

# **PROGRAMAREA ORIENTATĂ OBIECT C++**

**Conf.univ.dr. Ana Cristina DĂSCĂLESCU**

**Universitatea Titu Maiorescu**

### ➤ STL

- Este o bibliotecă de șabloane utilă pentru crearea și managementul structurilor dinamice de date (tablou, liste, mulțime, tabelă de asocieri etc).
- STL are o arhitectură bazată pe următoarele componente:
  - Containere
  - Algoritmi polimorfici
  - Iteratori
- Programele dezvoltate folosind STL beneficiază de o viteză de dezvoltare și o viteză de executare sporite, fiind mai eficiente, mai robuste, mai portabile și mai ușor de modificat.

### ➤ Containerere

- Un *container* este un obiect care grupează mai multe elemente într-o structura unitară.
- Intern, elementele dintr-un container se află într-o relație specifică unei structuri de date (lineară, asociativă, arborescentă etc.), astfel încât asupra lor se pot efectua operații de căutare, adăugare, modificare, ștergere, parcurgere etc.
- În STL există trei tipuri de containere:
  - *containere secvențiale* (vector, listă, coadă cu două capete – deque)
  - *containere asociative* (mulțime, relație)
  - *containere adaptor* (queue, stack, priority queue)
- Toate template-urile sunt definite în namespace-ul standard și au fișiere-antet de forma <vector>, <stack>, etc

### ➤ **Containere secvențiale**

- Elementele din colecție sunt memorate într-o ordine strict liniară.
- Elementele se regăsesc în ordinea în care sunt adăugate.
- Accesarea elementelor se realizează pe baza poziției acestora în cadrul containerului.
- Clasele predefinite care modelează containere secvențiale sunt:
  - *vector*
  - *deque*
  - *list*
- Fiecare clasă predefinită conține următoarele tipuri de constructori:
  - impliciți
  - cu un argument ce specifică dimensiunea containerului
  - de copiere
  - cu argumente care specifică elementele dintr-un alt container

## ➤ **Șablonul <vector>**

- Stochează datele într-o **zonă continuă de memorie** ce este redimensionabilă.
- Elementele sunt alocate dinamic.
- Prezintă eficiență pentru operațiile de accesare, respectiv inserare la sfârșitul vectorului.

### ▪ **Constructori**

Constructor	Efect	Complexitate
<code>vector&lt;T&gt;v</code>	crează un vector vid	<b>O(1)</b>
<code>vector&lt;T&gt;v(n)</code>	crează un vector cu <b>n</b> elemente	<b>O(n)</b>
<code>vector&lt;T&gt;v(n,val)</code>	crează un vector cu <b>n</b> elemente inițializate cu val	<b>O(n)</b>
<code>vector&lt;T&gt;v(v1)</code>	crează un vector inițializat cu vectorul v1	<b>O(n)</b>

- Operații specifice șablonului <vector>

Accesor	Efect	Complexitate
<b>v[i]</b>	întoarce elementul i	<b>O(1)</b>
<b>v.front()</b>	întoarce primul element	<b>O(1)</b>
<b>v.back()</b>	întoarce ultimul element	<b>O(1)</b>
<b>v.capacity()</b>	întoarce numărul maxim de elemente	<b>O(1)</b>
<b>v.size()</b>	întoarce numărul curent de elemente	<b>O(1)</b>
<b>v.empty()</b>	întoarce true dacă vectorul este vid	<b>O(1)</b>
<b>v.begin()</b>	întoarce un iterator la începutul vectorului	<b>O(1)</b>
<b>v.end()</b>	întoarce un iterator după sfârșitul vectorului	<b>O(1)</b>

### ▪ Operații specifice șablonului <vector>


Modificator	Efect	Complexitate
<code>v.push_back(val)</code>	adaugă o valoare la sfârșit	<b>O(1)</b>
<code>v.insert(iter,val)</code>	inserează valoarea în poziția indexată de iterator	<b>O(n)</b>
<code>v.pop_back()</code>	șterge valoarea de la sfârșit	<b>O(1)</b>
<code>v.erase(iter)</code>	șterge valoarea indexată de iterator	<b>O(n)</b>
<code>v.erase(iter1,iter2)</code>	șterge valorile din domeniul de iteratori	<b>O(n)</b>

- `T at(int index)` – returnează elementul aflat pe poziția index
- `T front()`, `T back()` – returnează primul element, respectiv ultimul element

### ➤ Exemplu

```
vector<int> vi;  
for (int i = 1; i <= 5; i++)  
    vi.push_back(i);  
for (int i = 1; i <= 5; i++)  
    cout<<vi[i];
```

```
.....  
vector<Persoana> vp;  
Persoana ob;  
vp.push_back(ob);  
for (int i = 1; i <= 5; i++)  
    cout<<vp[i];
```



Clasa Persoana conține  
supraîncărcarea  
operatorului <<



### ➤ Șablonul de tip `<list>`

- Este o colecție de elemente ordonate care permite inclusiv memorarea elementelor duplicate.
- Modelează o structură de date de tip listă dublu înlănțuită
- Elementele sunt stocate într-un spațiu de memorie necontiguă!!!
- Elementele nu pot fi accesate direct
- Permite inserarea elementelor la ambele capete
- Este util pentru stocarea obiectelor mari sau pentru aplicații care necesită inserări sau ștergeri multiple în interiorul listei.

