# Structuri de date în C++ - Biblioteca STL

Universitatea "Transilvania" din Brașov

April 28, 2017

# Biblioteca STL - Generalități

- STL = Standard Template Library.
- set ce clase template C++ care implementează algoritmi și structuri de date frecvent utilizate, precum vectori, liste, cozi sau stive.
- Componente principale:
  - Containere - utilizate pentru managementul multimilor de obiecte - vector, list, etc.
  - Algoritmi - se aplică asupra containerelor - ex. sortare, căutare.
  - Iteratori - permit navigarea printre obiectele unui container.

# Containere - General

- **Containere**: sunt tipuri de date din STL care conțin date
- **Adaptori** -***adapters***: - sunt tipuri de date din STL care adaptează un container la o interfață specifică (coadă, stivă).

# Containere simple - *pair*

**Pair**

- container asociativ simplu constând dintr-o pereche de elemente : *first* și *second*
- elemente de tip *pair* pot fi atribuite, copiate și comparate
- obiectele stocate într-un *map* sau *hash_map* sunt prin definiție de tip *pair*

# Containere simple - *pair*

**Clasa:** `template <class T1, class T2> struct pair;`

**Membrii: first** și **second**

**Funcții membre**

- **Constructori:**
  - constructor predefinit: `std::pair <int, int> coord1;`
  - constructor de inițializare `std::pair <int, int> coord2 (500,200);`
  - constructor de copiere
- Funcția **swap**: `coord1.swap(coord2);`

# Containere simple - *pair*

**Funcții non-membre**

- Funcția **make_pair**: `coord1 = std::make_pair(70, 100);`
- Funcția **get**:
  ```
  coord_x = std::get<0>(coord1);
  coord_y = std::get<1>(coord1);
  ```
- Operatorii relaționali. Comparația se face în ordinea componentelor.

# Containere simple - *tuple*

**tuple**

- o colecție de dimensiune fixă de elemente de diferite tipuri = generalizare pentru **pair**
- elemente de tip *tuple* pot fi atribuite, copiate și comparate

# Containere simple - *tuple*

**Clasa:** `template< class... Types > class tuple;`

**Funcții membre**

- **Constructori:**
  - constructor predefinit: `std::tuple <std::string, int, int, double> student;`
  - constructor de inițializare
    `std::tuple <std::string, int, int, double> student("Ionescu",10,9,9.50);`
  - constructor de copiere
- Funcția **swap**: `student.swap(stud2);`

# Containere simple - *tuple*

**Funcții non-membre**

- Funcția **make_tuple**: `student1 = std::make_tuple("Anton", 8,9,8.50);`
- Funcția **get**:
  `nume = std::get<0>(student1);`
  `nota1 = std::get<1>(student1); nota2 = std::get<2>(student2); ...`
- **tie**: `std::tie(a, b, c, d) = student1;`
- Operatorii relaționali. Comparația se face în ordinea componentelor.

**Observație:** - pentru restul funcțiilor membre vezi documentație online.

# Containere simple - *tuple*

```
#include <tuple>
#include <iostream>
void main()
{
      std::tuple<int, int, double> t1;
      t1 = std::make_tuple(10, 10, 2.5);
      int a, b; double c;
      std::tie(a, b, c) = t1;
      std::cout << "a=" << a << ", b=" << b << ", c=" << c;
}
```

# Containere de tip secvență - *vector*

**Vector**:

- container de tip secvență de dimensiune modificabilă.
- utilizează zone de memorie alăturate pentru stocarea elementelor.
- utilizează intern memorie alocată dimanic $\Rightarrow$ necesită uneori realocarea unei zone noi de memorie
- acces eficient la elemente, comparativ cu listele înlănțuite.

# Containere de tip secvență - *vector*

```
template <class T, class Alloc=allocator<T>> class vector;
```

**Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::vector<int> v:`
  - Contructor de inițializare:
    ```
    std::vector<int>v2 = {1, 2, 3};
    std::vector<int>v3(10);
    std::vector<int>v4(10,5);
    std::vector<int>v5(s.begin(),s.end());
    ```
  - Constructor de copiere:
    ```
    std::vector<int>v6(v4);
    std::vector<int>v7=v6;
    ```

