STUDIA UNIVERSITATIS
Revistă științifică a Universității de Stat din Moldova, 2010, nr.9(39)

DESPRE UNELE ASPECTE ALE PROGRAMĂRII GENERICE

Silviu GÎNCU

Universitatea de Stat din Tiraspol

In this article we show how to use the generic programming in C++. Are describes examples of creating and using the function and classes template.

Introducere

Stilurile de programare au progresat foarte mult, principala cauză constând în necesitatea acomodării la complexitatea crescândă a produselor software. Odată cu aceasta a devenit tot mai evident că programarea structurată este ineficientă. Programele sunt tot mai mari, având o complexitate sporită. De aceea, un rol substanțial îl joacă abstractizarea datelor. În 1984 Shankar afirma că „natura abstractizărilor ce pot fi obținute prin utilizarea procedurilor este adecvată descrierii operațiilor abstracte, dar nu este adecvată descrierii obiectelor abstracte".

S-a impus, deci, găsirea unui nou model de programare capabil să depășească limitele programării structurate care să efectueze o abstractizare adecvată a datelor. Astfel a apărut clasa limbajelor *bazate pe obiecte*, apoi a celor *orientate pe obiecte.*

Un avantaj al utilizării programării orientate pe obiect este programarea generică. Aceasta este o metodă de programare în care funcțiile şi clasele au parametri formali de tip nedefinit.

În limbajul C++ programarea generică poate fi realizată prin intermediul *template*. „*Template* (sau clasa parametrizată) implementează conceptul de tip parametrizat. O clasă parametrizată reprezintă un şablon (sau container) ce defineşte o mulțime de clase" [1, p.187].

Funcții template

Funcțiile template (şablon) sunt concepute pentru a facilita scrierea funcțiilor cu algoritmi similari, deosebindu-se doar prin tipul datelor prelucrate. O funcție template are drept parametru formal tipul acesteia.

Declararea unei funcții template se realizează conform sintaxei:

```
template <class T1,class T2,..., class Tn>
[tip_returnat] nume_functie ([lista_parametri_formali]){
//instructiuni
}
```

Pentru a apela o funcție-şablon vom scrie:

```
nume_functie< T1,T2,...,Tn>([lista_parametri_actuali]);
```

unde `template` este cuvânt-cheie;

`T1,T2,...,Tn` sunt o serie de tipuri abstracte.

Şabloanele permit utilizarea unei funcții pentru o gamă largă de mai multe tipuri. Drept exemplu considerăm problema 1, care prezintă o funcție-şablon pentru determinarea elementului maximal dintre două valori.

Problema 1

```
#include<iostream.h>
template <class T> T max(T a,T b){
 if(a>b) return a; else return b;}
int main(){
 cout<<"Numere intregi : "<<max<int>(4,10)<<endl;
 cout<<"Numere reale : "<<max<float>(3.56,2.3)<<endl;
 cout<<"Caractere : "<<max<char>('i','h')<<endl;
 return 0;
}
```

Astfel, compilatorul creează câte o funcție pentru determinarea elementului maximal. La apelul funcției parametrizate, tipul argumentului determină versiunea şablonului care este folosit.

Clase template

În cazul în care într-un program sunt utilizate mai multe funcţii-şablon, care prelucrează tipuri de date similare, se recomandă a utiliza clase template (şablon). O clasă template reprezintă o formă generică care în momentul utilizării va fi folosită pentru crearea de tipuri concrete. Declararea unei clase template se realizează conform sintaxei:

```
template <class T1,class T2,..., class Tn>
class nume_clasa{
//date şi metode
};
```

Descrierea metodelor clasei:

```
template <class T1,class T2,..., class Tn> [tip_returnat]
nume_clasa<T1,T2,...,Tn>::nume_metoda([lista_parametri_actuali]){
//instructiuni
}
```

Crearea obiectelor:

```
nume_clasa<T1,T2,...,Tn> lista_obiecte;
```

Metodele obiectelor template vor fi apelate în mod tradiţional prin intermediul operatorului săgeată („->”), dacă obiectul este pointer şi prin intermediul operatorului punct („.”), dacă acesta nu este pointer.

Prefixul template `template <class T1,class T2,..., class Tn>` specifică declararea unui template cu argumentele `T1,T2,...,Tn`. După această introducere, argumentele `T1,T2,...,Tn` sunt folosite exact la fel ca orice tip de date, în tot domeniul clasei template declarate.

Drept exemplu se consideră clasa-vector cu metodele de citire, afişare, sortare.

