Standard Template Library - STL

Ce oferă bibliotecile standard C++?

- Suport pentru proprietățile limbajului (*gestionarea memoriei*, *RTTI*).
- Informații despre implementarea compilatorului (de exemplu cea mai mare valoare pentru numere de tip `float`).
- Funcții ce nu pot fi implementate optimal în limbaj pentru orice sistem (`sqrt()`, `memmove()` etc.).
- Suport pentru lucrul cu şiruri de caractere şi stream-uri (include suport pentru internaționalizare şi localizare).
- Un framework pentru containere (`vector`, `map`, …) şi algoritmi generici pentru ele (*parcurgere*, *sortare*, *reuniune*, …).
- Suport pentru prelucrări numerice.

STL - Standard Template Library

- Bibliotecă C++ ce conține clase pentru *containere*, *algoritmi* şi *iteratori*.
- Furnizează majoritatea algoritmilor de bază şi a structurilor de date necesare în dezvoltarea de programe.
- *Bibliotecă generică* – componentele sunt *parametrizate* – aproape fiecare componentă din STL este un *template*.
- Prima bibliotecă generică a limbajului C++ folosită pe scară largă.

STL – Conținut

- **Containere -** obiecte ce conțin alte obiecte.
- **Iteratori -** “pointeri” pentru parcurgerea containerelor.
- **Algoritmi generici -** funcții ce se aplică diverselor tipuri de containere.
- **Clase adaptor -** clase ce adaptează alte clase (variații ale altor containere).
- **Alocatori** - obiecte responsabile cu alocarea spațiului.

STL – `String`

- Un tip special de container.
- Capacitate:

  `size, length, max_size, resize, capacity, reserve, clear, empty`
- Acces la elemente: `[], at`
- Modificatori:

  `+=, append, push_back, assign, insert, erase, replace, copy, swap`

STL – `String`

- `c_str` – returnează reprezentarea C corespunzătoare şirului de caractere (secvență de caractere terminată cu valoarea ‘\0’).
- `data` – returnează un vector de caractere fără valoarea ‘\0’ la sfârşit.
- `find` – caută prima apariție a unui caracter/şir de caractere.
- `rfind` – caută ultima apariție a unui caracter/şir de caractere.
- `find_first(last)_of` – caută în şirul de caractere oricare din caracterele date ca parametru.
- `find_first(last)_not_of` – caută în şirul de caractere primul caracter ce nu apare în parametru.
- `substr` - returnează un şir de caractere cu proprietatea că el este un subşir al obiectului curent.
- `compare` – compară 2 şiruri de caractere (similar cu funcția `strcmp`).

STL – String

- header

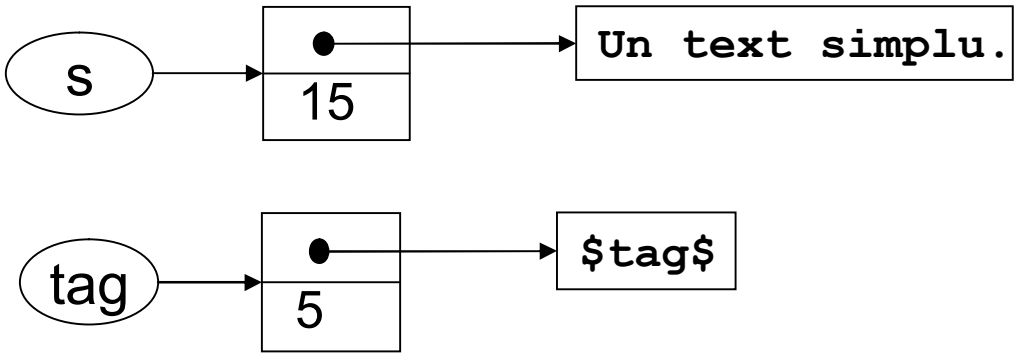
```
#include <string>
```

- spațiul de nume folosit

```
using namespace std;
```

- declarații şiruri:

```
string s("Un text simplu.");
string tag("$tag$");
```



STL - String. Exemple

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string str1 = "Hello";
  string str2 = "World";
  string str3;
  int  len;

  str3 = str1;
  cout << "str3 : " << str3 << endl;

  str3 = str1 + str2;
  cout << "str1 + str2 : " << str3 << endl;

  len = str3.size();
  cout << "str3.size() :  " << len << endl;

  return 0;
}
```

```
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

int main() {
  // Conversie de la string la double
  stringstream ssio;
  string txt("3.14159");
  ssio << txt;
  double x;
  ssio >> x;
  cout << x << endl;

  // Conversie de la double la string
  stringstream ssio2;
  double y = 2.71828;
  ssio2 << y;
  string ctxt(ssio2.str());
  cout << ctxt << endl;

  return 0;
}
```

STL - `String`. Exemple

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string quote1("There are two sides to every issue: one side is right\n");
  string quote2("and the other is wrong, but the middle is always evil.");

  quote1.append(quote2);
  cout << quote1 << endl;

  string substring = quote1.substr(36, 17);
  cout << substring << endl;

  quote1.erase(34, 42);
  cout << quote1 << endl;

  return 0;
}
```

STL - `String`. Exemple

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string quote1("A little learning is a dangerous thing.");
  cout << (unsigned int)quote1.find('i') << endl;

  string quote2("To err is human; to forgive, divine.");
  string letters("dfimh");
  cout << (unsigned int)quote2.find_first_of(letters) << endl;

  string line1("If white and black blend, soften and unite");
  string line2("a thousand ways, is there no black or white?");

  string quote3(line1 + line2);
  string black("black");

  cout << (unsigned int)quote3.find(black) << endl;
  cout << (unsigned int)quote3.rfind(black) << endl;

  return 0;
}
```

STL – Avantaje

- micşorează timpul de implementare
- structuri de date deja implementate şi testate
- cod mai uşor de citit
- creşte robusteţea codului
- structurile STL sunt actualizate automat
- cod portabil
- creşte mentenabilitatea codului
- uşurinţă în programare.

STL - Containere

- *Vector*
- *List*
- *Deque*

Containere secvență

- *Queue*
- *Stack*
- *Priority queue*

Adaptori de containere

- *Set*
- *Map*
- *Multiset*
- *Multimap*
- *Bitset*

Containere asociative (cheie-valoare)

STL - `Vector`

- Elimină problema realocării dinamice a spațiului de memorie.
- Se redimensionează în partea finală astfel încât să poată fi realizată adăugarea de noi elemente.
- Compus dintr-un bloc de memorie alocată secvențial.
- *Timp constant* pentru inserție şi eliminarea componentelor de la sfârşit.
- *Timp liniar* pentru inserarea şi eliminarea elementelor de la început şi interior.
- Devine ineficient când depăşirile de memorie alocată pot determina copierea întregului vector.

