AVANSAT · POO ÎN C++ · LECȚIA 6

Șabloane (templates): funcții și clase generice, specializare, variadice, if constexpr

Durată: 90 min Nivel: avansat / pregătire pentru facultate și industrie Înainte: lecțiile oop-03, oop-04

CE VEI ȘTI DUPĂ LECȚIE

- Scrii șabloane de funcții și de clase și explici cum le instanțiază compilatorul.
- Folosești parametri de șablon care nu sunt tipuri (de exemplu, o dimensiune N).
- Specializezi un șablon pentru un tip anume.
- Scrii șabloane variadice cu expresii fold și alegi cod la compilare cu if constexpr și type_traits.
- Înțelegi cum este construită biblioteca standard: containere, iteratori și algoritmi generici.

1. Același cod pentru tipuri diferite

Ai scris `maxim` pentru `int`. Acum îți trebuie pentru `double`, `string`, `Fractie`. Copierea funcției de patru ori încalcă DRY, iar o ierarhie cu funcții virtuale ar fi lentă și nefirească pentru `int`. **Șabloanele** (*templates*) permit scrierea codului **o singură dată**, cu tipul ca **parametru**. Compilatorul generează automat câte o versiune pentru fiecare tip folosit. Întreaga bibliotecă standard (STL) este construită astfel: `vector<T>`, `map<K, V>`, `sort`, `max`.

	Polimorfism cu `virtual` (oop-04)	**Șabloane**
când se alege funcția	la rulare	la compilare
cost la rulare	apel indirect prin vtable	zero (cod generat, poate fi *inline*)
tipurile	trebuie să moștenească o bază comună	orice tip care are operațiile folosite
colecții de tipuri diferite	da (`vector<unique_ptr<Baza>>`)	nu direct
mesaje de eroare	simple	uneori foarte lungi

2. Șabloane de funcții

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

template <typename T>                       // T este un tip, ales la fiecare apel
const T& maxim(const T &a, const T &b)
{
    return (a < b) ? b : a;                 // cere doar ca T să aibă operatorul <
}

template <typename T>
void schimba(T &a, T &b)
{
    T aux = std::move(a);                   // mutare, nu copiere (oop-02)
    a = std::move(b);
    b = std::move(aux);
}