### ➤ Operații specifice șablonului <list>

Accesor	Efect	Complexitate
<b>L.front()</b>	întoarce primul element	<b>O(1)</b>
<b>L.back()</b>	întoarce ultimul element	<b>O(1)</b>
<b>L.size()</b>	întoarce numărul curent de element	<b>O(1)</b>
<b>L.empty()</b>	întoarce <b>true</b> dacă lista este vidă	<b>O(1)</b>
<b>L.begin()</b>	întoarce un iterator la începutul listei	<b>O(1)</b>
<b>L.end()</b>	întoarce un iterator după sfârșitul liste	<b>O(1)</b>

Modificator	Efect	Complexitate
<b>L.push_front(val)</b>	adaugă o valoare la început	<b>O(1)</b>
<b>L.push_back(val)</b>	adaugă o valoare la sfârșit	<b>O(1)</b>
<b>L.insert(iter,val)</b>	inserează valoarea în poziția indexată de iterator	<b>O(1)</b>
<b>L.pop_front()</b>	șterge valoarea de la început	<b>O(1)</b>
<b>L.pop_back()</b>	șterge valoarea de la sfârșit	<b>O(1)</b>
<b>L.erase(iter)</b>	șterge valoarea indexată de iterator	<b>O(1)</b>
<b>L.erase(iter1,iter2)</b>	șterge valorile din domeniul de iteratori	<b>O(1)</b>
<b>L.remove(val)</b>	șterge toate aparițiile lui <b>val</b> în listă	<b>O(n)</b>
<b>L.remove_if(pred)</b>	șterge toate elementelele care satisfac predicatul	<b>O(n)</b>
<b>L.reverse(pred)</b>	inversează lista	<b>O(n)</b>

### ➤ Șablonul <deque>

- Modelează o structură de tip listă ce poate fi accesată la ambele capete
- Ocupă o zonă de memorie necontiguă, împărțită în blocuri mari contigui, alocate pe măsura extinderii containerului și gestionate uzual prin vectori de pointeri
- Permite toate operațiile de bază specifice șablonului vector și în plus, permite operațiile `push_front`, respectiv `pop_front`.
- Structura este eficientă pentru operații de tip `FIFO`, permitând adresarea unui element prin index

### ➤ Example:

```
list<int> lista;

for(int i = 0; i < 10; ++i)
    lista.push_back(i * 2);

cout << "Primul element " << lista.front();
cout << "Ultimul element: " << lista.back();

.....
deque<int> lista;
lista.push_back(10);
lista.push_front(20);
lista.push_back(30);
lista.push_front(15);
cout << "Numar elemente: " << lista.size();
```

### ➤ **Iteratori**

- Pentru a parcurge elementele unui container sunt necesare următoarele informații:
  - adresa primului element
  - modalitatea de a obține următorul element celui curent
  - adresa ultimului element
- O soluție unitară pentru a parcurge elementele unui container este dată de **iteratori**
- Rolul principal al iteratorilor este de a accesa elementele unui container, indiferent de tipul acestuia!!!
- **Iteratorul** este un obiect care permite idirectarea elementelor dintr-un container (pointer către un obiect)

- Tipuri de iteratori

- `iterator` - permite modificarea elementelor pe măsură ce acestea sunt accesate
- `const_iterator` – nu permite modificarea elementelor indirectate

- Definirea iteratorilor de către programator

```
container<T>:: iterator it;  
container<T>::const_iterator cit;  
container<T>::reverse_iterator rit;
```

- Definirea iteratorilor prin metodele specifice containerului

```
begin(), rbegin()  
end(), rend()
```

### ➤ Operații specifice iteratorilor

Operație	Intrare	Ieșire	Intrare/Ieșire	Bidirecționali	Acces direct	Exemplu
<b>++</b>	*	*	*	*	*	it++, ++it
<b>=</b>	*	*	*	*	*	it1 = it2
<b>==, !=</b>	*		*	*	*	it1 == it2
<b>* (rvalue), -&gt;</b>	*		*	*	*	*it
<b>*(lvalue)</b>		*	*	*	*	*it = val
<b>--</b>				*	*	it--, --it
<b>+, -, &gt;, &gt;=, &lt;, &lt;=</b>					*	it + d, d + it
<b>[]</b>					*	it[d]

