Elemente de programare generica în limbajul C/C++

Lect.dr. Marian DÂRDALA
Catedra de Informatica Economica, A.S.E. Bucuresti

În domeniul programarii calculatoarelor se urmareste o implementare cît mai generala a algoritmilor. Generalitatea poate fi privita în raport de tipurile de date asociate parametrilor de apel functiilor, dar si de posibilitatea de a particulariza algoritmul prin intermediul unor secvente de cod furnizate la apel.
Acest mod de a programa este strîns legat de facilitatile pe care le ofera un limbaj de programare. Se poate spune ca limbajul C/C++ este unul dintre cele mai performante în acest sens, iar facilitatile au fost adaugate pe parcursul aparitiei de noi versiuni. În articol se vor prezenta modalitatile de construire a functiilor generice specifice programarii procedurale si nu obiectuale.
Cuvinte cheie: *programare generica, pointeri generici, sabloane de functii, substituire simbolica, tip generic de data.*

Folosirea pointerilor în scrierea functiilor generice

Dupa cum se cunoaste, prin intermediul pointerilor se pot trimite si returna date în/din functii. Aritmetica de pointeri impune ca pointerii sa fie spre un tip de data bine precizat. Pentru a asigura o mai mare generalitate în raport de tipul de data a variabilelor procesate în cadrul functiilor se vor folosi pointeri generici (*void **).
Sa presupunem ca vrem sa construim functia de cautare secventiala a unui element într-un vector de întregi; functia va returna pozitia elementului sau –1 daca el nu exista.

```
int cauta(int *v, int n, int k)
{
for(int i=0; i<n; i++)  if( v[i]==k ) return i;
return -1;
}
```

Functia primeste ca parametri vectorul (*int *v*), numarul de elemente (*int n*) si elementul de cautat (*int k*).
Pentru a nu fi dependenta de tipul de data al vectorului, functia va fi reconstruita astfel încît vectorul sa fie primit prin intermediul unui pointer la *void*. Lucru cu o adresa spre un tip nedefinit are implicatii în sensul ca e necesar sa se transmita un parametru suplimentar care sa exprime lungimea tipului de data a elementelor vectorului.
Functia are forma:

```
#include <memory.h>
int cauta(void *v,int n,int dim_el,void *el)
{
char *ec=(char*)v;
for(int i=0; i<n; i++)
   if(!memcmp(ec+i*dim_el,el,dim_el))
       return i;
return -1;
}
```

Parametrul *int dim_el* indica dimensiunea în baiti a tipului elementelor vectorului, iar elementul de cautat este dat prin adresa lui (parametrul *void *el*), lungimea se considera a fi aceeasi cu a elementelor vectorului.
Se observa ca pointerul primit în functie (*v*) este convertit din *void** în *char** pentru ca dimensiunea unui caracter este de un octet, iar dimensiunea unui element (*dim_el*) este exprimata în baiti.
Compararea dintre elemente se face folosindu-se functia de biblioteca *memcmp* care primeste doua adrese si lungimea în baiti a zonei de memorie de comparat. Prototipul ei se afla în fisierul header *memory.h*.

Modalitatile diferite de apel ale functiei *cauta* se vor prezenta construind functia *main* dupa cum urmeaza:

```
#include <stdio.h>
#include <string.h>
void main()
{
        int a[]={3,6,4,1,8},t=4,k;
        (k=cauta(a,sizeof(a)/sizeof(int),sizeof(int),&t)) == -1 ? printf("\nElement inexis-
                tent!!!") : printf("\nElementul pe pozitia %d",k);
        double b[]={3.5,6.2,4.9,28.15},el_r=6.2;
        (k=cauta(b,sizeof(b)/sizeof(double),sizeof(double),&el_r)) == -1 ?
                printf("\nElement inexistent!!!") : printf("\nElementul pe pozitia %d",k);
        char *sir="Sir de caractere!!!", el_c='i';
        (k=cauta(sir,strlen(sir),sizeof(char),&el_c)) == -1 ?
                printf("\nElement inexistent!!!") : printf("\nElementul pe pozitia %d",k);
}
```

Definirea functiilor generice utilizînd directive de preprocesare

În esenta, aceasta modalitate se concretizeaza în construirea de macrodefinitii cu parametri în care se definesc functii. Se stie ca o macrodefinitie substituie parametri, la momentul apelului la nivel de simbol, deci codul se poate parametriza nu numai în sensul redenumirii unor nume de variabile, ci chiar în sesul substituirii unor tipuri de date.

Se va construi o macrodefinitie cu parametri în vederea definirii functiei de cautare; parametrul va fi un tip generic de data.

```
#define CAUT(TIP)  int cauta(TIP *v, int
n, TIP k)\
        {\ for(int i=0; i<n; i++)\
           if( v[i] == k ) return i;\
           return -1;\
        }
```

Apelul CAUT(int); va determina definirea functiei *cauta* pentru tipul de data întreg (*int*). Se pot expanda, în acest fel, functii pentru diferite tipuri de date, lucru posibil datorita facilitatii de supraîncarcare a functiilor, permis în limbajul C++.

