LABORATOR XI
TEMPLATE-URI, STANDARD TEMPLATE LIBRARY (STL)

XI.1 SCOPUL LUCRĂRII

Lucrarea are ca scop analizarea dezvoltării şi modului de utilizare a *template*-urilor, fără a pierde din vedere contextul, care conţine construcţii şi idei care au fost deja discutate. Este urmărit modul de abstractizare a tipurilor de date în direcţia pe care se regăseşte STL. STL este o bibliotecă de template-uri dezvoltată la HP[2] şi SGI[3] care face parte din draft-ul de standard ANSI/ISO pentru C++.

XI.2 TEMPLATE-URILE – ABSTRACTIZAREA TIPURILOR DE DATE

Pe scurt, template-urile reprezintă o formă de abstractizare suplimentară prin tipurile de date. Din alt punct de vedere, în cazul template-urilor de funcţii, este vorba de o soluţie de supradefinire automată a funcţiilor şi operatorilor. Supradefinirea este o formă slabă de polimorfism de tip *early-binding*. Acelaşi lucru se poate afirma despre template-uri.

Polimorfismul de tip early-binding este mai ales eficient pentru programator: reduce spaţiul numelor, favorizează reutilizarea codului şi creşte coerenţa acestuia. Template-urile folosite cu eleganţă pot conduce la o creştere foarte accentuată a productivităţii programatorului, la uşurarea întreţinerii aplicaţiilor şi la creşterea performanţelor acestora.

Template-urile de funcţii se pot considera extensii la supradefinirea funcţiilor considerând că tipurile de date reprezintă parametri. De cele mai multe ori template-urile de funcţii reprezintă o formă de abstractizare a unui set de funcţii supradefinite. Utilizarea template-urilor impune şi o serie de condiţii asupra tipurilor de date parametrizate cum ar fi: existenţa unor operatori definiţi pentru tipurile respective, eventual o anumită structurăă, etc. Dintre aceste restricţii este interesant să trecem în revistă un concept general: clase (în sensul de tipuri de date) *simpatice*[4]. O clasă T este simpatică dacă are definiţi:

```
T (const T&) //constructor de copiere
T& operator= (const T&) //operator de atribuire
int operator== (const T&, const T&) //operatorul egal
int operator!= (const T&, const t&) //operatorul diferit
```

[2] Hewlett-Packard Company

[3] Silicon Graphics Computer Systems, Inc.

[4] Nice classes – concept rezultat în urma colaborării HP – Bell Labs.

astfel încât:

T a(b); implică a == b
a = b; implică a == b;
a == a;
a == b dacă şi numai dacă b == a
(a == b) && (b == c) implică (a == c)
a != b dcacă şi numai dacă ! (a == b)

O clasă T este *super simpatică*[5] dacă este simpatică şi suplimentar toate metodele *conservă relaţia de egalitate*[6]. O metodă a clasei T::s(...) Conservă relaţia de egalitate dacă:

a == b implică a.s(...) == b.s(...)

XI.3 UN EXEMPLU FOARTE SIMPLU

Un exemplu, de fapt extrem de simplu, este funcţia Swp care are ca scop interschimbarea valorilor a două obiecte. Pentru tipul int funcţia este:

```
void Swp(int &x, int &y)
{
        int Temp(x);
        x = y;
        y = Temp;
}
```

Pentru tipul de date float aceeaşi funcţie este:

```
void Swp(float &x, float &y)
{
        float Temp(x);
        x = y;
        y = Temp;
}
```

Imediat se poate analiza contextul în care funcţionează această funcţie: un tip de date pentru care este definit constructor de copiere şi operator de atribuire, acestea fiind ingredientele necesare pentru exprimarea algoritmului. Deci algoritmul de interschimbare a valorilor a două obiecte poate fi făcut independent de tipul obiectelor în momentul în care tipul respectiv satisface nişte cerinţe. Varianta parametrizată a algoritmului scris sub formă de template este:

[5] Extra nice

[6] Equality preserving

```
template<class T>
void Swp(T &x, T &y)
{
        T Temp(x);
        x = y;
        y = Temp;
}
```

Parametrul în acest caz fiind tipul T.

