergerile, pot fi implementate prin intermediul acestei structuri astfel:

- Parcurgerea nodurilor listei:

```
  nod<E> *a;      // adresa nodului actual
  // ...
  while ( a ) {   // adresa ultimului element are valoarea 0
    // ...          prelucrarea informatiei a->val
    a = a->next;  // notatie echivalenta cu a = (*a).next
  }
```

- Inserarea unui nou nod in lista:

```
nod<E> *a;      // adresa nodului dupa care se face inserarea
nod<E> *pn;     // adresa nodului de inserat
// ...
pn->next = a->next;
a->next  = pn;
```

- Stergerea unui nod din lista (operatie care necesita cunoasterea nu numai a adresei elementului de eliminat, ci si a celui anterior):

```
nod<E> *a;          // adresa nodului de sters
nod<E> *pp;         // adresa nodului anterior lui a
// ...
pp->next = a->next; // stergerea propriu-zisa

// ...
// eliberarea spatiului de memorie alocat nodului de
// adresa a, nod tocmai eliminat din lista
```

Structura de nod este suficienta pentru manipularea listelor cu elemente de tip E, cu conditia sa cunoastem primul nod:

```
nod<E> head;  // primul nod din lista
```

Exista totusi o lista imposibil de tratat prin intermediul acestei implementari, si anume lista vida. Problema de rezolvat este oarecum paradoxala, deoarece variabila head, primul nod din lista, trebuie sa reprezinte un nod care nu exista. Se pot gasi diverse solutii particulare, dependente de tipul si natura informatiilor. De exemplu, daca informatiile sunt valori pozitive, o valoare negativa ar putea reprezenta un nod inexistent. O alta solutie este adaugarea unei noi date membre pentru validarea existentei nodului curent. Dar este inacceptabil ca pentru un singur nod si pentru o singura situatie sa incarcam toate celelalte noduri cu inca un camp.

Imposibilitatea reprezentarii listelor vide nu este rezultatul unei proiectari defectuoase a clasei nod<E>, ci al confuziei dintre lista si nodurile ei. Identificand lista cu adresa primului ei nod si adaugand functiile uzuale de manipulare (inserari, stergeri etc), obtinem tipul abstract lista<E> cu elemente de tip E:

```
template <class E>
class lista {
// ...
private:
  nod<E> *head;  // adresa primul nod din lista
};
```

Conform principiilor de incapsulare, manipularea obiectelor clasei abstracte `lista<E>` se face exclusiv prin intermediul functiilor membre, structura interna a listei si, desigur, a nodurilor, fiind invizibila din exterior. Conteaza doar tipul informatiilor din lista si nimic altceva. Iata de ce clasa `nod<E>` poate fi in intregime nepublica:

```
template <class E>
class nod {
  friend class lista<E>;
  // ...
protected:
  nod( const E& v ): val( v ) { next = 0; }

  E       val;  // informatia propriu-zisa
  nod<E> *next; // adresa nodului urmator
};
```

In lipsa declaratiei `friend`, obiectele de tip `nod<E>` nici macar nu pot fi definite, datorita lipsei unui constructor `public`. Prin declaratia `friend` se permite accesul clasei `lista<E>` la toti membrii privati ai clasei `nod<E>`. Singurul loc in care putem utiliza obiectele de tip `nod<E>` este deci domeniul clasei `lista<E>`.

Inainte de a trece la definirea functiilor de manipulare a listelor, sa remarcam un aspect interesant la constructorul clasei `nod<E>`. Initializarea membrului `val` cu argumentul `v` nu a fost realizata printr-o atribuire `val = v`, ci invocand constructorul clasei `E` prin lista de initializare a membrilor:

```
nod( const E& v ): val( v ) { // ... }
```

In acest context, atribuirea este ineficienta, deoarece `val` ar fi initializat de doua ori: o data in faza de initializare prin constructorul implicit al clasei `E`, iar apoi, in faza de atribuire, prin invocarea operatorului de atribuire.

