


# Capitolul 6. STL

## Standard Template Library

# Standard Template Library

 Standard Template Library (STL) = o bibliotecă de template-uri (clase și funcții), utilizate frecvent în rezolvarea problemelor de programare.

Componentă:

1. Containere (containers)
2. Iteratori (iterators)
3. Algoritmi (algorithms)

---

4. Obiecte funcții (functors)
5. Alocatori (allocators)

# 1. Containere (Containers)

 O serie de template-uri ce implementează structuri de date comune. Un container gestionează spațiul de stocare alocat elementelor sale și conține funcții membre care accesează respectivele elemente.

Toate containerele conțin o serie de funcții, printre care:

- `==` (operatorul de testare a egalității)
- `!=` (operatorul de testare a inegalității)
- `swap(ob_1, ob_2)` (interschimbare a 2 elemente)
- `size()` (numărul de elemente aflate în container)
- `max_size()` (numărul de elemente conținute de cel mai mare container posibil)
- `empty()` (indicator al lipsei elementelor în container)

# 1. Containere (Containers)

Containererele conțin mai multe subtipuri:

- 1.1. Containere de secvențe (sequence containers);
- 1.2. Containere asociative (associative containers);
- 1.3. Containere asociative neordonate (unordered associative containers);
- 1.4. Adaptori de containere (container adaptors).

# 1.1. Containere de secvențe

Containere de secvențe = structuri de date ce pot fi accesate secvențial

1.1.1. `array`

1.1.2. `vector`

1.1.3. `deque`

1.1.4. `forward_list`

1.1.5. `list`

# 1.1.1. array

Container care reține tablouri multidimensionale, de dimensiune **fixă**, alocate **static**, în locații de memorie consecutive. Sintaxă:

```
template<class T, std::size_t N> struct array;
```

```
#include <iostream>
```

```
#include <array>
```

```
int main(){
    std::array<int, 5> a={1, 2, 3, 4, 5}, b={8, 7, 6};
    std::cout << a[0] << std::endl;
    std::cout << a.size() << std::endl;
    a.swap(b);
    std::cout << a[0] << std::endl;
    std::cout << a.size() << std::endl;
    return 0;
}
```

```
1
5
8
5

Process returned 0 (0x0)   execution time : 0.026 s
Press any key to continue.
```

# 1.1.2. vector

Container care reține tablouri multidimensionale, de dimensiune **variabilă**, alocate **dinamic**, în locații de memorie consecutive. Sintaxă:

```
template<class T, class Allocator = std::allocator<T>> class vector;

#include <iostream>
#include <vector>

void display(std::vector<int> v){
    for (size_t n=0; n<v.size(); ++n){
        std::cout << v[n] << " ";
    }
    std::cout << "\n";
}

int main()
{
    std::vector<int> v = { 7, 5, 16, 8, 2, 4, 9};
    std::cout << "Capacity: " << v.capacity() << "\n";
    display(v); // afisare

    v.push_back(25);
    v.push_back(13);
    v.insert(v.begin()+2, 18); // iterator pe prima pozitie
    display_2(v); // afisare

    std::cout << "Size: " << v.size() << "\n";
    std::cout << "Capacity: " << v.capacity() << "\n";
    v.erase(v.begin()+7);
    display(v); // afisare

    v.pop_back();
    display(v); // afisare
    return 0;
}
```

```
Capacity: 7
7 5 16 8 2 4 9
7 5 18 16 8 2 4 9 25 13
Size: 10
Capacity: 14
7 5 18 16 8 2 4 25 13
7 5 18 16 8 2 4 25

Process returned 0 (0x0)   execution time : 0.110 s
Press any key to continue.
```

# 1.1.3. deque

Container care reține tablouri multidimensionale, de dimensiune **variabilă**, alocate **dinamic**, pentru care inserția și ștergerea elementelor se poate face la ambele capete (double-ended queue). Sintaxă:

```
template<class T, class Allocator = std::allocator<T>> class deque;
```

```
#include <iostream>
#include <deque>

void display(std::deque<int> deq) {
    for (size_t n=0; n<deq.size(); ++n) {
        std::cout << deq[n] << " ";
    }
    std::cout << "\n";
}
```

```
Size: 7
7 5 16 8 2 4 9
13 7 18 5 16 8 2 4 9 25
7 18 5 16 8 2 9
```

```
Process returned 0 (0x0)   execution time : 0.094 s
Press any key to continue.
```

```
int main()
{
    std::deque<int> deq = { 7, 5, 16, 8, 2, 4, 9};
    std::cout << "Size: " << deq.size() << "\n";
    display(deq); // afisare

    deq.push_back(25);
    deq.push_front(13);
    deq.insert(deq.begin()+2, 18);
    display(deq); // afisare

    deq.erase(deq.begin()+7);
    deq.pop_back();
    deq.pop_front();
    display(deq); // afisare
    return 0;
}
```