### ➤ Example

```
list<int> lista;
```

```
for(int i = 0; i < 10; ++i)  
    lista.push_back(i * 2);
```

```
list<int> :: iterator it;
```

Se atașează listei un  
iterator

```
for(it = lista.begin(); it != lista.end(); ++it)  
    cout << *it;
```

Parcurgerea containerului



```
list<int> list1;
```

```
// utilizare assign() pentru inserare multiplă  
list1.assign(3, 2);
```



Inserează 3 valori egale cu 2

```
// inițializarea unui iterator  
list<int>::iterator it = list1.begin();
```

```
// poziționarea iteratorului pe poziția 2  
advance(it, 2);
```

```
// inserarea valorii 5 pe poziția a 3-a  
list1.insert(it, 5);
```

```
for (list<int>::iterator i = list1.begin();  
     i != list1.end();  
     i++)  
    cout << *i << " ";
```



Afișarea listei

### ➤ Containerul asociativ de tip mulțime <set>

- Șablonul <set> modelează o colecție de elemente care nu conțin duplicate, respectiv o colecție de tip mulțime.
- Fiecare element din container este o cheie unică!!!
- Permite stocarea și regăsirea rapidă a elementelor.
- Ordinea elementelor nu este dată de ordinea înserarii lor în container, ci de către o funcție comparator!!!
- Containerul asociativ set este implementat intern prin arbori binari de căutare

## ➤ Operații specifice șablonului <set>

Constructor	Efect	Complexitate
<code>map&lt;tip_ch,tip_val,comp_ch&gt;r</code>	crează o relație vidă; sortarea elementelor după cheie se face cu <b>comp_ch</b>	<b>O(1)</b>

Accesor	Efect	Complexitate
<code>r[ch]</code>	întoarce valoarea accesată prin cheie	<b>O(log n)</b>
<code>r.find(ch)</code>	întoarce un iterator la perechea cheie-valoare ( <b>r.end()</b> , dacă nu găsește cheia în <b>r</b> )	<b>O(log n)</b>
<code>r.size()</code>	întoarce numărul curent de elemente	<b>O(1)</b>
<code>r.empty()</code>	întoarce <b>true</b> dacă relația este vidă	<b>O(1)</b>
<code>r.begin()</code>	întoarce un iterator la prima pereche din <b>r</b>	<b>O(1)</b>
<code>r.end()</code>	întoarce un iterator la ultima pereche din <b>r</b>	<b>O(1)</b>

Modificator	Efect	Complexitate
<code>r[ch]=val</code>	memorează perechea cheie-valoare în <b>r</b>	<b>O(log n)</b>
<code>m.insert(pereche)</code>	are același efect	<b>O(log n)</b>

### ➤ Exemplu

```
class Persoana
{
    int varsta;
    char nume[50];
public:
    Persoana(char* nume, int varsta);
    bool operator<(const Persoana p2)
    {
        return varsta<p2.varsta;
    }
    char* getNume();
    int getVarsta();
}
```

```
set<Persoana> echipa;
Persoana p1("Matei", 23);
Persoana p2("Daniel", 20);

echipa.insert(p1);
echipa.insert(p2);

set<Persoana>::iterator it;

for(it=echipa.begin(); it!=echipa.end(); it++)
    cout<<(*it).getNume()<<" "
        <<(*it).getVarsta();
```

### ➤ Algoritmi polimorfici

- Sunt funcții independente care implementează operații specifice colecțiilor de date
- Algoritmii sunt generici, respectiv nu depind de tipul de date
- Sunt definiți în antetul `<algorithm>`

### ➤ Algoritmul sort

- definește o funcție care sortează elementele unui container
- este o implementare a lui `Quick Sort` ( $O(n \log n)$ )
- criteriul de sortare este definit printr-o funcție comparator ce este transmisă ca argument al funcției sort
- funcția comparator primește doua argumente de tipul elementelor aflate în container și returnează o valoare de tipul `bool`

```
qsort(container.begin(), container.end(), comparator);
```

### ➤ Exemplu

```
bool cmpPersoana(Persoana ob1, Persoana ob2)
{
    return ob1.getVarsta() < ob2.getVarsta();
}

int main()
{
    Vector<Persoana> vp;
    vp.insert(p1);
    vp.insert(p2);
    sort(v.begin(), v.end(), cmpPersoana);
}
```

### ▪ Algoritmul **find**

- Caută un element într-o colecție și returnează un iterator ce referă elemental căutat (prima apariție)
- Dacă elementul nu se află în colecție, atunci se returnează un iterator ce marchează sfârșitul colecției)

```
vector<int> vec { 10, 20, 30, 40 };
int ser = 30;
it = find (vec.begin(), vec.end(), ser);
if (it != vec.end())
{
    cout << "Element " << ser << " found at position : " ;
    cout << it - vec.begin();
}
else
    std::cout << "Element not found";
```