# Containere de tip secvență - *vector*

**Vector**: `std:: vector<int>v;`

**Funcții membre - de acces la elemente:**

- funcția **at**:

```
int i = v.at(5);
```

- operatorul **[]**:

```
int i = v[5];
```

- funcțiile **front** și **back**:

```
std::cout<<v.front()<<v.back();
```

# Containere de tip secvență - *vector*

**Vector**: `std::vector<int>v;`

**Funcții membre - de verificare a dimensiunii:**

- funcția **size**:

```
int i = v.size();
```

- operatorul **capacity**:

```
int i = v.capacity();
```

- funcția **empty**:

```
bool el=v.empty();
```

# Containere de tip secvență - *vector*

**Vector**: `std:: vector<int>v=`$\{1,2,3,4\}$`;`

**Funcții membre - de gestionare a dimensiunii:**

- funcția **reserve**:

```
v.reserve(10);
```

- funcția **clear**:

```
v.clear();
```

- funcția **resize**:
  - `v.resize(6);` - obținem $v = \{1,2,3,4,0,0\}$
  - `v.resize(3);` - obținem $v = \{1,2,3\}$

# Containere de tip secvență - *vector*

**Vector**: `std:: vector<int>v={1,2,3,4};`

**Funcții membre - de adăugare / extragere de elemente:**

- funcția **push_back**:

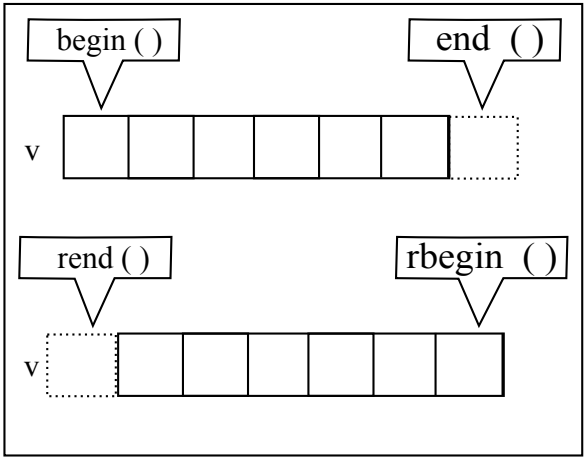
  `v.push_back(5);` - obținem $v = \{1, 2, 3, 4, 5\}$

- funcția **pop_back**:

  `v.pop_back();` - obținem $v = \{1, 2, 3\}$

- funcția **insert**: - permite inserția de elemente pe pe o poziție date de un iterator.
- funcția **implace**: - creearea și inserarea unui element
- funcția **implace_back**: - creearea și adăugarea unui element la sfarșit

# Containere de tip secvență - *vector*




**Funcții membre -iteratori**

- permit iterarea prin elementele unui vector/ liste înlănțuite
- principlalii iteratori **begin, end, rbegin, rend**
- funcția **insert**: - permite inserția de elemente pe o poziție date de un iterator.
- funcția **erase**: - permite ștergerea unui element de pe o poziție dată de un iterator.

# Containere de tip secvență - *vector*

**Funcții membre -iteratori - Exemplu**

```
#include <iostream>
#include <vector>
void main ()
{
      std::vector<int> v;
      for(int i=1; i<=10; i++)
         v.push_back(i);
      std::vector<int>::iterator it = v.begin();
      it+=3; advance(it,3);
      v.insert(it,100);
      ++it;
      v.erase(it);
}
```

# Containere de tip secvență - *list*

**List**:

- listă dublu înlănțuită ale cărei elemente nu sunt memorate în zone de memorie alăturate
- acces în timp liniar la elemente.
- inserție și ștergere de elemente în timp constant.

# Containere de tip secvență - *list*

**List**: `template<class T, class Allocator = std::allocator<T>> class list;` **Funcții membre**

- **Constructori**:

# Containere de tip secvență - *list*

**List**: `template<class T, class Allocator = std::allocator<T>> class list;` **Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::list<int> L1;`- listă vidă

# Containere de tip secvență - *list*

**List**: `template<class T, class Allocator = std::allocator<T>> class list;` **Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::list<int> L1;`- listă vidă
  - Contructor de inițializare:
    `std::list<int>L2(4,10);`- listă cu 4 elemente cu val 10
    `std::list<int>L3(L2.begin(), L2.end());`- listă ce conține aceleași elemente ca L2

