Functii de STL (Standard template library)- Introducere

- **std::pair**
  - Scriere in C++:

    `pair<T1,T2>a` **,unde T1,T2 determina ce tipuri de date contine pair-ul.**

    Exemplu : `pair<int,int>a`**.**

    Pentru a accesa elementele unui pair vom folosi `a.first` pentru a accesa valoarea primului element si `a.second` pentru a accesa al doilea element .Pentru a face un pair putem folosi functia `make_pair(val1,val2)` sau putem folosi `{val1,val2}`;

    Exemplu:

    ```
    int x=7;
    int y=6;
    pair<int,int>a;
    pair<int,int>b;
    a=make_pair(x,y);
    b={x,y};
    //Ambele pair-uri vor avea aceleasi elemente
    //a.first=6;a.second=7;
    //b.first=6;b.second=7;
    ```

- **std::tuple**
  - Scriere in C++:

    Tuple este o varianta generalizata a pair-ului,iar ea se declara in forma `tuple<T1,T2,...,Ti>a`**,unde T1,T2,...,Ti sunt tipurie de date ce contine tuple-ul si i>=1;**

    Exemplu:`tuple<int,int,int>a`**;**

    Pentru a accesa elementele unui tuple vom folosi functia `get<k>()`,unde k este indicele la care se afla elementul in tuple(*indexarea in tuple se face de la*

0) .Pentru a adauga elemente in tuple avem aceeasi functie ca si la pair `make_tuple(val1,va2,...,valn)`sau `{va1,val2,...,valn}`;

Exemplu:

```
#include <iostream>
#include<tuple>///Biblioteca specifica pentru tuple
using namespace std;

int main()
{
tuple<int,int,int>a;
tuple<int,int,int>b;
a={6,7,8};
b=make_tuple(6,7,8);
cout<<get<0>(a)<<" "<<get<1>(a)<<" "<<get<2>(a)<<"\n";
//Se va afisa primul tuple a;
cout<<get<0>(b)<<" "<<get<1>(b)<<" "<<get<2>(b)<<"\n";
//Se va afisa al doiea tuple b;
return 0;
}
```

- # Random access iterator
  - **Un iterator cu acces aleatoriu este cea mai puternică categorie de iteratoare din C++. Acesta permite accesul direct la elementele unui container în timp constant, ceea ce înseamnă că poți sări la orice element din container fără a fi nevoie să parcurgi secvențial elementele anterioare(trece din pointer in pointer).El se poate scrie sub forma** `container::iterator nume iterator`

Exemplu:

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

int main()
{

std::vector<int> vec = {1, 2, 3, 4, 5};

// Obținem un iterator pentru începutul vectorului
```

```
std::vector<int>::iterator it = vec.begin();

// Sărim la al treilea element (indicele 2)
it += 2;

std::cout << *it << std::endl;
//Se afiseaza valoarea 3
return 0;
}
```

OBS:Iteratorul trece prin pointeri,nu prin indici!!! **Un pointer în C++ este o variabilă care stochează adresa de memorie a altei variabile. Cu alte cuvinte, un pointer "îndreaptă" către locația în care este stocată o valoare în memorie, în loc să stocheze acea valoare direct.**

Pentru a afla valoarea la care indica un pointer putem folosi `*numepointer`.

- Tipul de declarare ***auto***

Cand declaram o variabila de tip auto,compilatorul deduce automat ce fel de tip de date este folosit pentru acea variabila

Exemplu:

```
#include <iostream>
using namespace std;
int main() {
auto x=10;       // x se deduce ca fiind de tip int
auto y=3.14;     // y se deduce ca fiind de tip double
auto z="h";  // z se deduce ca fiind de tip char
cout<<x<<" "<<y<<" "<<z<<"\n";
return 0;
}
```

- STL Containers

1. std::vector

   - Scriere in C++:

     `vector<tip de date>a`

     Exemplu:`vector<int>a`

- Parcurgerea unui vector

  Vectorul este o structura de date care admite si parcurgerea prin indici,dar si prin iteratori

  Exemplu:

```
#include <iostream>
#include <vector>

using namespace std;

int main() {

   vector<int> vec = {1, 2, 3, 4, 5};

     // 1. Parcurgerea vectorului prin indecsi

   for (int i = 0; i < vec.size(); ++i) {
         cout << vec[i] << " "; //
     }
     cout << endl;

     // 2. Parcurgerea vectorului prin iteratori

     vector<int>::iterator it ;
   for(it=vec.begin();it!=vec.end();it++)
     cout<<*it<<" ";
     cout << endl;

     // 3. Parcurgerea vectorului folosind auto

   for (auto x:vec) {
         cout << x << " ";
     }
     cout << endl;

     return 0;
}
```

- Functii

  1. `size()`**-returneaza numarul de elemente al unui vector**

  2. `begin()`**-returneaza iteratorul primului element**

3. `end()`**-returneaza iteratorul unei pozitii imediate iteratorului de la sfarsit**
4. `rbegin()`**-returneaza iteratorul utimului element**
5. `rend()`**-returneaza iteratorul unei pozitii inaintea iteratorului de inceput**
6. `push_back(val)`**-adauga un element la finalul vectorului**
7. `pop_back`**-sterge un element de la finalul vectorului**
8. `erase(it)`**-sterge elementul de pe pozitia iteratorului it**
9. `clear()`**-sterge toate elementele vectorului;**
10. `back()`**-returneaza valoarea ultimului element din vector**
11. `front()`**-returneaza valoarea primului element din vector**

Exemplu:

```
#include <iostream>
#include <vector>

using namespace std;

int main() {

    vector<int> v;

    // Adauga elemente la finalul vectorului
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    //Afiseaza primul si ultimul element
    cout<<v.front()<<" "<<v.back();
    // Afiseaza dimensiunea vectorului
    cout << "Dimensiunea vectorului: " << v.size()
<< endl;

    // Afiseaza elementele vectorului folosind
iteratorul begin() si end()
    cout << "Elementele vectorului: ";
    for (auto it = v.begin(); it != v.end(); ++it)
```

```
{
        cout << *it << " ";
    }
    cout << endl;

    // Foloseste rbegin() si rend() pentru a afisa
vectorul invers
    cout << "Elementele vectorului in ordine
inversa: ";
    for (auto rit = v.rbegin(); rit != v.rend();
++rit) {
        cout << *rit << " ";
    }
    cout << endl;

    // Sterge ultimul element folosind pop_back()
    v.pop_back();
    cout << "Dupa pop_back(), dimensiunea este: "
<< v.size() << endl;

    // Sterge un element folosind erase() - sterge
al doilea element (iterator begin + 1)
    v.erase(v.begin() + 1);
    cout << "Dupa erase(), elementele vectorului:
";
    for (auto it = v.begin(); it != v.end(); ++it)
{
        cout << *it << " ";
    }
    cout << endl;

    // Sterge toate elementele folosind clear()
    v.clear();
    cout << "Dupa clear(), dimensiunea vectorului
este: " << v.size() << endl;
    //se afiseaza 0
    return 0;
 }
```

2. std::stack(Stiva)

- Scriere in C++:

  `stack<tip de date>a`

  Exemplu:`stack<int>a`

- Proprietati si functii:

  Stiva este o structura de date te tipul LIFO(Last in ,first out),astfel ultimul element care este adaugat este primul scos din stiva.

  `push(val)`**-adauga un element in varful stivei**

  `pop()`**-sterge elementul din varfu stivei**

  `top()`**-acceseaza elementu din varful stivei**

  `empty()`**-returneaza 1 daca stiva este goala si 0 daca stiva mai are elemente**

  Exemplu:

```
#include <iostream>
#include <stack>  // Biblioteca pentru stiva

 using namespace std;

 int main() {
     // Declara o stiva de intregi
     stack<int> st;

     // Adauga elemente in varful stivei
     st.push(10);
     st.push(20);
     st.push(30);

     // Afiseaza elementul din varful stivei
     cout << "Elementul din varful stivei: " <<
st.top() << endl;  // Va afisa 30

     // Sterge elementul din varful stivei
     st.pop();
     cout << "Dupa pop, elementul din varful
stivei: " << st.top() << endl;  // Va afisa 20