```
#include <vector>
std::vector<int> tab;

std::vector<int>::iterator tabIt;
```

Declarație vector de numere întregi

Declarație iterator pt vector de numere întregi

- Cerințe pentru tipul T:
  - `T(); T(const T&); ~T(); T& operator=(const T&);`

STL – `Vector`. Operații

Operație	Descriere	Complexitate
`operator=`	atribuire	*O(n)*
`begin`	returnează un *iterator* ce referă primul element	*O(1)*
`end`	returnează un *iterator* ce referă elementul (teoretic) ce urmează ultimului element; *nu referă un element fizic, deci nu trebuie dereferențiat*	*O(1)*
`rbegin`	returnează un *reverse_iterator;* indică *ultimul element* din cadrul vectorului	*O(1)*
`rend`	returnează un *reverse_iterator;* indică elementul teoretic ce *precede primul element* al vectorului	*O(1)*
`size`	returnează dimensiunea vectorului	*O(1)*
`max_size`	returnează dimensiunea maximă a vectorului	
`resize`	redimensionează vectorul la valoarea specificată	*O(1)/O(n)*

STL – `Vector`. Operații

`capacity`	returnează capacitatea vectorului	*O(1)*
`empty`	returnează *true* dacă vectorul este vid	*O(1)*
`reserve`	schimbă capacitatea vectorului	*O(1)/O(n)*
`operator[]`	returnează o referință la obiectul de pe poziția specificată	*O(1)*
`at`	accesează elementul de la poziția parametrului	*O(1)*
`front`	accesează primul element al vectorului	*O(1)*
`back`	accesează ultimul element al vectorului	*O(1)*
`assign`	atribuie vectorului un nou conținut	*O(n)*
`push_back`	adaugă un element la sfârşit	*O(1)/O(n)*

STL – `Vector`. Operații

`pop_back`	şterge ultimul element din vector	*O(1)*
`insert`	inserează elemente înaintea unei poziții	*O(n)*
`erase`	şterge din vector un element de la o poziție sau elementele dintr-un domeniu `[first, last)`	*O(n)*
`swap`	interschimbă 2 vectori	*O(1)*
`clear`	distruge elementele din vector şi dimensiunea devine 0	*O(n)*

STL – List

- Compusă din obiecte ce au structura *anterior-info-urmator.*
- Nu deține proprietatea (*ownership*) asupra elementelor.
- Este folosită atunci când se fac inserări/ştergeri oriunde în cadrul listei.
- *Timp constant (amortizat)* pentru inserție şi eliminare la început, la sfârşit sau în interior.
- Cerințe pentru tipul T:
  - `T(); T(const T&); ~T(); T& operator=(const T&); T* operator&(); int operator<(const T&, const T&); int operator==(const T&, const T&);`

```
#include <list>

std::list<int> lista;

std::list<int>::iterator listaIt;
```

Declarație listă de numere întregi

Declarație iterator pt lista de întregi

STL – `List`

Operație	Descriere	Complexitate
`operator=`	atribuire	*O(n)*
`swap`	interschimbă elementele a două liste (pointerii `head` şi `tail`)	*O(1)*
`begin`	returnează un iterator către primul element al listei	*O(1)*
`end`	returnează un iterator către un element ce urmează ultimului element al listei	*O(1)*
`front`	returnează primul element al listei	*O(1)*
`back`	returnează ultimul element al listei	*O(1)*

STL – `List`

`empty`	returnează *true* dacă lista e vidă	*O(1)*
`size`	returnează numărul de elemente al listei	*O(n)* sau *O(1)*
`push_front`	inserează un element în capul listei	*O(1)*
`push_back`	adaugă un element la sfârşitul listei	*O(1)*
`insert`	inserează un element înaintea elementului indicat de iterator	*O(1)*
`pop_front`	şterge primul element din listă	*O(1)*
`pop_back`	şterge ultimul element din listă	*O(1)*
`remove`	şterge toate aparițiile unui element din listă (foloseşte operatorul “==“)	*O(n)*
`erase`	2 forme – şterge elementul indicat de iterator sau secvența dintre 2 iteratori	*O(1)* sau *O(n)*

STL – `List`

`sort`	ordonează elementele listei în ordine ascendentă. Operatorul ‘<‘ trebuie să fie supraîncărcat.	$O(n\log_2 n)$
`reverse`	inversează ordinea elementelor unei liste	$O(n)$
`merge`	interclasează elementele a două liste. În urma operației, elementele din lista argument sunt inserate în lista apelantă. Ambele liste trebuie să fie ordonate.	$O(m+n)$ m,n lungimile listelor

STL – `Deque`

- *Deque* = *double ended queue* (coadă având două capete; practic operațiile de inserare/ştergere se realizează la ambele capete).
- Nu este o structură de date cu acces de tip FIFO.
- Permite adăugarea/ştergerea elementelor la ambele capete.
- Permite inserarea sau ştergerea de elemente la poziții arbitrare.

```
#include <deque>
std::deque<int> deque;
std::deque<int>::iterator dequeIt;
```

Declaratie coadă de numere întregi

Declarație iterator coadă de întregi

- Accesul la elemente se poate realiza similar vectorilor, prin intermediul operatorului '`[]`' sau metodei `at()`.

STL – `Deque`. Operații

Operație	Descriere	Complexitate
`operator=`	atribuire	*O(n)*
`swap`	interschimbă elementele a două cozi (pointerii `head` şi `tail`)	*O(1)*
`begin`	returnează un iterator către primul element al cozii	*O(1)*
`end`	returnează un iterator către un element ce urmează ultimului element al cozii	*O(1)*
`back`	returnează ultimul element al cozii	*O(1)*
`empty`	returnează *true* dacă coada este vidă	*O(1)*
`front`	returnează primul element al cozii	*O(1)*

STL – `Deque`. Operații

`size`	returnează numărul de elemente al cozii	*O(n) sau O(1)*
`push_front`	inserează un element la începutul cozii	*O(1)*
`push_back`	adaugă un element la sfârşitul cozii	*O(1)*
`insert`	inserează un element înaintea elementului indicat de iterator	*O(1)*
`pop_front`	şterge primul element din coadă	*O(1)*
`pop_back`	şterge ultimul element din coadă	*O(1)*
`remove`	şterge toate aparițiile unui element din coadă (foloseşte operatorul '==')	*O(n)*

STL – `Deque`. Operații

`erase`	2 forme – şterge elementul indicat de iterator sau secvența dintre 2 iteratori	*O(1)* sau *O(n)*
`sort`	sortează elementele cozii în ordine ascendentă. Operatorul '<' trebuie să fie supraîncărcat	$O(n\log_2 n)$
`reverse`	inversează ordinea elementelor unei cozi	*O(n)*
`merge`	interclasează elementele a două cozi. În urma operației, elementele din coada argument sunt inserate în coada apelantă. Ambele cozi trebuie să fie ordonate.	*O(m+n)* *m,n* lungimile cozilor

STL - `Vector` **vs** `Deque`

- *Deque* permite *inserarea/ştergerea* elementelor de la *începutul/sfârşitul* cozii în timp **constant**.
- *Vectorul* permite *inserarea/ştergerea* elementelor doar la *sfârşit* în timp **constant**.
- *Deque* permite *inserarea/ştergerea* de elemente la *poziții arbitrare* mai eficient decât *vectorul*, *dar nu în timp constant*.
- *Accesul aleator* la elemente este mai rapid la *vector* decât la *deque*.
- Pentru secvențe mari de elemente, *vectorul* va aloca o zonă mare de *memorie contiguă*, pe când *deque* va aloca mai multe blocuri de memorie de dimensiuni mai mici – mai eficient din punctul de vedere al sistemelor de operare.
- *Deque* se comportă atât ca o stivă, cât şi ca o coadă.
- *Deque* se expandează mai eficient decât vectorii.

STL – Clase adaptor

- Simple variații ale containerelor secvență.
- Clase ce nu suportă iteratori.
- `stack<T> : <stack>` – LIFO (*last in, first out*)
- `queue<T> : <queue>` – FIFO (*first in, first out*)
- `priority_queue<T> : <queue>` – obiectul cu valoare maximă/minimă este întotdeauna primul în coadă.

STL – `Stack`

- O structură de date având un comportament de acces de tip **LIFO**.
- *Stivă* = *adaptor* d.p.v. al implementării în STL – preia ca parametru al template-ului un container secvență şi pune la dispoziție operațiile ce implementează comportamentul unei stive.

`#include <stack>`

`#include <C>` - C containerul dorit să fie adaptat la stivă

