Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

Programare orientată pe obiecte

- suport de curs -

Andrei Păun
Anca Dobrovăț

An universitar 2021 – 2022
Semestrul II
Seriile 13, 14 şi 15

Curs 12

Agenda cursului

Biblioteca Standard Template Library - STL

- Containere, iteratori şi algoritmi.
- Clasele vector, list, map / multimap.
- Elemente avansate

Standard Template Library (STL)

-bibliotecă de clase C++, parte din Standard Library

Ofera:
- structuri de date și algoritmi fundamentali → dezvoltarea de programe in C++;
- componente generice, parametrizabile. Aproape toate clasele din STL sunt parametrizate (Template).

Componentele STL se pot compune cu usurință fără a sacrifica performanța (generic programming)

STL conține clase pentru:
- **containere**
- **iteratori**
- **algoritmi**
- functori (function objects)
- allocators

Standard Template Library (STL)

Container

"Containers are the STL objects that actually store data" (H. Schildt)

- grupare de date în care se pot adauga (insera) si din care se pot sterge (extrage) obiecte;

- gestionează memoria necesară stocarii elementelor, oferă metode de acces la elemente (direct si prin iteratori);

- **funcționalități (metode)**:
  - accesare elemente (ex.: [ ])
  - gestiune capacitate (ex.: size())
  - modificare elemente (ex.: insert, clear)
  - iterator (begin(), end())
  - alte operații (ie: find)

Standard Template Library (STL)

Container

Tipuri de containtere:

- ***de tip secventa*** (in terminologia STL, o secventa este o lista liniara):
  - vector
  - deque
  - list

- ***asociativi*** (permit regasirea eficienta a informatiilor bazandu-se pe chei)
  - set
  - multiset
  - map (permite accesul la informatii cu cheie unica)
  - multimap

- ***adaptor de containere***
  - stack
  - queue
  - priority_queue

Standard Template Library (STL)

Adaugarea si stergerea elementelor din containere:

De tip secventa (vector, deque, list) si - ***asociativi*** (set, multiset, map, multimap):

- insert()
- erase()

De tip secventa (vector, deque, list) permit si:

- push_back()
- pop_back()

List , Deque

- pop_front()
- push_front()

Containerele asociative:

- find()

Accesarea uzuala: prin iteratori.

Standard Template Library (STL)

Containere de tip secventa

Vector (Dynamic Array)

Vector, Deque, List sunt containere de tip secvență, folosesc reprezentări interne diferite, astfel operațiile uzuale au complexități diferite.

template <class T, class Allocator = allocator <T> > class vector
T = tipul de data utilizat
Allocator = tipul de alocator utilizat (in general cel standard).

- elementele sunt stocate secvențial in zone continue de memorie.

Vector are performanțe bune la:
- Acesare elemente individuale de pe o poziție dată (constant time).
- Iterare elemente în orice ordine (linear time).
- Adăugare/Ștergere elemente de la sfârsit (constant amortized time).

Standard Template Library (STL)

Containere de tip secventa

Vector (Dynamic Array)

Constructori:

explicit vector(const Allocator &a = Allocator( ));

// expl. ***vector<int> iv*** *- vector de int cu zero elemente;*

explicit vector(size_type num, const T &val = T ( ), const Allocator &a = Allocator( ));

// expl. ***vector<int> iv(10)*** *- vector de int cu 10 elemente;*

// expl. ***vector<int> iv(10,7)*** *- vector de int cu 10 elemente, fiecare egal cu 7;*

vector(const vector<T,Allocator> &ob);

// expl. ***vector<int> iv2(iv)*** *- vector de int reprezentand copia lui iv;*

template <class InIter>
vector(InIter start, InIter end, const Allocator &a = Allocator( ));

Standard Template Library (STL)

Containere de tip secventa

Vector (Dynamic Array)

Functii membre uzuale (sursa H. Schildt)

- **back**( ): returneaza o referinta catre ultimul element;
- **front**( ): returneaza o referinta catre primul element;
- **begin**( ): returneaza un iterator catre primul element;
- **end**( ): returneaza un iterator catre zona de memorie de dupa ultimul element;
- **clear**( ): elimina toate elementele din vector.
- **empty**( ): “true(false)” daca vectorul e (sau nu) gol.
- **erase(iterator i)**: stergerea elementului pointat de i; returneaza un iterator catre elementul de dupa cel sters.
- **erase(iterator start, iterator end):** stergerea elementelor intre start si end.

Standard Template Library (STL)

Containere de tip secventa

Vector (Dynamic Array)

Functii membre uzuale (sursa H. Schildt)

- **insert(iterator i, const T &val):** insereaza val inaintea elementului pointat;
- **insert(iterator i, size_type num, const T & val)** : insereaza un numar de “num” elemente de valoare “val” inaintea elementului pointat de i.
- **insert(iterator i, InIter start, InIter end):** insereaza o secventa start-end inaintea elementului pointat de i.
- **operator[ ](size_type i)** : returneaza o referinta la elementul specificat de i;
- **pop_back( ):** sterge ultimul element.
- **push_back(const T &val):** adauga la final valoarea “val”
- **size( )** : dimensiunea vectorului.

