Capitolul 9

Funcții şi clase prietene. Clase imbricate

În afară de relația de compunere şi cea de derivare, limbajul C++ permite şi alte metode prin care o clasă poate avea acces la membrii altor clase: ***clasele prietene*** şi ***clasele definite în interiorul altor clase***. Spre deosebire de primele două variante, în cazul claselor prietene sau imbricate în alte clase, accesul nu se poate face direct, ci doar prin intermediul unor obiecte instanță ale claselor respective.

9.1 Funcții şi clase prietene

Este necesar ca în unele cazuri funcțiile ce utilizează obiecte ale unei anumite clase să facă referire la membrii privați (`private` sau `protected`) ai acesteia. De exemplu, în cazul în care clasa *Punct* ar fi definită astfel:

```
class Punct {
protected:
  double x, y;
Punct(double a, double b): x(a), y(b) { }
};
```

o funcție ce determină distanța între două puncte ar necesita referirea la membrii *x* şi *y* ai unui obiect din clasa *Punct*:

```
double Dist(Punct& c, Punct& b) {
  double d = sqrt((a.x-b.x)*(a.x-b.x) +
     (a.y-b.y)*(a.y-b.y);
  return d;
}
```

O soluție la această problemă ar fi ca funcția *Dist* să fie definită ca ***funcție prietenă*** clasei *Punct* şi să aibă acces la membrii privați ai acesteia:

```
class Punct {
friend double Dist(Punct& c, Punct& b);
protected:
  double x, y;
public:
  Punct(double a, double b) : x(a), y(b) { }
};
```

În acest caz, un exemplu de utilizare a funcției *Dist* poate fi:

```
void Prelucrare() {
```

```
  Punct p1(1, 1), p(2, 2);
  double d = dist(p1, p2);
  // ...
}
```

O ***funcție prietenă*** unei anumite clase este externă clasei respective, dar ea poate avea acces la membrii clasei cu care este prietenă, chiar dacă aceştia sunt privați.

Observații:

1) Declarația de ***funcție prietenă*** se face în interiorul clasei unde funcția este prietenă şi nu în interiorul funcției.
2) Indiferent de locul unde se declară funcția prietenă în cadrul clasei, ea va fi întotdeauna vizibilă în exteriorul clasei (funcția prietenă nu aparține clasei căreia îi este prietenă, este exterioară ei).
3) Funcția are acces la toți membrii clasei prietene, indiferent că aceştia sunt publici sau nu.
4) Funcția nu poate avea acces la membrii clasei prietene în mod direct, ci doar prin intermediul unor obiecte instanță ale clasei. De exemplu, în corpul funcției *Dist* membrii *x* şi *y* nu pot fi accesați direct prin numele lor, ceea ce înseamnă că nu pot exista instrucțiuni de forma:

```
d = sqrt(x*x + y*y);
```

Din punct de vedere sintactic, declarația unei ***funcții prietene*** se face prin declarația funcției respective precedată de cuvântul cheie `friend`.

Datorită locului de declarare a funcțiilor `friend`, pentru adăugarea de noi funcții `friend` la o anumită clasă trebuie modificată declarația clasei respective. Rezultă deci că declarația funcțiilor `friend` trebuie facută în etapa de proiectare a unei aplicații.

În mod uzual, funcțiile prietene unei clase ar putea fi definite ca funcții membru ale clasei respective, caz în care ele pot avea acces direct la membrii clasei. De exemplu, funcția *Dist* ar putea fi definită astfel:

```
class Punct {
protected:
  double x, y;
public:
  Punct(int a, int b) : x(a), y(b) { }
  Double Dist(Punct& b) {
    double d = sqrt((x-b.x)*(x-b.x)+(y-b.y)*(y-b.y);
    return d;
  }
};

void Prelucrare() {
  Punct p1(1, 1), p2(2, 4);
  double d1 = p1.Dist(p2);
  double d2 = p2.Dist(p1); //acelasi lucru
  // ...
}
```

În cazul în care se utilizează varianta funcției membru, o aemenea funcție poate fi apelată ca o metodă a unui obiect al clasei respective. Diferența între utilizarea funcției *Dist* ca funcție membru şi ca funcție prietenă în acest exemplu, constă într-o anumită *asimetrie* în cazul funcției membru. În cazul general, când o anumită funcție este privită ca implementare a unei operații asupra a două obiecte (echivalentul unui operator binar), metoda funcției prietene poate apare ca fiind mai naturală, deoarece metoda funcției membru necesită un singur parametru explicit (celalalt fiind pointerul ascuns `this` spre obiectul curent).

Există situații în care se dorește ca o anumită funcție membru a unei clase să fie prietenă unei alte clase.