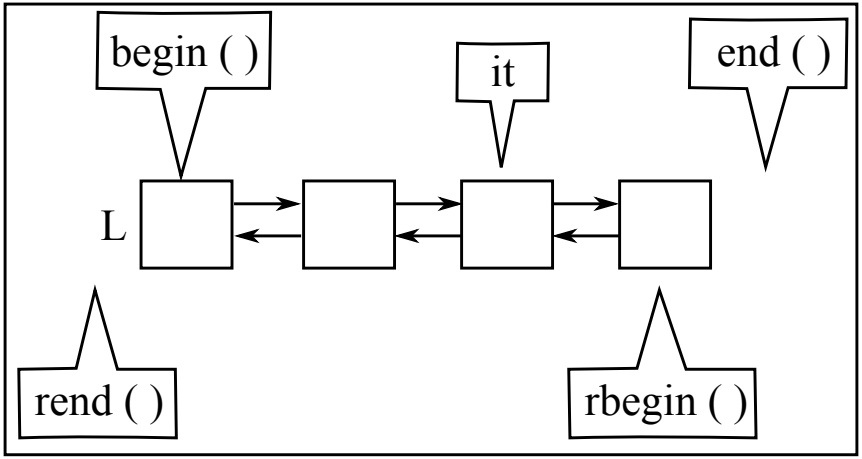
# Containere de tip secvență - *list*

**List**: `template<class T, class Allocator = std::allocator<T>> class list;` **Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::list<int> L1;`- listă vidă
  - Contructor de inițializare:
    `std::list<int>L2(4,10);`- listă cu 4 elemente cu val 10
    `std::list<int>L3(L2.begin(), L2.end());`- listă ce conține aceleași elemente ca L2
  - Constructor de copiere:
    `std::list<int>L4(L3);`

# Containere de tip secvență - *list*

**Funcții membre- iteratori**



# Containere de tip secvență - *list*

**List**: `std::list<int>L(5,10);`

**Accesul la al *N*-lea element al listei** - cu ajutorul iteratorilor

- cu funcția **advance**:

```
if(L.size() > N)
{
   std::list<int>::iterator it = L.begin()
   std::advance(it,N)
   std::cout<<*it;
}
```

# Containere de tip secvență - *list*

**List**: `std::list<int>L(5,10);`

**Accesul la al *N*-lea element al listei** - cu ajutorul iteratorilor

- cu funcția **advance**:

```
if(L.size() > N)
{
   std::list<int>::iterator it = L.begin()
   std::advance(it,N)
   std::cout<<*it;
}
```

- cu un ciclu for:

```
for(i=0,it=L.begin();i<N && it!=L.end();i++,it++) ;
if(it==L.end()) std::cout<<"N prea mare";
else std::cout<<*it;//
```

# Containere de tip secvență - *list*

**List**: `std::list<int>L(5,10);`

**Accesul la capetele listei** - cu ajutorul iteratorilor

- `int prim = L.front();`
- `int ultim = L.back();`

# Containere de tip secvență - *list*

**List**: `std::list<int>L;`

**Funcții membre - de verificare a dimensiunii:**

- funcția **size**:

```
int i = L.size();
```

- funcția **empty**:

```
bool el=L.empty();
```

# Containere de tip secvență - *list*

**List**: Se consideră lista L: 1 2 3

**Funcții membre - modificare a listei**

- funcții de adăugare a unui element:
  `L.push_back(5);` - L: 1 2 3 5
  `L.push_front(4);` - L: 4 1 2 3

- funcții de eliminare a unui element:
  `L.pop_back();` - L: 1 2
  `L.pop_front();` - L: 2 3

# Containere de tip secvență - *list*

**List**:

**Funcții membre - modificare a listei**

- inserția în listă:

# Containere de tip secvență - *list*

**List**:

**Funcții membre - modificare a listei**

- inserția în listă:
  - a unui element pe o anumită poziție
    `iterator insert (iterator position, const value_type& val);`

# Containere de tip secvență - *list*

**List**:

**Funcții membre - modificare a listei**

- inserția în listă:
  - a unui element pe o anumită poziție
    `iterator insert (iterator position, const value_type& val);`
  - a unui element de *n* ori începând de la o poziție dată
    `void insert (iterator position, size_type n, const value_type& val);`

# Containere de tip secvență - *list*

**List**:

**Funcții membre - modificare a listei**

- inserția în listă:
  - a unui element pe o anumită poziție
    `iterator insert (iterator position, const value_type& val);`
  - a unui element de *n* ori începând de la o poziție dată
    `void insert (iterator position, size_type n, const value_type& val);`
  - a unor elemente din alt container
    `void insert (iterator position, InputIterator first, InputIterator last);`