# 1.1.4. forward\_list

Container care implementează conceptul de listă simplu înlănțuită. Toate operațiile se fac prin intermediul primului nod al listei. Sintaxă:

```
template<class T, class Allocator = std::allocator<T>> class forward_list;

#include <iostream>
#include <forward_list>

void display(std::forward_list<int> f_list)
{
    for (int n : f_list) {
        std::cout << n << " ";
    }
    std::cout << "\n";
}

int main()
{
    std::forward_list<int> fwd_list = { 7, 5, 16, 8, 2, 4, 9 };
    display(fwd_list); // afisare

    fwd_list.push_front(13);
    display(fwd_list); // afisare

    fwd_list.erase_after(fwd_list.begin());
    fwd_list.pop_front();
    display(fwd_list); // afisare
    return 0;
}
```

```
7 5 16 8 2 4 9
13 7 5 16 8 2 4 9
5 16 8 2 4 9
```

```
Process returned 0 (0x0)   execution time : 0.109 s
Press any key to continue.
```

Singurul container căruia îi lipsește funcția `size()` datorită implementării axate pe eficiență maximă.

# 1.1.5. list

Container care implementează conceptul de listă dublu înlănțuită.  
Sintaxă:

```
template<class T, class Allocator = std::allocator<T>> class list;
```

```
#include <iostream>
#include <list>

void display(std::list<int> lista){
    for (int n:lista){
        std::cout << n << " ";
    }
    std::cout << "\n";
}
```

```
Size: 7
7 5 16 8 2 4 9
18 13 7 5 16 8 2 4 9 25
13 7 5 16 8 2 4 9

Process returned 0 (0x0)   execution time : 0.036 s
Press any key to continue.
```

```
int main()
{
    std::list<int> lista = { 7, 5, 16, 8, 2, 4, 9};
    std::cout << "Size: " << lista.size() << "\n";
    display(lista); // afisare

    lista.push_back(25);
    lista.push_front(13);
    lista.insert(lista.begin(), 18);
    display(lista); // afisare

    lista.pop_back();
    lista.pop_front();
    display(lista); // afisare
    return 0;
}
```

## 1.2. Containere asociative

Containere asociative = structuri de date ordonate (sortate) ce pot fi accesate rapid (complexitate de calcul  $O(\log_n)$ ). Sortarea implică și definirea unei relații de ordine între două elemente de același tip.

1.2.1. `set`

1.2.2. `map`

1.2.3. `multiset`

1.2.4. `multimap`

# 1.2.1. set

Container care reține o mulțime de obiecte **unice**, sortate, de tipul `Key`. Cheile sunt sortate folosind funcția de comparare. Este implementat sub forma unui arbore binar roșu-negru. Sintaxă:

```
template<class Key, class Compare = std::less<Key>,  
        class Allocator = std::allocator<Key>> class set;
```

```
#include <iostream>  
#include <set>  
  
void display(std::set<int> multime){  
    for (int n:multime){  
        std::cout << n << " ";  
    }  
}
```

```
Size: 4  
2 4 7 16  
Numar aparitii 3:0  
Numar aparitii 4:1  
Contine 7  
Nu contine 8  
2 4 7 16 25  
  
Process returned 0 (0x0)   execution time : 0.035 s  
Press any key to continue.
```

```
int main()  
{  
    std::set<int> multime = { 7, 4, 16, 7, 2, 4, 4};  
    std::cout << "Size: " << multime.size() << "\n";  
    display(multime); // afisare  
  
    multime.insert(25);  
    std::cout << "Numar aparitii 3:" <<  
    multime.count(3) << std::endl;  
    std::cout << "Numar aparitii 4:" <<  
    multime.count(4) << std::endl;  
  
    contine(multime, 7);  
    contine(multime, 8);  
  
    display(multime); // afisare  
    return 0;  
}
```