Standard Template Library (STL)

Containere de tip secventa
Vector (Dynamic Array)

Exemplu citirea unui vector cu accesarea elementelor prin iterator si prin operatorul [ ]

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
     vector<int> v(3);
   v[0] = 12;
   v[1] = 34;
   v[2] = 56;
   //v.resize(5); // necesar pentru reactualizarea dimensiunii vectorului
   v[3] = 78;
   v[4] = 90;
for(int i = 0; i<v.size(); i++)
   cout<<v[i]<<" ";
}
```

Standard Template Library (STL)

Containere de tip secventa
Vector (Dynamic Array)

Exemplu citirea unui vector cu accesarea elementelor prin iterator si prin operatorul [ ]

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
     vector<int> v(3);
   v.push_back(12);
   v.push_back(34);
   v.push_back(56);
   v.push_back(78);
   v.push_back(90);
for(int i = 0; i<v.size(); i++)
   cout<<v[i]<<" ";// Afisare 8 valori, resize automat
}
```

Standard Template Library (STL)

Containere de tip secventa
Vector (Dynamic Array)

Exemplu

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<char> v(10);
    unsigned int i;
    cout << "Size = " << v.size() << endl;
    for(i=0; i<10; i++) v[i] = i + 'a';
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;
    for(i=0; i<10; i++) v.push_back(i + 10 + 'a');
    cout << "Size now = " << v.size() << endl;
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;
    for(i=0; i<v.size(); i++) v[i] = toupper(v[i]);
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
}
```

```
Size = 10
a b c d e f g h i j
Size now = 20
a b c d e f g h i j k l m n o p q r s t
A B C D E F G H I J K L M N O P Q R S T
```

Standard Template Library (STL)

Containere de tip secventa
Vector (Dynamic Array)

Exemplu - accesarea unui vector cu iterator

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<char> v(10); //unsigned int i = 0;
    vector<char>::iterator p;

     for( p = v.begin(); p != v.end(); p++,i++)
               *p = i + 'a';

     for( p = v.begin(); p != v.end(); p++)
        cout << *p << " ";

     for( p = v.begin(); p != v.end(); p++)
        *p = toupper(*p);
}
```

Standard Template Library (STL)

Containere de tip secventa
Vector (Dynamic Array)

Exemplu - inserarea si stergerea elementelor intr-un vector

```
vector<int> v(3);
   v[0] = 12;    v[1] = 34;    v[2] = 56; /// 12,34,56
   v.resize(4);
   v[3] = 78;
   v.push_back(90); /// 12, 34, 56, 78, 90

   vector<int>::iterator p,pp;
   p = v.begin();
   p++;
   v.insert(p,3,100); /// 12, 100, 100, 100, 34, 56, 78, 90

p = v.begin();
v.erase(p); /// 100, 100, 100, 34, 56, 78, 90

p = v.begin()+3;
v.erase(p,p+2); /// 100, 100, 100, 78, 90
```

Standard Template Library (STL)

Containere de tip secventa
Vector (Dynamic Array)

Exemplu - inserarea elementelor de tip definit de utilizator

```
class Test { int i;
public:
   Test(int x = 0):i(x){cout<<"C ";};
   Test(const Test& x){i = x.i; cout<<"CC ";}
   ~Test(){cout<<"D ";} };

int main() {
   vector<Test> v;
   v.push_back(10); cout<<endl;    /// C CC D
   v.push_back(20); cout<<endl;    /// C CC CC D D
   Test ob(30);
   v.push_back(ob); cout<<endl;    /// C CC CC CC D D
   Test& ob2 = ob;
   v.push_back(ob2);    /// CC D D D D
}
```

Standard Template Library (STL)

Containere de tip secventa

List

template <class T, class Allocator = allocator <T> > class list

T = tipul de data utilizat
Allocator = tipul de alocator utilizat (in general cel standard).

- implementat ca și listă dublu înlănțuită

List are performanțe bune la:
▪ Ștergere/adăugare de elemente pe orice poziție (constant time).

▪ Mutarea de elemente sau secvențe de elemente în liste sau chiar și intre liste diferite(constant time).

▪ Iterare de elemente in ordine (linear time)

Standard Template Library (STL)

Containere de tip secventa

List

Constructori:

explicit list(const Allocator &a = Allocator( ));

// expl. list<int> iv

explicit list(size_type num, const T &val = T( ), const Allocator &a = Allocator());

// expl. list<int> iv(10);

// expl. list<int> iv(10,7);

list(const list <T,Allocator> &ob);

// expl. list<int> iv2(iv);

template <class InIter>
list(InIter start, InIter end, const Allocator &a = Allocator( ));

Standard Template Library (STL)

Containere de tip secventa
List

Functii membre uzuale (sursa H. Schildt)

- **back**( ): returneaza o referinta catre ultimul element;
- **front**( ): returneaza o referinta catre primul element;
- **begin**( ): returneaza un iterator catre primul element;
- **end**( ): returneaza un iterator catre zona de memorie de dupa ultimul element;
- **clear**( ): elimina toate elementele din vector.
- **empty**( ): “true(false)” daca vectorul e (sau nu) gol.
- **erase(iterator i)**: stergerea elementului pointat de i; returneaza un iterator catre elementul de dupa cel sters.
- **erase(iterator start, iterator end):** stergerea elementelor intre start si end.

