



# Tutoriat 6

Ana-Maria Rusu & Ionuț-Daniel Nedelcu

Grupa 242

Facultatea de Matematică și Informatică a Universității din București

**Mențiune:**

**toate programele sunt rulate în Visual Studio Code  
compilator g++, comanda terminal: g++ sursa.cpp -o sursa && ./sursa**

# Ce vom face azi?

STL

Template 🥵

Exercitii



01

STL

# STL (Standard Template Library)

- În C++, **STL (Standard Template Library)** oferă un set de structuri de date generice și algoritmi foarte utilizați în programare. Scopul STL este să eficientizeze scrierea codului prin folosirea unor structuri de date și funcții gata implementate, dar personalizabile prin template-uri.
- STL funcționează cu template-uri, deci putem avea `vector<string>`, `map<int, vector<int>>` etc.
- Integrează perfect cu algoritmi STL precum `sort`, `find`, `count`, `accumulate`, etc.
- În cadrul acestei materii vom discuta doar despre structuri de date preimplementate în STL.

# Structuri de date

- **Vector** - `vector<tip_date>` - Similar cu un array dinamic, elemente accesibile prin index, dimensiune variabilă
- **Set** - `set<tip_date>` - Colecție de elemente unice, sortate automat. Bazat pe arbori binari de căutare.
- **List** - `list<tip_date>` - Listă dublu înlănțuită (doubly linked list). Inserarea/ștergerea sunt eficiente oriunde în listă.
- **Map** - `map<tip_cheie, tip_valoare>` - Colecție de perechi cheie -> valoare, unde cheile sunt unice și sortate.

# Alte exemple (optionale, extra)

→ **Liste** - forward\_list, dequeue

→ **Mape/Mapări** - multimap, unordered\_map, unordered\_multimap

→ **Seturi(mulțimi)** - multiset, unordered\_set, unordered\_multiset

→ **Stive/Cozi** - stack, priority\_queue, dequeue

# Vector

→ Creare vectori:

```
vector<int> v1;           // vector gol
vector<int> v2(5);        // 5 elemente inițializate cu 0
vector<int> v3(5, 10);    // 5 elemente inițializate cu 10
vector<int> v4 = {1, 2, 3}; // listă de inițializare (C++11)
```



## Inserare și ștergere

### Funcție

### Descriere

`push_back(val)`

Adaugă `val` la finalul vectorului

`pop_back()`

Elimină ultimul element

`insert(pos, val)`

Inserează `val` înainte de poziția `pos`

`erase(pos)`

Șterge elementul de la poziția `pos`

`erase(start, end)`

Șterge elementele din intervalul `[start, end)`

`clear()`

Elimină toate elementele

`emplace_back(args)`

Construiește un obiect direct în vector



## Acces la elemente

Funcție	Descriere
<code>v[i]</code>	Acces direct prin index (fără verificare)
<code>at(i)</code>	Acces cu verificare (aruncă excepție dacă e invalid)
<code>front()</code>	Primul element
<code>back()</code>	Ultimul element
<code>data()</code>	Pointer la array-ul intern

## Informații despre vector

Funcție	Descriere
<code>size()</code>	Numărul de elemente
<code>capacity()</code>	Cât spațiu a fost alocat
<code>empty()</code>	Verifică dacă vectorul este gol
<code>max_size()</code>	Numărul maxim de elemente posibile



## Alte funcții utile

Funcție	Descriere
<code>resize(n)</code>	Schimbă dimensiunea vectorului la <code>n</code>
<code>reserve(n)</code>	Alocă spațiu pentru cel puțin <code>n</code> elemente
<code>shrink_to_fit()</code>	Eliberează memoria neutilizată (capacitate → dimensiune)
<code>swap(v2)</code>	Schimbă conținutul între doi vectori
<code>assign(n, val)</code>	Reîncarcă vectorul cu <code>n</code> copii ale lui <code>val</code>
<code>begin(), end()</code>	Iteratori pentru parcurgere
<code>rbegin(), rend()</code>	Iteratori inversați

# Exemplu Vector Constructori

```
#include <iostream>
#include <vector>
using namespace std;
class A {
    int x;
public:
    A(): x(0) { cout<<"C "; }
    A(const A& other): x(other.x)
    { cout<<"CC "; }
    ~A() { cout<<"D "; }
};
```

```
int main()
{
    vector<A> v;
    v.push_back(A()); cout<<endl; // 1 element
    v.push_back(A()); cout<<endl; // 2 elemente
    v.push_back(A()); cout<<endl; // 3 elemente
    v.push_back(A()); cout<<endl; // 4 elemente
    v.push_back(A()); cout<<endl; // 5 elemente
    v.push_back(A()); cout<<endl; // 6 elemente
    v.push_back(A()); cout<<endl; // 7 elemente
    v.push_back(A()); cout<<endl; // 8 elemente
    v.push_back(A()); cout<<endl; // 9 elemente
    return 0;
}
```

# Ce se afiseaza?

C CC D

C CC CC D D

C CC CC CC D D D

C CC D

C CC CC CC CC CC D D D D D

C CC D

C CC D

C CC D

C CC CC CC CC CC CC CC CC CC D D D D D D D D D

D D D D D D D D D



02

Template

# Template

→ **Definitie:** mecanism care permite definirea unor clase si functii generice, astfel incat putem opera cu diferite tipuri de date fara a fi nevoie de rescrierea codului pentru fiecare tip.