int main()
{
    int x, y;
    double p, q;
    string s, t;
    cin >> x >> y >> p >> q >> s >> t;
    cout << maxim(x, y) << ' ';             // T dedus: int
    schimba(x, y);
    cout << y << '\n';
    cout << maxim(p, q) << ' ' << maxim<double>(q, 1) << '\n';   // tip dat explicit: 1 devine 1.0
    schimba(s, t);
    cout << s << ' ' << t << '\n';
    return 0;
}
```

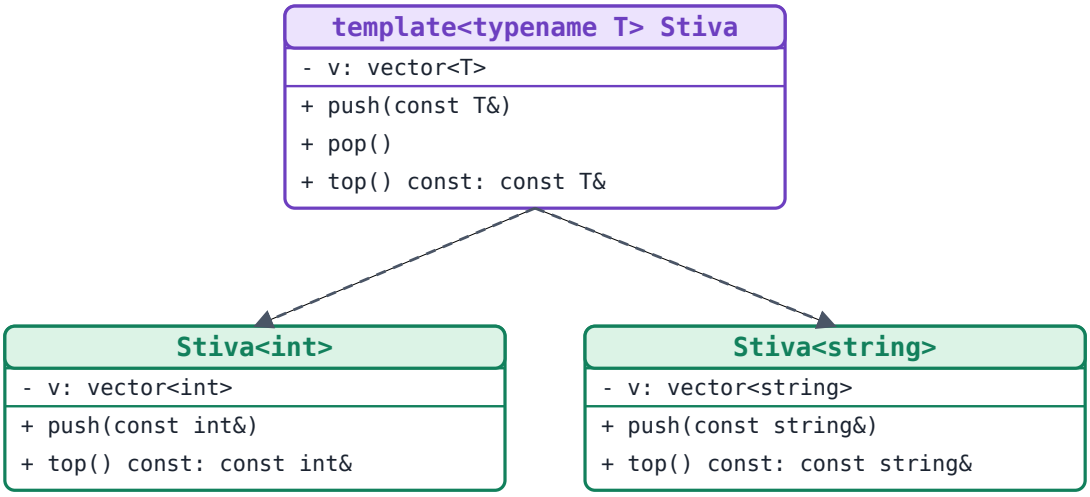
DE CE AȘA?

Cum funcționează? Compilatorul **deduce** `T` din argumente (`maxim(x, y)` cu `int` ⇒ `T = int`) și **instanțiază** funcția: generează `maxim<int>` ca și cum ai fi scris-o de mână. Nu există nicio funcție `maxim` „generală" la rulare. **De ce `maxim(p, 1)` nu ar merge?** Din `p` rezultă `T = double`, din `1` rezultă `T = int`: conflict. Soluția este să dai tipul explicit, `maxim<double>(p, 1)`. **De ce `const T&`?** Ca să nu copiem obiecte mari (`string`, `Fractie`).

DE ȘTIUT

`typename` și `class` înseamnă același lucru în `template <typename T>` / `template <class T>`. `typename` este mai clar, pentru că T poate fi și `int`, care nu este o clasă.

3. Șabloane de clase



compilatorul generează câte o clasă pentru fiecare tip folosit (Stiva<int>, Stiva<string>)

Instanțierea unui șablon

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <stdexcept>
using namespace std;

template <typename T>
class Stiva
{
    vector<T> v;
public:
    void push(const T &x) { v.push_back(x); }
    void pop()
    {
        if (v.empty()) throw out_of_range("pop pe stiva goala");
        v.pop_back();
    }
    const T& top() const
    {
        if (v.empty()) throw out_of_range("top pe stiva goala");
        return v.back();
    }
    bool empty() const { return v.empty(); }
    size_t size() const { return v.size(); }
};

int main()
{
    Stiva<int> a;                          // o clasă generată pentru int
    a.push(10); a.push(20); a.push(30);
    cout << a.top() << ' ';
    a.pop();
    cout << a.top() << '\n';

    Stiva<string> b;                       // altă clasă, generată pentru string
    b.push("templates"); b.push("cpp"); b.push("invat");
    cout << b.size() << '\n';
    b.pop();
    cout << b.top() << ' ';
    b.pop();
    cout << b.top() << '\n';
    return 0;
}
```

ATENȚIE — GREȘELI FRECVENTE

Codul unui șablon trebuie să fie **vizibil** acolo unde este folosit, deci șabloanele se scriu de obicei **în întregime în fișierul header** (`.h` / `.hpp`), nu separat într-un `.cpp` . Dacă pui doar declarația în header și definițiile într-un `.cpp` , primești erori de legare (*undefined reference*), pentru că acolo compilatorul nu știa ce tipuri vor fi folosite.

SFAT

Din C++17, tipul poate fi dedus și la clase, din constructor: `vector v = {1, 2, 3};` este `vector<int>`, `pair p(1, 2.5);` este `pair<int, double>` (*class template argument deduction*, CTAD).

4. Parametri care nu sunt tipuri

Un parametru de șablon poate fi și o **valoare constantă** cunoscută la compilare, de exemplu o dimensiune. Așa este construit `std::array<T, N>`: dimensiunea face parte din tip, deci tabloul stă pe stivă, fără alocare dinamică.

