Capitolul 10

12 Clase şi funcţii parametrizate. Mecanismul template

Parametrizarea datelor reprezintă un element important al paradigmei abstractizării datelor şi permite definirea unor clase care conţin tipuri de date nespecificate complet, în mod asemănător parametrilor funcţiilor. În acest mod, o clasă parametrizată reprezintă un şablon ce defineşte o mulţime de clase.

Clasele parametrizate ale limbajului C++ sunt asemănătoare celor din limbajul Ada, dar mecanismul de creare şi utilizare al lor este diferit şi se bazează pe noţiunea de `template`. În plus, C++ permite definirea şi utilizarea funcţiilor `template`, într-un mod asemănător claselor `template`.

12.1 Clase template

Principala problemă a containerelor în limbajul C++ o constituie instanţierea tipului de date al elementelor componente. Dacă, de exemplu, s-a definit o clasă *stack* ce defineşte un container de tip stivă, pentru a utliza într-un program o stivă de numere întregi, va trebui creată o clasă distinctă, *istack* de exemplu, care este derivată din clasa *stack* şi dintr-o clasă ce defineşte numerele întregi.

În mod uzual, un container este o interfaţă abstractă ce defineşte principalele operaţii ce se pot efectua asupra unor elemente componente, fără să fie precizat tipul acestora. Soluţia aleasă de Stroustrup pentru limbajul C++ se bazează pe reutilizarea codului şi este numită `template`.

Pentru a crea în acest mod o stivă de întregi, se va instanţia clasa parametrizată *stack* la tipul `int`. De exemplu, dacă s-a definit clasa parametrizată *stack*, atunci următoarele declaraţii definesc două obiecte de tip stivă, unul conţine întregi şi celălalt caractere:

```
stack<int> sint;
stack<char> schar;
```

În acest mod s-au creat două clase distincte, deşi codul scris de programator a fost doar pentru clasa *stack*.

12.1.1 Definirea şi utilizarea claselor parametrizate

Definirea claselor parametrizate este relativ simplă: definirea clasei respective este precedată de o construcţie sintactică de forma următoare:

```
template '<' <lista parametri> '>'
```

unde `<lista parametri>` reprezintă lista parametrilor clasei. În mod uzual, parametrii unei clase `template` sunt tipuri de date, fiind specificaţi de cuvântul cheie `class` sau `typename`.

Exemplu. Clasa următoare defineşte un tablou unidimensional la care elementele componente au un tip de date generic, *T*. Acesta este instanțiat în funcția `main` de doua ori.

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

template <class T>
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index) {
    return A[index];
  }
};

void main() {
  Array<int> ia;
  Array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 0.5;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j] << ", " << fa[j] << endl;
}
```

Tipul parametrizat *T* este utilizat în cadrul calsei ca orice tip de date cunoscut. Singura restricție care se impune este ca toate clasele sau tipurile de date utilizate în eventualele instanțieri ale clasei parametrizate să posede funcțiile şi operatorii utilizați în clasa parametrizată pentru clasa *T*.

De exemplu, dacă în clasa *Array* s-ar defini o funcție *Print* de forma următoare:

```
template <class T>
class Array {
  // ...
  void Print() {
    for (int k=0; k<size; k++)
      A[k].Print();
  }
  // ...
};
```

nu s-ar putea crea instanțe de forma `Array<int>` deoarece tipul `int` nu posedă funcția *Print*.

Este posibil să se specifice într-o clasă parametrizată şi alți parametri decât nume de tipuri de date, în mod asemănător parametrilor funcțiilor. În acest caz la instanțiere trebuie să fie expresii constante care să poată fi evaluate de compilator.

Pentru exemplificare, în exemplul următor se va rescrie clasa *Array* cu doi parametri, al doilea specificând dimensiunea tabloului.

Exemplu.

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

template <class T, unsigned size>
class Array {
  T A[size];
public:
  T& operator[](int index) {
    return A[index];
  }
};

void main() {
  Array<int, 20> ia;
  Array<float, 10> fa;
  // ...
}
```

De asemenea, este posibil ca parametrii unei clase parametrizate să aibeă valori implicite, în mod asemănător valorilor implicite ale funcțiilor. De exemplu, definiția următoare:

```
template <class T, unsigned size = 100>
class Array {
  // ...
};
```

permite instanțieri de forma:

```
Array<int> ia;
Array<float, 10> fa;
```

Instanțierea claselor parametrizate înseamnă de fapt generearea automată de către compilator a codului pentru clasa instanță. Instanțierea se produce în mod uzual în acele locuri din program în care sunt definite obiecte ale clasei respective. În momentul instanțierii, toate argumentele clasei `template` trebuie să fiec cunoscute.