# 1.2.2. map

Container asociativ, sortat, care conține perechi **unice** de tipul Key-Value. Cheile sunt sortate folosind funcția de comparare. Este implementat sub forma unui arbore binar roșu-negru. Sintaxă:

```
template<class Key, class T, class Compare = std::less<Key>,  
        class Allocator = std::allocator<std::pair<const Key, T>>> class map;
```

```
#include <iostream>  
#include <map>  
#include <string>  
  
void print_map(const std::map<std::string, int>& m)  
{  
    for (const auto& [key, value] : m) {  
        std::cout << key << " = " << value << ";\t";  
    }  
}
```

```
i5 = 10;      i7 = 15;      i9 = 20;  
Threadripper = 30; i5 = 10;      i7 = 25;      i9 = 20;  
  
Process returned 0 (0x0)   execution time : 0.035 s  
Press any key to continue.
```

```
int main()  
{  
    std::map<std::string, int> fct { {"i5", 10},  
    {"i7", 15}, {"i9", 20}, {};  
  
    print_map(fct);  
  
    fct["i7"] = 25;  
    fct["Threadripper"] = 30;  
  
    print_map(fct);  
    return 0;  
}
```

Tipul variabilei este dedus automat din inițializarea ei

# 1.2.3. multiset

Container care reține o mulțime de obiecte, sortate, de tipul `Key`. Cheile sunt sortate folosind funcția de comparare. Spre deosebire de `set`, pot exista mai multe chei cu aceeași valoare. Este implementat sub forma unui arbore binar roșu-negru. Sintaxă:

```
template<class Key, class Compare = std::less<Key>,  
        class Allocator = std::allocator<Key>> class multiset;
```

```
#include <iostream>  
#include <set>  
  
void display(std::set<int> multime){
```

```
Size: 7  
2 4 4 4 7 7 16  
Numar aparitii 3:0  
Numar aparitii 4:3  
Contine 7  
Nu contine 8  
2 4 4 4 7 7 16 25  
  
Process returned 0 (0x0)   execution time : 0.033 s  
Press any key to continue.
```

```
int main()  
{  
    std::set<int> multime = { 7, 4, 16, 7, 2, 4, 4};  
    std::cout << "Size: " << multime.size() << "\n";  
    display(multime); // afisare  
  
    multime.insert(25);  
    std::cout << "Numar aparitii 3:" <<  
    multime.count(3) << std::endl;  
    std::cout << "Numar aparitii 4:" <<  
    multime.count(4) << std::endl;  
  
    contine(multime, 7);  
    contine(multime, 8);  
    display(multime);  
    return 0;  
}
```

## 1.2.4. multimap

Container asociativ, sortat, care conține perechi de tipul Key-Value. Cheile sunt sortate folosind funcția de comparare. Spre deosebire de map, pot exista mai multe chei cu aceeași valoare. În acest caz, sunt reținute în ordinea în care au fost introduse. Este implementat sub forma unui arbore binar roșu-negru. Sintaxă:

```
template<class Key, class T, class Compare = std::less<Key>,
        class Allocator = std::allocator<std::pair<const Key, T>>> class map;

#include <iostream>
#include <map>
#include <string>

void print_map(const std::multimap<std::string, int>& m)
{
    for (const auto& [key, value] : m) {
        std::cout << key << " = " << value << ";\t";
    }
    std::cout << std::endl;
}

int main()
{
    std::multimap<std::string, int> fct {
        {"i5", 10}, {"i7", 15}, {"i9", 20}, {"i5", 3},
        {"i5", 19}};

    print_map(fct);
    fct.insert(std::pair{"Threadripper", 30});
    fct.insert(std::pair{"i7", 25});

    print_map(fct);
    return 0;
}
```

```
i5 = 10;      i5 = 3; i5 = 19;      i7 = 15;      i9 = 20;
Threadripper = 30;      i5 = 10;      i5 = 3; i5 = 19;      i7 = 15;      i7 = 25;      i9 = 20;
```

# 1.3. Containere asociative neordonate (C++11)

Containere asociative neordonate = structuri de date neordonate ce pot fi accesate rapid (complexitate de calcul medie  $O(1)$ ; worst-case:  $O(n)$ ).