Principalele operatii asupra listelor sunt inserarea si parcurgerea elementelor. Pentru a implementa parcurgerea, sa ne amintim ce inseamna parcurgerea unui tablou – pur si simplu un indice si un operator de indexare:

```
tablou<int> T( 32 );
T[ 31 ] = 1;
```

In cazul listelor, locul indicelui este luat de elementul curent. Ca si indicele, care nu este memorat in clasa tablou, acest element curent nu are de ce sa faca parte din structura clasei `lista<T>`. Putem avea oricate elemente curente, corespunzatoare oricator parcurgeri, tot asa cum un tablou poate fi adresat prin oricati indici. Analogia tablou-lista se sfarseste aici. Locul operatorului de

indexare [] nu este luat de o functie membra, ci de o clasa speciala numita iterator<E>.

Intr-o varianta minima, datele membre din clasa iterator<E> sunt:

```
template <class E>
class iterator {
// ...
private:
  nod<E>* const *phead;
  nod<E>        *a;
};
```

adica adresa nodului actual (curent) si adresa adresei primului element al listei. De ce adresa adresei? Pentru ca iteratorul sa ramana functional si in situatia eliminarii primului element din lista. Operatorul (), numit in terminologia specifica limbajului C++ *iterator*, este cel care implementeaza efectiv operatia de parcurgere

```
template <class E>
iterator<E>::operator ()( E& v ) {
  if( a ) { v = a->val;  a = a->next; return 1; }
  else    { if( *phead ) a = *phead;  return 0; }
}
```

Se observa ca parcurgerea este circulara, adica, odata ce elementul actual a ajuns la sfarsitul listei, el este initializat din nou cu primul element, cu conditia ca lista sa nu fie vida. Atingerea sfarsitului listei este marcata prin returnarea valorii *false*. In caz contrar, valoarea returnata este *true*, iar elementul curent este "returnat" prin argumentul de tip referinta la E. Pentru exemplificare, operatorul de inserare in ostream poate fi implementat prin clasa iterator<E> astfel:

```
template <class E>
ostream& operator <<( ostream& os, const lista<E>& lista ) {
  E v; iterator<E> l = lista;

  os << " { ";
  while ( l( v ) ) os << v << ' ';
  os << "} ";

  return os;
}
```

Initializarea iteratorului l, realizata prin definitia iterator<E> l = lista, este implementata de constructorul

```
template <class E>
iterator<E>::iterator( const lista<E>& l ) {
  phead = &l.head;
  a = *phead;
}
```

Declaratia `const` a argumentului `lista<E>& l` semnifica faptul ca `l`, impreuna cu datele membre, este o variabila read-only (constanta) in acest constructor. In consecinta, `*phead` trebuie sa fie constant, adica definit ca

```
nod<E>* const *phead;
```

Aceeasi initializare mai poate fi realizata si printr-o instructiune de atribuire `l = lista`, operatorul corespunzator fiind asemanator celui de mai sus:

```
template <class E>
iterator<E>& iterator<E>::operator =( const lista<E>& l ) {
  phead = &l.head;
  a = *phead;

  return *this;
}
```

Pentru a putea defini un iterator neinitializat, se va folosi constructorul implicit (fara nici un argument):

```
template <class E>
iterator<E>::iterator( ) {
  phead = 0;
  a = 0;
}
```

In finalul discutiei despre clasa `iterator<E>`, vom face o ultima observatie. Aceasta clasa trebuie sa aiba acces la membrii privati din clasele `nod<E>` si `lista<E>`, motiv pentru care va fi declarata `friend` in ambele.

In sfarsit, putem trece acum la definirea completa a clasei `lista<E>`. Functia `insert()` insereaza un nod inaintea primului element al listei.

```
template <class E>
lista<E>& lista<E>::insert( const E& v ) {
  nod<E> *pn = new nod<E>( v );
  pn->next = head; head = pn;

  return *this;
}
```

O alta functie membra, numita `init()`, este invocata de catre constructorul de copiere si de catre operatorul de atribuire, pentru intializarea unei liste noi cu o alta, numita lista `sursa`.

```
template <class E>
void lista<E>::init( const lista<E>& sursa ) {
  E v; iterator<E> s = sursa;

  for ( nod<E> *tail = head = 0; s( v ); ) {
    nod<E> *pn = new nod<E>( v );
    if ( !tail ) head = pn; else tail->next = pn;
    tail = pn;
  }
}
```

Functia `reset()` elimina rand pe rand toate elementele listei:

```
template <class E>
void lista<E>::reset( ) {
  nod<E> *a = head;

  while( a ) {
    nod<E> *pn = a->next;
    delete a;
    a = pn;
  }
  head = 0;
}
```

Instructiunea `head = 0` are, aparent, acelasi efect ca intreaga functie `reset()`, deoarece lista este redusa la lista vida. Totusi, aceasta instructiune nu se poate substitui intregii functii, deoarece elementele listei ar ramane alocate, fara sa existe posibilitatea de a recupera spatiul alocat.