# Containere de tip secvență - *list*

**Exemplu**- inserția unor elemente într-o listă L

```
#include <iostream>
#include <list>
#include <vector>
void main (){
      std::list<int> L;
      std::list<int>::iterator it;
      for(int i=1; i<=5; i++)
         L.push_back(i);
      it=L.begin(); ++it;
      L.insert(it,10);
      L.insert(it,2,20);
      --it;
      std::vector<int> v (2,30);
      L.insert (it,v.begin(),v.end());
}
```

# Containere de tip secvență - *list*

**List**:

**Funcții membre - modificare a listei**

- Eliminarea unui element din listă: - funcția **remove**

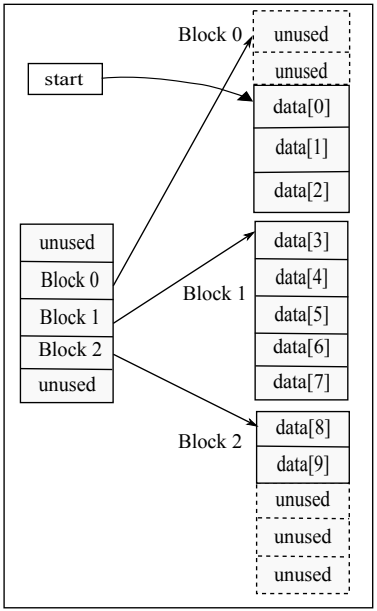
```
L.remove(3)
```

# Containere de tip secvență - *deque*

**Dequeue** - *double-ended queue*

- organizată ca un *array* de *array*-uri sau o listă înlănțuită de *array*-uri
- permite adăugare și extragere la ambele capete în complexitate $O(1)$
- permite acces la orice element prin poziție în complexitate $O(1)$
- necesită pentru inserție sau ștergerea unui element din interior cel mult $N/2$ operații

# Containere de tip secvență - *deque*




- blocuri de dimensiune constantă
- **push_back()** adaugă un element la ultimul bloc sau se alocă un nou bloc în care se inserează; similar pentru **push_front()**
- funcția **insert** deplasează elementele de la punctul de inserție înspre cel mai apropiat capăt.
- container utilizat pentru implementarea structurilor **queue** - coadă - și **stack** - stivă.

# Containere de tip secvență - *deque*

**Deque**: `template<class T, class Allocator = std::allocator<T>> class deque;` **Funcții membre**

- **Constructori**:

# Containere de tip secvență - *deque*

**Deque**: `template<class T, class Allocator = std::allocator<T>> class deque;` **Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::deque<std::string> DQ1;`- structură vidă

# Containere de tip secvență - *deque*

**Deque**: `template<class T, class Allocator = std::allocator<T>> class deque;` **Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::deque<std::string> DQ1;`- structură vidă
  - Contructor de inițializare:
    `std::deque<std::string>DQ2(3, "super");`- 3 copii ale cuvântului "super".
    `std::deque<std::string>DQ3={"ana", "este", "aici "};`- inițializare cu 3 elemente.
    `std::deque<std::string>DQ4(DQ3.begin(), DQ3.end());`- aceleași elemente ca DQ3

# Containere de tip secvență - *deque*

**Deque**: `template<class T, class Allocator = std::allocator<T>> class deque;` **Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::deque<std::string> DQ1;`- structură vidă
  - Contructor de inițializare:
    `std::deque<std::string>DQ2(3, "super");`- 3 copii ale cuvântului "super".
    `std::deque<std::string>DQ3={"ana", "este", "aici "};`- inițializare cu 3 elemente.
    `std::deque<std::string>DQ4(DQ3.begin(), DQ3.end());`- aceleași elemente ca DQ3
  - Constructor de copiere:
    `std::deque<std::string>DQ5(DQ4);`

