C11: STL

Standard Template Library (STL)

- Clase container
- Iteratori
- Algoritmi

C11:STL

STL - Standard Template Library face parte din C++ Standard Library.

STL contine 3 componente cheie:

- clase container
- iteratori
- algoritmi

Containere:

Un container e un obiect care contine o colectie de alte obiecte (elementele containerului).

Clasele container:

- implementeaza structuri de date clasice sub forma template => flexibilitate mare in utilizare
- se ocupa cu managementul spatiului de memorie pentru elemente
- pun la dispozitie functii membre pentru accesul la elemente – direct sau prin intermediul iteratorilor

- **Containere secventiale (sequence containers)**

- Contin o succesiune de elemente de acelasi tip T, fiecare element are o anume pozitie in container. Pozitia depinde de locul sau momentul cand obiectul a fost inserat in structura. Parcurgerea se face secvential.

  Clase puse la dispozitie: **vector**, **deque**, **list; C++11: array, forward_list**

- **Adaptoare de containere (container adaptors)**

- Clase ce pun la dispozitie o interfata specifica – bazandu-se pe un obiect container de baza, de exemplu, deque sau list.

  Clase puse la dispozitie: **stack, queue, priority_ queue**

- **Containere asociative (associative containers)**

- Contin colectii de elemente, in care pozitia unui element depinde de valoarea lui in functie de un criteriu de sortare; permit accesul rapid la un element prin intermediul valorii sale sau intermediul unei chei.

  Clase puse la dispozitie: **set**, **multiset**, **map**, **multimap**…

- **Containere asociative neordonate (unordered associative containers)**

- C++11: **unordered_set, unordered_multiset, unordered_map, unordered_multimap**

http://www.cplusplus.com/reference/stl/

C11:STL

Iteratori

- permit accesul la elementele unui container, independent de modul in care acestea sunt stocate
- sunt asemanatori cu pointerii. Mai exact, sunt o generalizare a pointerilor, fiind obiecte ce indica (point) alte obiecte (elementele din container)
- fiecare clasa container are definiti iteratorii proprii

http://www.cplusplus.com/reference/iterator/

Algoritmi

- STL pune la dispozitie algoritmi pentru procesarea elementelor din colectii; algoritmi de: cautare, partitionare, ordonare, lucru cu multimi, gasirea minimului/maximului etc.
- acestia sunt functii template care nu apartin claselor container
- algoritmii au ca parametri iteratori.

http://www.cplusplus.com/reference/algorithm/

Containere

Structura de date implementata	Nume STL	#include
Tablou dinamic	vector	<vector>
Lista dublu inlantuita	list	<list>
Stiva	stack	<stack>
Coada	queue	<queue>
Multime ordonata (Arbore binar de cautare)	set	<set>
Multime ordonata(se pot repeta valorile)	multiset	<set>
Multime ordonata dupa cheie	map	<map>
Multime ordonata dupa cheie (se pot repeta val.)	multimap	<map>
Max-heap (coada cu prioritati)	priority_queue	<queue>

Incepand cu C++11 mai exista inca o serie de containere: array, forward_list; unordered_set, unordered_multiset, unordered_map, unordered_multimap (acestea din urma fiind implementate ca tabele de dispersie).

C11:STL

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

class complex
{double re,im;
public:


complex(double i=0,double j=0)
{ re=i;
  im=j;
}

 friend ostream& operator<<(ostream
                &dev,const complex&x)
  {
          dev<<x.re<<"+j*"<<x.im;
  }
};
```

```
int main(int argc, char *argv[])
{ vector<int> v(5);//vector specializat pt. int
  vector<complex> cv(5);// specializat pt. complex
//dimensiunea lui v si cv e 5;
// in complex trebuie sa existe  implementat
//constructorul fara parametri

 for (int i=0;i<v.size();i++)
      {v[i]=i;
       cv[i]=complex(i,i);
//operatorul [] e implementat pentru vector
       }

  for (int i=0;i<v.size();i++)
      cout<<v[i]<<" ";
  cout<<endl;

  for (int i=0;i<cv.size();i++)
      cout<<cv[i]<<" ";

//se foloseste operatorul de << din complex

return 0;
}
```

C11:STL

Containere - Similaritati

1. Toate containerele sunt template:

```
vector<complex>    vector_nr_complexe;
set<string>    cuvinte;
list<int>    list_int;
```

2. Containerele au iteratori asociati

Tipul iteratorului e specificat folosind sintaxa: **container<T>::iterator nume_iterator**