Standard Template Library (STL)

Containere de tip secventa

List

Functii membre uzuale (sursa H. Schildt)

- **insert(iterator i, const T &val):** insereaza val inaintea elementului pointat;

- **insert(iterator i, size_type num, const T & val)** : insereaza un numar de “num” elemente de valoare “val” inaintea elementului pointat de i.

-**insert(iterator i, InIter start, InIter end):** insereaza o secventa start-end inaintea elementului pointat de i.

- **merge(list <T, Allocator> &ob):** concateneaza lista din ob; aceasta din urma devine vida.

- **merge(list &ob, Comp cmpfn):** concateneaza si sorteaza;

Standard Template Library (STL)

Containere de tip secventa

List

Functii membre uzuale (sursa H. Schildt)

- **pop_back**( ): sterge ultimul element.
- **pop_front**( ): sterge primul element.
- **push_back(const T &val)**: adauga la final valoarea “val”
- **push_front(const T &val)**: adauga la inceput valoarea “val”
- **remove(const T &val)**: elimina toate valorile “val” din lista;
- **reverse( )**: inverseaza lista.
- **size( )** : numarul de elemente.
- **sort( )**: ordoneaza crescator
- **sort(Comp cmpfn)**: - sorteaza cu o functie de comparatie.

Standard Template Library (STL)

Containere de tip secventa
List

Exemplu

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
     list<int> lst; // create an empty list
     int i;
     for(i=0; i<10; i++) lst.push_back(i);
     cout << "Size = " << lst.size() << endl;
     list<int>::iterator p;
     for(p = lst.begin();p != lst.end();p++)
        cout << *p << " ";
     for(p = lst.begin();p != lst.end();p++)
         *p = *p + 100;
     for(p = lst.begin();p != lst.end();p++)
        cout << *p << " ";
     return 0;
}
```

Standard Template Library (STL)

Containere de tip secventa
List

Exemplu

```
 list<int> lst; // lista vida
 for(int i=0; i<10; i++) lst.push_back(i); // insereaza 0 .. 9

    // Lista afisata in ordine inversa - mod 1"
list<int>::iterator p;
p = lst.end();
while(p != lst.begin()) {
 p--; // decrement pointer before using
cout << *p << " ";
 }
```

```
// Lista afisata in ordine inversa - mod 2”
list<int>::reverse_iterator q;
 for(q = lst.rbegin(); q!=lst.rend();q++)
 cout<<*q<<" ";
```

Standard Template Library (STL)

Containere de tip secventa
List

Exemplu - push_front, push_back si sort

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list lst1, lst2;
int i;
for(i=0; i<10; i++) lst1.push_back(i);
for(i=0; i<10; i++) lst2.push_front(i);

list::iterator p;
for( p = lst1.begin(); p != lst1.end(); p++) cout << *p << " ";
for( p = lst2.begin(); p != lst2.end(); p++) cout << *p << " ";

// sort the list
lst1.sort();
}
```

Standard Template Library (STL)

Containere de tip secventa
List

Exemplu - ordonare crescatoare si descrescatoare

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

bool compare(int a, int b) {return a>b;}

int main() {
list<int> l;
l.push_back(5); l.push_back(3); l.push_front(2);
list<int>::iterator p;
l.sort(); //crescator
for(p=l.begin(); p!=l.end();p++) cout<<*p<<" ";

l.sort(compare); // descrescator prin functia comparator
for(p=l.begin(); p!=l.end();p++) cout<<*p<<" ";
}
```

Standard Template Library (STL)

Containere de tip secventa
List

Exemplu - concatenarea a 2 liste

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    list<int> lst1, lst2; /*...creare liste …*/
    list<int>::iterator p;
// concatenare
    lst1.merge(lst2);
    if(lst2.empty())
        cout << "lst2 vida\n";
    cout << "Contents of lst1 after merge:\n";
    for(p = lst1.begin();p != lst1.end(); p++) cout << *p << " ";
    return 0;
}
```

Standard Template Library (STL)

Containere de tip secventa
List

Exemplu - stocarea obiectelor in list

```
class Test { int i;
public:
  Test(int x = 0):i(x){cout<<"C ";};
  Test(const Test& x){i = x.i; cout<<"CC ";}
  ~Test(){cout<<"D ";} };

int main() {
  list<Test> v;
  v.push_back(10); cout<<endl;    /// C CC D
  v.push_back(20); cout<<endl;    /// C CC D
Test ob(30);
  v.push_back(ob); cout<<endl;    /// C CC
Test& ob2 = ob;
  v.push_back(ob2);    /// CC D D D D D
}
```

Standard Template Library (STL)