# Containere de tip secvență - *deque*

**Deque**: `std:: deque<int>DQ;`

**Funcții membre - de acces la elemente:**

- funcția **at**:

```
int i = DQ.at(5);
```

- operatorul **[]**:

```
int i = DQ[5];
```

- funcțiile **front** și **back**:

```
std::cout<<DQ.front()<<DQ.back();
```

# Containere de tip secvență - *deque*

**Deque**: `std:: deque<int>DQ;`

**Funcții membre - de gestiune a dimensiunii**

- **size**

**Funcții membre - de modificare a structurii**

# Containere de tip secvență - *deque*

**Deque**: `std:: deque<int>DQ;`

**Funcții membre - de gestiune a dimensiunii**

- **size**
- **empty**

**Funcții membre - de modificare a structurii**

# Containere de tip secvență - *deque*

**Deque**: `std:: deque<int>DQ;`

**Funcții membre - de gestiune a dimensiunii**

- **size**
- **empty**
- **clear**

**Funcții membre - de modificare a structurii**

# Containere de tip secvență - *deque*

**Deque**: `std:: deque<int>DQ;`

**Funcții membre - de gestiune a dimensiunii**

- **size**
- **empty**
- **clear**
- **resize**

**Funcții membre - de modificare a structurii**

# Containere de tip secvență - *deque*

**Deque**: `std:: deque<int>DQ;`

**Funcții membre - de gestiune a dimensiunii**

- **size**
- **empty**
- **clear**
- **resize**

**Funcții membre - de modificare a structurii**

- **push_back**, **push_front**

# Containere de tip secvență - *deque*

**Deque**: `std:: deque<int>DQ;`

**Funcții membre - de gestiune a dimensiunii**

- **size**
- **empty**
- **clear**
- **resize**

**Funcții membre - de modificare a structurii**

- **push_back**, **push_front**
- **pop_back**, **pop_front**

# Containere de tip secvență - *deque*

**Deque**: `std:: deque<int>DQ;`

**Funcții membre - de gestiune a dimensiunii**

- **size**
- **empty**
- **clear**
- **resize**

**Funcții membre - de modificare a structurii**

- **push_back**, **push_front**
- **pop_back**, **pop_front**
- **insert**

# Containere de tip secvență - *deque*

**Deque**: `std:: deque<int>DQ;`

**Funcții membre - de gestiune a dimensiunii**

- **size**
- **empty**
- **clear**
- **resize**

**Funcții membre - de modificare a structurii**

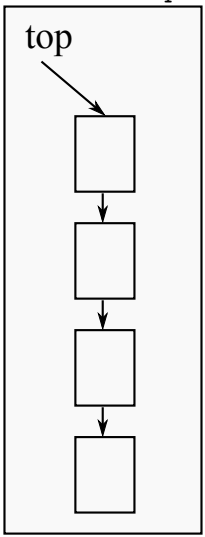
- **push_back**, **push_front**
- **pop_back**, **pop_front**
- **insert**
- **erase**

# Adaptori de containere

- tipuri de date din STL care adptează containere pentru o interfață specifică
- ex: coadă, stivă, coadă de priorități

# Adaptori de containere - stiva

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




- disciplină de intrare/ieșire LIFO - acces doar prin vârf
- utilizează un container pentru stocarea elementelor - în mod predefinit *dequeue*
- elementele sunt inserate/ extrase din capătul *back* al containerului care este *top* -ul stivei

# Adaptori de containere - stiva

**Funcții membre**

- **Constructori**:

# Adaptori de containere - stiva

**Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::stack<int> Stiva1;`- stivă vidă

# Adaptori de containere - stiva

**Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::stack<int> Stiva1;`- stivă vidă
  - Constructor predefinit cu modificarea tipului de container: `std::stack<int,std::vector<int>> Stiva2;`

# Adaptori de containere - stiva

**Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::stack<int> Stiva1;`- stivă vidă
  - Constructor predefinit cu modificarea tipului de container:
    `std::stack<int,std::vector<int>> Stiva2;`
  - Contructor de inițializare:
    `std::deque<int>DQ(3,10);`
    `std::stack<int> Stiva3(DQ);`

    - utilizând un vector drept container:
    `std::stack<int,std::vector<int>>Stiva4(Stiva2);`

# Adaptori de containere - stiva

**Funcții membre**

- **empty** - testează dacă stiva este vidă

# Adaptori de containere - stiva

**Funcții membre**

- **empty** - testează dacă stiva este vidă
- **size** - determină câte elemente sunt în stivă

# Adaptori de containere - stiva

**Funcții membre**

- **empty** - testează dacă stiva este vidă
- **size** - determină câte elemente sunt în stivă
- **top** - returnează elementul din vârf