```
vector<complex>::iterator  it   = vector_nr_complexe.begin();
set<string>::iterator   poz   =   cuvinte.find("ceva");
list<int>::iterator   primul_el  =  lista_int.begin();
```

Iteratorii se pot dereferentia ca orice alt pointer:

```
cout<<* primul_el;    //afiseaza elementul de la adresa primul_el
```

C11:STL

Iteratori-continuare

- Urmatoarele operatii sunt definite pentru aproape toti iteratorii:

  ++, -- muta iteratorul inainte, inapoi in cadrul structurii de date;

  = atribuire

  * dereferentiaza iteratorul pentru a obtine elementul stocat pe acea pozitie

  ==, != compara pozitiile iteratorilor

- Containerele STL cu iteratori asociati au metodele begin si end care returneaza iteratori

  - metoda begin() pointeaza la primul element

  - metoda end() pointeaza la nodul santinela – care nu contine niciun element

  Ex: cout<<*(l.end()); //run-time error

- Cand o functie de cautare nu gaseste elementul pe care il cauta - returneaza iteratorul end().

C11:STL

3. Toate containerele STL au urmatoarele functii membre:

```
int size()          //returneaza nr de elemente din container
iterator begin()    //returneaza iterator (pointer) la primul element
iterator end()      //returneaza iterator la ultimul nod (santinela)
bool empty()        //returneaza true daca obiectul container  e gol
```

Exemplu:

```
vector<string> vs(2);
vs[0]=string("primul");
vs[1]=string("al doilea");

vector<string>::iterator it=vs.begin();
while (it!=vs.end())
{cout<<*it<<" ";    //afisez elementul de pe pozitia it
 it++;                //trec la elementul urmator
 }
//echivalent cu:
//for (vector<string>::iterator it=vs.begin(); it!=vs.end(); it++) cout<<*it<<" ";
```

*nodul santinela (obtinut cu vs.end()) se gaseste imediat dupa ultimul element din vector.

C11:STL

Containere - Diferente

Fiecare tip de container din STL are functii membre specifice.

Exemplu: - al i-lea element dintr-un vector se poate accesa cu operatorul de indexare []:

```
vector<int> vi(2);
vi[0]=2;
```

DAR nu exista operator[] pentru liste:

```
list<int> li(2);
li[0]=2; //ERROR
```

Se poate adauga un element la finalul unei liste sau unui vector cu functia push_back, care are ca parametru elementul care se doreste introdus:

```
vi.push_back(3);
li.push_back(3);
```

DAR nu exista “back end” pentru un set sau map (arbore binar de cautare), un element e adaugat in asa fel incat stuctura ramane sortata:

```
set<int> ai;
ai.push_back(3);   //ERROR
```

C11:STL Containere secventiale

vector vs list

vector

- Zona de memorie continua.
- Daca prealoc spatiu pentru elemente viitoare, s-ar putea sa ocup mai mult spatiu decat necesar.
- Fiecare element necesita spatiu pentru el si atat (la liste mai am nevoie de un pointer pentru fiecare element).
- Realoc memorie pentru tot vectorul daca vreau sa adaug un element.
- Inserarea/stergerea de elemente la/de la finalul vectorului are costuri mici daca nu se realoca memorie, oriunde altundeva costul creste.
- Putem sa accesam aleator elementele.

list

- Zona de memorie ocupata nu e continua.
- Nu prealoc spatiu.
- Fiecare element are nevoie de un pointer in plus (sau doi, daca e dublu inlantuita).
- Nu trebuie niciodata sa realoc memorie daca vreau sa adaug un element.
- Inserarea, mutarea si stergerea unui element se face cu costuri mici, indiferent de pozitie.
- Nu pot accesa elementele aleator – asa ca - gasirea unui element poate fi costisitoare.

C11:STL

Vector

Clasa **vector** permite realizarea unui vector alocat dinamic, ce poate contine elemente de orice tip.
Clasa permite acces aleator facil la orice element (operator de indexare sau cu iteratori).
Implementarea din STL pune la dispozitie mai multi constructori, operator=, operator[], metode pentru manipularea obiectelor din vector, realocarea spatiului in functie de dorinta utilizatorului, etc.

Pentru a vedea toate metodele puse la dispozitie puteti folosi:
http://www.cplusplus.com/reference/vector/vector/

Cateva exemple de apel al constructorilor pusi la dispozitie:

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

vector<int> vec1;     // creaza un vector gol de int

vector<double> vec2(3);    //creaza un vector cu 3 elemente double

vector<int> vec3(3,10);  //creaza un vector cu 3 elemente int initializate cu valoarea 10

vector<int> vec4 (vec3); //creaza un vector – copie a lui vec3
```