```
#include <stack>
#include <list>
std::stack<int, std::list<int>> stiva;
```

Implicit `deque` (lista consumă mai multă memorie, vectorii se expandează mai greu)

```
std::stack<int> stiva;
```

STL – `Stack`

Operație	Descriere
`empty`	Returnează *true* dacă stiva este vidă
`size`	Returnează numărul de elemente din stivă
`top`	Returnează o referință la elementul din vârful stivei
`push`	Adaugă un nou element în stivă
`pop`	Extrage elementul din vârful stivei

STL – Containere asociative

- `map<T> : <map>` – componentele sunt perechi `<cheie, valoare>` cu cheie unică (nu există două înregistrări cu aceeaşi cheie).
- `multimap<T> : <map>` – componentele sunt perechi `<cheie, valoare>` cu cheie multiplă (pot exista mai multe înregistrări cu aceeaşi cheie).
- `set<T> : <set>` – componentele sunt doar de tip cheie şi NU pot exista mai multe chei având aceeaşi valoare.
- `multiset<T> : <set>` – componentele sunt doar de tip cheie şi POT exista mai multe chei având aceeaşi valoare.

STL – `Set`

- Set = *mulțime*. Container în care toate elementele sunt distincte. Este implementată de obicei sub forma unui arbore binar de căutare.
- Păstrează elementele ordonate.
- Funcția `find()` are complexitate logaritmică.

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

int main(){
  set<int> intSet;
  for(int i = 0; i < 25; i++)
    for(int j = 0; j < 10; j++)
      intSet.insert(j);

  set<int>::iterator it = intSet.begin();

  while(it != intSet.end()){
    cout << *it << " ";
    it++;
  }
  return 0;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

STL – Multiset

- Permite apariția unui element de mai multe ori.

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

int main(){
  multiset<int> intSet;
  for(int i = 0; i < 25; i++)
    for(int j = 0; j < 10; j++)
      intSet.insert(j);

  multiset<int>::iterator it = intSet.begin();

  while(it != intSet.end()){
    cout << *it << " ";
    it++;
  }
  return 0;
}
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
```

STL – `Set`. Operații

Operație	Descriere	Complexitate
`swap`	interschimbă elementele a două mulțimi	*O(1)*
`lower_bound`	returnează un iterator către primul element >= o cheie. Iteratorul este setat pe *end()* dacă un astfel de element nu există.	*O($log_2 n$)*
`upper_bound`	returnează un iterator către elementul imediat următor parametrului. Returnează *end()* dacă un astfel de element nu există.	*O($log_2 n$)*
`insert`	adaugă un element la o mulțime	*O($log_2 n$)*
`begin`	returnează un iterator către primul element al mulțimii	*O(1)*

STL – `Set`. Operații

`size`	returnează dimensiunea mulțimii. Pentru testarea dacă este vidă, se preferă folosirea funcției *empty()* – este mai rapidă.	*O(n)*
`empty`	returnează *true* dacă mulțimea este vidă	*O(1)*
`find`	caută o cheie şi returnează un iterator către elementul găsit, altfel *end()*	$O(log_2 n)$

STL – Set. Multiset. unordered_{set, multiset}

set	multiset	unordered_set	unordered_multiset
Păstrează valorile în ordine. Păstrează numai valori unice. Elementele pot fi doar inserate sau şterse, şi nu pot fi modificate. Se pot şterge mai multe elemente. Se pot parcurge cu iteratori.	Păstrează valorile în ordine. Pot exista mai multe valori identice. Elementele pot fi doar inserate sau şterse, şi nu pot fi modificate. Se pot şterge mai multe elemente.	Elementele pot fi păstrate în orice ordine (nu sunt ordonate). Păstrează numai valori unice. Sunt utilizate tabele de dispersie pentru a păstra elemente.	Păstrează numai valori unice.

STL – `Map`

- Asociază chei de un anumit tip cu valori de alt tip.
- Elementele sunt accesate direct, pe baza cheii: `aMap["John Doe"] = 3.2;`
- **Cheile trebuie să fie unice.**

```
#include <map>
std::map<string, double> aMap;
std::map<string,double>::iterator it;
```

Declarație dicționar

Declarație iterator pt dicționar

- Operatorul '`[]`' – mod de lucru:
  - caută cheia în dicționar;
  - dacă este găsită se returnează o referință spre valoarea asociată;
  - dacă nu este găsită, se creează un nou obiect de tipul valorii şi se returnează o referință spre el.
- Pentru a verifica apartenența unui element la dicționar se foloseşte metoda `find()` – aceasta nu creează un obiect dacă cheia nu există.
- `Find` returnează un iterator către o pereche de obiecte (*cheie, valoare*) dacă cheia există, altfel returnează *`end()`*.

STL – `Pair`

- `pair` este o clasă template ce are două date membre publice, accesibile folosind `pair::first` (*cheia*), respectiv `pair::second` (*valoarea*).
- `<utility>`
- Pentru a crea un obiect de tip `pair` – se foloseşte funcția template `make_pair` ce are 2 parametri:

```
pair<string,double> p;
p = make_pair(string("Microsoft Share Price"), double(85.27));
```

- Cerințe asupra tipului cheilor/valorilor:
  - Să aibă operatorul '=' supraîncărcat.
  - Tipul cheilor trebuie să respecte următoarele constrângeri (ordine strictă):
    - `a<a` este fals;
    - Egalitatea se poate determina din expresia `(!(a<b) && !(b<a))`;
    - Dacă `a<b` şi `b<c`, atunci `a<c`.
  - Tipul valorilor trebuie să aibă definit un constructor implicit.

STL – `Map`. Operații

`swap`	interschimbă elementele a două dicționare	*O(1)*
`begin`	returnează un iterator către primul element al dicționarului	*O(1)*
`end`	returnează un iterator către ultimul element al dicționarului	*O(1)*
`size`	returnează numărul de elemente din dicționar	*O(n)*
`empty`	returnează *true* dacă dicționarul este vid	*O(1)*
`operator []`	returnează o referință către obiectul asociat cheii. Dacă cheia nu există, va crea un obiect nou.	$O(log_2 n)$
`find`	caută o cheie în dicționar. Dacă o găseşte, returnează un iterator către perechea *(cheie, valoare)*, altfel returnează un iterator setat pe `end()`.	$O(log_2 n)$
`erase`	- şterge elementul indicat de iterator - şterge secvența dintre 2 iteratori - primeşte ca parametru un obiect de tipul cheii şi şterge elementele cu cheia egală cu parametrul (cel mult 1 element)	

STL – Multimap

- Permite existența aceleiaşi chei de mai multe ori:

```
#include <map>

multimap<string, int> mMap;
multimap<string, int>::iterator itr;
```

- Pentru a găsi toate perechile cheie-valoare corespunzătoare unei chei, vom folosi 2 iteratori - unul setat pe prima pereche din dicționar şi altul setat pe ultima pereche din dicționar ce corespunde cheii.

```
multimap<string, int>::iterator itr;
multimap<string, int>::iterator lastElement;
itr = mMap.find(key);
if (itr == mMap.end())
  return;
cout << "Urmatoarele elemente sunt asociate cheii de valoare "
     << key << ":" << endl;
lastElement = mMap.upper_bound(key);
for ( ; itr != lastElement; ++itr)
  cout << itr->second << endl;
```