Declaratiile claselor `nod<E>`, `lista<E>` si `iterator<E>`, in forma lor completa, sunt urmatoarele:

```
template <class E>
class nod {
  friend class lista<E>;
  friend class iterator<E>;

protected:
  nod( const E& v ): val( v ) { next = 0; }

  E       val;  // informatia propriu-zisa
  nod<E> *next; // adresa nodului urmator
};
```

```
template <class E>
class lista {
  friend class iterator<E>;
public:
  lista(  ) { head = 0; }
  lista( const lista<E>& s ) { init( s ); }
  ~lista( ) { reset( ); }

  lista& operator =( const lista<E>& );
  lista& insert( const E& );

private:
  nod<E> *head;  // adresa primul nod din lista

  void init( const lista<E>& );
  void reset( );
};

template <class E>
class iterator {
public:
  iterator( );
  iterator( const lista<E>& );

  operator ()( E& );
  iterator<E>& operator =( const lista<E>& );

private:
  nod<E>* const *phead;
  nod<E>        *a;
};
```

4.4 Exercitii

4.1 In cazul alocarii dinamice, este mai rentabil ca memoria sa se aloce in blocuri mici sau in blocuri mari?

Solutie: Rulati urmatorul program. Atentie, stiva programului trebuie sa fie suficient de mare pentru a "rezista" apelurilor recursive ale functiei `alocareDinmica()`.

```
#include <iostream.h>

static int nivel;
static int raport;
```

```
void alocareDinamica( unsigned n ) {
  ++nivel;
  char *ptr = new char[ n ];
  if ( ptr )
    alocareDinamica( n );

  // memoria libera este epuizata
  delete ptr;
  if ( !raport++ )
    cout << "\nMemoria libera a fost epuizata. "
      << "S-au alocat "
      << (long)nivel * n * sizeof( char ) / 1024 << 'K'
      << ".\nNumarul de apeluri " << nivel
      << "; la fiecare apel s-au alocat "
      << n * sizeof( char ) << " octeti.\n";
}


main( ) {
  for ( unsigned i = 1024; i > 32; i /= 2 ) {
    nivel = 1; raport = 0;
    alocareDinamica( 64 * i - 1 );
  }

  return 1;
}
```

Rezultatele obtinute sunt clar in favoarea blocurilor mari. Explicatia consta in faptul ca fiecarui bloc alocat i se adauga un antet necesar gestionarii zonelor ocupate si a celor libere, zone organizate in doua liste inlantuite.

4.2 Explicati rezultatele programului de mai jos.

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

int main( ) {
  tablou<int> y( 12 );

  for ( int i = 0, d = y.size( ); i < d; i++ )
    y[ i ] = i;

  cout << "\nTabloul y      : " << y;
  y = 8;
  cout << "\nTabloul y      : " << y;

  cout << '\n';
  return 0;
}
```

Solutie: Elementul surprinzator al acestui program este instructiunea de atribuire `y = 8`. Surpinzator, in primul rand, deoarece ea "trece" de compilare, desi nu s-a definit operatorul de atribuire corespunzator. In al doilea rand, instructiunea `y = 8` surprinde prin efectele executiei sale: tabloul `y` are o alta dimensiune si un alt continut. Explicatia este data de o conventie a limbajului C++, prin care un constructor cu un singur argument este folosit si ca operator de conversie de la tipul argumentului, la tipul clasei respective. In cazul nostru, tabloului `y` i se atribuie un tablou temporar de dimensiune 8, generat prin invocarea constructorului clasei `tablou<T>` cu argumentul 8. S-a realizat astfel modificarea dimensiunii tabloului `y`, dar cu pretul pierderii continutului initial.

4.3 Exercitiul de mai sus contine o solutie pentru modificarea dimensiunii obiectelor de tip `tablou<T>`. Problema pe care o punem acum este de a rezolva problema, astfel incat continutul tabloului sa nu se mai piarda.

Solutie: Iata una din posibilele implementari:

