Genericitatea în limbaje de programare

Prof.dr. Ion LUNGU
Catedra de Informatică Economică, A.S.E. Bucureşti
Asist. Iuliana TĂNASE
Universitatea "Constantin Brâncoveanu"

Articolul realizează o prezentare a noţiunii de genericitate şi a avantajelor acesteia considerând limbajul C++, care are implementat acest mecanism. Prima parte demonstrează uşurinţa şi eleganţa folosirii claselor template şi a funcţiilor template (template este echivalentul genericităţii în C++), iar cea de a doua simulează o funcţie template în limbajul Pascal, deşi limbajul nu deţine acest mecanism.

Cuvinte cheie: *genericitate, clasa template, functii template, polimorfism.*

Vom aborda noţiunile de moştenire şi genericitate, printr-o analiză comparativă, demonstrând că moştenirea este mecanismul mai puternic, că putem simula genericitatea folosind moştenirea.

Genericitatea reprezintă capacitatea de a defini module parametrizate, iar moştenirea posibilitatea de a construi module prin extensie succesivă şi specializare.

La baza acestor tehnici stau noţiunile de polimorfism şi supraîncărcare, unde polimorfismul este abilitatea de definire a entităţilor de program care pot lua mai mult decât o formă, iar supraîncărcarea (o simplă formă a polimorfismului) este abilitatea de a ataşa mai mult de un înţeles la acelaşi nume.

Prezenţa mecanismului de genericitate într-un limbaj de programare oferă o serie întreagă de avantaje: reduce spaţiul necesar memorării unui text sursă; constituie un mecanism de abstractizare a subprogramelor şi încurajează programatorul să se concentreze asupra algoritmului şi nu asupra detaliilor; prezintă un mare grad de portabilitate la nivelul tipurilor de date (modificarea declaraţiei unui tip de date într-un program nu necesită modificarea programului).

Utilizarea "claselor template"

Cei care utilizează un limbaj de programare îşi dau seama că foarte multe subprograme sunt identice din punct de vedere al algoritmului, singura diferenţă constând în tipul variabilelor folosite. Acelaşi algoritm se foloseşte şi pentru a determina minimul dintr-un şir de numere întregi şi pentru a determina minimul dintr-un şir de caractere. Pentru a înlătura aceste neajunsuri, încă de la primele versiuni ale limbajului C++ s-a introdus conceptul "template", care permite descrierea subprogramelor o singură dată pentru o familie de tipuri.

O clasă template este o generalizare a mai multor clase inviduale, acestea obtinându-se din clasa template prin specificarea concretă a tipului generic. De exemplu, pentru a crea un tabel cu înregistrări, a cărui structură se modifică de la o aplicaţie la alta se va folosi o clasă generică în care tipul "T" va inlocui clasa care defineşte structura tabelului. Tabelul va fi implementat sub forma unei stive, înzestrată cu operaţiile specifice.

```
template<class T>      // descrierea tabloului
template "AStiva"
class AStiva
{ public:
   void Push(T* a);
   T* Pop(void);
   void List(void);    // listarea stivei
   Astiva(void) { Varf=0;}
   ~AStiva(void);
 private:
  T* Spatiu[MaxDim];//elementele tabloului
  int Virf;    //nr. elem. ale tabloului tabloul
};
```

Prefixul "template<class T>" specifică faptul că a fost declarat un mecanism template, iar argumentul T este tipul ce va fi folosit

exact ca un alt nume de tip obişnuit. Domeniul lui T se extinde până la sfârşitul declaraţiei ce începe cu "template <class >". Funcţiile membru ale unei clase template care nu sunt declarate "inline" vor fi ele însele parametrizate cu tipul argumentului *clase* template.

```
template<class T>
void AStiva<T>::Push(T* a)
{ if(Virf==MaxDim)
   { cout<<"stiva este plina"<<endl;
    exit(1);      }
 else
  {   Spatiu[Virf]=new T;
      Spatiu[Virf++]=a; }
}
template<class T>
T* AStiva<T>::Pop(void)
{ T* a;
 if (Virf==0) {
    cout<<"stiva este goala"<<endl;
    return NULL; }
  else {
    a = new T;
    a=Spatiu[Virf--];
    return a;}
}
template<class T>
void AStiva<T>::List(void)  // listare stivă
{ if(Virf==0){
  cout<<"Stiva este goala"<<endl;
  exit(1);  }
 else
  for(int i=0;i<Virf;i++)
    Spatiu[i]->Afiseaza();}
template<class T>
AStiva<T>::~AStiva(void)    // destructorul
stivei ; dealocare de memorie
{ for(int i=0;i<Virf;i++)
    delete Spatiu[i];}
```