STL – `Multimap`. Operații

Operație	Descriere	Complexitate
`swap`	interschimbă 2 dicționare cu chei multiple	*O(1)*
`count`	returnează numărul valorilor asociate unei chei	*$O(\log_2 n+m)$* *m - numărul de valori asociate cheii*
`upper_bound`	returnează un iterator situat cu o pozitie după ultima apariție a cheii sau *end()*	*$O(\log_2 n)$*
`lower_bound`	returnează un iterator situat pe o pereche a cărei cheie este mai mare sau egală cu cheia dată ca parametru sau *end()*	*$O(\log_2 n)$*
`begin`	returnează un iterator pe prima pereche din dicționar	*O(1)*
`end`	returnează un iterator situat după ultimul element al dicționarului	*O(1)*

STL – `Multimap`. Operații

`size`	returnează numărul de perechi din dicționar	*O(n)*
`empty`	returnează *true* dacă dicționarul este vid	*O(1)*
`find`	primeşte o cheie ca parametru şi returnează un iterator pe prima pereche ce are cheia egală cu parametrul, sau *end()* dacă nu există	$O(log_2 n)$
`insert`	adaugă o pereche în dicționar	$O(log_2 n)$
`erase`	3 variante supraîncărcate: - şterge elementul indicat de iterator - şterge elementele dintre 2 iteratori - şterge perechile ce au cheia egală cu parametrul	$O(log_2 n+m)$ m - numărul de valori asociate cheii

STL – Iteratori

- Sunt similari pointerilor - elementele indicate sunt accesate indirect.
- *Reprezintă o abstractizare între container şi utilizatorul său: sunt folosiți pentru a naviga prin containere, fără să ştim ce tip de dată este utilizat pentru memorarea obiectelor*.
- Determină reducerea cuplarii (*GRASP - low coupling*) între funcții şi containerele accesate de acestea.
- Pentru a accesa elementul corespunzator, iteratorul trebuie **dereferențiat**.
- Fiecare container include funcțiile membru `begin()`, `end()` pentru specificarea valorilor iterator corespunzătoare *primului element*, respectiv *primului element după ultimul obiect* din container.
- Concepte cheie:
  - elementul curent la care face referire (**`p->`**, **`*p`**)
  - elementul următor/precedent (**`++p`**, **`--p`**)
  - comparații (‘**<**’ ‘**<=**’ ‘**==**’).

STL – Iteratori

Categoria	Output	Input	Forward	Bidirectional	Random
Citire		`=*p`	`=*p`	`=*p`	`=*p`
Acces		`->`	`->`	`->`	`-> []`
Scriere	`*p=`		`*p=`	`*p=`	`*p=`
Iterație	`++`	`++`	`++`	`++ --`	`++ -- + - += -+`
Comparare		`== !=`	`== !=`	`== !=`	`== != < > <= >=`

STL – Iteratori

- Majoritatea containerelor acceptă iteratori, cu excepția *stivei*, *cozii* şi *cozii cu priorități*.

```
<container>::iterator
<container>::const_iterator
```

Declarație iteratori

- Parcurgerea elementelor containerului folosind iteratori:

```
for (itr = container.begin(); itr != container.end(); ++itr)
    @proceseaza(*itr);
```

- Accesul la elementul curent se poate face:
  - folosind '*' sau '->' (dacă obiectul referit este un agregat)
    - `*itr = 3;`
    - `struct pair { int first, second; };`
    - `itr->first = 5; itr->second = 6;`

STL – Iteratori pe containere reversibile

- *Container reversibil* – produce iteratori ce parcurg o secvență de la sfârşit spre început.
- Toate containerele standard permit existența iteratorilor reversibili.

<container>::reverse_iterator
<container>::const_reverse_iterator

- Inițializare: rbegin(), rend()

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

using namespace std;

int main(){
 vector<int> v;
 v.push_back(3);
 v.push_back(4);
 v.push_back(5);

 vector<int>::reverse_iterator  rit
   = v.rbegin();
 while (rit < v.rend()) {
  cout << *rit << ' ';
  ++rit;
 }

 return 0;
}
```

5 4 3

STL – Iteratori pe stream-uri

- STL furnizează metode de aplicare a unor algoritmi generici care să înlocuiască nevoia de a itera prin containere.
- Un iterator pe stream foloseşte un stream fie pentru intrare, fie pentru ieşire.
- `ostream_iterator`
- `istream_iterator`

```
int main () {
  vector<int> a;
  int value;

  cout << "Introduceti valorile: (oprire CTRL+Z) ";
  istream_iterator<int> eos; //end-of-stream iterator
  istream_iterator<int> iit(cin); // stdin iterator

  while (iit != eos) {
    a.push_back(*iit);
    iit++;
  }

  ifstream fin("date.in");
  istream_iterator<int> fiit(fin); //file iterator
  while (fiit != eos) {
    a.push_back(*fiit);
    fiit++;
  }

  ostream_iterator<int> out_it(cout, ", ");
  copy(a.begin(), a.end(), out_it);

  ofstream fout("date.out");
  ostream_iterator<int> fout_it(fout,"|");
  copy(a.begin(), a.end(), fout_it);

  return 0;
}
```

STL – Iteratori de inserare

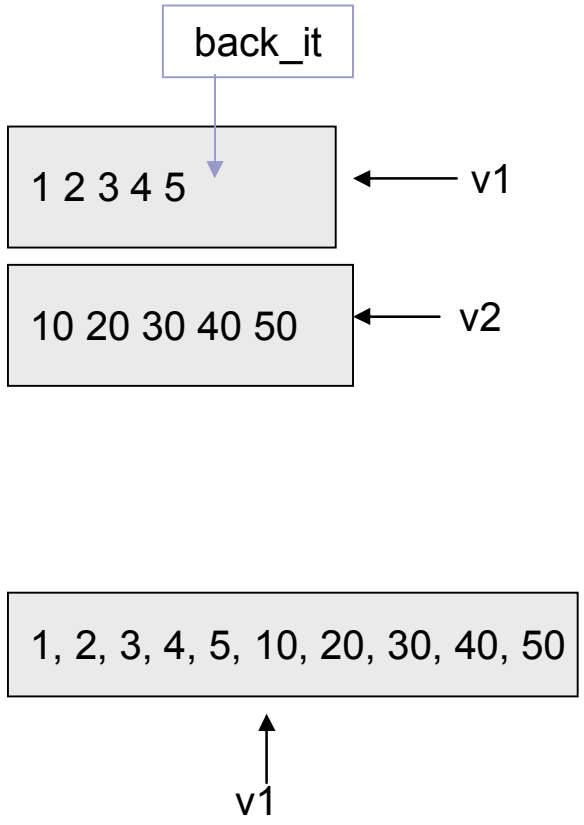
- x iterator

  `*x = 3` ⇦ Suprascrie valoarea existentă

- Daca x este un *insert iterator,* **`*x = 3`** generează adăugarea elementului 3 la secvența pe care iterează.
- Sunt necesari unor algoritmi ce au scopul de a umple containerele, nu de suprascriere.
- `insert_iterator` – permite inserarea de elemente în mijlocul unei secvențe.
- 2 clase adaptor:
  - **`front_inserter`** – containerul trebuie să aibă metoda `push_front`
  - **`back_inserter`** - containerul trebuie să aibă metoda `push_back`.
- Constructorii iau un container secvențial (`vector`, `list`, `deque`) şi produc un iterator ce apelează `push_back` sau `push_front`.

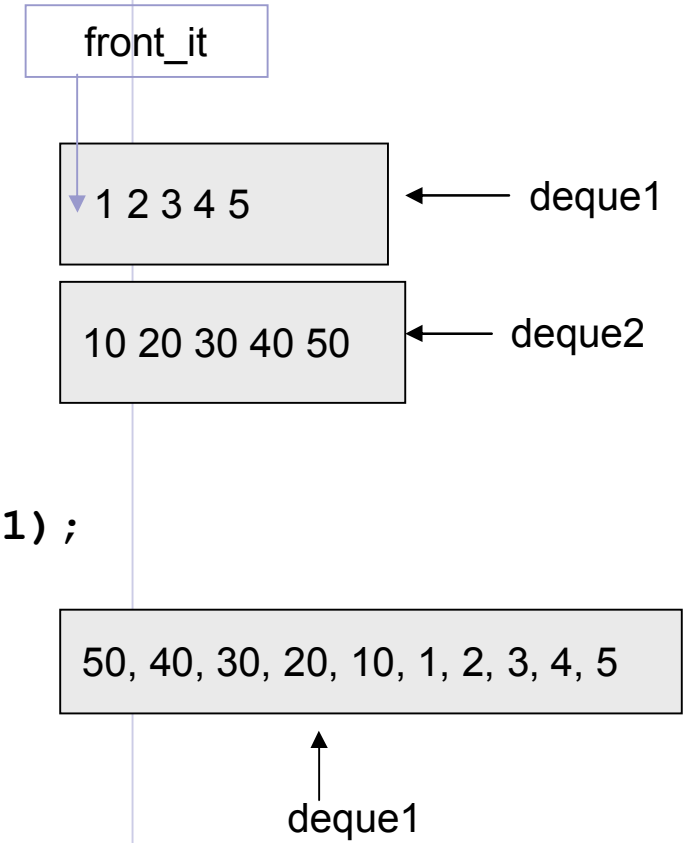
STL – back insert iterator

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

int main() {
  vector<int> v1, v2;
  for (int i = 1; i <= 5; i++) {
    v1.push_back(i);
    v2.push_back(i * 10);
  }

  back_insert_iterator<vector<int> > back_it(v1);
  copy(v2.begin(), v2.end(), back_it);

  ostream_iterator<int> out_it(cout, ", ");
  copy(v1.begin(), v1.end(), out_it);

  return 0;
}
```