Containere de tip secventa
List

Exemplu - stocarea obiectelor in list

```
#include <iostream>
#include <list>
#include <cstring>
using namespace std;
class myclass
{
    int a, b, sum;
public:
    myclass(){a = b = 0;}
    myclass(int i, int j){a = i;b = j;sum = a + b;}
    int getsum(){return sum;}
    friend bool operator<(myclass &o1,myclass &o2) {return o1.sum < o2.sum;}
    friend bool operator>(myclass &o1,myclass &o2) {return o1.sum > o2.sum;}
    friend bool operator==(myclass &o1,myclass &o2){return o1.sum == o2.sum;}
    friend bool operator!=(myclass &o1,myclass &o2){return o1.sum != o2.sum;}
};
```

Standard Template Library (STL)

Containere de tip secventa
List

Exemplu - stocarea obiectelor in list

```
int main() {
    int i;
    list<myclass> lst1;
    list<myclass>::iterator p;

    for(i=0; i<10; i++) lst1.push_back(myclass(i, i));
    for (p = lst1.begin(); p != lst1.end(); p++) cout<< p->getsum()<< " ";
    cout << endl;
// create a second list
    list<myclass> lst2;
    for(i=0; i<10; i++) lst2.push_back(myclass(i*2, i*3));
    for (p = lst2.begin(); p != lst2.end(); p++) cout<< p->getsum()<< " ";
    cout << endl;
// now, merget lst1 and lst2
    lst1.merge(lst2);
    return 0;
}
```

Standard Template Library (STL)

Containere de tip secventa

Deque (double ended queue) - Coadă dublă (completă)

- elementele sunt stocate în blocuri de memorie (chunks of storage)
- elementele se pot adăuga/șterge eficient de la ambele capete

Vector vs Deque

- Accesul la elemente de pe orice poziție este mai eficient la vector
- Inserare/ștergerea elementelor de pe orice poziție este mai eficient la Deque (dar nu e timp constant)
- Pentru liste mari Vector alocă zone mari de memorie, deque aloca multe zone mai mici de memorie – Deque este mai eficient în gestiunea memoriei

Standard Template Library (STL)

Adaptor de container

- încapsulează un container de tip secvență, și folosesc acest obiect pentru a oferi funcționalități specifice containerului (stivă, coadă, coadă cu priorități).

Stack: strategia LIFO (last in first out) pentru adaugare/ștergere elemente
Operații: empty(), push(), pop(), top().
template < class T, class Container = deque<T> > class stack;

Queue: strategia FIFO (first in first out)
Operații: empty(), front(), back(), push(), pop(), size();
template < class T, class Container = deque<T> > class queue;

Priority_queue: se extrag elemente pe baza priorităților
Operații: empty(), top(), push(), pop(), size();
template < class T, class Container = vector<T>, class Compare = less<typename Container::value_type> > class priority_queue;

Standard Template Library (STL)

Adaptor de container

Exemplu stiva

```
#include <stack>
#include<iostream>
using namespace std;
void sampleStack() {
     stack<int> s;
     s.push(3);
     s.push(4);
     s.push(1);
     s.push(2);

      while (!s.empty())      {
         cout << s.top() << " ";
         s.pop();     }
}
int main() {
sampleStack(); // 2, 1, 4, 3
}
```

Standard Template Library (STL)

Adaptor de container

Exemplu stiva

```
#include <stack>
#include <vector>
#include<iostream>
using namespace std;
void sampleStack() {

    stack<int,vector<int>> s;  // primul parametru = tipul elementelor, al doilea parametru, stilul de stocare

    s.push(3);
    s.push(4);
    s.push(1);
    s.push(2);

     while (!s.empty())      {
        cout << s.top() << " ";
        s.pop();     }
}
int main() {
sampleStack(); // 2, 1, 4, 3
}
```

Standard Template Library (STL)

Adaptor de container

Exemplu stiva - influenta celui de-al doilea parametru (eventual)

```
class Test { int i;
public:
Test(int x = 0):i(x){cout<<"C ";};
Test(const Test& x){i = x.i; cout<<"CC ";}
~Test(){cout<<"D ";}};

int main()
{
    stack<Test> s;
    s.push(1); // C CC D
    cout<<endl;
    s.push(3); // C CC D
    cout<<endl;
    s.push(5); // C CC D urmat de D D D (distrugerea listei)
}
```

Standard Template Library (STL)

Adaptor de container

Exemplu stiva - influenta celui de-al doilea parametru (eventual)

```
class Test { int i;
public:
Test(int x = 0):i(x){cout<<"C ";};
Test(const Test& x){i = x.i; cout<<"CC ";}
~Test(){cout<<"D ";}};

int main()
{
    stack<Test, vector<Test> > s;
    s.push(1); // C CC D
    cout<<endl;
    s.push(3); // C CC CC D D
    cout<<endl;
    s.push(5); // C CC CC CC D D D urmat de D D D
}
```

Standard Template Library (STL)