C11:STL

```
//realizarea unei matrici de intregi cu L – linii si C  – coloane folosind vector
//elementele sunt initializate cu 0
#include <iostream>
#include <vector>
using namespace std;

#define L 3
#define C 3

int main()
{
// vector cu L linii, fiecare cu C  coloane cu valoare initiala a elementelor 0

vector<vector<int> > mat(L ,   vector<int>(C ,0));

for(int i = 0; i < L; ++i) {
   cout << endl;
   for(int j = 0; j < C ; ++j) {
       cout << mat[i][j] << " "; //folosesc operator[] pentru a accesa elementele
}
}

 return 0;
}
```

C11:STL

Cateva dintre metodele puse la dispozitie de clasa vector:

push_back() – adauga un element la finalul vectorului si creste dimensiunea cu 1

pop_back() – scoate ultimul element din vector si reduce dimensiunea cu 1

clear() – scoate toate elementele din vector si seteaza dimensiunea la 0

empty() - returneaza true daca vectorul e gol si altfel false

resize() - modifica dimensiunea vectorului

capacity() - returneaza capacitatea setata prin constructor sau cu functia resize

insert() - insereaza element pe pozitia specificata; creste dimensiunea

erase() - scoate elementul de pe pozitia specificata; scade dimensiunea

*pentru specificarea pozitiei se folosesc iteratori

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

int main ()
{
  vector<int> myvector (3,100);
  for (int i=0; i<myvector.size(); i++) cout<<myvector[i]<<" "; cout<<endl<<endl;

  vector<int>::iterator it;    //declar un iterator it
  it = myvector.begin();       //pozitionez iteratorul la inceputul vectorului

  it = myvector.insert( it , 200 );
  //inserez pe pozitia data de iterator (la inceputul vectorului) elementul 200; dimensiunea
//va creste cu 1; it va fi pozitionat acum la inceputul vectorului

  for (int i=0; i<myvector.size(); i++) cout<<myvector[i]<<" "; cout<<endl<<endl;

 it= myvector.insert (it,2,300);
//inserez doua elemente consecutive cu valoarea 300 - incepand cu pozitia data de it
 //se returneaza iterator catre primul obiect adaugat

   //! Functiile insert si erase folosesc iteratori ca parametru- pentru specificarea pozitiei
//unde vreau sa fac adaugarea sau stergerea
```

```
for (it=myvector.begin(); it!=myvector.end(); it++)//parcurg vectorul cu un iterator si afisez
    cout <<" "<< *it;                                    //obiectul de la adresa iteratorului
    cout<<endl<<endl;

  // it se gaseste la finalul vectorului; ma repozitionez la inceput:
  it = myvector.begin();

  vector<int> anothervector (2,400);   //creez alt vector

  myvector.insert (it+2,anothervector.begin(),anothervector.end());
  //inserez pe pozitia it+2, o secventa din alt vector : de la inceputul lui pana la finalul lui

  for (int i=0;i<myvector.size();i++) cout<<myvector[i]<<" "; cout<<endl<<endl;

  int myarray [] = { 501,502,503 };

   myvector.insert (myvector.begin(), myarray, myarray+2);
  //inserez la inceputul vectorului elementele din myarray astfel: incepand cu primul - pana la
//al doilea inclusiv (myarray+2)

  for (int i=0;i<myvector.size();i++) cout<<myvector[i]<<" "; cout<<endl<<endl;
  //501 502 300 300 400 400 200 100 100