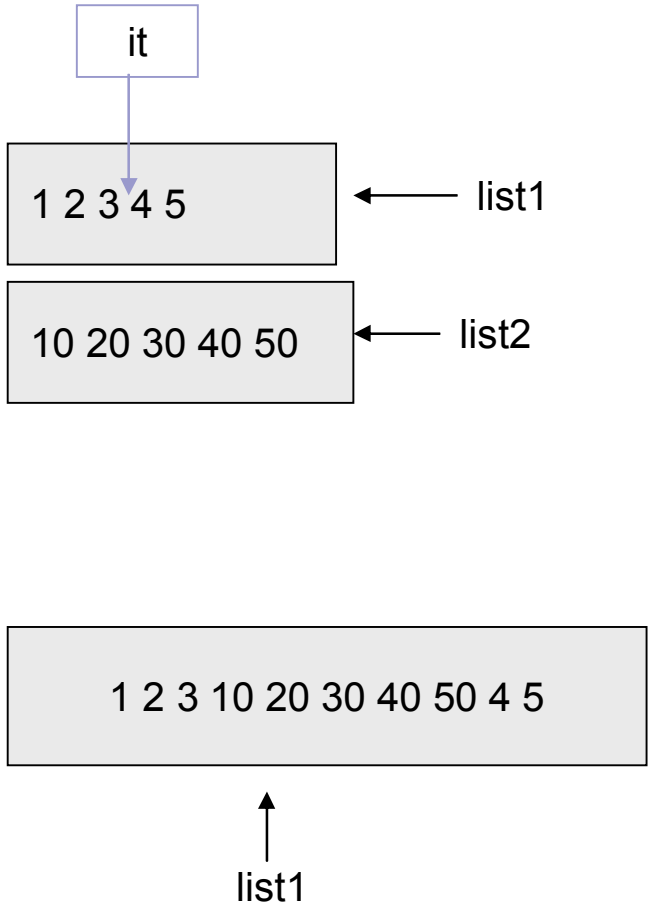
STL – front insert iterator

```
#include <iostream>
#include <iterator>
#include <deque>
using namespace std;

int main() {
  deque<int> deque1, deque2;
  for (int i = 1; i <= 5; i++) {
    deque1.push_back(i);
    deque2.push_back(i*10);
  }

  front_insert_iterator<deque<int> > front_it(deque1);
  copy(deque2.begin(), deque2.end(), front_it);

  deque<int>::iterator it;
  ostream_iterator<int> oit(cout, ",");
  copy(deque1.begin(), deque1.end(), oit);

  return 0;
}
```



STL – insert iterator

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

int main() {
  list<int> list1, list2;
  for (int i = 1; i <= 5; i++) {
    list1.push_back(i);
    list2.push_back(i*10);
  }

  list<int>::iterator it;
  it = list1.begin();
  advance(it, 3);

  insert_iterator<list<int> > insert_it(list1, it);

  copy(list2.begin(), list2.end(), insert_it);

  for (it = list1.begin(); it != list1.end(); ++it)
    cout << *it << " ";
  cout << endl;
  return 0;
}
```



STL – Algoritmi

- Un set de template-uri (sabloane) de funcții ce acționează asupra secvențelor de elemente.
- Clasificare
  - Nonmodifying algorithms
  - Modifying algorithms
  - Removing algorithms
  - Mutating algorithms
  - Sorting algorithms
  - Sorted range algorithms
  - Numeric algorithms

STL – Obiecte de tip funcție

- *Obiect funcție* (sau *functor*) = un obiect ce are **`operatorul()`** definit astfel încât în exemplul următor

```
FunctionObjectType fo;
  ...
fo(...);
```

expresia `fo()` reprezintă un apel al **operatorului ()** pentru obiectul funcție **fo**, în loc de un apel al funcției `fo()`, cum ar putea să pară la prima vedere.

- În loc să scriem toate instrucțiunile în interiorul corpului funcției `fo()`

```
void fo() {
  instructiuni
}
```

acestea vor fi amplasate în cadrul corpului metodei **operator()** definită în clasa obiectului funcție:

```
class FunctionObjectType {
  public:
    void operator()() { instructiuni }
};
```

STL – Obiecte de tip funcție

- Un *obiect de tip funcție* poate fi mai inteligent decât o *simplă funcție*, deoarece acesta posedă o *stare*.
- Pot exista două instanțe ale aceleiaşi funcții, reprezentată de un obiect de tip funcție, ce poate avea stări diferite în acelaşi timp. Acest lucru nu este posibil pentru funcțiile obişnuite.
- *Fiecare funcție are propriul tip*. Astfel, există posibilitatea să fie transmis *tipul unui obiect de tip funcție* către un *şablon* (*template*) pentru a specifica un anumit *comportament*, cu avantajul de a avea tipuri de containere distincte cu diferite obiecte de tip funcție.
- Un obiect de tip funcție este, de obicei, mai *rapid* decât un pointer către o funcție.

STL – Obiecte de tip funcție

```
class Person {
  public:
    string firstname() const;
    string lastname() const;
    //...
};

class PersonSortCriterion {
  public:
    bool operator() (const Person& p1, const Person& p2) const {
      return p1.lastname() < p2.1astname() ||
             ((p2.1astname() == p1.lastname()) &&
               p1.firstname() < p2.firstname());
  }
};

int main() {
  // declara un tip multime cu un criteriu particular de ordonare
  typedef set<Person,PersonSortCriterion> PersonSet;
  // creaza a astfel de colectie
  PersonSet coll;
  ...
  // realizeaza o prelucrare a elementelor
  PersonSet::iterator pos;
  for (pos = coll.begin(); pos != coll.end(); ++pos) {
    ...
  }
  ...
}
```

STL – Obiecte de tip funcție

- De exemplu, un obiect de tip funcție folosit pentru a genera o secvență de numere întregi.