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

template <typename T, int N>               // N este o valoare, nu un tip
struct Matrice
{
    T a[N][N] = {};

    Matrice operator*(const Matrice &o) const
    {
        Matrice r;
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                for (int k = 0; k < N; k++)
                    r.a[i][j] += a[i][k] * o.a[k][j];
        return r;
    }
    void afiseaza() const
    {
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                cout << a[i][j] << (j + 1 < N ? ' ' : '\n');
    }
};

int main()
{
    Matrice<long long, 2> m;               // matrice 2 x 2 de long long
    cin >> m.a[0][0] >> m.a[0][1] >> m.a[1][0] >> m.a[1][1];
    m.afiseaza();
    (m * m).afiseaza();
    // Matrice<long long, 3> t; m * t;      // nu se compilează: 2 x 2 și 3 x 3 sunt tipuri
diferite
    return 0;
}
```

DE CE AȘA?

Ce câștigi? Dimensiunea fiind cunoscută la compilare, buclele pot fi optimizate (chiar desfăcute), iar greșelile de dimensiune devin **erori de compilare**, nu căderi la rulare. Același cod de ridicare la putere din lecția oop-03 merge acum pentru orice dimensiune și orice tip numeric.

5. Specializare

Uneori varianta generală nu se potrivește pentru un tip anume. **Specializarea completă** oferă o implementare separată pentru acel tip:

```
template <typename T>
string descrie(const T &x) { return "valoare " + to_string(x); }

template <>                                  // specializare pentru bool
string descrie<bool>(const bool &x) { return x ? "adevarat" : "fals"; }
```

La clase se poate face și **specializare parțială** (de exemplu, o versiune pentru toate tipurile pointer `T*`). Exemplul celebru din biblioteca standard este `vector<bool>`, specializat să folosească un bit pe element.

6. Șabloane variadice și `if constexpr`

Un **șablon variadic** acceptă oricâte argumente, de tipuri diferite (`typename... Args`). În C++17, **expresiile fold** aplică un operator peste toate: `(args + ...)`. Cu `if constexpr` alegi o ramură **la compilare**, după proprietățile tipului (din `<type_traits>`); ramura neleasă nici nu este compilată pentru acel tip.

```cpp
#include <iostream>
#include <string>
#include <type_traits>
using namespace std;

template <typename... Args>
auto suma(Args... args)
{
    return (args + ...);                    // fold: a1 + (a2 + (a3 + ...))
}

template <typename T>
void eticheta(const T &x)
{
    if constexpr (is_same_v<T, bool>) cout << "[bool " << x << "] ";
    else if constexpr (is_integral_v<T>) cout << "[int " << x << "] ";
    else if constexpr (is_floating_point_v<T>) cout << "[real " << x << "] ";
    else cout << "[text " << x << "] ";
}

template <typename... Args>
int afiseazaToate(const Args &... args)
{
    (eticheta(args), ...);                  // fold peste operatorul virgulă: apel pentru
fiecare argument
    cout << '\n';
    return sizeof...(args);                 // numărul de argumente
}

int main()
{
    cout << suma(1, 2, 3, 4) << '\n';
    cout << suma(2.5, 5) << '\n';
    int n = afiseazaToate(42, 3.14, string("abc"), true);
    cout << n << '\n';
    return 0;
}
```

DE ȘTIUT

Înainte de C++17, variadicele se scriau prin **recursivitate** pe argumente (un caz de bază și un caz care „consumă” primul argument). Expresiile fold fac același lucru într-o singură linie. Așa sunt implementate `make_unique`, `emplace_back`, `tuple` și `printf`-ul sigur din bibliotecile moderne.

7. Cum este construită STL: algoritmi pe iteratori

Algoritmii STL (`sort`, `find`, `count_if`) nu lucrează pe containere, ci pe **iteratori**: obiecte care se comportă ca pointerii (`*it`, `++it`, `it != sfarsit`). Un singur șablon funcționează astfel pentru `vector`, `list`, `array`, `string` și chiar pentru tablouri C (unde iteratorii sunt pointeri).