Exemplul 9.1. Clasa *contorA* conține un pointer la un obiect al unei clase *A* şi o funcție ce contorizează numarul de referiri la acest obiect.

```
class A;

class contorA {
  A *a;
public:
  contorA(int k = 0);
  int increment();
};

class A {
  int n;
public:
  A(int k = 0) { n = k; }
friend int contorA::increment();
};

int contorA::increment() { return a->n++; }
contorA::contorA(int k) { a = new A; }

void main() {
  contorA c;
  cout << c.increment() << endl;
  cout << c.increment() << endl;
}
```

Observația. Funcțiile *increment* şi constructor ale clasei *contorA* nu au fost definite inline, deoarece clasa *A* nu a fost complet definită în momentul definirii clasei *contorA*.

Funcția *increment* din clasa *contorA* trebuie să incrementeze membrul privat *n* al obiectului *a*, ce aparține clasei *A*. Din acest motiv funcția a fost declarată ca funcție `friend` în clasa *A*. Spre deosebire de exemplul precedent, funcția a fost prefixată cu numele clasei de care aparține.

În cazul în care se dorește ca mai multe funcții membru ale unei anumite clase *A* să aibă acces la membrii privați ai unei alte clase *B*, se poate declara întreaga clasă *A*, ca o ***clasă***

prietenă clasei *B*. În acest caz declarațiile cuprind doar cuvântul cheie `friend`, urmat de declarația clasei care este prietenă.

Pentru exemplul precedent, se poate declara *contorA* ca o ***clasă prietenă*** clasei *A*.

```
class A;

class contorA {
  A *a;
public:
  contorA(int k = 0);
  int increment();
};

class A {
  int n;
public:
friend class contorA;
  A(int k = 0) { n = k; }
};
```

Observație: Relația de ***clasă prietenă*** nu este biunivocă, în sensul că declarația :

```
friend class contorA;
```

nu înseamnă că membrii clasei *A* au acces la membrii privați ai clasei *contorA*.

Exemplul 9.2. Ca un exemplu intuitiv de utilizare a claselor prietene, se rescriu clasele *node* şi *list* din exemplul 5.5 (capitolul 5), pentru definirea unei liste liniare simplu înlănțuite. Deoarece funcțiile din clasa *list* accesează membrul privat *next* al clasei *node*, clasa *list* poate fi declarată ca o clasă prietenă lui *node*.

```
class list;

class node {
friend class list;
  int val;
  node* next;
public:
  node(int v, node* p = 0) { val = v; next = p; }
  ~node() { next =0; }
  void Add(int v) {
    node* q = new node(v);
    next = q;
  }
  int Val() const { return val; }
  void Print() const { cout << val << endl; }
};
class list {
  node* first;
  void Delete();
```

```
  void Copy(node* p);
public:
  list() { first = 0; }
  list(list& l) { first = 0; Copy(l.first); }
  ~list() { Delete(); first = 0; }
  //adauga un element la sfarsitul listei
  void AddLast(int v);
  void Print() const {
    for (node* p=first; p; p=p->next)
      p->Print();
  }
  int ListaVida() const { return first == 0; }
};

void list::AddLast(int v) {
  if (!first)
    first = new node(v);
  else {
    for (node* q=first; q->next; q=q->next);
    q->Add(v);
  }
}

void list::Copy(node* p) {
  first = 0;
  for (node* q=p; q; q=q->next)
    AddLast(q->val);
}

void list::Delete() {
  node *p = first, *q;
  while (p) {
    q = p;
    p = p->next;
    delete q;
  }
}

void main() {
  list l;
  l.AddLast(7);
  l.AddLast(5);
  l.AddLast(9);
  l.Print();
}
```

Observație: Trebuie făcută distincție între o clasă prietenă altei clase şi o clasă derivată din altă clasă.

Fie de exemplu:

```
class D1 {
  // ...
};

class B1 {
friend class D1;
  // ...
};

class B {
  // ...
};

class D: public B {
  // ...
};

void Prelucrare() {
  D1 d1;
  D d;
  // ...
}
```

Obiectul *d* al clasei *D* conține toți membrii clasei *B*, la care se adaugă anumiți membri suplimentari, care sunt proprii clasei *D*. Spre deosebire de acesta, obiectul *d1* al clasei *D1* conține doar membrii proprii clasei *D1*, nu şi pe cei ai clasei *B1* (declarația `friend class D1;`). Deci funcțiile membru ale clasei *D1* pot accesa membrii privați ai clasei *B1* doar prin intermediul unor obiecte instanță ale lui *B1*).

9.2 Clase definite în interiorul altor clase (clase imbricate)

Prezența mai multor clase sau funcții prietene într-o ierarhie de clase denotă o proiectare ineficientă a ierarhiei. În aceste cazuri se impune o priectare a ierarhiei care să minimizeze apariția funcțiilor şi claselor prietene: redefinirea funcțiilor `friend` ca funcții membre, iar a claselor `friend` drept ***clase definite în interiorul altor clase***

Această observație este justificată de faptul că funcțiile şi clasele prietene nu reprezintă o caracteristică specifică unui limbaj pur orientat pe obiecte, ci un compromis în favoarea pragmatismului proiectării aplicațiilor.