     // Verifica daca stiva este goala
     if (st.empty()) {
         cout << "Stiva este goala!" << endl;
     } else {
         cout << "Stiva nu este goala!" << endl;
     }
```

```
    // Stergem toate elementele ramase
    while (!st.empty()) {
        cout << "Stergem elementul: " << st.top()
<< endl;
        st.pop();
    }

    // Verifica din nou daca stiva este goala
    if (st.empty()) {
        cout << "Stiva este acum goala" << endl;
    }

    return 0;
 }
```

3. std::queue(Coada)

- Scriere in C++:

  `queue<tip de date>a`

  Exemplu:`queue<int>q`

- Proprietati si functii:

  Coada este o structura de date de tip FIFO(First In,first out),astfel primul care este adaugat este si primul scos din coada.

  `push(x)`**-adauga elementul x in coada**

  `pop()`**-sterge primul element din coada**

  `empty()`**-returneaza daca coada este vida sau nu**

  `front()`**-returneaza primul element din coada**

  `back()`**- returneaza ultimul element din coada**

  `size()`**-returneaza numarul de elemente ale cozii**

  Exemplu:

```
#include <iostream>
#include <queue>  // Biblioteca pentru coadă
```

```
using namespace std;

 int main() {

     queue<int> coada;

     // Adauga elemente in coada
     coada.push(10);
     coada.push(20);
     coada.push(30);

     // Afiseaza numarul de elemente din coada
     cout << "Numarul de elemente din coada: " <<
coada.size() << endl;  // Va afisa 3

     // Afiseaza primul si ultimul element din
coada
     cout << "Primul element din coada: " <<
coada.front() << endl;  // Va afisa 10
     cout << "Ultimul element din coada: " <<
coada.back() << endl;  // Va afisa 30

     // Sterge primul element din coada
     coada.pop();
     cout << "Dupa pop, primul element din coada: "
<< coada.front() << endl;  // Va afisa 20

     // Verifica daca coada este goala
     if (coada.empty()) {
         cout << "Coada este goala!" << endl;
     } else {
         cout << "Coada nu este goala!" << endl;
     }

     // Stergem toate elementele ramase
     while (!coada.empty()) {
         cout << "Stergem elementul: " <<
coada.front() << endl;
         coada.pop();
     }

     // Verifica din nou daca coada este goala
     if (coada.empty()) {
         cout << "Coada este acum goala!" << endl;
     }
```

```
    return 0;
}
```

4. std::dequeue(Double ended queue)

- Scriere in C++

  `dequeue<tip de date>a`

  Exemplu:`dequeue<int>q`

- Proprietati si functii:

  Dequeue este o structura de date in care poti adauga si sterge elemente din ambele capete ale ei.

  `push_back(x)`**-adauga un element x la final**

  `pop_back()`**-sterge ultimul element din dequeue**

  `back()`**-returneaza ultimul element din dequeue**

  `push_front(x)`**-adauga un elementx la inceput**

  `pop_front()`**-sterge primul element din dequeue**

  `front()`**-returneaza primul element din dequeue**

  `empty()`**-returneaza daca dequeue-ul este este vid sau nu**

  Exemplu:

```
#include <iostream>
#include <deque>  // Biblioteca pentru deque

using namespace std;

int main() {
    // Declara o dequeue de intregi
    deque<int> dq;

    // Adauga elemente la final si la inceput
    dq.push_back(10);    // Adaugă 10 la final
    dq.push_back(20);    // Adaugă 20 la final
```

```
        dq.push_front(5);    // Adaugă 5 la început
        dq.push_front(1);    // Adaugă 1 la început

        // Afiseaza elementele din dequeue
        cout << "Primul element din dequeue: " <<
dq.front() << endl;  // Va afisa 1
        cout << "Ultimul element din dequeue: " <<
dq.back() << endl;  // Va afisa 20

        // Afiseaza dimensiunea deque-ului
        cout << "Dimensiunea dequeue-ului: " <<
dq.size() << endl;  // Va afisa 4