# Adaptori de containere - stiva

**Funcții membre**

- **empty** - testează dacă stiva este vidă
- **size** - determină câte elemente sunt în stivă
- **top** - returnează elementul din vârf
- **push** - pune un element în stivă

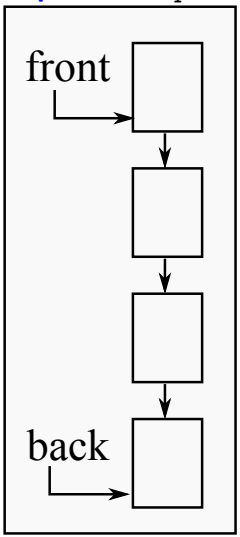
# Adaptori de containere - stiva

**Funcții membre**

- **empty** - testează dacă stiva este vidă
- **size** - determină câte elemente sunt în stivă
- **top** - returnează elementul din vârf
- **push** - pune un element în stivă
- **pop** - elimină elementul din vârful stivei.

# Adaptori de containere - coada

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




- disciplină de intrare/ieșire FIFO - acces prin cele două capete
- utilizează un container pentru stocarea elementelor - în mod predefinit *dequeue*
- elementele sunt inserate în capătul *back* și extrase din capătul *front* al containerului cozii

# Adaptori de containere - coada

**Funcții membre**

- **Constructori**:

# Adaptori de containere - coada

**Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::queue<int> Coada1;`- coadă vidă

# Adaptori de containere - coada

**Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::queue<int> Coada1;`- coadă vidă
  - Constructor predefinit cu modificarea tipului de container: `std::queue<int,std::list<int>> Coada2;`

# Adaptori de containere - coada

**Funcții membre**

- **Constructori**:
  - Constructor predefinit: `std::queue<int> Coada1;`- coadă vidă
  - Constructor predefinit cu modificarea tipului de container:
    `std::queue<int,std::list<int>> Coada2;`
  - Contructor de inițializare:
    `std::deque<int>DQ(3,10);`
    `std::queue<int> Coada3(DQ);`

    - utilizând o listă drept container:
    `std::queue<int,std::list<int>>Coada4(Coada2);`

# Adaptori de containere - coada

**Funcții membre**

- **empty** - testează dacă coada este vidă

# Adaptori de containere - coada

**Funcții membre**

- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă

# Adaptori de containere - coada

**Funcții membre**

- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă
- **front** și **back**

# Adaptori de containere - coada

**Funcții membre**

- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă
- **front** și **back**
- **push** - pune un element în coadă

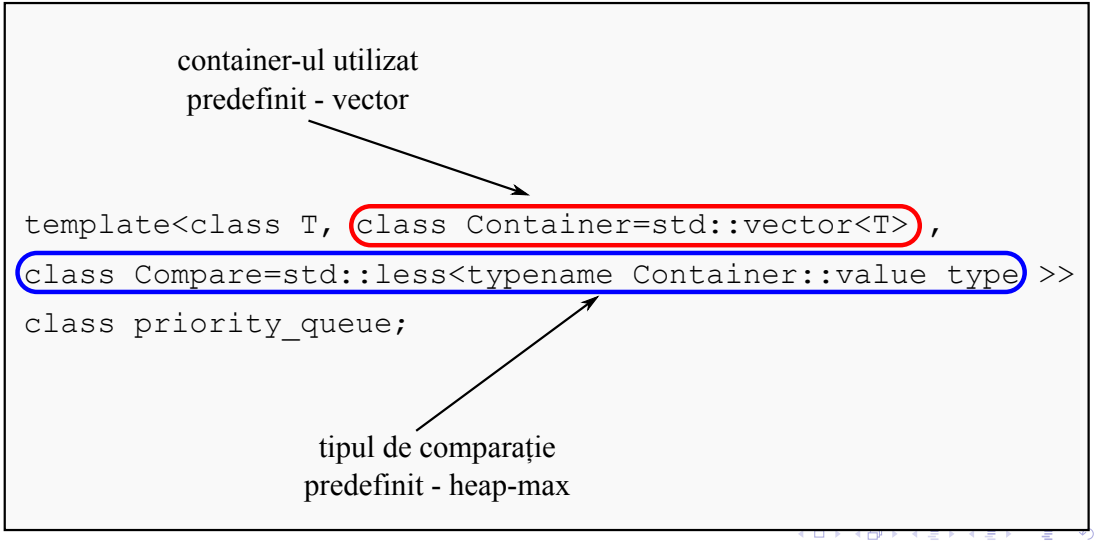
# Adaptori de containere - coada

**Funcții membre**

- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă
- **front** și **back**
- **push** - pune un element în coadă
- **pop** - elimină elementul din coadă.