Exemplul 9.3. Clasa *list* poate fi definită într-un stil pur orientat pe obiecte astfel (implmementările funcțiilor clasei *list* sunt identice exemplului precedent 9.2):

```
class list {
  struct node {
    int val;
    node* next;
    node(int v, node*p = 0): val(v), next(p) { }
    ~node() { next = 0; }
```

```
    void Add(int v) {
      node* q = new node(v);
      next = q;
    }
    void Print() const { cout << val << endl; }
  };
  node *first;
  void Delete();
  void Copy(node* p);
public:
  list() { first = 0; }
  list(list& l) { first = 0; Copy(l.first); }
  ~list() { Delete(); first = 0; }
  void Add(int v);  //adauga un element la sfarsitul
listei
  void Print() const {
    for (node* p=first; p; p=p->next)
      p->Print();
  }
  int ListaVida() const { return first == 0; }
};
```

Clasa *node* din exemplul precedent este definită în interiorul clasei *list*, în secțiunea `private` a acesteia.

O ***clasă imbricată*** în altă clasă poate fi considerată ca un membru al clasei respective şi poate fi definită în oricare parte a clasei respective: în secțiunea `public`, `protected` sau `private`. Accesibilitatea acesteia depinde de secțiunea clasei în care a fost definită:

- o clasă definită în secțiunea `public` a unei alte clase este vizibilă în afara clasei respective;
- o clasă definită în secțiunea `proctected` este vizibilă doar în subclasele derivate din clasa respectivă;
- o clasă definită în secțiunea `private` este vizibilă doar în interiorul clasei din care face parte.

Accesibilitatea membrilor unei clase imbricate în altă clasă respectă regulile generale de acces ale membrilor unei clase. În concluzie, membrii `private` dintr-o clasă *B* imbricată într-o clasă *A* nu pot fi accesați în clasa *A*, indiferent de secțiunea în care a fost definită clasa *B*. În cazul în care se doreşte ca întraga clasă *A* să aibă acces la membrii `private` din clasa *B*, sau clasa *B* să aibă acces la membrii `private` din clasa *A*, clasele pot fi definite prietene. Declarația de clasă prietenă pentru o clasă imbricată poate preceda sau succeda definiția clasei.

Exemplul 9.4. Clasele *B* şi *C* sunt definite în interiorul clasei *A*, astfel încât fiecare dintre ele să aibă acces la membrii `private` ai celorlalte clase.

```
class A {
  int n;

  class B {
```

```
  friend class A;
    int k;
  public:
    B(int n = 0): k(n) { }
    int K() const { return k; }
  };
  friend class B;

public:
  class C {
  friend class A;
    int l;
  public:
    C(int n = 0): l(n) { }
    int L() const { return l; }
  };
  friend class C;

  A(int a): n(a) { }
  int N() const { return n; }

  // ...
};
```

Accesibilitatea unei clase imbricate în altă clasă *se referă doar la numele clasei privit ca un tip de date şi nu la membrii acesteia.* Accesul la membrii unei clase imbricate se poate face doar prin intermediul unui obiect instanță ala cesteia, ca şi în cazul claselor prietene.

În exemplul 9.3 al clasei *list*, clasa *node* nu este accesată direct în funcțiile membru ale clasei *list*, pentru aceasta se utilizându-se o dată membru, *first*, care este pointer la clasa *node*.

Implementarea funcțiilor care nu sunt `inline` ale unei clase imbricate se poate face în exteriorul clasei unde este definită clasa imbricată, folosindu-se operatorul de rezoluție.

De exemplu, în cazul în care funția *Add* din clasa *node* ar fi fost definită astfel:

```
class list {
  struct node {
    int val;
    node* next;
    node(int v, node*p = 0): val(v), next(p) { }
    ~node() { next = 0; }
    void Add(int v);
    void Print() const { cout << val << endl; }
  };
  node *first;
  // ...
};
```

Atunci implementarea sa în exteriorul clasei *list* poate fi următoarea:

```
void list::node::Add(int v) {
  node* q = new node(v);
  next = q;
}
```

În cazul în care o clasă imbricată conține date statice, accesul acestora se poate realiza tot cu ajutorul operatorului de rezoluție.

