Clase template

Funcții şi clase template

Clasele şi funcțiile template sunt şabloane pe baza cărora se pot genera clase şi funcții folosind un mecanism de expandare asemănător cu macrodefinițiile cu parametri. Spre deosebire de macrodefiniții, template-urile oferă mecanisme de înlocuire doar pentru tipuri de date şi constante, însă au avantajul unei verificări mai bune de către compilator.

Sintaxa utilizată pentru definirea şabloanelor de funcții este:

template <class T_1, class T_2, …, class T_n, tip_1 c_1, …, tip_m c_m> prototip_functie

Exemplu de construire şi utilizre a unei funcții template pentru calculul sumei elementelor unui vector:

```
#include <iostream>
using namespace std;

// sablon cu parametri pentru tipul
// elementelor (T) si dimensiunea vectorului
// (constanta n) pentru calculul sumei
// elemenetelor unui vector
template <class T, int n>
T suma(T vector[])
{
	T suma = 0;

	for (int i = 0; i < n; i++)
		suma = suma + vector[i];

	return suma;
}

void main()
{
	// declaram si initializam doi vectori de
	// tipuri si dimensiuni diferite
	int v1[] = {1, 5, 7, 9};
	double v2[] = {1.3, 5.7, 7.9, 9.11, 11.9};

	// folosim functia template suma pentru a genera
	// functii de calcul suma pentru cei doi vectori
	cout << "Suma 1 este: " << suma<int, 4>(v1) << endl;
	cout << "Suma 2 este: " << suma<double, 5>(v2) << endl;
}
```

Funcția poate fi folosită pentru orice tip de date care suportă operațiile utilizate în cadrul funcției: inițializare cu 0 şi operatorul +. De exemplu, putem construi o clasă Complex şi apela funcția pentru clasa construită de noi:

```
#include <iostream>
using namespace std;


class Complex
{
public:
	// constructor cu valori implicite care
	// suporta o initializare de forma
	// Complex c = 0;
	Complex(double r = 0, double i = 0)
		: real(r), imaginar(i) {}

	// operatorul + pentru adunarea a doua numere complexe
```

```
        const Complex operator + (const Complex c)
        {
                real += c.real;
                imaginar += c.imaginar;

                return (*this);
        }

        double real, imaginar;
};

// operator pentru afisare
ostream& operator << (ostream& out, const Complex& c)
{
        out << "(" << c.real << "," << c.imaginar << ")";
        return out;
}

// sablon cu parametri pentru tipul
// elementelor (T) si dimensiunea vectorului
// (constanta n) pentru calculul sumei
// elemenetelor unui vector
template <class T, int n>
T suma(T vector[])
{
        T suma = 0;

        for (int i = 0; i < n; i++)
                suma = suma + vector[i];

        return suma;
}

void main()
{
        // declaram si initializam doi vectori de
        // tipuri si dimensiuni diferite
        int v1[] = {1, 5, 7, 9};
        double v2[] = {1.3, 5.7, 7.9, 9.11, 11.9};

        // folosim functia template suma pentru a genera
        // functii de calcul suma pentru cei doi vectori
        cout << "Suma 1 este: " << suma<int, 4>(v1) << endl;
        cout << "Suma 2 este: " << suma<double, 5>(v2) << endl;

        Complex v3[] = { Complex(3.2, 4.2), Complex(1.1, 2.2),
                         Complex(5, 1) };
        cout << "Suma 3 este: " << suma<Complex,3>(v3) << endl;
}
```

Mecanismul se poate folosi asemănător şi pentru construirea şabloanelor de clase. Sintaxa este:

template <class T_1, class T_2, …, class T_n, tip_1 c_1, …, tip_m c_m> class nume_clasă {…}

Metodele implementate inline în cadrul clasei folosesc sintaxa obişnuită. În cazul în care se doreşte implementarea metodelor în afara clasei se va folosi sintaxa:

template <class T_1, class T_2, …, class T_n, tip_1 c_1, …, tip_m c_m>
tip_returnat nume_clasă< T_1, T_2, …, T_n, c_1, …, c_m >:: nume_functie(param){…}