Problema 2

```
#include<iostream.h>
#include<iomanip.h>
#define n 5
template <class T> class vector{
public:
T v[n];
void citire();
void afisare();
void sortare();
};
template <class T> void vector<T>::citire(){
 for(int i=0;i<n;i++) {
  cout<<"Introdu elementul "<<i<<" ";
  cin>>v[i];
 } }
template <class T> void vector<T>::afisare(){
 for(int i=0;i<n;i++) cout<<setw(4)<<v[i];
 cout<<endl;
}
template <class T> void vector<T>::sortare(){
 int i,j;
 T x;
 for(i=1; i<n; i++) {
 x = v[i];  j = i - 1;
 while((j >= 0) && (x < v[j])) {
  v[j+1] = v[j]; j--;
 } v[j+1] = x;
 }}
template <class T> void apel(vector<T> a){
a.citire();
a.afisare();
a.sortare();
cout<<"Elementele Sortate"<<endl;
a.afisare();
}
```

```
int main(){
vector<int> vi;
 vector<char> vc;
 cout<<"Dati 5 numere intregi"<<endl;
 apel(vi);
 cout<<"Dati 5 caractere"<<endl;
 apel(vc);
return 0;
}
```

Declarația unui şablon cere compilatorului să utilizeze un tip care va fi precizat mai târziu. La începutul declarației se foloseşte următoarea sintaxă:

```
template <class T> vector
```

Aceasta arată compilatorului că un utilizator al clasei-vector va furniza un tip când şablonul va fi multiplicat şi că acel tip trebuie folosit oriunde este plasat `T` în întreaga declarație de şablon.

Ierarhizarea şabloanelor

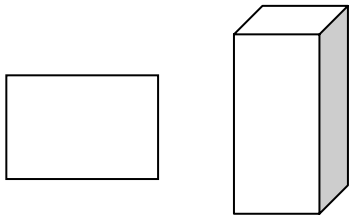
Clasele pot fi ierarhizate prin intermediul şabloanelor. Ierarhizarea se poate efectua prin două metode:

✓ Prin moştenire: „Atunci când o clasă transmite parametrii sau funcționalitatea altei clase, care la rândul său se consideră clasă de bază pentru o altă ierarhie de moştenire” [3, p.50].

✓ Prin agregare: „Agregarea este relația dintre două obiecte în care unul dintre obiecte aparține celuilalt obiect. Agregarea redă apartenența unui obiect la un alt obiect. Semantic, agregarea indică o relație de tip "***part of***" ("*parte din"*)” [2, p.70].

Moştenirea şabloanelor

Clasele template ca şi clasele obişnuite utilizează mecanismul de moştenire. Toate principiile de bază ale procedeului de moştenire rămân neschimbate. Astfel, se oferă posibilitatea de a construi diferite modele ierarhice de clase. Fie dată ierarhia (figura):



Figură.

Se consideră drept bază clasa *dreptunghi*, iar în calitate de derivată clasa *prismă*. În această ierarhie va fi realizat polimorfismul pentru metodele *citire*, *afisare*, *suprafata* şi *volum*. Se vor descrie şi constructorii ambelor clase.

Problema 3

```
#include<iostream.h>
template <class T> class drept {
public:
 T a,b;
drept(){};
drept(T,T);
virtual void citire();
virtual void afisare();
virtual T suprafata();
virtual T volum(){return 0;}//metodă virtuala pura
};
template <class T> drept<T>::drept(T x, T y){a=x;b=y;}
template <class T> void drept<T>::citire(){
cout<<"a=";cin>>a;
cout<<"b=";cin>>b;
}
```

```
template <class T> void drept<T>::afisare(){
cout<<"Dreptunghi lungimile laturilor: "<<a<<" "<<b<<endl;
cout<<"Suprafata: "<<suprafata()<<endl;
}
template <class T> T drept<T>::suprafata(){return a*b;}
template <class T> class prisma : public drept< T>  {
public:
 T h;
prisma(){};
prisma(T,T,T);
void citire();
void afisare();
T suprafata();
T volum();
};
template <class T> prisma<T>::prisma(T x,T y,T z):drept<T>(x,y){h=z;}
template <class T> void prisma<T>::citire(){
drept<T>::citire();
cout<<"h=";cin>>h;
}
template <class T> void prisma<T>::afisare(){
cout<<"Prisma lungimile laturilor bazei: "<<a<<" "<<b<<"Inaltimea: "<<h<<endl;
cout<<"Suprafata: "<<suprafata()<<" Volumul: "<<volum()<<endl;
}
template <class T> T prisma<T>::suprafata(){return 2*(a*b+a*h+b*h);}
template <class T> T prisma<T>::volum(){return drept<T>::suprafata()*h;}
int main(){
int i; double st,vt;
drept<int> *p[4];
p[0]=new drept<int>(2,3);
p[1]=new prisma<int>(4,2,7);
p[2]=new drept<int>; p[2]->citire();
p[3]=new prisma<int>;p[3]->citire();
cout<<"Datele introduse de tipul int"<<endl;
for(i=0;i<4;i++) p[i]->afisare();
drept<double> *t[4];
t[0]=new drept<double>(2.5,3);
t[1]=new prisma<double>(4.3,2,7.4);
t[2]=new drept<double>; t[2]->citire();
t[3]=new prisma<double>;t[3]->citire();
cout<<"Datele introduse de tipul double"<<endl;
for(i=0;i<4;i++) t[i]->afisare();
 prisma<double> b[3];
 cout<<"Dati datele a 3 prisme"<<endl;
 for(i=0;i<3;i++)
 b[i].citire();
 vt=st=0.0;
 cout<<"Datele introduse"<<endl;
 for(i=0;i<3;i++){  b[i].afisare();
 vt+=b[i].volum();
 st+=b[i].suprafata();
 }
 cout<<"Volumul total:="<<vt<<endl;
 cout<<"Suprafata totala:="<<st<<endl;
return 0;
}
```