Compilatorul generează automat varianta corectă a funcţiei pe care urmează să o utilizeze deducând tipul din parametrii de apel. În consecinţă nu există cod pentru obiectul template ci doar pentru instanţele particulare ale acestuia obţinute prin particularizarea tipurilor de date. O altă concluzie este că este nevoie atât de partea declarativă cât şi de definiţie în fiecare unitate a proiectului unde se utilizează template-uril. Este deci foarte natural ca template-urile sa fie plasate în fişiere header. Bibliotecile de template-uri, în general, conţin numai fişiere header.

XI.4 PROGRAMUL I

În acest program este vorba despre utilizarea unei funcţii care implementează algoritmul de căutare binară (binary search). Este vorba de un algoritm foarte simplu care caută o valoare într-un container în care sunt memorate o mulţime de valori de acelaşi tip. Containerul în acest caz trebuie să fie ordonat crescător conform cu o relaţie de ordonare definită pe tipul de date al obiectelor conţinute în acesta. Utilizând ordonarea containerului algoritmul de căutare binară realizează o căutare extrem de rapidă.

Se spune ca pentru rezolvarea problemei căutării unui leu în deşert un programator cere un leu pe care să-l aibă ca referinţă şi se apucă să parcurgă sistemetic deşertul, de la nord la sund, pe fâşii de vest la est, comparând fiecare obiect întâlnit cu leul de referinţă. Dacă găseşte un obiect care seamănă cu leul asta este, dacă nu trece mai departe. Pentru ca programul să se terminte cu bine se asigură şi fixează un leu la capătul deşertului. Pentru aceeaşi problemă un matematician cântăreşte din ochi şi împarte deşertul în două. Într-o parte va fi leul în cealaltă nu. Ia partea în care se află leul şi o mai împarte o dată şi tot aşa până când pune mâna pe leu. Strategia adoptată de algoritmul de căutare binară este strategia matematicianului.

Programul conţine o serie de construcţii care vă sunt de acum familiare. Spre exemplu tratarea excepţiilor. Dacă este cineva căruia îi scapă subiectul atunci să nu mai stea pe gânduri şi să se apuce de învăţat.

Prototipul funcţiei este:

binary_search.hpp

```
#ifndef _BINARY_SEARCH_
#define _BINARY_SEARCH_

const int* binary_search (const int* , int, int);
```

```
#endif
```

Pentru tipul int funcţia de sortare este:

BinarySearchInt.cpp

```
//Header inclus numai pentru macrodefinitai lui NULL.
#include <stddef.h>

//Algoritmul nu-si propune modificarea valorilor din container.
//Nici a elementului a carui adresa este returnata ...
const int* binary_search (const int* array, int n, int x)
{
   //Nici a vreunui alt element ...
   //Modificarea valorii vreunui pointer este alceva ...
   const int* lo = array, *hi = array + n, *mid;
   while(lo != hi)
   {
      mid = lo + (hi - lo) / 2;
      if(x == *mid)
            return mid;
      if(x < *mid)
            hi = mid;
      else
            lo = mid + 1;
   }
   return NULL;
}
```

Este interesant de analizat contextul în care funcţionează funcţia care implementează algoritmul. Deocamdată containerul pe care lucrează funcţia este un vector de elemente întregi. Operaţile cu obiecte din container, doar comparaţii, se bazează pe existenţa operatorilor == şi < . Nu mai subliniem algebra pe spaţiul pointerilor.

