

Seminar 5

1 Funcții template

Atunci când avem nevoie de o funcție pentru mai multe tipuri de date, polimorfismul nu este întotdeauna bun (implementarea de mai multe ori ale aceleiași funcții pentru diverse tipuri de parametrii nu este o practică foarte bună pentru a menține cod). Soluția este data de template-uri (sabloane).

O funcție template este o funcție în care tipul de date este parametrizat, deci putem avea multiple variante ale aceleiași funcții scriind o singură dată cod. Sintaxa pentru declararea unei funcții template este:

```
1 template <typename T1, typename T2, ..., typename Tn>
2 <tip - retur> <nume - functie> (<lista - parametrii>) {
3     /* corp functie */
4 };
```

În loc de **typename** putem folosi și **class** pentru a enumera tipurile de date parametrizate în template-ul nostru.

Exemplu:

```
1 #include <iostream>
2
3 template <typename T> T add (T a, T b) {
4     return a + b;
5 }
6
7 int main () {
8     const int i = 4;
9     const int j = 5;
10
11     std::cout << add<int>(i, j); // instantiere explicita
12     std::cout << add(i, j);      // instantiere implicita, compilatorul
13                                 // deduce tipul de date
14 }
```

Codul asociat cu un template este compilat mai întâi din punct de vedere al sintaxei. Compilarea semnatică se face la instanțiere. Orice eroare care ar putea rezulta datorită tipurilor de date cu care este instanțiat un template apare la instanțiere (daca nu instanțiem niciodată un template compilatorul nu va indica posibile erori de semnatică pentru acel template).

Pentru o funcție template putem oferi o specializare pentru anumite tipuri de date. Scopul unei specializări este să oferim un comportament special pentru instanțiere template-ului cu un anumit tip de date.

```
1 // specializarea functie template add pentru intregi
2 template <> int add <int> (int a, int b) {
3     std::cout << "integer specialization ";
4     return a + b;
5 }
6
7 int main () {
8     const int i = 4;
9     const int j = 5;
10 }
```

```

11     std::cout << add(i, j); // integer specialization 9
12 }

```

In C++, o functie template nu e vazuta ca o functie propriu-zisa, transformarea template-ului intr-o functie are loc la instantiere, cand compilatorul genereaza automat of functie in care tipurile parameterizate primesc valori (e.g. `add<int>`). Instantierea functie template care este marcata ca functie si poate fi folosita pentru apeluri.

Putem avea in clase non template metode (statice sau nu) si functii prieten template:

```

1  #include <iostream>
2  using namespace std;
3
4  class Point {
5      int x, y;
6  public:
7      Point (int a = 0, int b = 0) : x(a), y(b) {}
8      friend ostream& operator<< (ostream&, const Point&);
9
10     template <typename T>
11     void foo (const T&);
12
13     template <typename T>
14     static Point convert (const T&, const T&);
15 };
16
17 template <typename T>
18 void Point::foo(const T& val) {
19     x *= val;
20     y *= val;
21 }
22
23 template <typename T>
24 Point Point::convert(const T& a, const T& b) {
25     Point p;
26
27     p.x = a; p.y = b;
28
29     return p;
30 }
31
32 ostream& operator<< (ostream& out, const Point& p) {
33     out << "(" << p.x << ", " << p.y << ")";
34     return out;
35 }
36
37 int main () {
38     Point p (2, 10);
39     p.foo(2.4f);
40     cout << p << endl;
41     p = Point::convert(1.3f, 5.678f);
42     cout << p << endl;
43     return 0;
44 }

```

2 Clase template

C++ suporta si clase template pentru situatiile in care clasele pe care le scriem pot fi avea ca proprietati mai multe tipuri de date (e.g. stiva, vector, pereche, lista, matrice, hashmap etc.). Sintaxa pentru declararea unei clase template este urmatoarea:

```

1  template <typename T1, typename T2, ..., typename Tn>
2  class <nume - clasa> {
3      /* definitie clasa */
4  };