Adaptor de container

Exemplu stiva - influenta celui de-al doilea parametru (eventual)

```
class Test { int i;
public:
Test(int x = 0):i(x){cout<<"C ";};
Test(const Test& x){i = x.i; cout<<"CC ";}
~Test(){cout<<"D ";}};

int main()
{
    stack<Test, list<Test> > s;
    s.push(1); // C CC D
    cout<<endl;
    s.push(3); // C CC D
    cout<<endl;
    s.push(5); // C CC D urmat de D D D (distrugerea listei)
}
```

Standard Template Library (STL)

Adaptor de container

Exemplu coada

```
#include <queue>
#include<iostream>
using namespace std;
void sampleQueue() {
    queue<int> s;
//queue<int,deque<int> >
//queue<int,list<int> >

    s.push(3);
    s.push(4);
    s.push(1);
    s.push(2);

     while (!s.empty())      {
        cout << s.top() << " ";
        s.pop();     }
}
int main() {
sampleQueue(); // 3, 4, 1, 2
}
```

Standard Template Library (STL)

Adaptor de container

Exemplu coada cu prioritate

```
#include <queue>
#include<iostream>
using namespace std;
void samplePriorQueue() {
    priority_queue<int> s;
//priority_queue<int,deque<int> >
//priority_queue<int,list<int> >

    s.push(3);
    s.push(4);
    s.push(1);
    s.push(2);

     while (!s.empty())      {
        cout << s.top() << " ";
        s.pop();     }
}
int main() {
samplePriorQueue(); // 4, 3, 2, 1
}
```

Standard Template Library (STL)

Containere asociative

eficiente în accesare elementelor folosind chei (nu folosind poziții ca și în cazul containerelor de tip secvență).

- **Set**
  - mulțime - stochează elemente distincte.
  - se folosește arbore binar de căutare ca și reprezentare internă

- **Map**
  - dictionar - stochează elemente formate din cheie și valoare
  - nu putem avea chei duplicate
  - **multimap** poate avea chei duplicate

- **Bitset**
  - container special pentru a stoca biti.

Standard Template Library (STL)

Containere asociative
Map

- in sens general, map = lista de perechi cheie - valoare

template <class Key, class T, class Comp = less<Key>, class Allocator = allocator <pair<const Key,T> > class map

Key = tipul cheilor

T = tipul valorilor

Comp = functie care compara 2 chei

Allocator = tipul de alocator utilizat (in general cel standard).

Inserarea se face ordonat dupa chei.

Standard Template Library (STL)

Containere asociative
Map

Constructori:

explicit map(const Comp &cmpfn = Comp( ), const Allocator &a = Allocator( ));

// expl. map<char,int> m; *- map cu zero elemente;*

map(const map<Key,T,Comp,Allocator> &ob);

template <class InIter>
map(InIter start, InIter end, const Comp &cmpfn = Comp( ), const Allocator &a = Allocator( ));

Standard Template Library (STL)

Containere asociative
Map

Functii membre uzuale (sursa H. Schildt)

Member	**Description**
iterator begin( ); const_iterator begin( ) const;	returneaza un iterator catre primul element;
iterator end( ); const_iterator end( ) const;	returneaza un iterator catre ultimul element;
void clear( );	elimina toate elementele din map.
size_type count(const key_type &k) const;	- numarul de aparitii ale lui k
bool empty( ) const;	“true(false)” daca vectorul e (sau nu) gol.

Standard Template Library (STL)

Containere asociative
Map

Functii membre uzuale (sursa H. Schildt)

Member	**Description**
iterator erase(iterator i);	stergerea elementului pointat de i; returneaza un iterator catre elementul de dupa cel sters.
iterator erase(iterator start, iterator end);	stergerea elementelor intre start si end.
size_type erase(const key_type &k)	- stergerea tuturor valorilor k
iterator find(const key_type &k); const_iterator find(const key_type &k) const;	- returneaza un iterator catre valoarea cautata k sau catre sfarsitul map daca nu exista.

Standard Template Library (STL)

Containere asociative
Map

Functii membre uzuale (sursa H. Schildt)

Member	**Description**

size_type size( ) const;

iterator insert(iterator i, const value_type &val); - insereaza val pe pozitia pointata de i sau dupa;

template <class InIter>
void insert(InIter start, InIter end); - insereaza o secventa start-end.

pair <iterator,bool>
insert(const value_type &val);

mapped_type & operator[ ](const key_type &i) - returneaza o referinta la elementul cheie i, daca nu exista, atunci se insereaza.