  return 0;
}
```

```
//Un alt exemplu
#include <vector>
#include <iostream>
#include <string>
#include <string.h>
using namespace std;
class Persoana
{char*nume=NULL;
 public :
        Persoana(){nume=new char[strlen("doe")+1];
                    strcpy(nume,"doe");}
        Persoana(char*c){
              nume=new char[strlen(c)+1];
              strcpy(nume,c);}
        Persoana(const Persoana&c){
              nume=NULL;
              *this=c;}
        Persoana& operator=(const Persoana&c){
              if (nume!=NULL) delete[]nume;
              nume=new char[strlen(c.nume)+1];
              strcpy(nume,c.nume);return *this;}
        friend ostream& operator<<(ostream & dev,Persoana &ceva){
             dev<<ceva.nume;
             return dev;}
        ~Persoana(){ if (nume!=NULL) delete[]nume;}
    };
```

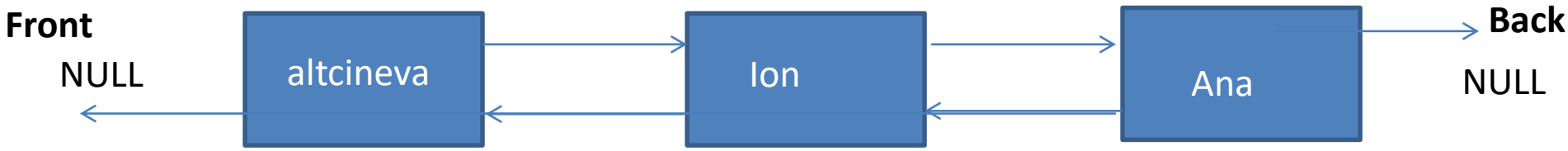
```
int main()
{vector<Persoana> persoane(1);  //aici am nevoie de constructorul fara parametri din Persoana
persoane.push_back(Persoana("Ana")); //constructor cu parametri din Persoana
persoane. push_back(Persoana("Maria")); //adaug elemente la finalul vectorului; creste dim
cout << "dimensiunea vectorului e " << persoane.size() <<endl;        //2;

persoane.pop_back(); //scot ultimul element din vector; scade dimensiunea cu 1
cout << "dimensiunea e " << persoane.size() << endl;          //1

persoane.resize(4);   //redimensionez vectorul; realocare spatiu
cout << "dimensiunea vectorului e " << persoane.size() <<endl;  //4
cout << "capacitatea vectorului e " << persoane.capacity() <<endl; //4

persoane[2]=Persoana("X");  //adaug o persoana pe pozitia 2; se foloseste Persoana::operator=

vector<Persoana>::iterator iter;   //declar un iterator cu care parcurg vectorul
for (iter = persoane.begin(); iter != persoane.end(); ++iter)
cout << *iter <<endl; //afisez ce gasesc la adresa lui iter; am nevoie de operator<< din Persoana

persoane.clear();  //sterg toate elementele; dimensiunea e 0; apel destructor din Persoana
 if(persoane.empty()) cout << "Gol"<<endl;  //testez daca vectorul e gol
else cout << "Avem ceva in vector";
return 0;
}
```

//Alt exemplu: clase.h

```
 #include <iostream>
#include <string.h>
using namespace std;

class Student
{ protected:    string nume;
                int id;
 public :
   Student(){}
   Student(int i, string c):id(i),nume(c){}
   virtual void afisare(){
         cout<<"Nume: "<<nume<<"  id: "<<id;    }
   virtual ~Student(){}
 };

 class Student_Ang:public Student
 { int salariu;
 public:
   Student_Ang(){}
   Student_Ang(int i, string c,int s):Student(i,c),salariu(s){}
   void afisare(){
         Student::afisare();
         cout<<" salariu: "<<salariu<<endl;
         }
 };
```

Creati un vector de studenti (nume, id) si studenti angajati (nume, id, salariu), populati-l si afisati toate atributele obiectelor stocate.
Reprezentati grafic!

```
#include <vector>
#include “clase.h”


int main()
{
   vector<Student*> vsa;
   vsa.push_back(new Student(1,"Stud1"));
   vsa.push_back(new Student_Ang(2,"Stud2",2000));
   vsa.push_back(new Student(3,"Stud3"));

   vector<Student*>::iterator its;

   for (its=vsa.begin();its!=vsa.end();its++)      //vreau sa afisez fiecare element al lui vsa
                  (*its)->afisare();

   vsa.clear();

return 0;
}

//cum as putea sa caut un anumit student x in vector?
```



List

O lista dublu inlantuita este o lista cu elemente “imprastiate” prin memorie, conectate prin pointeri; lista poate fi parcursa in ambele sensuri: inainte si inapoi. Ca functionalitati seamana cu clasa vector, dar elementele nu sunt stocate in zone de memorie succesive si nu se pot accesa elemente in mod aleator ([]).

```
list<Persoana> lp;
```

Putem sa adaugam, scoatem elemente de la inceputul sau finalul listei cu functiile: push_back; push_front; pop_back; pop_front. Inserarea de elemente (oriunde) este necostisitoare.