Observație. În cazul în care se utilizează o referință sau un pointer la un asemenea obiect, instanțierea nu mai are loc. De exemplu, declarația următoare nu produce o instanțiere a clasei *Array*:

```
Array<int> *pa;
```

În cazul în care definirea unei funcții membru a unei asemena clase nu se face `inline`, ci în afara clasei, definirea trebuie precedeată de aceeaşi contrucție `template` ca şi clasa de care aparține, care specifică parametrii `template` ai funcției. În plus, orice referire la clasa de care aparține funcția trebuie specificată ca o instanță.

Pentru exemplificare, la clasa *Array* anterioară s-a adăugat o funcție *Print*.

Exemplu.

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

template <class T, unsigned size>
class Array {
  T A[size];
public:
  T& operator[](int index) {
    return A[index];
  }
  void Print() const;
};

template <class T, unsigned size>
void Array<T>::Print() const {
  for (int k=0; k<size; k++)
    cout << A[k] << ' ';
  cout << endl;
}

void main() {
  Array<int, 10> ia;
  for(int i = 0; i < 10; i++)
    ia[i] = i * i;
  cout << "Elementele sunt:" << endl;
  ia.Print();
}
```

În mod uzual, definiția unei clase parametrizate, precum şi implementarea funcțiilor care nu sunt inline se face în acelaşi fişier header. Motivul pentru această regulă se referă la acțiunile pe care compilatorul trebuie să le execute în momentul alocări memoriei pentru instanțierile unei clase parametrizate. Există însă şi compilatoare care permit fişiere distincte pentru definiția clasei şi pentru implementare.

12.1.2 Cuvântul cheie `typename`

Aşa cum s-a specificat anterior, în cadrul construcției template, se poate utiliza atât cuvântul chaie `class`, cât şi `typename`, pentru a specifica un parametru care este un tip de date. Inițial s-a utilizat cuvântul `class`, dar `typename,` care a fost introdus mai târziu în limaj, este mai sugestiv.

Există însă situații în care ecasta trebuie folosit în mod obligatoriu. Cea mai frecventă este cea în care într-o clasă parametriztă se utilizează un tip de date definit înăuntrul clasae care este parametru. În această situație, compilatorul nu cunoaşte suficiente informații pentru a deosebi un nume care reprezintă un membru al clasei parametru, sau un tip de date imbricat în aceasta.

Exemplu. Clasele *X* şi *Y* conțin ambele clasă imbricată numită *M*. Clasa parametrizată *A* care este instanțiată pentru fiecare din aceste două clase, are ca membru, o instanță a clasei *M*.

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

```
class X {
public:
  // ...
  class M {
  public:
    void g() { cout << "g in clasa X\n"; }
  };
  // ...
};

class Y {
public:
  // ...
  class M {
  public:
    void g() { cout << "g in clasa Y\n"; }
  };
  // ...
};

template <class T>
class A {
  typename T::M m;
public:
  void f() { m.g(); }
};

void main() {
  A<X> ax;
  A<Y> ay;
  ax.f();
  ay.f()
}
```

În cazul în care nu s-ar fi utilizat `typename` în declarația:

```
  typename T::M m;
```

aceasta ar fi fost ambiguă, deoarece clasa *T* este o clasă generică (în caz contrar nu ar fi existat ambiguități).

12.1.3 Clase `template` şi declaraţii `friend`

În cadrul claselor parametrizate se pot declara alte clase sau funcţii ca fiind prietene clasei respective. Există trei tipuri de declaraţii friend (după cum se va discuta ulterior, se pot defini şi funcţii `template`, în mod asemănător claselor `template`):

a) declararea unor clase sau funcţii ne-template;
b) declararea unor clase sau funcţii `template` cu aceiaşi parametri ca şi clasa curentă (legate);
c) declararea unor clase sau funcţii `template` cu alţi parametri (nelegate).

În primul caz, funcţiile sau clasele se declară prietene utilizând aceleaşi regului ca şi la clasele ne-template.

În cazul al doilea, parametrii clasei template curentă sunt folosiți pentru a lega parametrii claselor (şi funcțiilor) prietene. În acest caz, o instanțiere a unei clase (sau funcții) prietene are acces la membrii privați doar ai unei instanțe a clasei `template` curente cu aceleaşi valori ale argumentelor.

În ultimul caz, listele de argumente ale clesei `template` curente şi cele ale claselor (şi funcțiilor) prietene sunt diferite, astfel încât acestea se numesc nelegate. În acest caz, orice instanțiere a unei clase prietene are acces la membrii privați ai oricărei instanțieri a clasei `template` curente.

Exemplu. Pentru exemplificare se va defini o clasă parametrizată prietenă unei alte clase parametrizate în cazul doi anterior.

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

template <class T>
class A;

template <class T>
class B {
  void g() { cout << "g in clasa B\n"; }
  friend class A<T>;
};

template <class T>
class A {
public:
  void f(B<T> b) { b.g(); }
};

void main() {
  A<int> ax;
  A<float> ay;
  B<int> bx;
  B<float> by;

  ax.f(bx);
  ay.f(by);
  //ax.f(by);  // Eroare de conversie a parametrului
  //ay.f(bx);  // Eroare de conversie a parametrului
}
```