```

Exemplu:

```
1 template <typename T> class Point;
2
3 template <typename T>
4 istream& operator >> (istream&, Point<T>&);
5
6
7 template <typename T> class Point {
8     T x, y;
9 public:
10     Point(const T&, const T&);
11     int cadran ();
12
13     template <typename U>
14     void templateMethod (const U&);
15
16     static void staticMethod ();
17
18     template <typename U>
19     static void templateStaticMethod (const U&);
20
21     template <typename U>
22     friend ostream& operator<<(ostream&, const Point<U>&);
23
24     friend istream& operator>><T>(istream&, Point<T>&);
25 };
26
27 template <typename T>
28 Point<T>::Point(const T& a, const T& b) : x(a), y(b) { }
29
30 template <typename T>
31 int Point<T>::cadran() {
32     if (x > 0) {
33         if (y > 0) {
34             return 1;
35         } else if (y < 0) {
36             return 2;
37         }
38     } else if (x < 0) {
39         if (y < 0) {
40             return 3;
41         } else if (y > 0) {
42             return 4;
43         }
44     }
45     // the point is in origin or on axis
46     return 0;
47 }
48
49
50
51 template<typename T>
52 template<typename U>
53 void Point<T>::templateMethod (const U& a) {
54     cout << a;
55 }
56
57 template <typename T>
58 void Point<T>::staticMethod() {
59     cout << "This is a static method in a template class";
60 }
61
62 template <typename T>
63 template <typename U>
64 void Point<T>::templateStaticMethod(const U& a) {
65     cout << "This is template static method in a template class " << a;
66 }
```

```

67
68 template <typename T>
69 ostream& operator<<(ostream& out, const Point<T>& p) {
70     out << "(" << p.x << ", " << p.y << ")";
71     return out;
72 }
73
74 template <typename T>
75 istream& operator>>(istream& in, Point<T>& p) {
76     in >> p.x >> p.y;
77     return in;
78 }
79
80 int main () {
81     Point<int> p(0, 0);
82     cin >> p;
83     cout << p << " " << p.cadran() << endl;
84     return 0;
85 }

```

Datorită modului în care sunt compilate template-urile, o clasă template nu poate fi scrisă în header (.h) și sursă (.cpp). Pentru a putea defini separat clasa trebuie să fie instanțiat template-ul la sfârșitul fișierului cpp.

Ca în cazul funcțiilor template, codul pentru clasa template nu denota un tip de date. Când are loc instanțierea template-ului compilatorul declară un nou tip de date. Fiecare instanțiere a unei clase template creează un nou tip de date care nu este legat în niciun fel de celelalte instanțieri ale template-ului.

```

1 template class Point <int>; // instanțiere explicită
2 Point<int> p; // instanțiere implicită

```

Ca în cazul funcțiilor template, putem avea specializări pentru o clasă template, în care putem avea metode și proprietăți noi (care nu există în definiția originală a clasei template).

```

1 template <>
2 class Point <float> {
3     /* template specialization */
4 };

```

3 Valori default in template

Ca în cazul parameterilor unei funcții putem să furnizăm valori default pentru tipurile de date parameterizate într-un template.

```

1 #include <iostream>
2 using namespace std;
3
4 template <typename T = int>
5 struct Pair {
6     public:
7         T first, second;
8 };
9
10 template <class T>
11 ostream& operator<< (ostream& out, const Pair<T>& p) {
12     out << "(" << p.first << ", " << p.second << ")";
13     return out;
14 }
15
16
17 int main () {
18     Pair<> p1 = {2, 4};

```

```

19 // in mod normal suntem obligati sa spune cu ce tip da date instantiem
20 // dar in acest caz putem omite si compilatorul va stii ca e vorba de int
21 Pair<float> p2 = {3.4f, 10.6f};
22
23 cout << p1 << " " << p2 << endl;
24 return 0;
25 }

```

Exerciții

1. Implementați o clasă template pentru listă liniară înlănțuită, cu următoarea interfață:

- constructor cu parametrii și de copiere;
- metodă pentru adăugarea de element nou;
- metodă pentru căutarea unui element (rezultat boolean);
- metodă pentru ștergerea unui element după valoare;
- supraîncărcarea operatorului `[]` pentru obținerea elementului de pe poziția `i` ;
- supraîncărcarea operatorilor `<<` și `>>` pentru citire si afișare;
- destructor;
- operator de atribuire;

2. Implementați o clasă template pentru stivă, cu următoarea interfață:

- constructor cu parametrii și de copiere;
- metodă pentru adăugarea de element nou;
- metodă pentru intoarcerea elementului din varful stivei;
- metodă pentru ștergerea unui element;
- supraîncărcarea operatorilor `<<` și `>>` pentru citire si afișare;
- destructor;
- operator de atribuire;
- metoda care elimina 1 sau mai multe elemente din stivă si le intoarce sub forma de vector.