1.3.1. `unordered_set`

1.3.2. `unordered_map`

1.3.3. `unordered_multiset`

1.3.4. `unordered_multimap`



## 1.3.1. unordered\_set

Container care reține o mulțime de obiecte **unice**, nesortate, de tipul `Key`. Este implementat sub forma unui hash table. Acest container se găsește în biblioteca `<unordered_set>`. Sintaxă:

```
template<class Key,  
        class Hash = std::hash<Key>,  
        class KeyEqual = std::equal_to<Key>,  
        class Allocator = std::allocator<Key>  
> class unordered_set;
```

Diferențe între `set` și `unordered_set`:

1. `set` utilizează mai puțină memorie;
2. elementele din `set` sunt ordonate;
3. obiectele de tipul `set` pot fi comparate lexicografic;
4. operațiile cu `unordered_set` sunt, în general, mai rapide.

## 1.3.2. unordered\_map

Container care reține o mulțime de perechi **unice**, nesortate, de tipul `Key-Value`. Este implementat sub forma unui hash table. Acest container se găsește în biblioteca `<unordered_map>`. Sintaxă:

```
template<class Key,  
         class T,  
         class Hash = std::hash<Key>,  
         class KeyEqual = std::equal_to<Key>,  
         class Allocator = std::allocator<std::pair<const Key, T>>  
> class unordered_map;
```

Diferențe între `map` și `unordered_map`:

1. `map` utilizează mai puțină memorie;
2. elementele din `map` sunt ordonate;
3. operațiile cu `unordered_map` sunt, în general, mai rapide. Excepție: inserția unui număr mare de elemente, datorită rehashing-ului.

## 1.3.3. unordered\_multiset

Container care reține o mulțime de obiecte, nesortate, de tipul `Key`. Pot exista mai multe chei cu aceeași valoare. Este implementat sub forma unui hashtable. Acest container se găsește în biblioteca `<unordered_set>` Sintaxă:

```
template<class Key,  
         class Hash = std::hash<Key>,  
         class KeyEqual = std::equal_to<Key>,  
         class Allocator = std::allocator<Key>  
> class unordered_multiset;
```

Diferențe între `multiset` și `unordered_multiset`:

1. `multiset` utilizează mai puțină memorie;
2. elementele din `multiset` sunt ordonate;
3. operațiile cu `unordered_multiset` sunt, în general, mai rapide. Excepție: inserția unui număr mare de elemente, datorită rehashing-ului.

## 1.3.4. unordered\_multimap

Container care reține o mulțime de perechi, nesortate, de tipul Key-Value. Este implementat sub forma unui hash table. Acest container se găsește în biblioteca `<unordered_map>`. Sintaxă:

```
template<class Key,  
         class T,  
         class Hash = std::hash<Key>,  
         class KeyEqual = std::equal_to<Key>,  
         class Allocator = std::allocator<std::pair<const Key, T>>  
> class unordered_multimap;
```

**Diferențe între multimap și unordered\_multimap:**

1. multimap **utilizează** mai puțină memorie;
2. elementele din multimap sunt ordonate;
3. operațiile cu unordered\_multimap sunt, în general, mai rapide.  
Excepție: inserția unui număr mare de elemente, datorită rehashing-ului.

# 1.4. Adaptorii de containere

Adaptorii de containere = containere cu interfețe diferite pentru containere secvențiale.

1.4.1. `stack`

1.4.2. `queue`

1.4.3. `priority_queue`

## 1.4.1. stack

Container care implementează conceptul de stivă (LIFO – Last-In, First-Out). Acționează ca un wrapper peste containerele `vector`, `deque` sau `list` => preia doar o parte dintre funcționalitățile acestora. Se găsește în biblioteca `<stack>`. Sintaxă:

```
template<class T, class Container = std::deque<T>> class stack;
```

Operațiile principale permise:

- `push()` <-> `Container::push_back()`
- `pop()` <-> `Container::pop_back()`
- `top()` <-> `Container::back()`

## 1.4.2. queue

Container care implementează conceptul de coadă (FIFO – First-In, First-Out). Acționează ca un wrapper peste containerele `deque` sau `list` => preia doar o parte dintre funcționalitățile acestora. Se găsește în biblioteca `<queue>`. Sintaxă:

```
template<class T, class Container = std::deque<T>> class queue;
```

Operațiile principale permise:

- `push()` <-> `Container::push_back()`
- `pop()` <-> `Container::pop_front()`
- `back()` <-> `Container::back()`
- `front()` <-> `Container::front()`

## 1.4.3. priority\_queue

Container care implementează conceptul de coadă (FIFO – First-In, First-Out). Acționează ca un wrapper peste containerele `vector` sau `deque` => preia doar o parte dintre funcționalitățile acestora. Se găsește în bibliotecă `<queue>`. Sintaxă:

```
template<class T,  
        class Container = std::vector<T>,  
        class Compare = std::less<typename Container::value_type>  
> class priority_queue;
```

Operațiile principale permise:

- `push()` <-> `Container::push_back()`
- `pop()` <-> `Container::pop_back()`
- `top()` <-> `Container::front()`



## 2. Iteratori (Iterators)

 Iteratori = obiecte care pointează (indică) către elementele din interiorul unui container.