12.1.4 Derivarea claselor parametrizate

O clasă parametrizată se poate deriva, caz în care clasa derivată va fi tot o clasă parametrizată cu aceiaşi parametri, la care eventual de pot adăuga alții noi.

Exemplu. Clasa parametrizată *Derived* moşteneşte public clasa parametrizată *Base*.

```
class A {
  int v;
public:
  A(int a = 0): v(a) {}
  void Print() const { cout << v << endl; }
};

class B {
  float v;
public:
  B(float a = 0): v(a) {}
  void Print() const { cout << v << endl; }
};

template<typename T>
class Base
{
  T& t;
public:
  Base(T& a): t(a) {}
  void Print() const { t.Print(); }
};

template<typename T, typename C>
class Derived: public Base<T>
{
  C c;
public:
  Derived(T& t, C& a): Base<T>(t), c(c)  {}
  void Print() const { Base<T>::Print(); c.Print(); }
};

void main() {
  A a(3);
  B b(3.3);
  Base<A> x(a);
  Derived<A, B> y(a, b);
  x.Print();
  y.Print();
}
```

12.1.5 Imbricarea claselor parametrizate

Legătura între clasele parametrizate şi noţiunea de imbricare a claselor se poate manifesta în două moduri diferite: definirea unei clase neparametrizate în interiorul unei clase template, precum şi definirea unei clase template în interiorul altei clase (parametrizate sau nu).

În primul caz, clasa neparametrizată imbricată în interiorul unei clase parametrizate devine ea însăşi o clasă parametrizată, putând utiliza în interiorul ei parametrii clasei exterioare.

Exemplu. Se va redefini clasa *Stack* ca o clasă parametrizată. Elementele unei asemenea stive sunt instanțe ale clasei *Link*, definită în interiorul clasei *Stack*. Clasa *Link* utilizează parametrul *T* al clasei *Stack*, devenind ea însăşi o clasă parametrizată.

```
template<class T>
class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt): data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack(){
    while(head)
      delete pop();
  }
  void push(T* dat) {
    head = new Link(dat, head);
  }
  T* pop(){
    if(head == 0) return 0;
    T* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};

class X {
  int n;
public:
  X(int a = 0): n(a) {}
  virtual ~X() { cout << "~X " << endl; }
  void Print() const { cout << n << endl; }
};

void main() {
  Stack<X> st;
  for(int i = 0; i < 10; i++)
    st.push(new X(i));
  for(int j = 0; j < 10; j++)
    st.pop()->Print();
}
```

În al doilea caz, o clasă imbricată în interiorul altei clase este ea însăşi o clasă parametrizată, care poate să fie sau nu o clasă parametrizată. În cazul în care clasa externă este tot o clasă parametrizată, clasa imbricată poate utiliza, ca în exemplul precedent, parametrii acesteia.