```
class IntSequence {
  private:
    int value;
  public:
    IntSequence (int initialValue) : value(initialValue) {  }
   //operatorul supraincarcat
   int operator() () {
     return value++;
   }
};
```

- Utilizarea unui obiect de tip funcție cu `generate_n()`:

```
list<int> ll;
//insereaza valorile de la 1 la 9
generate_n(back_inserter(ll), 9, IntSequence(1));
```

- Conținutul secvenței este:

```
1; 2; 3; 4; 5; 6; 7; 8; 9;
```

STL – Obiecte de tip funcție

- Utilizarea unui obiect de tip funcție cu `generate_n()`:

```
generate(++ll.begin(), --ll.end(), IntSequence(42));
```

- Conținutul secvenței este:

```
1; 42; 43; 44; 45; 46; 47; 48; 9;
```

- Obiectele de tip funcție de obicei sunt transmise prin valoare decât prin referință:

```
IntSequence seq(1);

//insereaza secventa de numere ce incepe cu valoarea 1
generate_n(back_inserter(ll), 9, seq);

// insereaza din nou secventa de numere ce incepe cu 1
generate_n(back_inserter(ll), 9, seq);
```

STL – Obiecte de tip funcție

- Uneori s-ar putea să fie necesar accesul la starea finală a obiectului, astfel încât se pune problema cum să obținem rezultatul dintr-un algoritm.
- Există două modalități de a obține un "*rezultat*" sau "*feedback*" dintr-un algoritm, folosind obiecte de tip funcție:
  - Puteți transmite obiectele de tip funcție ca referință.
  - Puteți utiliza valoarea de return a algoritmului `for_each()`.
- Pentru a transmite obiectul `seq` prin referință către funcția `generate_n()`, argumentele şablonului sunt calificate în mod explicit:

```
generate_n<back_insert_iterator<list<int> >, int, IntSequence&>
          (back_inserter(ll), 4, seq);
```

STL – Obiecte de tip funcție predefinite

Expresie	Efect
`negate<type>()`	- param
`plus<type>()`	param1 + param2
`minus<type>()`	param 1 - param2
`multiplies<type>()`	param1 * param2
`divides<type>()`	param1 / param2
`modulus<type>()`	param1 % param2
`equal_to<type>()`	param1 == param2
`not_equal_to<type>()`	param1 ! = param2
`less<type>()`	param1 < param2
`greater<type>()`	param1 > param2
`less_equal<type>()`	param1 <= param2
`greater_equal<type>()`	param1 >= param2
`logical_not<type>()`	! param
`logical_and<type>()`	param1 && param2
`logical_or<type>()`	param1 | | param2

STL – for_each

- Aplică o funcție dată ca parametru pentru fiecare element al secvenței specificate.

- *template <class InputIterator, class Function> Function for_each (InputIterator first, InputIterator last, Function f);*

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

using namespace std;

void f(int i) {
  cout << " " << i;
}

int main() {
  vector<int> v;
  v.push_back(10);
  v.push_back(20);
  v.push_back(30);

  cout << "Vectorul contine:";
  for_each(v.begin(), v.end(), f);

  return 0;
}
```

STL – `find`

- *template <class InputIterator, class T> InputIterator* ***find*** *(InputIterator first, InputIterator last, const T& value);*

- Returnează un iterator către primul element egal cu *value*, sau un iterator către `end()`, în cazul în care nu găseşte valoarea căutată.

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

using namespace std;

int main() {
  int a[] = {10, 20, 30, 40};

  vector<int> v(a, a + 4);
  vector<int>::iterator it;

  it = find(v.begin(), v.end(), 30);
  ++it;
  cout << "Elementul urmator lui 30 este"
       << *it << endl;

  return 0;
}
```

```
Elementul urmator lui 30 este 40
```

STL – `find_if`

- *template <class InputIterator, class Predicate> InputIterator find_if (InputIterator first, InputIterator last, Predicate pred);*
- `Predicate` – funcție ce are ca parametru un obiect de tipul template-ului şi returnează o valoare booleană.
- Găseşte un element în domeniul `[first, last)` care să respecte condiția specificată de `pred` şi returnează un iterator către element, altfel returnează un iterator către `end()`.

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

bool positive(int i) { return (i>0); }

int main() {
  vector<int> v;
  vector<int>::iterator it;
  v.push_back(-10);
  v.push_back(-95);
  v.push_back(40);
  v.push_back(-55);

  it = find_if(v.begin(), v.end(),
       positive);

  cout << "Prima valoare pozitiva "
       << *it << endl;
  return 0;
}
```

predicat

```
Prima valoare pozitiva 40
```

STL – Variante `find`

- `find_end` – găseşte ultima apariție a unei secvențe în altă secvență şi returnează un iterator.
- `find_first_of` – găseşte prima apariție a oricărui element dintr-o secvență în altă secvență.
- `adjacent_find` – caută prima apariție a două valori consecutive egale într-o secvență şi returnează un iterator la ea.

STL – equal

- Verifică dacă elementele din două domenii sunt egale efectuând compararea pe perechi de elemente corespunzătoare.

- *template <class InputIterator1, class InputIterator2> bool **equal** (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);*

- *template <class InputIterator1, class InputIterator2, class BinaryPredicate> bool **equal** (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate pred);*

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

bool pred(int i, int j) { return (i == j); }

int main() {
  int a[] = {20, 40, 60, 80, 100};
  vector<int> tab(a, a + 5);
  if (equal(tab.begin(), tab.end(), a))
    cout << "Continutul secventelor este
   acelasi." << endl;
  else cout << " Continutul secventelor este
   diferit." << endl;

  tab[3] = 81;
  if (equal(tab.begin(), tab.end(), a, pred))
    cout << "Continutul secventelor este
   acelasi." << endl;
  else
    cout << " Continutul secventelor este
   diferit." << endl;
  return 0;
```

Continutul secventelor este acelasi.
Continutul secventelor este diferit.

STL – count_if

- Returnează numărul de apariții ale unui element într-o secvență / numărul elementelor pentru care o condiție este adevarată.

- *template <class InputIterator, class T> typename iterator_traits<InputIterator>::difference_type count (ForwardIterator first, ForwardIterator last, const T& value);*

- *template <class InputIterator, class Predicate> typename iterator_traits<InputIterator>::difference_type count_if (ForwardIterator first, ForwardIterator last, Predicate pred);*

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

bool f(int i) { return (i > 0); }

int main () {
  vector<int> v;
  v.push_back(-10);
  v.push_back(-95);
  v.push_back(40);
  v.push_back(-55);

  int nrPos = (int)count_if(v.begin(),
                            v.end(), f);
  cout << "Nr valorilor pozitive este "
       << nrPos << endl;

  return 0;
}
```

-10 -95 40 -55

Nr valorilor pozitive este 1

STL – mismatch

- Compară două secvențe şi returnează poziția primei diferențe.