        // Sterge primul si ultimul element
        dq.pop_front();  // Sterge 1
        dq.pop_back();   // Sterge 20

        // Afiseaza elementele ramase
        cout << "Dupa stergerea primului si ultimului
element:" << endl;
        cout << "Primul element: " << dq.front() <<
endl;  // Va afisa 5
        cout << "Ultimul element: " << dq.back() <<
endl;   // Va afisa 10

        // Stergem toate elementele ramase
        while (!dq.empty()) {
            cout << "Stergem elementul: " <<
dq.front() << endl;
            dq.pop_front();
        }

        // Verifica daca deque-ul este gol
        if (dq.empty()) {
            cout << "Deque-ul este acum gol!" << endl;
        }

        return 0;
    }
```

5. std::set

- Scriere in C++:

  `set<tip de date>a`

  Exemplu:`set<int>s`

Set-ul este o structura de date in care elementele se pastreaza in ordine crescatoare ,iar elementele nu se pot repeta

- Functii:

  `insert(x)`**-adauga valoarea x in set;**

  `erase(x)`**-sterge valoarea x din set;**

  `lower_bound(x)`**- returnează un iterator către primul element dintr-un interval care nu este mai mic decât (adică este mai mare sau egal cu) valoarea specificată**

  `upper_bound(x)`**-returnează un iterator către primul element dintr-un interval care este mai mare decât valoarea specificată.**

  `find(x)`**-returneaza iteratorul lui x ,daca x este prezent in set,altfel returneaza iteratorul de sfarsit al set-ului**

  `count(x)`**-returneaza 1 daca elementul x este prezent in set sau 0 daca elementul nu exista in set.**

  Exemplu:

```
#include <iostream>
#include <set>  // Biblioteca pentru set

 using namespace std;

 int main() {
     // Declară un set de întregi
     set<int> s;

     // Adaugă valori în set
     s.insert(5);
     s.insert(10);
     s.insert(15);
     s.insert(20);
     s.insert(5);  // Aceasta nu va avea efect,
deoarece 5 este deja în set

     // Afișează elementele setului
     cout << "Elementele setului: ";
     for (const auto e : s) {
         cout << e << " ";
```

```
    }
    cout << endl;

    // Verifică și șterge o valoare
    int v = 10; // Valoarea de șters
    if (s.count(v)) { // Verifică dacă valoarea
există în set
        s.erase(v);
        cout << "Valoarea " << v << " a fost
stearsă din set." << endl;
    } else {
        cout << "Valoarea " << v << " nu este în
set." << endl;
    }

    // Afișează elementele setului după ștergere
    cout << "După ștergerea lui " << v << ", setul
este: ";
    for (const auto& e : s) {
        cout << e << " ";
    }
    cout << endl;

    // Utilizează lower_bound
    int l = 15; // Valoarea pentru lower_bound
    auto itLower = s.lower_bound(l);
    if (itLower != s.end()) {
        cout << "Lower bound pentru " << l << ": "
<< *itLower << endl; // Va afisa 15
    } else {
        cout << "Nu există lower bound pentru " <<
l << endl;
    }

    // Utilizează upper_bound
    int u = 15; // Valoarea pentru upper_bound
    auto itUpper = s.upper_bound(u);
    if (itUpper != s.end()) {
        cout << "Upper bound pentru " << u << ": "
<< *itUpper << endl; // Va afisa 20
    } else {
        cout << "Nu există upper bound pentru " <<
u << endl;
    }

    // Utilizează find
    int f = 5; // Valoarea de găsit
    auto itFind = s.find(f);
    if (itFind != s.end()) {
```

```
            cout << "Valoarea " << f << " este găsită
în set." << endl; // Va afisa Gasit
        } else {
            cout << "Valoarea " << f << " nu este în
set." << endl;
        }

