Silviu **GÎNCU**

doctorand, Universitatea de Stat din Tiraspol

Utilizarea şabloanelor în limbajul C++

Abstract: *In this article show how to use generic programing in C++. Are describes examples of creating and using the function and classes template.*

INTRODUCERE

Informatica, ca disciplină de studiu, participă la formarea şi dezvoltarea generală a personalităţii elevului, punctul forte fiind dezvoltarea gîndirii logice şi algoritmice. Gîndirea algoritmică se bazează pe operaţii,

pe treceri riguroase de la o stare la alta în succesiunea obligatorie a evenimentelor în timp.

Un algoritm (cuvîntul are la origine numele matematicianului persan Al-Khwarizmi) reprezintă o metodă sau o procedură de calcul alcătuită din paşii elementari necesari pentru rezolvarea unei probleme sau categorii de probleme. De obicei, algoritmii se implementează în mod concret prin programarea adecvată a unui calculator sau a mai multora. Şabloanele sînt o caracteristică a limbajului de programare C++, care permit scrierea de cod fără a lua în considerare tipul de date ce va fi utilizat.

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

FUNCŢII TEMPLATE

Funcţiile template (şablon) sînt concepute pentru a uşura scrierea funcţiilor cu algoritmi similari, deosebindu-se doar prin tipul datelor prelucrate. O funcţie template are în calitate de parametru formal tipul acesteia.

Declararea unei funcţii template se realizează conform sintaxei:

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

Pentru a apela o funcţie şablon, vom scrie:

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

unde `template` este cuvînt-cheie;
`T1,T2,...,Tn` sînt o serie de tipuri abstracte.

Şabloanele permit utilizarea unei funcţii pentru o gamă largă de tipuri. Drept exemplu se consideră *Problema 1*, care prezintă o funcţie 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 : ";
cout <<max<int>(4,10)<<endl;
cout<<"Numere reale : ";
cout <<max<float>(3.56,2.3)<<endl;
cout<<"Caractere : ";
cout <<max<char>('i','h')<<endl;
}
```

Astfel, compilatorul creează cîte o funcţie pentru determinarea elementului maximal. La apelul funcţiei parametrizate, tipul argumentului determină ce versiune a şablonului este folosită.

CLASE TEMPLATE

În cazul în care într-un program sînt 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 implementării 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 si 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` sînt folosite exact la fel ca orice tip de date, în tot domeniul clasei template declarate.

În calitate de exemplu se consideră clasa *vector* cu metodele *citire, afisare, 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);
}
```

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

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

Aceasta arată compilatorului că un utilizator al clasei vector va furniza un tip în care şablonul va fi multiplicat şi că acest tip trebuie folosit oriunde este plasat `T` în declarația de şablon.

IERARHIZAREA ŞABLOANELOR

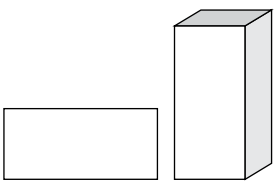
Şabloanele sînt utilizate la ierarhizarea claselor, care poate fi efectuată în două direcții:

- ✓ prin moştenire – „atunci cînd o clasă transmite parametri sau funcționalitatea altei clase care, la rîndul său, se consideră clasă de bază pentru o altă ierarhie de moştenire" [3, pag. 50];
- ✓ prin agregare – „agregarea este relația dintre două obiecte care aparțin unul celuilalt. Agregarea redă apartenența unui obiect la un alt obiect. Semantic, aceasta indică o relație de tip "***part of***" ("*parte din"*)" [2, pag. 70].

MOŞTENIREA ŞABLOANELOR

Clasele template, ca şi clasele obişnuite, susțin mecanismul de moştenire. Toate principiile de bază ale moştenirii rămîn neschimbate. Astfel, se oferă posibilitatea de a construi modele ierarhice de clase.

Fie dată ierarhia:



Se consideră drept bază clasa *dreptunghi*, iar derivată – clasa *prismă*. Această ierarhie va implica realizarea polimorfismului pentru metodele *citire*, *afisare*, *suprafata* şi *volum*. Se va 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: ";
cout<<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:";
cout<<a<<" "<<b<<"Inaltimea: "<<h<<endl;
cout<<"Suprafata: "<<suprafata();
cout<<" 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 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;
}
```

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: lista, coada, arbori, 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";
cout<<"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);
}
```

CONCLUZII

Aşadar, utilizarea şabloanelor va duce la dezvoltarea gîndirii algoritmice a elevului. Prin intermediul acestora elevul va învăţa:

- să creeze o funcţie şablon;
- să utilizeze un algoritm la rezolvarea unor probleme similare, cu tipuri de date distincte;
- să scrie un program în limbajul C++;
- să utilizeze mecanismele programării orientate pe obiecte la elaborarea de algoritmi;
- să aplice modele de algoritmizare, de analiză şi de programare pentru soluţionarea problemelor legate de prelucrarea automatizată a informaţiei;
- să scrie algoritmii frecvent utilizaţi în limbajul C++.

REFERINŢE BIBLIOGRAFICE:

1. Braicov, A.; Gîncu, S.; *„C++ Builder". Ghid de Iniţiere*, Tipografia Centrală, 2009.
2. Grady, B.; *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, California, 2nd edition, 1994.
3. Arnaut, V.; Putină, V.; Andrieş, I., *Programarea orientată pe obiecte în baza limbajului C++,* CEP USM, 2009.