Exemplu. Clasa parametrizată *B* este definită în interiorul clasei parametrizate *A*. În interiorul clasei *B* pot fi utilizați atât parametrii proprii ei, cât şi cei ai clasei *A*:

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

class X {
  int v;
public:
  X(int a = 0): v(a) {}
  void Print() const { cout << v << endl; }
};

template <class T1, class T2>
class A {
public:
  template <class T2>
  class B {
  public:
    B(T1 a): v1(a), v2(v1) {}
    T1 v1;
    T2 v2;
    void g() { v2.Print(); }
  };
  B<T2> m;
  A(T1 s): m(s) {}
  void f() { m.g(); }
};

void main() {
  A<int, X> a(7);
  a.f();
}
```

12.2 Funcții `template`

Funcțiile template reprezintă implementarea unor categorii de algoritmi numiți ***algoritmi generici***. Ei reprezintă descrierea unei clase de probleme doar prin specificarea acțiunilor ce se execută, fără să se specifice concret tipul de date al elementelor asupra cărora se acționează.

Un exemplu sugestiv îl constituie algoritmii generici de sortare a unui şir de elemente printr-o anumită metodă specifică (quicksort de exemplu). Presupunând că asupra elementelor unui şir se pot aplica operatorii de comparare, o metodă generică de sortare quicksort descrie acțiunile pentru sortarea şirului, fără să specifice tipul de date al elementelor componente.

Sintaxa de definire a unei funții `template` este asemănătoare cu cea a unei clase `template`, în sensul că înainte de definirea funcției propriu-zise trebuie specificată aceeaşi construcție `template`.

Parametrii generici ai unei asemenea funcții pot fi, atât nume de tipuri de date, cât şi elemente ale altor tipuri de date. Toate observațiile specificate la parametrii generici ai claselor rămân valabile şi în cazul funcțiilor `template`.

Ca şi în cazul claselor parametrizate, instanțierea unei funcții `template` se face în momentul apelului acesteia. Există însă o deosebire importantă față de clase: în cazul funcțiilor nu este nevoie să se specifice în mod explicit valorile de instanțiere, deoarece acestea sunt deduse în mod automat de către compilator.

Exemplu. Funcția *MinVector* determină valoarea minimă a elementelor unui şir de numere. În funcția `main` se generează două instanțe ale funcției *MinVector*.

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

template <typename T >
T MinVector(T* v, unsigned n) {
  T min = v[0];
  for (unsigned k=1; k<n; k++)
    if (v[k] < min)
      min = v[k];
  return min;
}

void main() {
  int v1[] = {1, 3, 0, 8}, m1;
  double v2[] = {-1, 0, -9}, m2;
  m1 = MinVector(v1, 4);
  m2 = MinVector(v2, 3);
  cout << m1 << ", " << m2 << endl;
}
```

Deducerea argumentelor de instanțiere se poate efectua doar pe baza tipurilor de date ale parametrilor uzuali de apel ai funcțiilor template. În exemplul precedent, valoarea de

instanțiere a parametrului generic *T* se deduce din tipul de date al primului parametru uzual al funcției *MinVector* (parametrul *v*).

Însă nu este obligatoriu ca numele unui argument generic să se regăsească între parametrii formali uzuali ai unei funcții template. Deoarece domeniul de vizibilitate al unui asemenea argument se extinde asupra întregii definiții a funcției, este posibil ca un argument generic să fie utilizat doar în cadrul corpului funcției.

În acest caz însă, compilatorul nu dispune de suficiente informații pentru deducerea argumentelor de instanțiere. Din acest motiv, este necesar ca instanțierea să fie făcută în mod explicit, în mod asemănator instanțierii claselor parametrizate. Specificarea argumentelor de instanțiere se face între numele funcției şi lista parametrilor actuali obişnuiți.

Exemplu. Presupunând că clasa generică *T* are un constructor implicit, precum şi funcția *Print*, funcția template *f* are o variabilă locală de tipul *T*, pe care o inițializează şi îi afişează apoi valorea. În cazul în care argumentul de instanțiere nu ar fi fost explicit, un apel de forma:

```
f();
```

ar fi generat o eroare.

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

class A {
  int n;
public:
  A(int v = 0): n(v) {}
  void Print() const { cout << n << endl; }
};

template <class T>
void f() {
  T t;
  t.Print();
}

void main(void) {
  f<A>();
}
```

Observație. Chiar dacă există mai multe apeluri, compilatorul nu generează decât o singură instanță pentru fiecare tip de argument generic. De exemplu, dacă în exemplul funcției *MinVect*, funcția `main` ar fi fost următoarea:

```
void main() {
  int v1[] = {1, 3, 0, 8}, m1;
  double v2[] = {-1, 0, -9}, m2;
  int v3[] = {9, 2, 4, 7}, m3;
  m1 = MinVector(v1, 4);
  m2 = MinVector(v2, 3);
  m3 = MinVector(v3, 4);
  cout << m1 << ", " << m2 << ", " << m3 << endl;
```

```
}
```

ar fi existat tot două instanțe ale ei: `MinVect<int>` şi `MinVect<double>`.