```
lp.push_back(Persoana(“Ana”));
lp.push_front(Persoana(“Ion”));
```

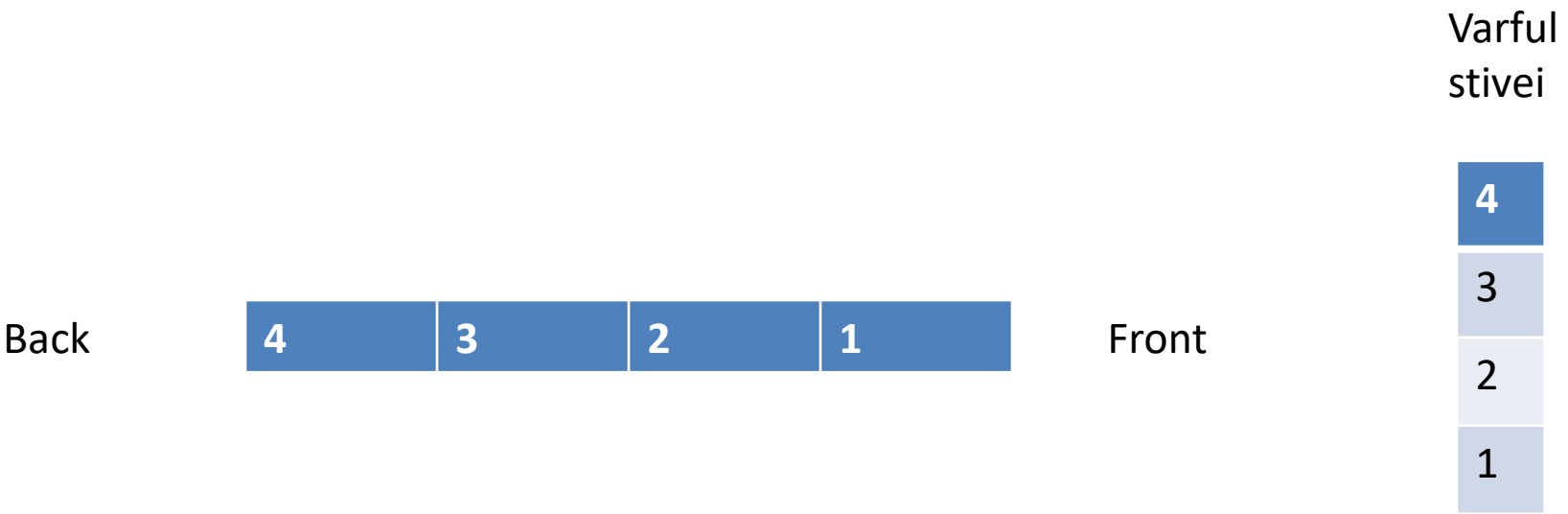
Putem sa adaugam, stergem elemente de pe o anumita pozitie folosind iteratori;

```
list<Persoana>::iterator p=lp.begin();
p=lp.insert(p,Persoana(“altcineva”));  //returneaza iterator care pointeaza catre noua valoare;
p=lp.begin();
while (p!=lp.end()) {cout<<*p<<endl; p++;}
p=lp.erase(lp.begin());              //returneaza iterator pointand catre urmatorul element
```

C11:STL

Stive si cozi (adaptoare de containere)

- stack si queue sunt clase ale STL
- o stiva este o lista inlantuita de tip LIFO. Putem accesa doar varful stivei.
- o coada este o lista inlantuita de tip FIFO. Putem sa ne uitam la inceputul si finalul cozii.
- in ambele containere putem adauga elemente cu functia push si scoate elemente cu functia pop.



C11:STL

- Push si pop se comporta diferit pentru stive si cozi.

```
stack<string>s;
s.push("Ana");
s.push("Ion");
s.push("cineva");
s.pop();//scoate ultimul element:cineva
s.push("X");
cout<<s.top();   //X
```


```
queue<string>q;
q.push("Ana");
q.push("Ion");
q.push("cineva");
q.pop(); //scoate primul element: Ana
q.push("X");
cout<<q.front()<<" "<<q.back(); //Ion X
```

C11:STL

Priority queue (heap) – coada cu prioritati

- O coada cu prioritati (adaptor de container) este o structura ce imita un max-heap: cea mai mare valoare este mereu la inceputul listei.

- Operatorul< trebuie definit pentru tipul de date pe care vrem sa il stocam intr-un astfel de container.