Iteratorii trebuie să implementeze cel puțin 2 funcționalități:

1. iterație – trebuie să poată trece de la un element la altul, prin incrementarea iteratorului (`operator++`);
2. dereferențiere – trebuie să poată citi din memorie elementul către care indică iteratorul (`operator*`).

Cea mai evidentă formă a unui iterator: pointer.

Fiecare container are un iterator specific.

## 2. Iteratori (Iterators)

Există 5 categorii de iteratori, ierarhizați, și încă o categorie în afara ierarhiei.

1. Legacy**Input**Iterator
2. Legacy**Forward**Iterator
3. Legacy**Bidirectional**Iterator
4. Legacy**RandomAccess**Iterator
5. Legacy**Contiguous**Iterator
  
6. Legacy**Output**Iterator

## 2. Iteratori (Iterators)

Din ierarhia de 5 iteratori, fiecare adaugă funcționalități peste nivelul precedent.

Categoria de iterator	Funcționalități suplimentare
<i>LegacyInputIterator</i>	+ citire – poate fi dereferențiată ca o rvalue (*a, a->m) + incrementare (++a, a++)
<i>LegacyForwardIterator</i>	+ incrementare cu mai multe parcurgeri (parcurgerea de mai multe ori nu modifică conținutul)
<i>LegacyBidirectionalIterator</i>	+ decrementare (--a, a--, *a--)
<i>LegacyRandomAccessIterator</i>	+ acces aleator la elemente(a+n, n+a, a-n, a-b, a<b, a>b, a<=b, a>=b, a+=n, a-=n, a[n], cu n întreg)
<i>LegacyContiguousIterator</i>	+ elementele consecutive se găsesc în locații de memorie consecutive
<i>LegacyOutputIterator</i>	scriere – poate fi dereferențiată ca o lvalue (*a=t, *a++=t)

## 2. Iteratori (Iterators) – funcții

```
std::vector<int> v{ 3, 1, 4 };
```

```
template<class InputIt, class Distance> void advance(InputIt& it, Distance n);  
// avanseaza iteratorul cu n pozitii  
auto vi = v.begin();  
std::advance(vi, 2);
```

```
template< class InputIt > typename std::iterator_traits<InputIt>::difference_type distance(  
InputIt first, InputIt last );  
// returneaza numarul de incrementari necesare pentru a ajunge de la first la last  
std::distance(v.begin(), v.end())
```

```
template< class InputIt> constexpr InputIt next(InputIt it, typename  
std::iterator_traits<InputIt>::difference_type n = 1);  
// returneaza al n-lea succesor al iteratorului it  
auto nx = std::next(v.begin(), 2);
```

```
template<class BidirIt> constexpr BidirIt prev(BidirIt it, typename  
std::iterator_traits<BidirIt>::difference_type n = 1);  
// returneaza al n-lea predecesor al iteratorului it  
auto nx = std::prev(v.end(), 2);
```

# 3. Algoritmi (algorithms)

 def Algoritmi = bibliotecă de funcții pentru numeroase mecanisme

“Câțiva” algoritmi importanți:

<code>for_each</code>	<code>count</code>	<code>find</code>	<code>search</code>	<code>copy</code>
<code>move</code>	<code>fill</code>	<code>transform</code>	<code>remove</code>	<code>replace</code>
<code>swap</code>	<code>reverse</code>	<code>sample</code>	<code>unique</code>	<code>is_sorted</code>
<code>sort</code>	<code>partial_sort</code>	<code>binary_search</code>	<code>merge</code>	<code>includes</code>
<code>set_difference</code>	<code>set_intersection</code>	<code>set_union</code>	<code>make_heap</code>	<code>push_heap</code>
<code>pop_heap</code>	<code>sort_heap</code>	<code>max</code>	<code>max_elem</code>	<code>min</code>
<code>min_elem</code>	<code>minmax</code>	<code>clamp</code>	<code>equal</code>	<code>accumulate</code>

Mai multe detalii aici: <https://en.cppreference.com/w/cpp/algorithm>

# Sfârșit capitol 6