I. Template-uri pentru funcții

Un *template de funcție* reprezintă descrierea unei familii de funcții; template-ul arată ca o funcție uzuală, cu excepția faptului că anumite elemente sunt parametrizate (nedeterminate). De exemplu, o familie de funcții care returnează maximul dintre două valori recepționate drept parametri se descrie astfel:

```
template <typename T> inline const T& max (const T& a,  const T& b)
{
	return a<b?b:a;
}
```

În exemplul anterior, T, introdus prin cuvântul cheie *typename*, este *parametru de template*; a şi b sunt *parametri de apel* şi au tipul const& T. T este un tip arbitrar, ce trebuie specificat în momentul apelului template-ului de funcție, cu restricția că el trebuie să ofere acele operații ce sunt utilizate în template (în exemplul anterior, pentru T trebuie să existe o definiție a operatorului <, altfel se va genera o eroare de compilare).

Obs: 1. Cuvântul cheie *class* poate fi utilizat în locul cuvântului *typename*.
2. Cuvântul cheie *struct* nu poate fi utilizat în locul cuvântului *typename*!

Template-ul de funcție max() se utilizează astfel:

```
int main()
{
	int i = 42;
	std::cout << "max(7,i): " << ::max(7,i) << std::endl;

	double f1 = 3.4;
	double f2 = -6.7;
	std::cout << "max(f1,f2): " << ::max(f1,f2) << std::endl;

	std::string s1 = "mathematics";
	std::string s2 = "math";
	std::cout << "max(s1,s2): " << ::max(s1,s2) << std::endl;
}
```

Obs: Deoarece în librăria standard C++ există definit template-ul *std::max()*, apelurile lui max() au fost prefixate cu operatorul de rezoluție *::*, pentru a specifica spațiul de nume global.

Template-urile NU sunt compilate în nişte entități capabile să gestioneze orice tip (aka *generics*). Pentru fiecare tip T cu care template-ul este folosit, se generează din template câte o entitate distinctă, numită *instanță*; procesul este numit *instanțiere*. (In exemplul anterior, max() a fost instanțiat de 3 ori.) Astfel, un template este compilat în 2 faze:

- fără instanțiere, codul template-ului este verificat d.p.d.v. sintactic;
- la instanțiere, se verifică validitatea tuturor apelurilor din codul template-ului.

Obs: Atunci când un template este instanțiat, compilatorul trebuie să aibă acces la definiția sa (o simplă declarație nu este suficientă pentru a efectua a doua fază); din acest motiv, definiția unui template se scrie într-un fişier header!

Instanțierea presupune *deducerea parametrilor de template* (T) din tipurile parametrilor de apel; conversiile automate NU sunt permise în acest proces. Astfel:

```
::max(4,7);                          // OK,  T este int pentru ambele argumente de funcție
::max(4,4.2);                        // eroare: int sau double?
::max(static_cast<double>(4),4.2);  // OK
::max<double>(4,4.2);                // OK, T este precizat explicit ca fiind double
```

Numărul parametrilor de template nu este limitat; astfel, max() ar putea recepționa parametri de apel de tipuri diferite:

```
template <typename T1, typename T2> inline T1 max (const T1& a, const  T2& b)
{
        return a < b ? b : a;
}
…
::max(42, 66.66)                     // OK, însă tipul primului argument definește tipul rezultatului!
```

Deoarece tipul valorii returnate trebuie precizat explicit şi deoarece în C++ nu există nici o cale de a preciza “cel mai puternic tip”, maximul dintr 42 şi 66.66 ar putea avea ca rezultat fie pe raționalul 66.66, fie pe întregul 66 (în funcție de ordinea argumentelor în apel).

Obs: Dacă rezultatul evaluării expresiei a<b este b, acesta va fi convertit către T1 (tipul rezultatului) în mod implicit, creând un obiect local, ceea ce înseamnă că nu se poate returna prin referință!

Putem să introducem un al treilea parametru de template pentru a preciza tipul valorii returnate:

```
template <typename T1, typename T2, typename RT> inline RT max (const T1& a, const T2& b);
```

Însă, asemeni mecanismului de rezoluție în cazul supraîncărcării, mecanismul deducerii parametrilor de template nu ia în considerare tipurile returnate; deoarece RT nu apare în tipurile parametrilor de apel, el nu poate fi dedus. Astfel, lista parametrilor de template trebuie precizată explicit:

```
max<int,double,double>(4,4.2)    // OK, însă incomod
```

Compilatorul ar putea deduce tipurile lui T1 şi T2, dar nu poate face acest lucru din cauză că RT este ultimul în listă. Schimbând ordine de declarare a parametrilor de template, putem să-l precizăm explicit doar pe RT, permitându-i compilatorului să-i deducă pe T1 şi T2:

```
template <typename RT, typename T1, typename T2> inline RT max (const T1& a, const T2& b);
max<double>(4,4.2)          // OK, RT este explicit double, T1 este dedus int, T2 este dedus int
```