→ Astfel, template reprezinta o forma de **polimorfism la compilare**.

→ **Mecanism (functii) :**

-la inceput, functia template este compilata fara a se tine cont de tipul de date

-cand apelam respectiva functie, compilatorul creeaza o noua functie ce va opera pe tipul de date pe care noi il specificam.



# Template

→ **Sintaxa generala :**

**template <typename T>** antet { corp } sau **template <class T>** antet { corp }

-specificatia de template trebuie să fie imediat inaintea definitiei funcției → altfel, eroare

-T reprezinta tipul de date generic pe care vom opera

# Exemplu

- cerinta: pentru o serie de vectori se cere sa se gaseasca elementul maxim din fiecare structura - tipurile de date pot fi diferite intre vectori.
- sa presupunem ca primim 2 vectori, de exemplu unul de int si unul de char.
- mecanismul de calcul al maximului este unul generic - logica e la fel pentru ambele tipuri de date (int si char).
- in mod normal, ar trebui sa facem doua functii: una care gaseste maximul pe un vector de int si alta care gaseste maximul pe un vector de char.
- solutia este corecta, dar foarte ineficienta.
- pentru a imbunatati eficienta si pentru a **reutiliza codul**, vom folosi sabloane (template)

# Exemplu – Fara Functii Template

```
#include<iostream>
using namespace std;
int maxim_int(int v[],int n)
{
    int max=v[0];
    for(int i=0;i<n;i++)
        if(max<v[i])
            max=v[i];
    return max;
}
```

```
char maxim_char(char v[], int n)
{
    char max=v[0];
    for(int i=0;i<n;i++)
        if(max<v[i])
            max=v[i];
    return max;
}
```

```
int main()
{
    int v1[]={1,5,3,7,3};
    char v2[]={'b','c','d','z','a'};

    cout << maxim_int(v1,sizeof (v1)/sizeof (int))<< endl;
    cout << maxim_char (v2,sizeof (v2)/ sizeof (char)) << endl;

    return 0;
}
```

# Exemplu – Cu Functii Template

```
#include<iostream>
using namespace std;
template <class T> T maxim(T v[],int n)
{
    T max=v[0];
    for(int i=0;i<n;i++)
        if(max<v[i])
            max=v[i];
    return max;
}

int main()
{
    int v1[]={1,5,3,7,3};
    char v2[]={'b','c','d','z','a'};

    cout << maxim<int> (v1, sizeof (v1)/sizeof (int))<< endl;
    cout << maxim<char> (v2, sizeof (v2)/ sizeof (char)) << endl;

    return 0;
}
```

→ T este tipul generic de date, care poate fi in acest caz int sau char

→ la apel vom specifica intre paranteze <> pentru ce tip de date apelam functia

# Exemplu – Class Template

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

template <typename T> class Test
{
    T variabila;
public:
    Test(T variabila) {this->variabila= variabila;}
    T getVar() const { return variabila;}

};

int main()
{
    Test<int> a(2);
    cout<<a.getVar()<<endl;

    Test<char> b('q');
    cout<<b.getVar()<<endl;

    Test<bool> c(false);
    cout<<c.getVar()<<endl;
}
```

→ Clasa Test poate avea tipuri de date generice - în acest caz, data membra variabila are tipul de date generic T

→ La instantiere, decidem tipul de date pentru "variabila"

→ Ca si la functii template, tipul de date se specifica intre paranteze <>

# Specializarea Functiilor Template

- Specializarea unui template reprezinta o particularizare a acelei functii pentru un anumit tip de date.
- Pentru functii, specializarea trebuie sa aiba exact **aceeasi semnatura** ca template-ul original.
- Scriem template <> pentru specializare.
- Nu putem sa **facem partial** specializarea functiilor (doar completa).

# Exemplu – Specializare

```
#include <iostream>

using namespace std;

template <typename T> void afisare(T val) {
    cout << "generic: " << val << endl;
}

// specializare pentru double
template <> void afisare<double>(double val) {
    cout << "double: " << val << endl;
}

// specializare pentru char
template <> void afisare<char>(char val) {
    cout << "char: '" << val << "'" << endl;
}
```

```
int main()
{
    afisare(5.345); // double
    afisare(10);    // generic (NU FACE CONVERSIE DE
// LA INT LA DOUBLE)
    afisare('A'); // char

    return 0;
}
```

# Prioritate la supraincarcare

1. Se caută o **functie obisnuita** care sa aiba parametri potriviti.
2. Daca nu s-a gasit, se cauta o **specializare template** cu parametri potriviti.
3. Daca nu s-a gasit, se cauta o **functie template** cu numarul de parametri potriviti.
4. Daca nu s-a gasit → eroare



# Exemplu

```
#include<iostream>

using namespace std;

template <typename T> void f (T x) {
    cout << "Template"<<endl;
}

template <> void f (int x) {
    cout << "Specializare Int"<<endl;
}

void f (char c) {
    cout << "Normal Char"<<endl;
}
```

```
void f (int c) {
    cout << "Normal Int"<<endl;
}

int main ()
{
    f(5); // Normal Int
    f('a'); // Normal Char
    f<int>(1); // Specializare Int
    f<char>('r'); // Template f
}
```



03

Exercitii