```cpp
#include <iostream>
#include <vector>
#include <list>
using namespace std;

template <typename It, typename Pred>      // orice iterator, orice condiție
int numaraDaca(It inceput, It sfarsit, Pred conditie)
{
    int k = 0;
    for (It it = inceput; it != sfarsit; ++it)
        if (conditie(*it)) k++;
    return k;
}

struct EstePar
{
    bool operator()(int x) const { return x % 2 == 0; }
};

int main()
{
    int n;
    cin >> n;
    vector<int> v(n);
    for (auto &x : v) cin >> x;
    list<int> l(v.begin(), v.end());       // aceleași valori, alt container
    int c[] = {5, 6, 7};

    cout << numaraDaca(v.begin(), v.end(), EstePar()) << ' ';          // vector
    cout << numaraDaca(l.begin(), l.end(), [](int x) { return x > 7; }) << ' ';   // listă
+ lambda
    cout << numaraDaca(c, c + 3, [](int x) { return x >= 5; });         // tablou C:
iteratorii sunt pointeri
    return 0;
}
```

DE CE AȘA?

De ce iteratori? Cu M containere și N algoritmi, fără iteratori ar trebui M × N implementări; cu iteratori sunt M + N. Fiecare container știe să ofere iteratori, iar fiecare algoritm știe să lucreze cu ei. Această separare este ideea de bază a lui Alexander Stepanov, autorul STL.

8. Ce cere un șablon de la tip: concepte

Un șablon funcționează cu orice tip care are operațiile folosite în corpul lui (`maxim` cere `<`). Dacă tipul nu le are, eroarea apare adânc în interiorul șablonului și poate avea zeci de rânduri. Remedii:

Unealta	Standard	Exemplu
`static_assert`	C++11	`static_assert(is_arithmetic_v<T>, "T trebuie să fie numeric");`
`if constexpr` + `type_traits`	C++17	ramuri diferite pentru tipuri diferite

Unealta	Standard	Exemplu
concepte	C++20	`template <std::integral T> T cmmdc(T a, T b);`

Conceptele (C++20) scriu cerințele direct în semnătură, iar mesajele de eroare devin scurte și clare: „tipul `double` nu satisface `integral`".

9. Greșeli frecvente

Greșeala	Ce se întâmplă	Corect
definițiile șablonului în `.cpp` separat	*undefined reference* la legare	totul în header
argumente de tipuri diferite pentru același `T`	deducerea eșuează	tip explicit `f<double>(…)` sau doi parametri `T`, `U`
tip fără operația cerută	erori lungi, greu de citit	`static_assert` sau concepte
copiere prin valoare a lui `T` mare	lent	`const T&`
`typename` uitat la tipuri dependente (`typename T::value_type`)	eroare de compilare	scrie `typename` în față
șabloane pentru orice, din reflex	cod greu de citit, compilare lentă	doar când chiar ai mai multe tipuri
specializare pusă după prima folosire	comportament diferit de cel așteptat	declară specializările înainte de folosire

10. Exersează pe hârtie

1. Ce tip deduce compilatorul pentru `T` în `maxim(3, 7)`, `maxim(2.5, 1.0)`, `maxim(string("a"), string("b"))`? De ce nu merge `maxim(3, 2.5)`?

2. Scrie un șablon `minim3(a, b, c)` care întoarce minimul a trei valori de același tip.

3. Câte clase `Stiva` generează compilatorul pentru un program care folosește `Stiva<int>`, `Stiva<int>` (în alt loc) și `Stiva<double>`?

4. De ce nu se compilează `Matrice<int, 2> a; Matrice<int, 3> b; a * b;`? Este un avantaj?

5. Ce întoarce `suma(1, 2.5, 3)` și de ce tip este?

6. Scrie un șablon `afiseazaVector(const vector<T> &v)` care afișează elementele separate prin spațiu.

7. Ce afișează `afiseazaToate('x', 7)`? (Atenție: `char` este tip întreg.)

8. Ce trebuie să aibă un tip ca să meargă cu `numaraDaca`?