La fel precum funcțiile uzuale, template-urile de funcții pot fi supraîncărcate, putând exista astfel diverse versiuni cu același nume. Mai mult, pot exista și funcții uzuale cu același nume, așa cum se observă în următorul exemplu:

```
// maximul dintre 2 valori de tip int
inline const int& max (const int& a, const  int& b)
{
        return a<b?b:a;
}

// maximul dintre 2 valori arbitrare
template <typename T> inline const  T& max (const  T& a, const T& b)
{
        return a<b?b:a;
}

// maximul dintre 3 valori arbitrare
template <typename T> inline const  T& max (const  T& a, const  T& b, const T& c)
{
        return max (max(a,b), c);
}

int main()
{
        ::max(7, 42, 68);          // apelează template-ul cu 3 argumente
        ::max(7.0, 42.0);          // apelează max<double>, prin deducerea argumentului
        ::max('a', 'b');           // apelează max<char>, prin deducerea argumentului
        ::max(7, 42);              // apelează versiunea ne-template pentru 2 de int
        ::max<>(7, 42);            // apelează max<int>, prin deducerea argumentului
        ::max<double>(7, 42);      // apelează max<double> (fără deducerea argumentului)
        ::max('a', 42.7);          // apelează versiunea ne-template pentru 2 de int (conversiile automate
                                   //                  sunt permise pentru funcțiile uzuale!)
}
```

Obs: În cazul în care versiunile template si ne-template sunt “la fel de bune”, este aleasă versiunea ne-template! Aceasta cu excepția cazului în care versiunea ne-template nu este vizibilă:

```
// maximul dintre 2 valori arbitrare
template <typename T> inline const T& max (const T& a, const T& b)
{
        return a<b?b:a;
}

// maximul dintre 3 valori arbitrare
template <typename T> inline const T& max (const T& a, const T& b, const T& c)
{
        // functia ne-template care calculeaza maximul dintre 2 valori de tip int nu este inca vizibila!
        return max (max(a,b), c);   // utilizeaza versiunea template (chiar si pentru int )
}

// maximul dintre 2 valori de tip int
inline const int& max (const int& a, const int& b)
{
        return a<b?b:a;
}
```

II. Template-uri pentru clase

În mod similar funcțiilor, clasele pot fi parametrizate cu unul sau mai multe tipuri, tipuri ce pot fi utilizate pentru a declara atribute şi metode. De exemplu:

```
// stack.h
#pragma once
template <typename T> class Stack
{
      int size;         // numarul "locurilor" in stiva
      int top;          // cate "locuri" sunt ocupate
      T* stackPtr;      // unde tinem elementele
public:
      Stack(int = 10) ;
      ~Stack() { delete [] stackPtr; }

      bool push(const T&);
      bool pop(T&);
      int isEmpty()const { return top == -1; }
      int isFull() const { return top == size - 1; }
} ;

template <typename T> Stack<T>::Stack(int s)
{
      size = (s > 0) && (s < 1000) ? s : 10;
      stackPtr = new T[size];
      top = -1 ;  // nici un loc ocupat
}

template <typename T> bool Stack<T>::push(const T& item)
{
      if ( !isFull() )
      {
            stackPtr[++top] = item;
            return true;      // succes
      }
      return false;  // nu mai sunt locuri...
}


template <typename T> bool Stack<T>::pop(T& popValue)
{
      if (!isEmpty())
      {
            popValue = stackPtr[top--] ;
            return true;  // succes
      }
      return false;  // stiva nu contine date....
}

// principal.cpp
#include <iostream>
#include "stack.h"
using namespace std;

void main()
{
      typedef Stack<float> FloatStack;
      typedef Stack<int> IntStack;

      FloatStack fs(5);
      float f = 1.1f;
```

```
	cout << "Pushing elements onto fs" << endl;
	while (fs.push(f))
	{
		cout << f << ' ' ;
		f += 1.1f ;
	}
	cout << endl << "Stack Full!" << endl;

	cout << endl << "Popping elements from fs" << endl;

	while (fs.pop(f))
		cout << f << ' ' ;
	cout << endl << "Stack Empty!" << endl;
	cout << endl;

	IntStack is;
	int i = 1.1;
	cout << "Pushing elements onto is" << endl;
	while (is.push(i))
	{
		cout << i << ' ' ;
		i += 1;
	}
	cout << endl << "Stack Full!" << endl;

	cout << endl << "Popping elements from is" << endl;
	while (is.pop(i))
			cout << i << ' ' ;
	cout << endl << "Stack Empty!" << endl ;
}
```

Obs:

- Numele clasei este Stack; tipul clasei este Stack<T>.
- Pentru template-urile de clasă, metodele sunt instanțiate doar dacă sunt utilizate!
- Atributele statice sunt instanțiate o singură dată, pentru fiecare valoare a parametrului de template.
- Se recomandă definirea de alias-uri cu typedef.

Un template de clasă poate fi *specializat* pentru anumite valori ale parametrilor template-ului. Aceasta permite corectarea unor comportamente improprii în cazul anumitor tipuri sau permite propunerea unor implementări alternative, care (eventual) să optimizeze o implementare general valabilă. De exemplu:

```
// stack.h
template<> class Stack<std::string>
{
	std::deque<std::string> elements;
public:
    bool push(std::string const&);
    bool pop( std::string& );
    int isEmpty() const {return elements.empty(); }
    int isFull() const { return 0; }
};

bool Stack<std::string>::push (std::string const& elem)
{
	elements.push_back(elem);
	return true;
```

```
}

bool Stack<std::string>::pop ( std::string& popValue)
{
	if (elements.empty())
		return false;

	popValue = elements[elements.size()-1];

	elements.pop_back();
	return true;
}

// principal.cpp
std::string messages[] = {"start", "stop", "save", "edit" };

typedef Stack<string> StringStack;
StringStack ss;

cout << "Pushing elements onto ss" << endl;
i = 0;
while ( (i < 4) && ss.push( messages[i] ) )
{
	cout << messages[i] << ' ' ;
	++i;
}
cout << endl;

cout << endl << "Popping elements from ss" << endl;
string m;
while (ss.pop(m))
{
	cout << m << ' ' ;
}
cout << endl << "Stack Empty!" << endl ;
```