Standard Template Library (STL)

Containere asociative

Map

Exemplu

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    map<int, int> m;
    m.insert(pair<int, int>(1,100));
    m.insert(pair<int, int>(7,300));
    m.insert(pair<int, int>(3,200));
    m.insert(pair<int, int>(5,400));
    m.insert(pair<int, int>(2,500));

    map<int, int>::iterator p;
    for(p = m.begin(); p!= m.end(); p++)
        cout<<p->first<<" "<<p->second<<endl;

//copierea intr-un alt map
map<int,int>m2(m.begin(),m.end());
}
```

Standard Template Library (STL)

Containere asociative

Map

Exemplu

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    map<char, int> m;
    for(int i=0; i<26; i++)
        m.insert(pair<char, int>('A'+i, 65+i));

    char ch;    cout << "Enter key: ";    cin >> ch;
    map<char, int>::iterator p;
// find value given key
    p = m.find(ch);
    if(p != m.end())
        cout << "Its ASCII value is " << p->second;
    else  cout << "Key not in map.\n";
    return 0;
}
```

Standard Template Library (STL)

Containere asociative

Multimap

Exemplu - catalog studenti (cheia = nume si valoarea = numar matricol)

```
#include <iostream>
#include <map>
using namespace std;
int main( ) {
    multimap<string, int> m;
    m.insert(m.end(), make_pair("Ionescu",100));
    // echivalent cu m.insert(pair<string,int>("Ionescu",100));

    m.insert(m.end(), make_pair("Popescu",355));
    m.insert(m.end(), make_pair("Ionescu",234));

    map<string, int>::iterator p;
    for(p = m.begin(); p!= m.end(); p++)
        cout<<p->first<<" "<<p->second<<endl;
```

Standard Template Library (STL)

Containere asociative

Multimap

Exemplu - catalog studenti (cheia = nume si valoarea = numar matricol)

```
//Afisarea elementelor cu un anumit nume
    string nume;
    cin>>nume;
    for(p = m.begin(); p!= m.end(); p++)
    if (p->first == nume)
        cout<<p->first<<" "<<p->second<<endl;

//stergerea primei aparitii a unui nume
    cin>>nume;
    map<string, int>::iterator f;
    f = m.find(nume);
    if (f!=m.end()) m.erase(f);
    for(p = m.begin(); p!= m.end(); p++)
        cout<<p->first<<" "<<p->second<<endl;
```

Standard Template Library (STL)

Containere asociative

Multimap

Exemplu - catalog studenti (cheia = nume si valoarea = numar matricol)

```
//Stergerea tuturor aparitiilor unui nume
    cin>>nume;

m.erase(nume);
```

Standard Template Library (STL)

Iterator

- un concept fundamental in STL, este elementul central pentru algoritmi oferiţi de STL;
- obiect care gestionează o poziție (curentă) din containerul asociat;
- suport pentru traversare (++,--), dereferențiere (*it);
- permite decuplarea intre algoritmi si containere.

Tipuri de iteratori:

• iterator input/output (istream_iterator, ostream_iterator)
• forward iterators, iterator bidirectional, iterator random access
• reverse iterators.

Adaptoarele de containere (container adaptors) - stack, queue, priority_queue - nu oferă iterator!

Standard Template Library (STL)

Iterator

Functii uzuale:

- **begin()** - returneaza pozitia de inceput a containerului
- **end()** - returneaza pozitia de dupa terminarea containerului
- **advance()** - incrementeaza pozitia iteratorului
- **next()** - returneaza noul iterator dupa avansarea cu o pozitie
- **prev()** - returneaza noul iterator dupa decrementarea cu o pozitie
- **inserter()** – insereaza elemente pe orice pozitie

Standard Template Library (STL)

Iterator

Exemplu - begin, end, advance, next, prev:

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

using namespace std;
int main()
{
    vector<int> v(5);
    for(int i = 0; i<5; i++) v[i] = (i+1)*10;

    vector<int>::iterator p;
    for (p = v.begin(); p < v.end(); p++)
        cout << *p << " ";
    cout<<endl;
    p = v.begin();
    advance(p, 2);
    vector<int>::iterator n = next(p);
    vector<int>::iterator d = prev(p);
    for (; p < v.end(); p++)
        cout << *p << " ";
    cout<<endl<<*n<<" "<<*d;

}
```

```
10 20 30 40 50
30 40 50
40 20
```

Standard Template Library (STL)

Iterator

Exemplu - inserter:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> v = { 1, 2, 3, 4, 5 };
    vector<int> v2 = { 10, 20, 30};
    vector<int>::iterator p= v.begin();
    advance(p, 2);
    copy(v2.begin(),v2.end(),inserter(v,p));
    for (p = v.begin(); p < v.end(); p++)
        cout << *p << " ";
    return 0;
}
```

1 2 10 20 30 3 4 5

Standard Template Library (STL)

Reverse iterator

Exemplu

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> v = { 1, 2, 3, 4, 5 };

    vector<int>::iterator p;
    for (p = v.begin(); p < v.end(); p++)
        cout << *p << " ";
    cout<<endl;
    vector<int>::reverse_iterator r;
    for (r = v.rbegin(); r < v.rend(); r++)
        cout << *r << " ";
    return 0;
}
```

Standard Template Library (STL)

Algoritm

- colecție de funcții template care pot fi folosite cu iteratori. Funcțiile operează pe un domeniu (range) definit folosind iteratori;

- domeniu (range) este o secvente de obiecte care pot fi accesate folosind iteratori sau pointeri.

- headere: <**algorithm**>, <**numeric**>

Standard Template Library (STL)

Algoritm

- **Operații pe secvențe**
  - care nu modifică sursa: accumulate, count, find, count_if, etc
  - care modifică : copy, transform, swap, reverse, random_shuffle, etc.

- **Sortări**: sort, stable_stort, etc.

- **Pe secvențe de obiecte ordonate**
  - **Căutare binară** : binary_search, etc
  - **Interclasare (Merge)**: merge, set_union, set_intersect, etc.

- **Min/max:** min, max, min_element, etc.

- **Heap**: make_heap, sort_heap, etc.

Standard Template Library (STL)

Algoritm

Exemplu: – accumulate : calculează suma elementelor