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

bool pred(int i, int j) { return (i == j); }

int main() {
  vector<int> v;
  for (int i = 1; i < 6; i++)
    v.push_back(i * 10);
  int a[] = {10, 20, 80, 320, 1024};
  pair<vector<int>::iterator, int*> mypair;
  mypair = mismatch(v.begin(), v.end(), a);
  cout << "First mismatching elements: " << *mypair.first;
  cout << " and " << *mypair.second << endl;
  mypair.first++; mypair.second++;
  mypair = mismatch(mypair.first, v.end(), mypair.second, pred);
  cout << "Second mismatching elements: " << *mypair.first;
  cout << " and " << *mypair.second << endl;

  return 0;
}
```

Comparație implicită (==)

Comparație cu predicat

First mismatching elements: 30 and 80
Second mismatching elements: 40 and 320

STL – search

- Caută prima apariție a unei secvențe în altă secvență şi returnează un iterator pe primul element al secvenței (comparația se face folosind '==' sau un predicat).

- *template <class ForwardIterator1, class ForwardIterator2> ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);*

- *template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate> ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2. BinaryPredicate pred);*

```
match1 found at position 3
match2 not found
```

```
bool pred(int i, int j) { return (i == j); }

int main() {
  vector<int> v;
  vector<int>::iterator it;
  for (int i=1; i<10; i++) v.push_back(i*10);

  int match1[] = {40, 50, 60, 70};
  it = search(v.begin(), v.end(),
              match1, match1 + 4);
  if (it != v.end())
    cout << "match1 found at position "
         << int(it - v.begin()) << endl;
  else
    cout << "match1 not found" << endl;

  int match2[] = {20, 30, 50};
  it = search(v.begin(), v.end(),
              match2, match2 + 3, pred);
  if (it != v.end())
    cout << "match2 found at position "
         << int(it - v.begin()) << endl;
  else
    cout << "match2 not found" << endl;
  return 0;
```

```
10 20 30 40 50 60 70 80 90
```

```
40 50 60 70
```

```
20 30 50
```

STL – search_n

- Caută prima apariție a unei succesiuni de n elemente egale în secvență şi returnează un iterator pe primul element.

- *template <class ForwardIterator, class Size, class T> ForwardIterator* ***search_n*** *(ForwardIterator first, ForwardIterator last, Size count, const T& value);*

- *template <class ForwardIterator, class Size, class T, class BinaryPredicate> ForwardIterator* ***search_n*** *(ForwardIterator first, ForwardIterator last, Size count, const T& value,*

  BinaryPredicate *pred);*

```
secv 10 10 gasita la pozitia 5
match not found
```

```
bool pred(int i, int j) { return (i == j); }

int main() {
  int a[] = {10, 20, 30, 30, 20, 10, 10,20};
  vector<int> v(a, a + 8);
  vector<int>::iterator it;

  it = search_n(v.begin(), v.end(), 2, 10);
  if (it != v.end())
    cout << "secv 10 10 gasita la pozitia "
         << int(it - v.begin()) << endl;
  else cout << "match not found" << endl;

  it = search_n(v.begin(), v.end(), 2, 20,
                pred);
  if (it != v.end())
    cout << "secv 20 20 gasita la pozitia "
         << int(it - v.begin()) << endl;
  else
    cout << "match not found" << endl;

  return 0;
}
```

STL – Operații ce modifică secvența

- `copy/copy_backward`
- `swap/swap_ranges/iter_swap`
- `transform`
- `replace/replace_if/replace_copy/replace_copy_if`
- `fill/fill_n`
- `generate/generate_n`
- `remove/remove_if/remove_copy/remove_copy_if`
- `unique/unique_copy`
- `reverse/reverse_copy`
- `rotate/rotate_copy`
- `random_shuffle`
- `partition`
- `stable_partition`

STL – copy

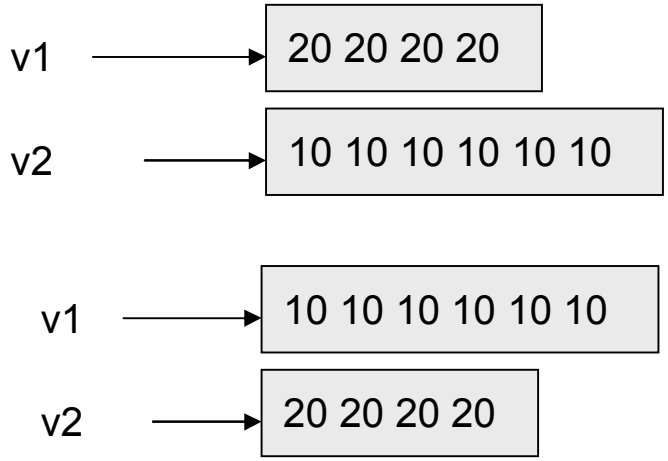
- *template <class InputIterator, class OutputIterator> OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result);*
- Copiază elementele din domeniul `[first, last)` într-un domeniu ce începe cu `result`.
- Returnează un *iterator* către ultimul element al secvenței destinație.
- *template <class BidirectionalIterator1, class BidirectionalIterator2> BidirectionalIterator2 copy_backward (BidirectionalIterator1 first, BidirectionalIterator1 last, BidirectionalIterator2 result);*
- Copiază elementele în ordine inversă (de la `last-1` la `first`) – se copiază `last-1` în `result-1`, ş.a.m.d.

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

int main() {
  int a[] = {10, 20, 30, 40, 50, 60, 70};
  vector<int> v(7);
  vector<int>::iterator it;

  copy(a, a + 7, v.begin());

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

  return 0;
}
```

```
10 20 30 40 50 60 70
```

STL – swap

- *template* <*class* T> *void* swap (T& a, T& b) {
  T c(a); a=b; b=c;
  }
- Interschimbă 2 valori.



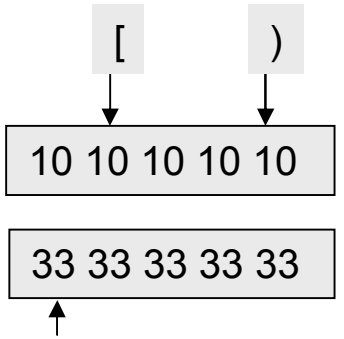

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main() {
  int x = 10, y = 20;
  swap(x, y);

  vector<int> v1(4, x), v2(6, y);
  swap(v1, v2);
  vector<int>::iterator it;
  for (it = v1.begin(); it != v1.end(); ++it)
    cout << " " << *it;
  cout << endl;

  for (it = v2.begin(); it != v2.end(); ++it)
    cout << " " << *it;
  cout << endl;

  return 0;
}
```

STL – swap_ranges

- *template < class ForwardIterator1, class ForwardIterator2 > ForwardIterator2 swap_ranges (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2);*

- Interschimbă valorile din domeniul `[first1,last1)` cu elementele începând de la `first2`.

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

int main() {
  vector<int> v1(5,10);
  vector<int> v2(5,33);
  vector<int>::iterator it;

  swap_ranges(v1.begin() + 1, v1.end() - 1,
              v2.begin());

  for (it = v1.begin(); it != v1.end(); ++it)
    cout << " " << *it;

  for (it = v2.begin(); it != v2.end(); ++it)
    cout << " " << *it;
  cout << endl;
  return 0;
}
```