Pentru exemplificare vom construi o clasă vector care să poată stoca orice tip de date:

```
#include <iostream>
using namespace std;

// clasa template Vector cu parametrii:
//    T - tipul de date stocat
//    n - constanta (numarul de elemente)
template <class T, int n>
class Vector
{
public:
```

```
        // constructor
        Vector()
        {
                // alocam spatiu pentru vector
                vector = new T[n];
        }

        // functie inline pentru aflarea dimensiunii
        int GetDim() { return n; }

        // operator pentru accesarea elementelor
        // definit in afara clasei
        T& operator[] (int i);

        // destructor pentru dealocarea memoriei
        ~Vector()
        {
                delete [] vector;
        }
private:

        // clasa nu va permite atribuirea
        // si constructia prin copiere
        Vector operator=(Vector&);
        Vector(const Vector&);

        // elementele vectorului (de tip T)
        T *vector;
};


// definirea in exterior a functiei operator[]
template<class T, int n>
T& Vector<T,n>::operator[] (int i)
{
        return vector[i];
}

// numarul maxim de caractere pentru un nume
const int DIM_MAX_NUME = 200;

// clasa persoana pentru testarea vectorului
class Persoana
{
public:

        // constructor
        Persoana(int cod = 0, char* nume = "Anonim")
        {
                // copiem datele primite ca parametri
                _cod = cod;
                strcpy(_nume, nume);
        }

        // functii de acces
        int GetCod()          const { return _cod;  }
        const char* GetNume() const { return _nume; }

private:
        // date membre
        int _cod;
        char _nume[DIM_MAX_NUME];
};

// operator pentru afisarea unei persoane
ostream& operator << (ostream& out, const Persoana& persoana)
{
        out << persoana.GetCod() << " " << persoana.GetNume();
        return out;
}

void main()
{
        int i;

        // exemplu de utilizare pentru un vector de intregi
        Vector<int, 3> v1;     // declarare vector
```

```
	// initializare elemente
	v1[0] = 4; v1[1] = 47; v1[2] = 2;

	// afisare vector
	for (i = 0; i < v1.GetDim(); i++)
		cout << v1[i] << endl;

	// exemplu de utilizare pentru un vector de persoane
	Vector<Persoana, 2> v2;		// declarare vector

	// initializare elemente
	v2[0] = Persoana(1, "Popescu Ion");
	v2[1] = Persoana(2, "Ionescu Maria");

	// afisare vector
	for (i = 0; i < v2.GetDim(); i++)
		cout << v2[i] << endl;
}
```

Se observă faptul că putem utiliza clasa template *Vector* pentru a stoca orice tip de date (predefinit sau definit de utilizator) care suportă operațiile utilizate în clasă (în exemplul prezentat doar constructorul implicit pentru: `vector = new T[n];`).

Specializări

În exemplele prezentate şabloanele sunt expandate după acelaşi model indiferent de tipul de date utilizat. Limbajul oferă posibilitatea de a preciza modele de expandare specifice pentru anumite tipuri de date. Acest mecanism se numeşte specializare.

Pentru exemplificare vom considera următoarea funcție template pentru sortarea unui vector:

```
template <class T>
void Sortare(T vector[], int n)
{
	for (int i = 0; i < n; i++)
		for (int j = i + 1; j < n; j++)
			if (vector[i] > vector[j])
			{
				// interschimbare elemente
				T temp = vector[i];
				vector[i] = vector[j];
				vector[j] = temp;
			}
}
```

Funcția astfel definită va funcționa corect pentru tipurile de date care suportă compararea folosind operatorul > şi atribuirea. Dacă dorim să folosim funcția pentru sortarea unui vector de şiruri de caractere vom obține un rezultat eronat deoarece tipul *char** nu suportă operațiile necesare. În această situație putem crea o specializare a funcției pentru lucrul cu şiruri de caractere. Specializările pentru funcții se face prin eliminarea tipului specializat din listă si utilizarea explicită a acestuia în prototipul funcției:

```
template <>
void Sortare<char*>(char* vector[], int n)
{
	for (int i = 0; i < n; i++)
		for (int j = i + 1; j < n; j++)
			// utilizam functia de comparare pentru stringuri
			if (strcmp(vector[i],vector[j]) > 0)
			{
				// interschimbare elemente
				char* temp = new char[strlen(vector[i]) + 1];
				strcpy(temp, vector[i]);
```

```
                                delete [] vector[i];
                                vector[i] = new char [strlen(vector[j]) + 1];
                                strcpy(vector[i], vector[j]);

                                delete [] vector[j];
                                vector[j] = new char [strlen(temp) + 1];
                                strcpy(vector[j], temp);
                        }
}
```

Exemplu de utilizare a funcției şi a specializării:

```
void main()
{
        // construim si sortam un vector de intregi
        int v1[] = {5, 3, 53, 2};
        Sortare(v1, 4);

        // construim vectorul de siruri
        char * v2[3];
        v2[0] = new char[8];
        strcpy(v2[0], "popescu");

        v2[1] = new char[8];
        strcpy(v2[1], "ionescu");

        v2[2] = new char[9];
        strcpy(v2[2], "adamescu");

        // si il sortam
        Sortare(v2, 3);
}
```

Se observă faptul că este posibilă omiterea parametrilor şablonului la apelul funcțiilor template atunci când compilatorul poate determina fără echivoc tipul pentru care trebuie făcută expandarea.

Specializarea se poate similar aplica şi la clase template. În cazul claselor este posibilă specializarea doar anumitor metode sau a întregii clase.

Aplicație

Să se construiască clasa template *Vector* pentru stocarea unui număr fix de elemente în alocate dinamic. Să se construiască o specializare pentru tipul *bool* care să folosească pentru stocare un vector de întregi fără semn, iar fiecare element va fi stocat pe un bit.