```
template< class T >
tablou<T>& tablou<T>::newsize( int dN ) {
  T *aN = 0;                // noua adresa

  if ( dN > 0 ) {
    aN = new T [ dN ];      // alocarea dinamica a memoriei
    for ( int i = d < dN? d: dN; i--; )
      aN[ i ] = a[ i ];     // alocarea dinamica a memoriei
  }
  else
    dN = 0;

  delete [ ] a;             // eliberarea vechiului spatiu
  d = dN; a = aN;           // redimensionarea obiectului

  return *this;
}
```

4.4 Implementati clasa parametrica `coada<T>`.

Solutie: Conform celor mentionate la sfarsitul Sectiunii 4.2.1, ne vom inspira de la structura clasei `stiva<T>`. Una din implementarile posibile este urmatoarea.

```
template <class T>
class coada: private tablou<T> {
public:
  coada( int d ): tablou<T>( d )
    { head = tail = 0; }
```

```
  ins_q( const T& );
  del_q(       T& );

private:
  int head; // indicele ultimei locatii ocupate
  int tail; // indicele locatiei predecesoare primei
            // locatii ocupate
};

template <class T>
coada<T>::ins_q( const T& x ) {
  int h = ( head + 1 ) % d;
  if ( h == tail )   return 0;
  a[ head = h ] = x; return 1;
}

template <class T>
coada<T>::del_q( T& x ) {
  if ( head == tail ) return 0;
  tail = ( tail + 1 ) % d;
  x = a[ tail ];      return 1;
}
```

4.5 Testati functionarea claselor `stiva<T>` si `coada<T>`, folosind elemente de tip `int`.

Solutie: Daca programul urmator furnizeaza rezultate corecte, atunci putem avea certitudinea ca cele doua clase sunt corect implementate.

```
#include <iostream.h>
#include "stiva.h"
#include "coada.h"

void main( ) {
  int n, i = 0;
  cout << "Numarul elementelor ... "; cin >> n;

  stiva<int> st( n );
  coada<int> cd( n );

  cout << "\nStiva push ... ";
  while ( st.push( i ) )  cout << i++ << ' ';

  cout << "\nStiva pop  ... ";
  while ( st.pop( i ) )   cout << i   << ' ';

  cout << "\nCoada ins_q... ";
  while ( cd.ins_q( i ) ) cout << i++ << ' ';
```

```
  cout << "\nCoada del_q... ";
  while ( cd.del_q( i ) ) cout << i   << ' ';

  cout << '\n';
}
```

4.6 Testati functionarea clasei parametrice `lista<E>` cu noduri de tip adrese de tablou si apoi cu noduri de tip `tablou<T>`.

Solutie (incorecta): Programul urmator nu functioneaza corect decat dupa ce a fost modificat pentru noduri de tip `tablou<T>`. Pentru a-l corecta, nu uitati ca toate variabilele din ciclul `for` sunt locale.

```
#include <iostream.h>

#include "tablou.h"
#include "lista.h"

typedef tablou<int> *PTI;

main( ) {
  lista<PTI> tablist;

  for ( int n = 0, i = 0; i < 4; i++ )
  { tablou<int> t( i + 1 );
    for ( int j = t.size( ); j--; t[ j ] = n++ );
    cout << "tablou " << i << ' '; cout << t << '\n';
    tablist.insert( &t );
  }

  cout << "\nLista "; cout << tablist << "\n";

  PTI t; iterator<PTI> it = tablist;
  while( it( t ) )
    cout << "Tablou din lista" << *t << '\n';

  return 1;
}
```

4.7 Destructorul clasei `lista<T>` “distruge” nodurile, invocand procedura iterativa `reset()`. Implementati un destructor in varianta recursiva.

Indicatie: Daca fiecare element de tip `nod<E>` are un destructor de forma `~nod( ) { delete next; }`, atunci destructorul clasei `lista<E>` poate fi `~lista( ) { delete head; }`.