```
priority_queue <int> Q;
Q.push(10);
Q.push(15);
Q.push(1);
Q.push(11);
Q.pop(); //scoate elementul de la inceputul listei: 15
cout<<Q.top();   //recupereaza valoarea de la inceput 11
```

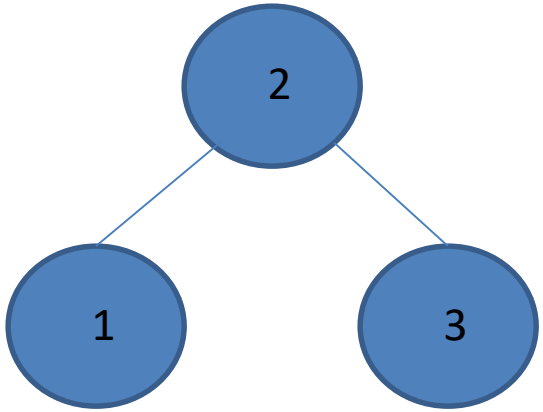
***Stivele si cozile nu au iteratori disponibili, am aces doar la elementele “din capetele” structuriilor.**

C11:STL

Containere asociative

Set (multime ordonata - implementata ca arbore binar de cautare echilibrat)

- pentru a se putea crea o multime, tipul de date trebuie sa aiba **operator<** definit.
- toate **elementele** inserate sunt **distincte** (in caz contrar folosim multiset) si nu mai pot fi modificate (pot sa fie doar sterse /adaugate alte elemente).
- containerele de tip set realizeaza in mod automat un **arbore binar echilibrat** astfel incat cautarea unui element sa se faca intr-un mod eficient.
- elementele din container sunt mereu ordonate.
- daca valorile urmatoare sunt introduse in set : 1 2 3; se obtine:



C11:STL

Operatii de baza pentru containerul set:

- insert(element)
- erase(iterator) sau erase(valoare element)
- iterator find(valoare element)

```
set<int>S;
set<int>::iterator x;

S.insert(-10); //nu se da pozitia
for (int i=10;i>0;i--)
S.insert(i); //nu se da pozitia
//-10 1 2 3 4 5 6 7 8 9 10

S.erase(8); //nu se da pozitia

x=S.begin();
while(x!=S.end())
{cout<<*x<<" ";
 x++;}                    // -10 1 2 3 4 5 6 7 9 10
```

```
x=S.find(7);

while(x!=S.begin())
{
   cout<<*x<<" ";
   x--;
}

cout<<*(S.begin());

//7 6 5 4 3 2 1 -10
```

C11:STL

Map

- Un container ce ne permite sa stocam perechi (cheie, obiect de stocat). Datele sunt tinute ordonate dupa cheie. Permit cautarea rapida a unui obiect folosind cheie unica asociata acestuia (ex: un element de tip Student dupa id)

- Pentru a crea un astfel de container avem nevoie de 2 tipuri template:

```
map<K,T>
```

unde K- este tipul de date al cheii si T tipul de date pe care vrem sa le stocam, de ex.:

```
map<int, Student>
```

Un container map contine elemente indexate – cu indecsi de tipul K al cheii.

Operatorul < trebuie sa fie definit pentru tipul de date al **cheii**. Daca nu exista – se utilizeaza un container de tipul: unordered_container

Elementele containerului sunt mereu sortate in functie de cheie (map e implementat cu arbori binari de cautare).

C11:STL

- Inserarea elementelor intr-un container map este ceva mai complicata deoarece trebuie introdus elementul propriu zis si valoarea pentru cheie.
- Biblioteca <utility> pune la dispozitie clasa template **pair** (cu atributele **first** si **second** – primul/al doilea element din pereche).
- Functia insert pentru map asteapta un argument de tip pair.

```
map<int,Student> ms;
pair<int,Student> aux(4,Student(4,“stud"));
ms.insert(aux);
```

Cel mai simplu pentru inserare se poate folosi operator[] (parametrul functiei operator[] e cheia).