Rezolvare:

```
#include<iostream>
#include<cmath>
using namespace std;

// Clasa template Vector
template<class T>
class Vector
{
public:

        // constructor
        Vector(int n) : n(n)
        {
                // alocam memorie pentru vector
                vector = new T[n];
        }
```

```
        // constructor de copiere
        Vector(const Vector& v) : n(v.n)
        {
                // alocam memorie pentru vector
                vector = new T[n];

                // copiem elementele
                for (int i = 0; i < n; i++)
                        vector[i] = v.vector[i];
        }

        // operatorul de atribuire
        const Vector& operator=(const Vector&)
        {
                // dealocam memoria alocata
                delete [] vector;

                // alocam memorie pentru noul vector
                vector = new T[n];

                // copiem elementele
                for (int i = 0; i < n; i++)
                        vector[i] = v.vector[i];
        }

        // destructorul
        ~Vector()
        {
                // dealocam memoria alocata
                delete [] vector;
        }

        // operatorul pentru citirea elementelor
        T operator[] (int i)
        {
                return vector[i];
        }

        // seteaza valoarea unui element din vector
        void Setare(int i, T valoare)
        {
                vector[i] = valoare;
        }

        // functie pentru obtinerea numarului de elemente
        int Lungime() { return n; }

private:
        int n;                  // numarul de elemente
        T *vector;              // vectorul de elemente
};

// specializarea clasei template Vector
// pentru stocarea eficienta a vectorilor
// de valori booleene
template<>                              // marcajul de clasa template
class Vector<bool>              // indicarea specializarii
{
public:

        // constructor
        Vector(int n) : n(n)
        {
                // alocam memorie pentru vector
                vector = new unsigned int[NumarIntregi(n)];
        }

        // constructor de copiere
        Vector(const Vector& v) : n(v.n)
        {
                int k = NumarIntregi(n);

                // alocam memorie pentru vector
                vector = new unsigned int[k];

                // copiem elementele
                for (int i = 0; i < k; i++)
```

```
                vector[i] = v.vector[i];
}

// operatorul de atribuire
const Vector& operator=(const Vector&v)
{
        // dealocam memoria alocata
        delete [] vector;

        int k = NumarIntregi(n);

        // alocam memorie pentru noul vector
        vector = new unsigned int[k];

        // copiem elementele
        for (int i = 0; i < k; i++)
                vector[i] = v.vector[i];
}

// destructorul
~Vector()
{
        // dealocam memoria alocata
        delete [] vector;
}

// operatorul pentru citirea elementelor
bool operator[] (int i)
{
        int dimElement = sizeof(int)*8;

        // obtinem elementul care contine bitul cautat
        unsigned int element = vector[i / dimElement];

        // calculam masca elementului
        unsigned int masca = 1 << (i % dimElement);

        // obtinem bitul cautat din element
        return (bool)(element & masca);
}

// seteaza valoarea unui element din vector
void Setare(int i, bool valoare)
{
        int dimElement = sizeof(int)*8;

        // obtinem elementul care contine bitul cautat
        unsigned int& element = vector[i / dimElement];

        if (valoare)
        {
                // trebuie sa setam bitul pe 1
                unsigned int masca = 1;

                // mutam bitul pe pozitia corespunzatoare
                masca = masca << (i % dimElement);

                // si setam bitul in vector
                element = element | masca;
        }
        else
        {
                // trebuie sa setam bitul pe 0
                unsigned int masca = 1;

                // mutam bitul pe pozitia corespunzatoare
                masca = masca << (i % dimElement);

                // inversam bitii din masca
                masca = masca ^ 0xFFFF;       // pp 32 biti

                // si resetam bitul in vector
                element = element & masca;
        }
}

// functie pentru obtinerea numarului de elemente
```

```
        int Lungime() { return n; }

private:
        int n;                // numarul de elemente

        // elementele sunt stocate intr-un vector
        // de intregi (cate 32 pe fiecare pozitie)
        unsigned int *vector;

        // calculeaza numarul de intregi necesari
        // pentru stocarea a k elemente booleene
        int NumarIntregi(int k)
        {
                return (int) ceil((double)k / (sizeof(int) * 8));
        }
};

void main()
{
        int i;

        Vector<int> v1(3);            // declarare vector de intregi

        v1.Setare(0, 7);              // setare elemente
        v1.Setare(1, 1);
        v1.Setare(2, 92);

        // afisare elemente vector
        for (i = 0; i < v1.Lungime(); i++)
                cout << v1[i] << endl;

        Vector<bool> v2(4);           // declarare vector de bool (foloseste
specializarea)

        v2.Setare(0, false);   // setare elemente
        v2.Setare(1, true);
        v2.Setare(2, true);
        v2.Setare(3, false);

        // afisare elemente vector
        for (i = 0; i < v2.Lungime(); i++)
                cout << boolalpha << v2[i] << endl;
}
```