Programul care apelează funcţia trebuie să-i creeze acesteia contextul de funcţionare. Fişierul în care se află elementele acestui program nu diferă foarte mult de la un exemplu la altul:

bin.cpp

```
#include <iostream.h>
#include "binary_search.hpp"

//Valori returnate de functia main.
enum Err
{
   Success,
   Error
};

//Clasa care este utilizata pentru tratarea exceptiilor
//care apar la alocarea dinamica de memorie si care pot fi
//rezolvate pe loc in urma unui compromis.
```

```
class AllocError {};
//Clasa care este utilizata pentru tratarea exceptiilor fatale la alocarea memoriei.
class AllocFatalError {};
//Clasa care este utilizata pentru tratarea erorilor de citire de la tastatura.
class ReadError
{
public:
        int *PtrInt;
        ReadError(void) : PtrInt(NULL) {}
        ReadError(int *P) : PtrInt(P) {}
};

//Alocarea memoriei se face numai cu aceasta functie (Alloc).
int * Alloc(int N)
{
   int *PtrInt;
   if(!N)
      throw AllocError(); //Eroare fatala ...
   PtrInt = new int[N];
   if(!PtrInt)
      throw AllocError();
   return PtrInt;
}

//Eliberarea memoriei se face numai cu aceasta functie (Free).
//Parametrul functiei are tipul: referinta la pointer la intreg.
void Free(int * &PtrInt)
{
   //Memoria a fost deja eliberata.
   if(!PtrInt)
      return;
   //Daca memoria nu a fost eliberata o eliberam acum.
   delete []PtrInt;
   //Nu uitam sa "marcam" pointerul respectiv
   //cu valoarea NULL in caz ca functia este
   //apelata din nou cu acelasi pointer.
   //Acesta este motivul pentru care am transferat
   //referinta la pointer si nu direct valoarea pointerului.
   PtrInt = NULL;
}


//Functie care incearca sa construiasca problema.
void DoEverithing(void)
{

   int NrElem;
   cout << "Numarul de elemente din vector : ";
   cin >> NrElem;
   if(!cin)
      //Inca nu avem nici un vector.
      throw ReadError();
   //Daca s-a citit corect NrElem ne asiguram ca valoarea citita este pozitiva.
```

```
   if(NrElem < 1)
      throw ReadError();

   //Aici apare si vectorul.
   int *VectorInt;

   //Se ilustreaza o strategie (nu cea mai buna)
   //de rezolvare a unei erori
   //fara sa se paraseasca programul ...
TryAgain: //Nu va speriati de eticheta de salt neconditionat !!!
   try //Incearca
   {
      VectorInt = Alloc(NrElem);
   }
   catch(AllocError) //In caz de eroare ...
   {
      if(NrElem < 1) //si daca N este suficient de mare ...
      {
         //Eroarea este FATALA: nu se poate
         //aloca memorie dinamica nici macar
         //pentru UN intrag!!!
         throw AllocFatalError();
      }
      cerr << "Nu sa putut aloca memorie pentru un vector de dimensiune " << NrElem;
      NrElem--; //Decrementeaza NrElem
      cerr << ". Se incearca alocarea pentru un vector de dimensiune " << NrElem << '!' << endl;
      //Nu va speriati nici de instructiune de salt neconditionat.
      goto TryAgain; //Mai incearca o data ...
   }
   //In acest moment vectorul este pregatit.
   //In alte circumstante valorile memorate in vector
   //ar rezulta dintr-un proces mai interesant.
   //Pentru utilizare functiei binary_search vectorul
   //trebuie sa fie si sortat in ordine crescatoare.
   //Pentru a nu complica prea mult lucrurile vom
   //incarca vectorul artificial:
   int *PtrVector(VectorInt);
   for(int i = 0; i < NrElem; i++)
      *PtrVector++ = i;

   int Val; //Valoarea elementului care este cautat.
   cout << "Valoarea elementului care trebuie cautat : " << endl;
   cin >> Val;
   if(!cin)
      throw ReadError(VectorInt);

   //In acest moment se poate incerca utilizarea functiei binary_search.
   const int *Result; //const -> vezi prototipul functiei binary_search ...
   Result = binary_search(VectorInt, NrElem, Val);
   if(!Result)
      cout << "Valoarea " << Val << " nu se afla in vector." << endl;
   else
      //conversia la int * este posibila deoarece operatorul << are ca parametru int const *.
```

```
      cout << "Valoarea " << Val << " este in vector la adresa " << (int *)Result << endl;
   Free(VectorInt);
}
int main(void)
{
   try
   {
      DoEverithing();
   }
   catch(AllocFatalError)
   {
      cerr << "Masina are ceva foarte grav ..." << endl;
      //Nu este nici un vector de eliberat.
      return Error;
   }
   //Informatia despre eroare este codificata in Error ...
   catch(ReadError Caught)
   {
      //Nu mai insistam, desi am fi putut ...
      cerr << "Eroare la citirea unei valori de la tastatura !" << endl;
      //Nu uitam sa eliberam memoria care a fost eventual alocata.
      //Informatia despre eroare este chiar valoarea pointerului.
      Free(Caught.PtrInt);
      return Error;
   }
   return Success;
}
```