Se va apela functia *cauta* pentru a cauta elemente în vectori de tip: întreg (*int*), real (*double*) si caracter (*char*).

```
CAUT(int);
CAUT(double);
CAUT(char);
#include <stdio.h>
#include <string.h>
void main()
{       int a[]={3,6,4,1,8},t=4,k;
        (k=cauta(a,sizeof(a)/sizeof(int),t)) == -1 ?
                 printf("\nElement inexistent!!!") : printf("\nElementul pe pozitia %d",k);
        double b[]={3.5,6.2,4.9,28.15},el_r=6.2;
        (k=cauta(b,sizeof(b)/sizeof(double),el_r)) == -1 ?
                printf("\nElement inexistent!!!") : printf("\nElementul pe pozitia %d",k);
        char *sir="Sir de caractere!!!", el_c='x';
        (k=cauta(sir,strlen(sir),el_c)) == -1 ?
                printf("\nElement inexistent!!!") : printf("\nElementul pe pozitia %d",k);}
```

De observat ca apelurile macrodefinitiei *CAUT* trebuie sa se faca în afara oricaror alte functii, deoarece o functie nu poate fi definita în alta functie în limbajul C.

Construirea de functii generice prin definirea sabloanelor de functii

Limbajul C++ pune la dispozitia programatorului o constructie speciala pentru a putea defini sabloane de functii. Un sablon de functie permite definirea unei functii parametrizata la nivelul tipurilor de date asociate unor variabile sau parametri de apel si a unor constante folosite în functie. Constructia are forma:

template < [lista_tipuri] [, [lista_argumente]] > *declaratie*

unde:

- lista_tipuri – reprezinta o lista de tipuri de data, indicata sub forma ***class id;***
- lista_argumente – reprezinta o lista de argumente, data în forma *nume_tip id.*

Se va construi sablonul functiei de cautare parametrizat la nivelul tipului elementelor masivului în care se va face cautarea:

```
template <class TIP>
int cauta(TIP *v, int n, TIP k)
{
        for(int i=0; i<n; i++)
                if(v[i] == k) return i;
        return -1;
}
```

Apelul se face în mod natural dupa cum se poate vedea în functia *main* de mai jos:

```
#include <stdio.h>
#include <string.h>
void main()
{
        int a[]={3,6,4,1,8},t=23,k;
        (k=cauta(a,sizeof(a)/sizeof(int),t)) == -1 ?
                printf("\nElement inexistent!!!") : printf("\nElementul pe pozitia %d",k);
        double b[]={3.5,6.2,4.9,28.15},el_r=6.2;
        (k=cauta(b,sizeof(b)/sizeof(double),el_r)) == -1 ?
                printf("\nElement inexistent!!!") : printf("\nElementul pe pozitia %d",k);
        char *sir="Sir de caractere!!!", el_c='a';
        (k=cauta(sir,strlen(sir),el_c)) == -1 ?
                printf("\nElement inexistent!!!") : printf("\nElementul pe pozitia %d",k);
}
```

Pentru exemplificarea definirii de sabloane parametrizate si la nivelul unor constante utilizate în functie se va defini functia ce realizeaza produsul a doi vectori:

$$z_i = x_i * y_i, \quad i = 0, n-1$$

Parametri sablonului sunt:

- tipul de data al elementelor vectorului;
- dimensiunea maxima a vectorului.

```
template <class T, int N_EL>
T *prod_v(T *x, T *y, int n)
{
        static T z[N_EL];
        for(int i=0; i<n; i++) z[i]=x[i]*y[i];
        return z;
}
```

Folosirea unui astfel de sablon se va exemplifica prin construirea functiei *main*, dupa cum urmeaza:

```
#include <stdio.h>
#define NMAX 20
void main()
{
        int a[NMAX]={3,6,4,8}, b[NMAX]={2,1,3,4}, n=4, *rez;
        rez=prod_v<int, NMAX>(a,b,n);
        for(int i=0;i<n;i++) printf("\n rez[%d]=%d",i,rez[i]);
}
```

Apelul functiei *prod_v* se face în forma rez = prod_v<int, NMAX>(a,b,n); deoarece altfel compilatorul nu poate deduce valoarea pe care sa o atribuie parametrului *N_EL* din sablon.

Se poate observa, din exemplele prezentate, ca sabloanele sunt particularizate la momentul preprocesarii, cînd sunt substituite tipurile generice si constantele, ulterior compilatorul va compila un cod sursa C valid.

Concluzii

O restrictie a acestui mod de abordare se refera la faptul ca doar anumiti algoritmi se preteaza la implementari generice.

Variantele prezentate sunt oarecum echivalente, prin faptul ca propun abordari generice, dar ultimile doua se pot aplica la o gama mai larga de algoritmi. Spre exemplu, folosind pointeri generici nu se poate implementa o functie care sa realizeze suma elementelor unui vector deoarece necunoscîndu-se tipul de la care s-a plecat, operatia de adunare nu s-ar efectua corect. Celelalte doua alternative care lucreaza cu tipuri generice de date substituite la momentul preprocesarii cu tipuri de date concrete, nu mai prezinta acest neajuns.

Bibliografie

- Mosich, D., Shammas, N., Flaming, B., *Advanced Turbo C Programmer's Guide*, John Wiley & Sons, New York, 1988
- Muslea, I., *C++ pentru avansati*, Ed. microinformatica, Cluj–Napoca, 1994
- Smeureanu, I., Ivan, I., Dârdala, M., *Structuri si obiecte în C++*, Ed. CISON, Bucuresti, 1998
- Smeureanu, I., Dârdala, M., *Programarea în limbajul C/C++*, Ed. CISON, Bucuresti, 2001