```
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(2);
v.push_back(7);
v.push_back(17);

//compute the sum of all elements in the vector
cout << accumulate(v.begin(), v.end(), 0) << endl;

//compute the sum of elements from 1 inclusive, 4 exclusive [1,4)
vector<int>::iterator start = v.begin()+1;
vector<int>::iterator end = v.begin()+4;
cout << accumulate(start, end, 0) << endl;
```

Standard Template Library (STL)

Algoritm

Exemplu: – copy

```
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(2);
v.push_back(7);
v.push_back(17);

//make sure there are enough space in the destination
//allocate space for 5 elements
vector<int> v2(5);

//copy all from v to v2
copy(v.begin(), v.end(), v2.begin());
```

Standard Template Library (STL)

Algoritm

Exemplu: – sort

Sorteaza elementele din intervalul [first,last) ordine crescătoare.
• Elementele se compară folosind **operator <**

```
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(2);
v.push_back(7);
v.push_back(17);

sort(v.begin(), v.end());
```

Standard Template Library (STL)

Algoritm

Exemplu: – sort cu o functie de comparare

```
bool asc(int i, int j) {
  return (i < j);
}

vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(2);
v.push_back(7);
v.push_back(17);

sort(v.begin(), v.end(), asc);
```

Standard Template Library (STL)

Algoritm

Exemplu: – for_each

```
void print(int elem) {
cout << elem << " ";
}

void testForEach() {
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(2);
v.push_back(7);
v.push_back(17);

for_each(v.begin(), v.end(), print);
cout << endl;
}
```

Standard Template Library (STL)

Algoritm

Exemplu: – transform

- aplică funcția pentru fiecare element din secvență ([first1,last1))
- rezultatul se depune în secvența rezultat

```
int multiply3(int el) {
return 3 * el;
}
void testTransform() {
vector<int> v;
v.push_back(3); v.push_back(4);
v.push_back(2);v.push_back(7);
v.push_back(17);

vector<int> v2(5);
transform(v.begin(), v.end(), v2.begin(), multiply3);
//print the elements
for_each(v2.begin(), v2.end(), print);
}
```

Standard Template Library (STL)

C++11

Lambda Expressions

- permite definirea functiei local, la momentul apelarii;

Sintaxa: ***[capture](parameters) mutable exception -> return-type {body}***

capture - partea introductiva, care spune compilatorului ca urmeaza o expresie lambda; aici se specifica si ce variabile si in ce mod (valoare sau referinta) se copiaza din blocul in care expresia lambda este definita;

parameters - parametrii expresiei lambda;

mutable (optional) – permite modificarea parametrilor transmisi prin valoare
exception (optional) – a se utiliza **noexcept** daca nu se arunca nicio expresie
return – type (optional) - tipul la care se evalueaza expresia lambda; aceasta parte este optionala, cel mai adesea compilatorul putand deduce implicit care este tipul expresiei.

body – corpul expresiei

Standard Template Library (STL)

C++11
Lambda Expressions

Blocul ***capture*** **[ ]**

- Expresiile lambda pot “captura” variabilele din program sau poate introduce variabile locale (incepand cu C++14).
- Variabilele transmise cu & sunt accesate prin referinta, iar celelalte prin valoare.
- Setul [ ] fara parametri semnifica faptul ca lambda nu acceseaza variabile din acelasi bloc de program.

```
int a = 0;                          // Define an integer variable
auto f = []()   { return a*9; }; // Error: 'a' cannot be accessed
auto f = [a]()  { return a*9; }; // OK, 'a' is "captured" by value
auto f = [&a]() { return a++; }; // OK, 'a' is "captured" by reference
                                    //      Note: It is the responsibility of the programmer
                                    //      to ensure that a is not destroyed before the
                                    //      lambda is called.
auto b = f();                       // Call the lambda function. a is taken from the capture
```

Standard Template Library (STL)

C++11
Lambda Expressions

Blocul ***capture*** **[ ] – observatii / restrictii**

- Daca [ ] contine & by default, atunci nu se poate declara un argument particular tot cu &
- Daca [ ] contine = by default, atunci nu se poate declara un argument particular tot prin valoare

```
struct S { void f(int i); };

void S::f(int i) {
    [&, i]{};       // OK
    [&, &i]{};      // ERROR: i preceded by & when & is the default
    [=, this]{};    // ERROR: this when = is the default
    [=, *this]{ }; // OK: captures this by value. See below.
    [i, i]{};       // ERROR: i repeated
}
```

Standard Template Library (STL)

C++11

Lambda Expressions – exemplu