9. De ce `std::sort` nu merge pe `std::list` (indiciu: ce operații pe iteratori cere)?

10. Când ai alege funcții virtuale în loc de șabloane?

11. Mini-grile

1. Un șablon de funcție este instanțiat: a) la rulare b) la compilare, pentru fiecare tip folosit c) o singură dată pentru toate tipurile d) de linker

2. Codul complet al unui șablon se pune de obicei: a) în header b) într-un fișier `.cpp` separat c) în `main` d) într-o bibliotecă dinamică

3. `template <typename T, int N>` înseamnă că N este: a) un tip b) o valoare constantă cunoscută la compilare c) o variabilă d) un pointer

4. `(args + ...)` este: a) o eroare b) un apel recursiv c) o expresie fold peste un pachet de argumente d) un macro

12. Mai departe (pentru cine vrea mai mult)

- **Concepte** și `requires` (C++20).
- **CRTP** (*Curiously Recurring Template Pattern*): polimorfism la compilare.
- **SFINAE** și `enable_if`: tehnica dinaintea conceptelor.
- **Metaprogramare:** calcule la compilare cu `constexpr` și șabloane.
- **Type erasure:** cum `std::function` ține orice obiect apelabil.

Rezumat

- Un șablon este o rețetă: compilatorul generează cod pentru fiecare tip folosit; cost zero la rulare.
- Șabloane de funcții (`template <typename T>`), deducere automată sau tip explicit.
- Șabloane de clase (`Stiva<T>`); codul lor stă în header.
- Parametri valoare (`int N`): dimensiuni cunoscute la compilare, erori prinse devreme.
- Specializare pentru tipuri speciale; variadice cu fold; `if constexpr` + `type_traits`.
- STL = containere + iteratori + algoritmi generici; conceptele (C++20) clarifică cerințele.

Citește și exersează în plus

- learncpp.com — Function templates — introducere pas cu pas în șabloane (engleză)
  https://www.learncpp.com/cpp-tutorial/function-templates/
- cppreference — Templates — toate regulile, cu exemple (engleză)
  https://en.cppreference.com/w/cpp/language/templates
- cppreference — fold expressions — expresii fold, C++17 (engleză)
  https://en.cppreference.com/w/cpp/language/fold

Răspunsuri

Exersează pe hârtie

1. `int`, `double`, `string`. Din `3` rezultă `int`, din `2.5` rezultă `double`: conflict de deducere.

2. `template <typename T> T minim3(const T &a, const T &b, const T &c) { T m = a; if (b < m) m = b; if (c < m) m = c; return m; }`

3. Două: `Stiva<int>` (o singură dată, oricâte folosiri) și `Stiva<double>`.

4. `Matrice<int, 2>` și `Matrice<int, 3>` sunt tipuri diferite, iar `operator*` acceptă doar același tip. Da: greșeala de dimensiune este prinsă la compilare.

5. `6.5`, de tip `double` (`1 + (2.5 + 3)` promovează la `double`).

6. `template <typename T> void afiseazaVector(const vector<T> &v) { for (const auto &x : v) cout << x << ' '; cout << '\n'; }`

7. `[int x] [int 7]`: `char` este tip întreg, deci intră pe ramura `is_integral_v`; se afișează totuși ca litera `x`, pentru că `cout << char` scrie caracterul.

8. Iteratorii trebuie să suporte `!=`, `++` și `*`; condiția trebuie să se poată apela cu elementul și să întoarcă ceva convertibil la `bool`.

9. `sort` cere iteratori cu acces direct (`it + k`, `it2 - it1`), iar iteratorii de listă merg doar câte un pas. Lista are metoda proprie `l.sort()`.

10. Când ai nevoie de obiecte de tipuri diferite în aceeași colecție, sau când tipul se decide la rulare (de exemplu, după datele de intrare), sau când vrei să ascunzi implementarea în spatele unei interfețe stabile.

Mini-grile

1. b). **2.** a) în header. **3.** b). **4.** c).