# Adaptori de containere - coadă de priorități

**priority_queue**:



# Adaptori de containere - coadă de priorități

**priority_queue**:

- acces în timp constant la elementul de prioritate maximă
- se poate utiliza pentru comparație `std::greater<T>` pentru heap-min
- se poate defini o clasă proprie pentru comparația obiectelor din heap
- structură de heap

# Adaptori de containere - coadă de priorități

**priority_queue**: Funcții membre

- Constructor:
  - predefinit: `std::priority_queue<int> first;`
  - de inițializare

```
int v[]= {10,60,50,20};
std::priority_queue<int> second (v,v+4);

std::priority_queue<int, std::vector<int>, std::greater<int> >
third (v,v+4);
```

# Adaptori de containere - coadă de priorități

**priority_queue**: Funcții membre

- **empty** - testează dacă coada este vidă

# Adaptori de containere - coadă de priorități

**priority_queue**: Funcții membre

- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă

# Adaptori de containere - coadă de priorități

**priority_queue**: Funcții membre

- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă
- **top** - returnează elementul de prioritate maximă

# Adaptori de containere - coadă de priorități

**priority_queue**: Funcții membre

- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă
- **top** - returnează elementul de prioritate maximă
- **push** - pune un element în coadă

# Adaptori de containere - coadă de priorități

**priority_queue**: Funcții membre

- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă
- **top** - returnează elementul de prioritate maximă
- **push** - pune un element în coadă
- **pop** - elimină elementul de prioritate maximă.

# Containere asociative

- implementează structuri de date sortate ce permit căutare rapidă în complexitate $O(\log n)$.
- căutarea se realizează pe baza unei chei (nu prin poziție).
- principalele tipuri de containere asociative: *set* și *map*

# Containere asociative - Set

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

- mulțime sortată de elemente, care nu permit dubluri

# Containere asociative - Set

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

- mulțime sortată de elemente, care nu permit dubluri
- sortarea se face pe baza funcției de comparare **Compare**

# Containere asociative - Set

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

- mulțime sortată de elemente, care nu permit dubluri
- sortarea se face pe baza funcției de comparare **Compare**
  - funcții predefinite de comparație: `std::less<Key>`, `std::greater<Key>`

# Containere asociative - Set

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

- mulțime sortată de elemente, care nu permit dubluri
- sortarea se face pe baza funcției de comparare **Compare**
  - funcții predefinite de comparație: `std::less<Key>`, `std::greater<Key>`
  - se pot defini funcții proprii (vezi documentație online)

# Containere asociative - Set

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

- mulțime sortată de elemente, care nu permit dubluri
- sortarea se face pe baza funcției de comparare **Compare**
  - funcții predefinite de comparație: `std::less<Key>`, `std::greater<Key>`
  - se pot defini funcții proprii (vezi documentație online)
- *Key* - trebuie să fie un tip primitiv - int, double, string, etc (tipuri care permit comparația cu $<$)

# Containere asociative - Set

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

- mulțime sortată de elemente, care nu permit dubluri
- sortarea se face pe baza funcției de comparare **Compare**
  - funcții predefinite de comparație: `std::less<Key>`, `std::greater<Key>`
  - se pot defini funcții proprii (vezi documentație online)
- *Key* - trebuie să fie un tip primitiv - int, double, string, etc (tipuri care permit comparația cu $<$)
- implementat printr-un arbore binar de căutare care se auto-balansează - de obicei Roșu - Negru

# Containere asociative - Set

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

- mulțime sortată de elemente, care nu permit dubluri
- sortarea se face pe baza funcției de comparare **Compare**
  - funcții predefinite de comparație: `std::less<Key>`, `std::greater<Key>`
  - se pot defini funcții proprii (vezi documentație online)
- *Key* - trebuie să fie un tip primitiv - int, double, string, etc (tipuri care permit comparația cu $<$)
- implementat printr-un arbore binar de căutare care se auto-balansează - de obicei Roșu - Negru
- operații în timp logaritmic