```
map<char,int> mci;
map<char,int>::iterator mit;

 mci['d']=200;
 mci['b']=100;
 mci['c']=150;
 mci['a']=50;

 mit=mci.find('b');
 mci.erase (mit); //sterge elementul cu cheia b
 //echivalent cu:
 mci.erase (mci.find('d'));

 cout << "a => " << mci.find('a')->second << endl;
 cout << "c => " << (*mci.find('c')).second << endl;
//gaseste elementul cu cheia ‘a’ si, respectiv, ‘c’ si
//afiseaza valoarea asociata acelei chei
```

//structura mci e mereu ordonata dupa cheie

```
class Persoana
{ string nume;
  int id;
  public :
        Persoana(){}

        Persoana(int i, string c):id(i),nume(c){}

        friend ostream& operator<<(ostream & dev,Persoana &ceva){
             dev<<ceva.id<<" "<<ceva.nume;
             return dev;}

         int getId(){return id;}

         string getNume(){return nume;}
//_____________________________________________________________
         friend bool operator<(const Persoana &p1,const Persoana &p2){
              return p1.id<p2.id;
        }

        friend bool operator==(const Persoana &p1,const Persoana &p2){
            return (p1.id==p2.id);
        }       //operator< si operator== sunt folositi la ex pentru biblioteca algorithm
};
```

```
void main(){
  map<int,Persoana> mp;
    Persoana p1(1,"Ana");
    Persoana p2(2,"Maria");
    Persoana p3(3,"AnaMaria");

    mp[p1.getId()]=p1;
    mp[p3.getId()]=p3;
    mp[2]=p2;
//elementele dintr-un container map sunt perechi (cheie, obiect stocat); adica pe o anumita
//pozitie in container avem un element de tip pair – structura template cu campurile first si
//second

    pair<int,Persoana> elem(4,Persoana(4,"cineva"));
    mp.insert(elem);

    map<int,Persoana>::iterator ip= mp.find(2);
 //cautarea se face dupa cheie; ip pointeaza catre perechea (cheie, persoana)

    cout<<ip->second.getNume()<<endl;
    // vreau sa afisez numele persoanei; second – al doilea element din pereche (Persoana)

    for (ip=mp.begin();ip!=mp.end();ip++)
        cout<<ip->second;
//afisez toate persoanele; care operator<< se foloseste?
return 0; }
```

C11:STL

Multimap

Putem sa avem mai multe inregistrari cu aceasi cheie.

```
multimap<char,int> mymm;
mymm.insert(pair<char,int>('a',10));
mymm.insert(pair<char,int>('b',20));
mymm.insert(pair<char,int>('b',40));
mymm.insert(pair<char,int>('c',50));
mymm.insert(pair<char,int>('d',60));
mymm.insert(pair<char,int>('b',30));
mymm.insert(pair<char,int>('c',60));

for (char ch='a'; ch<='d'; ch++) {
    pair <multimap<char,int>::iterator , multimap<char,int>::iterator> domeniu;
    domeniu = mymm.equal_range(ch);//caut elementele cu cheia ch si returnez pozitia de
                                   //inceput si de final (iteratori) in perechea domeniu
    cout << ch << " =>";
    for ( multimap<char,int>::iterator it=domeniu.first; it!=domeniu.second; ++it)
        cout << ' ' << it->second; std::cout << endl;
}
```

C11:STL

Ce si cand folosim?

- Daca este de interes cautarea rapida de date pe baza unei chei (dictionar, agenda telefonica, …) folositi (multi) map
- Daca elementele sunt unice, au operator< definit, si sunt de interes cautari rapide (inclusiv de domenii) sau mentinerea datelor ordonate-> (multi) set
- Daca e nevoie in principal de elementul cu cea mai mare valoare -> priority_queue.
- Daca e de interes adaugarea rapida de elemente si nu vrem sa le accesam des - > lista.
- Daca vrem acces rapid la elementele stocate pe o anumita pozitie - > vector.
- Daca vreau ca adaugarea, gasirea unui element sa se faca O(1) -> unordered (multi) map/set

Tema: Completati urmatorul tabel cu complexitatea operatiei/structura de date

Structura date	Adauga element la inceput/final/oriunde	Sterge element de la inceput/final/oriunde	Cauta element dupa un criteriu	Cauta min/max	Returneaza elementul de pe o anumita pozitie
Vector					
Lista					
BST					
BST + echilibrat					
Heap					
Hash table					

C11:STL

Biblioteca <algorithm>

Contine o serie de functii utile pentru lucrul cu containere:

- find
- remove
- count
- shuffle
- replace

Specificam, de obicei, iteratori pentru pozitiile de inceput si final al intervalului in care vrem sa realizam operatia.

Nu sunt functii membre!