Cu ajutorul acestei clase se pot crea stive în care elemetele au tipuri predefinite (întregi ("T=int"), de caractere ("T=char"), şiruri de caractere("T=char*")), dar şi stive în care tipul elemetelor este definit de utilizator. In cazul în care stiva are elemente de tip predefinit este necesară o specificare a funcţiei "afişează" sau crearea de clase care se bazează pe tipurile standard şi în care să specificăm o metodă de vizualizare. Dacă tipul "T" nu este predefinit el poate fi orice clasă definită de utilizator (am considerat în secvenţa următoare o astfel de clasă ce poate ţine locul lui "T" numită clasa "tip"):

```
class tip{
 public:
   char* st1 ;
   char* st2;  // structura are doua campuri
   tip(void);
   tip(char* ,char* );
   void Afiseaza(void);
   tip& operator=(tip&); };
```

Această clasă defineşte o structură de tabel de două câmpuri (st1, st2) şi conţine constructori, o funcţie care afişează structura şi o redefinire a operatorului de copiere.

```
tip::tip(void)
{ strcpy(st1," ");
 strcpy(st2," ");}
tip :: tip(char* nm,char* nn)
{ nm=new char;
  strcpy(st1,nm);
  nn=new char;
  strcpy(st2,nn);}
void tip::Afiseaza(void)
{ cout<<st1<<"  "<<st2<<endl;}
tip& tip::operator=(tip& a)
{ tip& b=* new tip(a.st1,a.st2);
 return b;}
```

De exemplu, o stivă de elemete de tipul "tip" se declară *AStiva<tip> st*, iar o stivă de numere întregi se declară *AStiva<int> si*, unde numele clasei template urmat de <> este nume de clasă şi poate fi utilizat exact ca o clasă obişnuită.

Funcţii template

Avantajul folosirii claselor template nu poate fi obţinut şi la nivelul funcţiilor?
Funcţiile template reprezintă o consencinţă directă a claselor template. O funcţie template poate defini o familie de funcţii, căci parametrii săi actuali sunt de tipuri diferite. Să considerăm problema determinării minimului a două elemetelor (algoritm universal şi pentru întregi, caractere sau

şiruri de caractere). Naturaleţea acestei funcţii este de-a dreptul dezarmantă.

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

Corectitudinea acestei funcţii depinde foarte mult de operatorul de comparare. Datorită faptului că nu toate tipurile au definit operatorul "<", sau îl au definit, cu alt efect, dar este indicat să se definească o clasă template special pentru comparare, care să cuprindă şi versiuni pentru tipurile care nu se regăsesc în definiţia generală (pentru tipul şir de caractere ("T=char*") compararea se realizează folosind funcţii speciale de comparare).
Clasa "Comparare" ar putea fi introdusă într-o bibliotecă de clase pentru a putea fi folosită să exprime cerinţele de comparare în mai multe imprejurări.
Avantajul acestei versiuni constă în separarea codului de operaţii. Separarea preocupărilor reprezintă un avantaj major, căci cel care implementează o funcţie template nu poate cunoaşte întotdeauna totalitatea "argumentelor template" ale tipurilor.

Generarea funcţiilor template în Pascal

Simularea funcţiilor template în Pascal se bazează pe moştenire şi polimorfism.
Considerăm o problemă asemănătoare cu cea descrisă în C++: să se manipuleze o stivă polimorfă (elementele sale au tipuri diferite) cu posibilitatea selectării mai multor opţiuni (creare, ştergere, ordonare, adăugare). Vom încerca să realizăm şi aici aceeaşi funcţie generică "minim" pentru a putea face o comparare între cele două limbaje de programare. Parametrii actuali ai acestei funcţii vor fi numere întregi, numere reale sau caractere. Dacă putem obţine tipuri diferite pentru parametrii actuali atunci am obţinut o funcţie generică şi în Pascal. Prototipul funcţiei va fi:

function minim(a,b:RParinte):boolean;