XI.5 PROGRAMUL II

Să modificăm funcţia binary_search astfel încât aceasta să funcţioneze pe un container de tip vector de elemnte de orice tip pentru care sunt definiţi operatorii == şi <. De această dată funcţia template va fi plasată într-un header:

binary_search.hpp

```
#ifndef _BINARY_SEARCH_
#define _BINARY_SEARCH_

template<class T>
const T* binary_search (const T* array, int n, const T& x)
{
   const T* lo = array, *hi = array + n, *mid;
   while(lo != hi)
   {
      mid = lo + (hi - lo) / 2;
      if(x == *mid)
            return mid;
      if(x < *mid)
```

```
            hi = mid;
        else
            lo = mid + 1;
    }
    return NULL;
}

#endif
```

De această dată programul care apelează funcţia template este identic cu cel care apela varianta clasică a funcţiei, deşi este plasat în fişierul cu numele bin1.cpp

XI.6 PROGRAMUL III

Următorul pas în abstractizare funcţiei binary_search este modificarea valorii obiectului returnat de aceasta în caz de eşec, de la valoarea NULL la o valoare care este adresa elementului care urmează după ultimul element al vectorului[7]. Este clar că nu există dubii odată ce valoarea returnată nu este o adresă din container (vector).

binary_search.hpp

```
#ifndef _BINARY_SEARCH_
#define _BINARY_SEARCH_

template<class T>
const T* binary_search (const T* array, int n, const T& x)
{
    const T* lo = array, *hi = array + n, *mid;
    while(lo != hi)
    {
        mid = lo + (hi - lo) / 2;
        if(x == *mid)
            return mid;
        if(x < *mid)
            hi = mid;
        else
            lo = mid + 1;
    }
    return array + n;
}

#endif
```

[7] Past the end

Modificarea în programul care apelează funcţia este la nivelul testului valorii returnate de aceasta (este reprodusă numai această modificare căreia sunteţi invitaţi să-i găsiţi locul în bin.cpp):

bin2.cpp

```
//In acest moment se poate incerca utilizarea functiei binary_search.
const int *Result; //const -> vezi prototipul functiei binary_search ...
Result = binary_search(VectorInt, NrElem, Val);
//De aceasta data nu se returneaza NULL ci adresa elementului
//de dupa sfarsitul vectorului.
if(Result == VectorInt + NrElem)
   cout << "Valoarea " << Val << " nu se afla in vector." << endl;
else
   //conversia la int * este posibila deoarece operatorul << are ca parametru int const *.
   cout << "Valoarea " << Val << " este in vector la adresa " << (int *)Result << endl;
```