Exemplul 9.5. Se reia exemplul 9.4 al claselor *A*, *B* şi *C*, în care se utilizează date şi funcții statice:

```
class A {
  int n;
  static int v;

  class B {
  friend class A;
    int k;
    static int v;
  public:
    B(int n = 0): k(n) { }
    int K() const { return k; }
    static void SetV(int n) {
      B::v = n;
    }
    static void SetAV(int n) {
      A::v = n;
    }
  };
  friend class B;

public:

  class C {
  friend class A;
    int l;
    static int v;
  public:
    C(int n = 0): l(n) { }
    int L() const { return l; }
    static void SetV(int n) {
      C::v = n;
    }
    static void SetAV(int n) {
      A::v = n;
    }
  };
  friend class C;

  A(int a): n(a) { }
  int N() const { return n; }
```

```
  static void SetV(int a) { v = a; }

  // ...
};

int A::v = 0;
int A::B::v = 0;
int A::C::v = 0;

void main() {
  A::C::SetV(1);
  A::C::SetAV(3);
  A::SetV(0);
  // ...
}
```

În cazul în care o clasă conține mai multe clase imbricate, ordinea de definire a acestora nu este în general restricționată. Excepție fac acele clase imbricate care sunt dependente unele de altele, caz în care unele trebuie declarate înainte de definire.

Deşi nu reprezintă clase, ***enumerările*** reprezintă tipuri de date şi ***pot fi definite în interiorul altor clase***. Atât numele acestor enumerări, cât şi valorile elementelor lor pot fi utilizate în clasele în interiorul cărora au fost definite, iar în cazul în care sunt definite într-o secțiune publică, pot fi utilizate şi în afara claselor.

Exemplul 9.6. Clasa *ceas* permite afişarea orei curente a unui ceas pentru diferite fuse orare predefinite: *oraLondrei*, *oraParisului*, *oraBucurestiului*, *oraMoscovei*. Un ceas se consideră că este potrivit după ora Bucureştiului.

```
#include <iostream>
using namespace std;

class ceas {
  int ora, min, sec;
public:
  enum OraAfisare {
    oraLondrei,
    oraParisului,
    oraBucurestiului,
    oraMoscovei
  };
  ceas(int o = 0, int m = 0, int s = 0):
     ora(o), min(m), sec(s) { }
  void AfisareOra(OraAfisare h) {
    cout << “ora “ << ora + h - oraBucurestiului;
    cout << “: min “ << min;
    cout << “: sec “ << sec << endl;
  }
};

void main() {
```

```
  ceas c(14, 20, 50);
  c.AfisareOra(ceas::oraBucurestiului);
  c.AfisareOra(ceas::oraParisului);
  c.AfisareOra(ceas::oraLondrei);
  c.AfisareOra(ceas::oraMoscovei);
}
```

9.3 Containere şi iteratori

Un exemplu uzual de utilizare a claselor prietene îl constituie cazul claselor de tip ***container*** şi ***iterator***. Un ***container*** este o colecţie de mai multe obiecte, la care se poate accesa un singur obiect la un anumit moment, aşa cum sunt listele şi vectorii. Un ***iterator*** este asociat întotdeauna la un container, el fiind responsabil cu accesul obiectului curent din container, dar nu permite accesul la implementarea containerului.

Exemplul 6.2. Se va implementa o listă ca un container. Accesul la obiectul curent din container este realizat cu ajutorul operatorului binar `->`, iar trecerea la elementul următor celui curent se realizează cu ajutorul operatorului ++. Ambii operatori aparţin clasei iterator, iar nu containerului.

```
class Node {   //clasa elementelor din container
  int val;
  Node *next;
public:
  Node(int v, Node *p = 0): val(v), next(p) {}
  ~Node() { next = 0; }
  int Val() const { return val; }
  void Print() const { cout << val << endl; }
  friend class List;
  friend class ListIterator;
};

class List {   //clasa container
  Node* first;
  void Copy(Node* p);
  void Delete();
public:
  List() { first = 0; }
  List(const List& l): first(0) { Copy(l.first); }
  ~List() { Delete(); first = 0; }
  void Add(int k) {      //inserare în fata listei
    Node* p = new Node(k);
    p->next = first;
    first = p;
       }
  friend class ListIterator;
};

class ListIterator {     // clasa iterator
```

```
  List l;                // Conteaza ordinea de declarare
  Node* current;         // a membrilor !!
public:
  ListIterator(List& ll): l(ll), current(l.first) {}
  Node* operator->() const { return current; }
  Node* operator++() {   //operator prefix
    current = current->next;
    return current;
  }
  Node* operator++(int) { //operator postfix
    Node* p = current;
    current = current->next;
    return p;
  }
  void BeginIterator()   //reinitialiazare iteratii
    { current = l.first; }
};

void main() {
  List l;
  l.Add(3);
  l.Add(7);
  l.Add(5);
  ListIterator it(l);
  // Eroare!! Dupa afisarea lui 5, it++ devine NULL
  do
     it->Print();
  while(it++);
  it.BeginIterator();
  // Corect. Se afiseaza 3,7,5.
  do
     it->Print();
  while(++it);
}
```