        // Verifică un element în set
        int c = 10; // Valoarea pentru count
        cout << "Count pentru " << c << ": " <<
s.count(c) << endl; // Va afisa 0

        c = 15; // Valoarea pentru count
        cout << "Count pentru " << c << ": " <<
s.count(c) << endl; // Va afisa 1

        return 0;
    }
```

6. std::multiset si std::unordered_set

- Scriere in C++

  `mutltiset<tip de date>a` si `unordered_set<tip de date>b`

  Exemple:

  `multiset<int>a`si `unordered_set<int>b`

  Proprietatile si functiile sunt aproape identice cu cele ale set-ului,doar ca in `multiset` se pot repeta valori (`count(val)>=1`),iar in `unordered_set`elementele nu sunt neaparat in ordine crescatoare.

7. std::unordered_map

- Scriere in C++:

  `unordered_map<key,value>a`**,unde key este cheia careia ii atribuim valoarea value**

  Exemplu :

```
#include <iostream>
#include <unordered_map>
```

```
 using namespace std;

 int main() {
     // Declară un unordered_map care stochează
perechi (cheie, valoare) de tip (int, string)
     unordered_map<int, string> umap;

     //  Adăugarea de elemente folosind `insert` și
operatorul `[]`
     umap.insert({1, "Alice"});
     umap[2] = "Bob";
     umap[3] = "Charlie";

     //  Afișarea elementelor folosind un iterator
     cout << "Elementele din unordered_map:" <<
endl;
     for (const auto elem : umap) {
         cout << "Cheie: " << elem.first << ",
Valoare: " << elem.second << endl;
     }

     //  Căutarea unei chei folosind `find`
     int cheie = 2;
     auto it = umap.find(cheie);
     if (it != umap.end()) {
         cout << "Găsit: Cheie: " << it->first <<
", Valoare: " << it->second << endl;
     } else {
         cout << "Cheia " << cheie << " nu a fost
găsită." << endl;
     }



     // Ștergerea unui element folosind `erase`
     umap.erase(2); // Șterge cheia 2

     // Verificarea dacă o cheie există folosind
`count`
     if (umap.count(2)) {
         cout << "Cheia 2 există." << endl;
     } else {
         cout << "Cheia 2 a fost ștearsă." << endl;
     }

     // Dimensiunea și verificarea dacă
unordered_map este gol
     cout << "Numărul de elemente din
```

```
unordered_map: " << umap.size() << endl;
     if (umap.empty()) {
         cout << "unordered_map este gol." << endl;
     } else {
         cout << "unordered_map nu este gol." <<
endl;
     }

     // Ștergerea tuturor elementelor folosind
`clear`
     umap.clear();
     cout << "Numărul de elemente după clear: " <<
umap.size() << endl;

     return 0;
 }
```

- Functii:

`insert({cheie,valoare})`**-adauga perechea cheie-valoare in unordered_map**

`find(cheie)`**-returneaza iteratorul elementului cheie daca exista in unordered_map**

Pentru a accesa cheia si valoarea cheii folosim

`it->first`si`it->second`sau `(*it).first`si `(*it).second`

`erase(cheie)`**-sterge cheia din unordered map**

`count(cheie)`**-returneaza 1 daca cheia exista si 0 daca nu exista in unordered map.**

8. std::map si std::multimap

- Scriere in C++:

  `map<key,value>a` si `multimap<key,value>b`

  Exemple:

  `map<int,int>a` si `multimap<int,int>b`

- Proprietatile si functiile lor sunt aproape identice cu `unordered_map-ul`.

  O diferenta este ca `map` si `multimap` au cheile in ordine crescatoare.In plus intr-un `multimap` se pot repeta perechile de tip `{cheie,valoare}`(`count(cheie)>=1`),in tip ce in `map` si `unordered_map` nu se pot repeta

9. std::priority_queue(Coada cu prioritati)

- Scriere in C++:

  `priority_queue<tip de date>a;`

  Exemplu:

  `priority_queue<int>pq`

- Proprietati si functii:

  Coada cu priorități este o structură de date specială care stochează elemente, fiecare având o anumită prioritate asociată. Într-o coadă cu priorități, elementele sunt extrase în funcție de prioritate, nu de ordinea în care au fost adăugate (ca în coada obișnuită).

  OBS:Coada cu prioritati in STL este ordonata descrescator dupa valoare.

  `push(x)`**-adauga valoarea x in queue**

  `pop()`**-sterge elementul din varf**

  `top()`**-returneaza elementul din varf**

  `size()`**-returneaza numarul de elemente din priority_queue**

  Exemplu:

```
#include <iostream>
#include <queue>  // pentru priority_queue

using namespace std;

int main() {
```

```
    priority_queue<int> pq;

    // 1. Adăugăm elemente în coadă folosind push
    pq.push(10);
    pq.push(5);
    pq.push(30);
    pq.push(20);

    // 2. Afișăm dimensiunea cozii
    cout << "Dimensiunea cozii cu priorități: " <<
pq.size() << endl;

    // 3. Verificăm elementul de pe vârf (elementul
cu prioritate maximă)
    cout << "Elementul din vârf (maxim): " <<
pq.top() << endl;

    // 4. Scoatem elementele din coadă până când
este goală
    cout << "Scoatem elementele din coadă în ordine
descrescătoare:" << endl;
    while (!pq.empty()) {
        cout << pq.top() << " ";  // Afișează
elementul din vârf
        pq.pop();  // Elimină elementul din vârf
    }
    cout << endl;

    // 5. Verificăm dacă coada este goală
    if (pq.empty()) {
        cout << "Coada cu priorități este goală."
<< endl;
    }

    return 0;
}
```

10. std::list

- Scriere in C++:

  `list<tip de date>a`

  Exemplu:

  `list<int>l`

- Proprietati si functii:

`std::list` **implementează o listă dublu înlănțuită.**

Spre deosebire de un array sau `std::vector`**, elementele dintr-o listă nu sunt stocate în locații de memorie adiacente.**

Fiecare element are un pointer către elementul anterior și următor, permițând inserarea și ștergerea eficientă a elementelor de la începutul, sfârșitul sau mijlocul listei.

Lista preia majoritatea functiilor vectorului,dar are si functii specifice precum:

`insert(it,x)`**-inserează elementul x la poziția indicată de iterator.**`O(1)-complexitate de timp`**;**

`advance(it,val)`**-muta iteratorul cu val pasi**

`reverse()`**-se inverseaza ordinea listei**

Exemplu:

```
#include <iostream>
#include <list>  // Pentru std::list

using namespace std;

int main() {

    list<int> l;

    // Adăugăm elemente la sfârșit și la început
    l.push_back(10);
    l.push_back(20);
    l.push_back(30);
    l.push_front(5);

    // Afișăm lista
    cout << "Elementele din listă: ";
    for (int x : l) {
        cout << x << " ";  // Afișare prin traversarea
listei
    }
    cout << endl;
```

```
        // Afișăm primul și ultimul element
        cout << "Primul element: " << l.front() << endl;
        cout << "Ultimul element: " << l.back() << endl;

        // Inserăm un element la a doua poziție (după
primul element)
        auto it = l.begin();
        advance(it, 1);  // Mutăm iteratorul pe a doua
poziție
        l.insert(it, 15);  // Inserăm valoarea 15

        // Afișăm lista după inserare
        cout << "Lista după inserarea lui 15: ";
        for (int x : l) {
            cout << x << " ";
        }
        cout << endl;

        // Ștergem primul și ultimul element
        l.pop_front();
        l.pop_back();

        // Afișăm lista după ștergere
        cout << "Lista după ștergerea primului și
ultimului element: ";
        for (int x : l) {
            cout << x << " ";
        }
        cout << endl;

        // Sortăm lista
        l.sort();

        // Afișăm lista sortată
        cout << "Lista sortată: ";
        for (int x : l) {
            cout << x << " ";
        }
        cout << endl;

        // Inversăm ordinea elementelor
        l.reverse();

        // Afișăm lista inversată
        cout << "Lista inversată: ";
        for (int x : l) {
            cout << x << " ";
        }
```

```
        cout << endl;

        return 0;
    }
```

- Probleme propuse:

Stiva

1. Pb Stiva
2. Pb Skyline
3. Pb nrpits
4. Pb treasure

Queue

1. Pb Coada
2. Pb BFS

Priority_Queue

1. Pb catmin
2. Dijkstra

Dequeue

1. Pb cuie

Vector(cu tuple)

1. Pb monezi