XI.7 PROGRAMUL IV

Este posibilă abstractizarea în continuare a funcţiei binary_search (mai mult decât poate aţi crede). Deocamdată să considerăm modul de transmitere a informaţiei despre domeniul în care urmează să fie căutată valoarea. De această dată se va renunţa la transmiterea pointerului de început şi a dimensiunii vectorului în favoarea transmiterii pointerului de început al domeniului şi a celui care urmează după ultimul element de la sfârşitul domeniului[8]. Această soluţie este ceva mai flexibilă şi devine de importanţă majoră în cazul în care abstractizarea merge mai departe, şi algoritmul este făcut să funcţioneze pe orice container (vectorul este un caz cu totul particular de container). În acest caz pointerul devine un caz particular de iterator. (Aceste probleme sunt păstrate pentru viitorul laborator.) De această dată funcţia returnează în caz de eşec valoarea pointer-ului past the last.

binary_search.hpp

```
#ifndef _BINARY_SEARCH_
#define _BINARY_SEARCH_

template<class T>
const T* binary_search(T* first, T* last, const T& value)
{
   const T* lo = first, *hi = last, *mid;
   while(lo != hi)
   {
      mid = lo + (hi - lo) / 2;
      if(value == *mid)
            return mid;
      if(value < *mid)
```

[8] Past the last

```
            hi = mid;
        else
            lo = mid + 1;
    }
    return last;
}

#endif
```

Şi în aceast caz programul care apelează funcţia binary_search este puţin diferit de prima variantă. În continuare este prezentată numai partea care diferă:

bin3.cpp

```
    //De aceasta data functia binary search are ca parametri
    //doi pointeri care marcheaza inceputul si urmatorul
    //element de dupa sfarsitul zonei din vector unde se face cautarea.
    int *begin, *end;
    int Depl;
    cout << "In domeniul : [0, " << NrElem -1 << ")" << endl;
    cout << "Primul element al domeniului de cautare : ";
    cin >> Depl;
    if(!cin)
        throw ReadError(VectorInt);
    //Sa ne asiguram ca stim ce facem. Altfel treburile scapa de sub control.
    if(Depl < 0 || Depl >= NrElem - 1)
        throw ReadError(VectorInt);
    begin = VectorInt + Depl;

    cout << "Ultimul element al domeniului de cautare : ";
    cin >> Depl;
    if(!cin)
        throw ReadError(VectorInt);
    //Sa ne asiguram ca stim ce facem. Altfel treburile scapa de sub control.
    if(Depl < 0 || Depl >= NrElem - 1)
        throw ReadError(VectorInt);
    end = VectorInt + Depl;
    //Sa vedem daca nu cumva intervalul specificat este convex.
    //Altfel lucrurile scapa de asemenea de sub control.
    if(begin >= end)
        throw ReadError(VectorInt);

    int Val; //Valoarea elementului care este cautat.
    cout << "Valoarea elementului care trebuie cautat : " << endl;
    cin >> Val;
    if(!cin)
        throw ReadError(VectorInt);

    //In acest moment se poate incerca utilizarea functiei binary_search.
    const int *Result; //const -> vezi prototipul functiei binary_search ...
    Result = binary_search(begin, end, Val);
    if(Result == end)
```

```
    cout << "Valoarea " << Val << " nu se afla in vector." << endl;
  else
    //conversia la int * este posibila deoarece operatorul << are ca parametru int const *.
    cout << "Valoarea " << Val << " este in domeniul de cautare la adresa " << (int *)Result << endl;
```

XI.8 TEMĂ

Răspundeţi în scris la întrebarea:

1. Care sunt consecinţele majore ale faptului că STL este inclusă în draft-ul de standard ANSI/ISO C++?

Analizaţi în scris programele furnizate şi comparaţi rezultatele pe care le-aţi notat cu rezultatele obţinute pe calculator, în laborator.

Modificaţi programul care apelează funcţia template din ultimul exemplu astfel încât să funcţioneze pentru tipul float, apoi pentru o clasă pe care o definiţi cu acest scop.

Scrieţi un template de funcţie care să realizeze sortarea unui container (de tip vector).