```
vector<string> l;
//adaug elemente
iter=find(l.begin(),l.end(),"Ana"); //tipul stocat trebuie sa aiba operatorul ==  implementat
int x=count(l.begin(),l.end(),"Ana"); //tipul stocat trebuie sa aiba operatorul ==  implementat
replace(l.begin(),l.end(),"Ana", "A"); //tipul stocat trebuie sa aiba operatorii ==,= implementati
```

```
vector<Persoana> vp(3);

vp[0]=Persoana(3,"A"); //operator= din clasa Persoana
vp[1]=Persoana(2,"B");
vp[2]=Persoana(1,"C");

sort(vp.begin(),vp.end());    //sortarea se face dupa id
//aici am nevoie ca operator< sa fie implementat in Persoana

for (vector<Persoana>::iterator vit= vp.begin();vit!=vp.end();vit++)
     cout<<*vit<<endl;

replace(vp.begin(),vp.end(),Persoana(1,"C"),Persoana(3,"A"));
 //aici am nevoie ca operator== si = sa fie implementati in Persoana

cout<<count(vp.begin(),vp.end(),Persoana(3,"A")); //2
 //aici am nevoie ca operator== sa fie implementat in Persoana
```

```
bool IsOdd (int i) { return ((i%2)==1); }

void main(){
   int myints[] = {10,20,30,30,20,10,10,20,0};
   vector<int> vv(myints,myints+8);          // 10 20 30 30 20 10 10 20

   sort (vv.begin(), vv.end());              // 10 10 10 20 20 20 30 30

   vector<int>::iterator low,up;
   low=lower_bound (vv.begin(), vv.end(), 20);
   up= upper_bound (vv.begin(), vv.end(), 20);
 //Returneaza un iterator ce pointeaza catre primul/ultimul element din intervalul
//[primul param, al doilea param)  care nu e </> ca valoarea precizata prin al 3-lea
//parametru

  cout << "lower_bound are pozitia " << (low- vv.begin()) << endl;   //3
  cout << "upper_bound are pozitia " << (up - vv.begin()) << endl;   //6

   for (int i=1; i<10; i++)
        vv.push_back(i); // vv: 10 10 10 20 20 20 30 30 1 2 3 4 5 6 7 8 9

  int cate = count_if (vv.begin(), vv.end(), IsOdd);
  cout << “am " << cate  << “elemente impare";
  return 0; }
```

Teme optionale pentru vacanta

1. Cititi materialul legat de tabele de dispersie (SDA AB).

Folosind tipurile de date Persoana (slide 29 - C11) sau orice alt tip de date implementat de voi, realizati scurte aplicatii ce folosesc clasele **unordered_set, unordered_multiset, unordered_map, unordered_multimap** din STL; folositi cat mai multe dintre metodele disponibile si verificati complexitatile lor.

2. O Persoana are un nume si o varsta.

Un Artist e o Persoana care are mai multe Albume.

Un Album e caracterizat prin an aparitie, nume si Trackuri.

Un track e caracterizat de pozitia in album, durata si numele sau.

Trebuie sa:

- se poata adauga/sterge/modifica un album al unui artist
- se poata adauga/sterge/modifica un track dintr-un album (continuare slideul urmator)

- sa se afiseze (eventual in fisier) in ordine cronologica toate albumele unui artist
- sa se afiseze toata discografia astfel (pot sa am 2 albume in acelasi an):

artistx - an apartite album1 - nume album1 - track1 (durata1)

- track2…

- an apartite album2 - nume album2 - track1

- track2…

- …

- sa se gaseasca toate albumele intre anii: an1 si an2
- pentru un artist sa se afiseze toate albumele intre an1 si an2.
- sa se gaseasca cel mai lung (nr de secunde) album al unui artist.
- pentru un album sa se calculeze cate trackuri sunt mai lungi de x secunde.

Implementarea trebuie sa foloseasca cele mai bune stucturi de date din punct de vedere al eficientiei.

2. Tinerea evidentei: filmelor dupa an aparitie/gen. Film: titlu; gen; regizor; buget; nota IMDb; an etc.

3. Evidenta facturilor dupa data. Factura: suma; nr; cine a emis-o; cand etc. Data: zi, luna, an.

4. Realizare dictionare; cataloage; etc.

5. Evidenta operatii pe conturi bancare/stocuri de produse.

6. Evidenta concurentilor dintr-un concurs. Acordarea de premii.