Ierarhizarea şabloanelor prin agregare

Un alt mecanism pentru crearea ierarhiilor de clase este agregarea. Aceasta presupune că un obiect este inclus în totalitate într-un alt obiect. Exemple de astfel de ierarhii pot servi: lista, coada, arborii etc. Problema 4 este un program prin intermediul căruia este creată o stivă.

Problema 4

```
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
template <class T> class celula{
public:
T elem;
celula *next;
celula(){next=NULL;}
void citire();
void afisare();
};
template <class T> void celula<T>::citire(){cin>>elem;}
template <class T> void celula<T>::afisare(){cout<<setw(6)<<elem;}
template <class T> class stiva{
 public:
 celula<T> *curent;
 stiva(){curent=NULL;}
 void creare();
 void parcurge();
 void inserare();
 void exclude();
 ~stiva();
};
template <class T> stiva<T>::~stiva(){while(curent!=NULL) exclude();}
template <class T> void stiva<T>::creare(){
 int c;
 cout<<"Introdu numarul de elemente din stiva"<<endl;cin>>c;
 for(int i=0;i<c;i++) {
  if(curent==NULL)  {
   curent=new celula<T>;
   curent->citire();
  }else  inserare();
 }
}
template <class T> void stiva<T>::parcurge(){
 celula<T> *p;
 p=curent;
 while(p!=NULL) {
  p->afisare();
  p=p->next;
 }cout<<endl;
}
template <class T> void stiva<T>::inserare(){
 celula<T> *q;
 q=new celula<T>;
 q->citire(); q->next=curent;
 curent=q;
}
template <class T> void stiva<T>::exclude(){
 celula<T> *q;
 q=curent; curent=curent->next;
 delete q;
}
```

```
template <class T> void meniu( stiva<T> a){
char c;
a.creare();clrscr();
do{
 cout<<"Alegeti una dintre optiuni:"<<endl;
 cout<<"1-Parcurge"<<endl;
 cout<<"2-Inserare"<<endl;
 cout<<"3-Exclude"<<endl;
 cout<<"0-iesire"<<endl;
 c=getch();clrscr();
 switch(c){
 case '1':a.parcurge();getch();break;
 case '2':a.inserare();break;
 case '3':a.exclude();break;
 }clrscr();
 }while(c!='0');
}
int main(){
 clrscr();
 cout<<"Stiva de numere intregi"<<endl;
 stiva<int> sn;
 meniu(sn);
 cout<<"Stiva de caractere"<<endl;
 stiva<char> sc;
 meniu(sc);
return 0;
}
```

Concluzii

Şabloanele reprezintă un instrument puternic şi eficient pentru manevrarea diferitelor tipuri de date. Totodată, trebuie amintit că aceste instrumente sunt destinate pentru utilizarea corectă şi necesită cunoştinţe aprofundate în domeniu. Există însă trei mari inconveniente întâlnite la utilizarea şabloanelor cum ar fi:

1. Foarte multe compilatoare au un suport limitat pentru şabloane, astfel încât utilizarea şabloanelor poate determina scăderea portabilităţii codului-sursă;
2. Aproape toate compilatoarele produc mesaje de eroare şi derutante când sunt detectate erori în codul şablonului. Aceasta poate face ca şabloanele să fie greu de programat.
3. Fiecare utilizare a unui şablon poate determina generarea de către compilator a unei noi versiuni de cod pentru noua instanţă a şablonului, deci, utilizarea fără discernământ a şabloanelor poate duce la încărcarea codului.

Referinţe:

1. Braicov Andrei, Gîncu Silviu. „C++ Builder". Ghid de iniţiere. - Chişinău: Tipografia centrală, 2009. - 196 p.
2. Grady Booch. Object-Oriented Design with Applications. Benjamin/Cummings, Redwood City, California, 2nd edition, 1994. - 534 p.
3. Arnaut Vsevolod, Putină V., Andrieş I. Programarea orientată pe obiecte în baza limbajului C++". - Chişinău: CEP USM, 2009. - 153 p.

Prezentat la 01.12.2010