```
#include <bits/stdc++.h>
#include <vector>

using namespace std;

int main() {
   vector<int> v={1,2,1,3,1,1};
//afisare prin lambda expresie
   for_each(v.begin(), v.end(), [ ](int i){ cout << i << ' '; });
   cout << '\n';

// contorizare prin lambda expresie
   int cont = 0; //modificat prin lambda expresie
   for_each(v.begin(), v.end(),[&cont](int i){
    if (i == 1)
     cont++;
    });
    cout<< cont <<" de 1 ";
return 0;
}
```

Standard Template Library (STL)

C++11

Lambda Expressions – exemplu

```
#include <iostream>
#include <string>
using namespace std;
int main()
   {
      auto sum = [](int a, int b) {
      return a + b;
      };

   cout <<"Sum of two integers:"<< sum(5, 6) << endl;

   return 0;
   }
```

Standard Template Library (STL)

C++14

Lambda Expressions – exemplu cu tipuri de date generalizate (incepand cu C++14)

```
#include <iostream>
#include <string>
using namespace std;
int main()
   {

      // generalized lambda
      auto sum = [](auto a, auto b) {
         return a + b;
         };

      cout <<"Sum(5,6) = "<< sum(5, 6) << endl; // sum of two integers

      cout <<"Sum(2.0,6.5) = "<<sum(2.0, 6.5) << endl; // sum of two floa

      cout <<"Sum((string(\"SoftwareTesting\"), string(\"help.com\")) =
      "<<sum(string("SoftwareTesting"), string("help.com")) << endl; // s

      return 0;
   }
```

Standard Template Library (STL)

C++14

Lambda Expressions – exemplu cu tipuri de date generalizate (incepand cu C++14)

```
#include <iostream>
#include <string>
using namespace std;
int main()
   {

      // generalized lambda
      auto sum = [](auto a, auto b) {
         return a + b;
         };

      cout <<"Sum(5,6) = "<< sum(5, 6) << endl; // sum of two integers

      cout <<"Sum(2.0,6.5) = "<<sum(2.0, 6.5) << endl; // sum of two floa

      cout <<"Sum((string(\"SoftwareTesting\"), string(\"help.com\")) =
      "<<sum(string("SoftwareTesting"), string("help.com")) << endl; // s

      return 0;
   }
```

Standard Template Library (STL)

C++11

Deducerea tipului in mod automat (auto si decltype)

In C++03, fiecare variabila trebuie sa aiba un tip de data;

Cuvantul cheie "auto"

In C++11, acesta poate fi dedus din initializarea variabilei respective;
In cazul functiilor, daca tipul returnat este ***auto*** , este evaluata de expresia returnata la runtime;

Variabilele ***auto*** TREBUIE initializate, altfel eroare.

Standard Template Library (STL)

C++11

Deducerea tipului in mod automat (auto si decltype)

```
#include <bits/stdc++.h>
using namespace std;
class Test{ };
int main() {
    auto x = 1; // x e int
    float z;
    auto y = z;// y e float
    auto t = 3.37; // t e double
    auto p = &x; // pointer catre int

    Test ob;
    auto A = ob;
    auto B = &ob;

    cout << typeid(x).name() << endl << typeid(y).name() << endl
         << typeid(t).name() << endl << typeid(p).name() << endl
         << typeid(A).name() << endl  << typeid(B).name() << endl;

    return 0;
}
```

Standard Template Library (STL)

C++11

Deducerea tipului in mod automat (auto si decltype)

```
#include <bits/stdc++.h>
#include <vector>
using namespace std;
int f(){}
int& g(){}

int main() {
vector<int> v = {1,2,4};

    auto x = v.begin();
    cout << typeid(x).name() << endl;
    for(auto p = v.begin(); p!=v.end(); p++)
        cout<<*p<<" ";
    cout<<endl;
    auto y = f();
    auto z = g();//    auto& z = g();
    cout << typeid(y).name() << endl << typeid(z).name() << endl;
    return 0;
}
```

Standard Template Library (STL)

C++11

Deducerea tipului in mod automat (auto si decltype)

```
int f(){}
float g(){}
// atentie: daca float& g(){}, eroare la decltype(g()) z, intrucat trebuie initializat

int main()
{

    decltype(f()) y;
    decltype(g()) z;
    cout << typeid(y).name() << endl;
    cout << typeid(z).name() << endl;

    return 0;
}
```

auto – permite declararea unei variabile cu un tip specific, iar decltype "ghiceste" tipul

Standard Template Library (STL)

C++11

Deducerea tipului in mod automat (auto si decltype)

```
#include <bits/stdc++.h>
using namespace std;

template <class T1, class T2>
auto findMin(T1 a, T2 b) -> decltype(a < b ? a : b)
{
    return (a < b) ? a : b;
}

int main()
{
    auto x = findMin(4, 3.44);
    cout << typeid(x).name() << endl;
    decltype(findMin(5.4, 3)) y;
    cout << typeid(y).name() << endl;

    return 0;
}
```

Perspective

Cursul 12:

Design Patterns