unde RParinte reprezintă o referinţă la tipul TParinte (tipul tuturor parametrilor actuali ai acestei funcţii vor fi în mod obligatoriu descendenţi ai acestui tip). Combinarea acestei reguli cu metodele virtuale oferă o formă foarte puternică de polimorfism. In componenţa obiectului părinte intră metoda virtuală "m_mic", care va fi necesară în implementarea "funcţiei generice" şi metode de actualizare şi afişare de câmpuri. Descendenţi ai acestui obiect sunt "TIntreg", "TChar" şi "TReal", crearea lor făcându-se după modelul tipurilor simple de data: "integer", "char" şi "real". Elementele stivei vor avea aceste tipuri. Pentru a realiza o funcţie generică va fi necesar să lucrăm cu referinţele la aceste obiecte ("RIntreg", "RChar", "RReal") şi nu cu obiectele propriu-zise. Descrierea acestor obiecte şi ale "funcţiei generice" este următoarea:

```
RParinte=^TParinte;
Tparinte=object
   function
   m_mic(a.RParinte):boolean;virtual;
   procedure transforma; virtual;
   procedure afiseaza;
   procedure actualizeaza(a:RParinte);
   {.....}
 private
        vs:string;
end;
RIntreg=^TIntreg;
TIntreg=object(TParinte)
   constructor init(val:integer);
   function
   m_mic(a:RParinte):boolean;virtual;
   procedure transforma;virtual;
 private:
        v:integer;
end;
function
   TParinte.m_mic(a:RParinte):boolean;
begin
        Abstract;
end;
procedure TParinte.actualizeaza(a:RParinte);
begin
  vs:=a^.vs;
end;
procedure Tparinte.afiseaza;
begin
  writeln(vs);
end;
```

```
Procedure Tparinte.transforma;
begin
        Abstract;
end;
constructor Tintreg.init(val:integer);
begin
   v:=val;
   str(v,vs);
end;
function
    TIntreg.m_mic(a:RParinte):boolean;
var n,coder:integer;
begin
val(a^.vs,n,coder);
m_mic:=false;
if v<n then m_mic:=true;
end;
procedure Tintreg.transforma;
var coder:integer;
begin
  val(vs,v,coder);
end;
```

Analog obiectului "TIntreg" se vor defini şi obiectele "TChar" si "TReal".

Funcţia generică "minim" care are tipul parametrilor formali "RParinte" va putea fi apelată şi cu parametrii actuali "RIntreg", "RReal" sau "RChar", deoarece aceştia reprezintă referinţe la obiecte ce moştenesc "TParinte", iar "RParinte" este referinţă şi el la "Tparinte":

```
function minim(c,d:RParinte):boolean;
   begin
      if c^.m_mic(d)=true then
                minim:=true
          else minim:=false;
   end;
```

Problema iniţială propunea manipularea unei stive în care elementele să fie întregi, reale sau caractere. În cazul în care conţinutul stivei este format din obiecte de acelaşi tip, este posibilă şi ordonarea acestora.

```
Stiva=object
 procedure Push(a:RParinte);
 procedure Pop(var a:RParinte);
 procedure list;
 destructor term;
 procedure ordonare;
private:
 Spatiu:array[0..MaxDim] of Rparinte;  //
elemetele stivei
 Varf:0..MaxDim;       // nr. de elemete din
stiva
end;
```

Metoda care prezintă importanţă pentru noi este "Ordonare" şi se implementează astfel:

```
procedure Stiva.Ordonare;
var i:1..MaxDim;
 aux:RParinte;
 sw:boolean;
begin
 if Varf=0 then WriteLn('Stiva vida')
 else begin
  repeat
     sw:=true;
     for i:=1 to Varf-1 do
      if minim(Spatiu[i],Spatiu[i-1])
     then begin
             aux:=Spatiu[i];
             Spatiu[i]:=Spatiu[i+1];
             Spatiu[i+1]:=aux;
             sw:=false;
           end
  until sw;
end;
end;
```

După cum se poate observa efectul genericităţii s-a obţinut şi într-un limbaj care nu are implemetat acest mecanism însă acesta cu preţul unei exprimări mai greoaie.

Bibliografie

1. STROUSTRUP, B., The C++ Programming Language, Addison-Wesley, 1987
2. MUSLEA, I., C++ pentru avansaţi, Ed. Microinformatica, luj-Napoca, 1994
3. SOMNEA, D., Iniţiere în C++. Programarea orientată pe biecte, Ed. Tehnică, Bucureşti, 1993
4. PÂRV, B., Fundamentele limbajelor de programare, lit. Universitatea Babeş-Bolyai, Cluj-Napoca, 1992