# Containere asociative - Set

## Construcție

```
#include <set>
bool fncomp (int lhs, int rhs) {return lhs<rhs;}
struct classcomp {
        bool operator() (const int& lhs, const int& rhs) const
        {return lhs<rhs;}
};
int main (){
    std::set<int> first; // empty set of ints
    int myints[]= {10,2,13,41,5};
    std::set<int> second (myints,myints+5); // range
    std::set<int> third (second); // a copy of second
    std::set<int>fourth (second.begin(), second.end()); // iterator constr.
    std::set<int,classcomp> fifth; // class as Compare
    bool(*fn_pt)(int,int) = fncomp; //function pointer
    std::set<int,bool(*)(int,int)> sixth (fn_pt); // function pointer as Compare
 return 0;}
```

# Containere asociative - Set

## Construcție - parcurgere

```
#include <iostream>
#include <set>
#include <vector>
void main (){
    std:set<int> m1;
    std::vector<int> v ={ 4, 1, 15, 0, 21, 7, 16, 9 };
    for(int i = 0; i ¡ v.size(); i++)
       m1.insert(v[i]);
    std::set<int>::iterator it = m1.begin();
    for (; it != m1.end(); ++it)
       cout<< *it << " ";
    cout << endl;
}
```

# Containere asociative - Set

## Funcții membre - dimensiune set

- **empty**: verifică dacă struct este vidă
- **size**: returnează numărul de elemente din structură

# Containere asociative - Set

## Funcții membre - modificare set

- **insert**:

  `s.insert(5);` - inserează cheia 5

- **erase**: poate avea ca parametru:
  - un iterator ⇒ șterge elem. indicat de către iterator
  - o valoare ⇒ șterge elementul cu cheia dată de valoarea respectivă
- **clear**: golește structura
- **emplace**: creează și adaugă un element

# Containere asociative - Set

## Funcții membre - căutare

- **find**: returnează un iterator care indică elementul căutat sau capătul **end** al structurii
- **count**: numără apraițiile unei chei. Rezultatul poate fi 0 sau 1

# Containere asociative - Set

**Exemplu**

```
#include <iostream>
#include <set>
void main (){
      int vv[] = {4, 1, 15, 0, 21, 7, 16, 9};
      std::set<int>m1(vv, vv + 8);
      cout <<"caut:"; cin >> x;
      std::set<int>::iterator it = m1.find(x);
      if (it != m1.end()){
        cout << "gasit"; m1.erase(it);
      }
      else cout << "not gasit";
}
```

# Containere asociative - Map

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

- asemănătoare cu *set*, doar că se păstrează perechi (cheie, valoare)
- accesul și căutarea/ sortarea se fac pe baza cheii
- elementele sunt de tip *pair*
- metodele sunt similare ca pentru set
- acces direct la elemente prin opeatorul [ ], pe baza cheii

# Containere asociative - Map

**Exemplul 1**

```
#include <iostream>
#include <map>
void main (){
    std::map<char, int> mymap;
    std::map<char, int>::iterator it;
    mymap['a'] = 50; mymap['b'] = 100;
    mymap['c'] = 150; mymap['d'] = 200;
    it = mymap.find('b');
    if (it != mymap.end())
      mymap.erase(it);
    // print content:
    std::cout << "elements in mymap:" << endl;
    std::cout << "a => " << mymap.find('a')- >second <<endl;
    std::cout << "c => " << mymap.find('c')- >second <<endl;
    std::cout << "d => " << mymap.find('d')- >second <<endl;
}
```

# Containere asociative - Map

**Exemplul 2**

```
#include <iostream>
#include <map>
void main (){
    std::map<std::string, int> mymap2 = {{ "alpha", 0 },
                                         { "beta", 0 }, { "gamma", 0 }};
    mymap2.at("alpha") = 10;
    mymap2.at("beta") = 20;
    mymap2.at("gamma") = 30;
    for (auto x : mymap2) {
        std::cout << x.first <<": "<<x.second << endl;
}
```

# Containere asociative - unordered_map

**unordered_map** - `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;`

- stochează perechi (cheie, valoare)
- elem. nu sunt sortate, ci inserate pe baza cheii in *bucket*-uri
- sunt mai rapide la cautare decât map, dar mai puțin eficiente pentru statistici de ordine sau iterare pe o submulțime
- acces direct la elemente prin opeatorul [ ], pe baza cheii