10 33 33 33 10

10 10 10 33 33

STL – transform

- Aplică o *transformare unară* fiecărui element din secvența de intrare sau o *transformare binară* elementelor corespunzătoare din două secvențe de intrare.
- *template < class InputIterator, class OutputIterator, class UnaryOperator > OutputIterator transform (InputIterator first1, InputIterator last1, OutputIterator result, UnaryOperator op);*
- Rezultatul – o nouă secvență.
- *template < class InputIterator1, class InputIterator2, class OutputIterator, class BinaryOperator > OutputIterator transform (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, OutputIterator result, BinaryOperator binary_op);*

```
int fchange(int i) { return (-1) * i; }
int fdiff(int i, int j) { return (i - j); }

int main() {
  vector<int> v1;
  vector<int> v2;
  vector<int>::iterator it;

  for (int i = 1; i < 6; i++)
    v1.push_back(i*10);

  v2.resize(v1.size());
  transform(v1.begin(), v1.end(), v2.begin(),
            fchange);

  for (it = v2.begin(); it != v2.end(); ++it)
    cout << " " << *it;
  cout << endl;
  transform(v1.begin(), v1.end(), v2.begin(),
            v1.begin(), fdiff);

  for (it = v1.begin(); it != v1.end(); ++it)
    cout << " " << *it;
  cout << endl;
  return 0;
```

10 20 30 40 50

-10 -20 -30 -40 -50

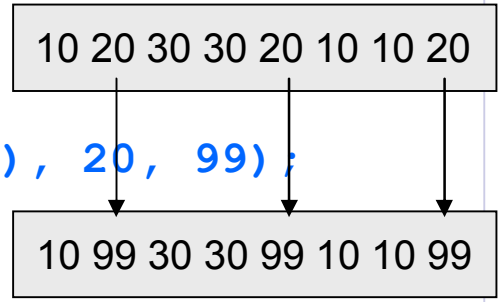
20 40 60 80 100

STL – replace

- *template < class ForwardIterator, class T > void replace (ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value);*

- Înlocuieşte toate aparițiile valorii `old_value` din domeniul `[first,last)` cu valoarea `new_value`.

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

int main() {
  int a[] = {10,20,30,30,20,10,10,20};
  vector<int> v(a, a + 8);

  replace(v.begin(), v.end(), 20, 99);

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

  return 0;
}
```



STL – fill / fill_n

- *template < class ForwardIterator, class T > void fill (ForwardIterator first, ForwardIterator last, const T& value);*
- *template < class OutputIterator, class Size, class T > void fill_n (OutputIterator first, Size n, const T& value);*
- Completează (primele n) elementele din domeniul `[first,last)` cu `value`.

```
int main() {
 vector<int> v1(8);
 fill(v1.begin(), v1.begin()+4, 5);
 fill(v1.begin()+3, v1.end()-2, 8);

 vector<int>::iterator it;
 for (it = v1.begin(); it != v1.end(); ++it)
   cout << " " << *it;
 cout << endl;
 vector<int> v2(8,10);
 fill_n(v2.begin(), 4, 20);

 fill_n(v2.begin()+3, 3, 33);

 for (it = v2.begin(); it != v2.end(); ++it)
   cout << " " << *it;
 return 0;
}
```

0 0 0 0 0 0 0 0

5 5 5 5 0 0 0 0

5 5 5 8 8 8 0 0

10 10 10 10 10 10 10 10

20 20 20 20 10 10 10 10

20 20 20 33 33 33 10 10

STL – generate / generate_n

- *template <class ForwardIterator, class Generator> void generate (ForwardIterator first, ForwardIterator last, Generator gen);*
- *template <class OutputIterator, class Size, class Generator> void generate_n (OutputIterator first, Size n, Generator gen);*
- Completează elementele din domeniul `[first,last)` cu valorile generate de funcția `gen()`.

```
int RandomNumber() { return (rand() % 100); }

int current(0);
int UniqueNumber() { return ++current; }

int main() {
  srand(unsigned(time(NULL)));
  vector<int> v(8);
  vector<int>::iterator it;

  generate(v.begin(), v.end(), RandomNumber);
  for (it = v.begin(); it != v.end(); ++it)
    cout << " " << *it;
  cout << endl;

  int a[9];
  generate_n(a, 9, UniqueNumber);
  for (int i = 0; i < 9; ++i)
    cout << " " << a[i];
  return 0;
}
```

STL – `remove`

- *template < class ForwardIterator, class T > ForwardIterator remove (ForwardIterator first, ForwardIterator last, const T& value);*
- Elimină toate elementele având valoarea `value` din domeniul `[first,last)`.
- Returnează un *iterator* spre noul sfârşit al secvenței.

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

int main() {
  int a[] = {10,20,30,40,50,20,70};

  int* pbegin = a;
  int* pend = a + sizeof(a)/sizeof(int);

  pend = remove(pbegin, pend, 20);

  cout << "Intervalul contine: ";
  for (int* p = pbegin; p != pend; ++p)
    cout << " " << *p;
  cout << endl;

  return 0;
}
```

```
10 20 30 40 50 20 70
```

```
10  30  40  50  70
```

STL – remove_copy

- *template <class InputIterator, class OutputIterator, class T> OutputIterator remove_copy (InputIterator first, InputIterator last, OutputIterator result, const T& value);*

- Realizează o copie a secvenței inițiale, eliminând toate elementele ce au valoarea egală cu `value`.

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

int main() {
  int a[] = {10,20,30,30,20,10,10,20};

  vector<int> v(8);
  vector<int>::iterator it;

  remove_copy(a, a + 8, v.begin(), 20);

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

  return 0;
}
```

10 20 30 30 20 10 10 20

10 30 30 10 10 0 0 0

STL – remove_copy_if

- *template <class InputIterator, class OutputIterator, class Predicate> OutputIterator remove_copy_if (InputIterator first, InputIterator last, OutputIterator result, Predicate pred);*

- Copiază elementele din domeniul `[first,last)` în domeniul ce începe cu `result`, cu excepția celor pentru care valoarea `pred` este *true*.

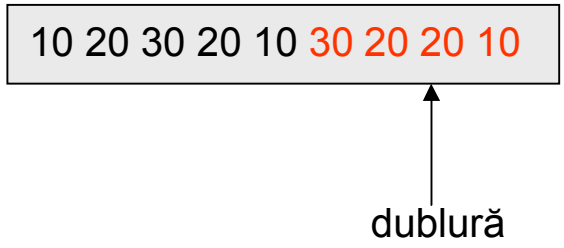
```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

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

int main() {
  int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  vector<int> v(9);
  vector<int>::iterator it;

  remove_copy_if(a, a+9, v.begin(), impar);

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

  return 0;
}
```

2 4 6 8 0 0 0 0 0

STL – unique

- *template <class ForwardIterator> ForwardIterator unique (ForwardIterator first, ForwardIterator last);*
- *template <class ForwardIterator, class BinaryPredicate> ForwardIterator unique (ForwardIterator first, ForwardIterator last, BinaryPredicate pred);*

- Elimină elementele egale ("==") situate pe poziții consecutive sau le compară folosind o funcție de comparare.




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

bool f(int i, int j) { return (i == j); }

int main() {
  int a[] = {10,20,20,20,30,30,20,20,10};

  vector<int> v (a, a+9);
  vector<int>::iterator it;
  it = unique(v.begin(), v.end());

  v.resize(it - v.begin());
  unique(v.begin(), v.end(), f);

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

  return 0;
}
```

10 20 30 20 10

STL – unique_copy

- *template <class InputIterator, class OutputIterator> OutputIterator unique_copy (InputIterator first, InputIterator last, OutputIterator result);*

- *template <class InputIterator, class OutputIterator, class BinaryPredicate> OutputIterator unique_copy (InputIterator first, InputIterator last, OutputIterator result, BinaryPredicate pred);*

0 0 0 0 10 10 20 30 30 40

- Copiază elementele din intervalul `[first, last)` cu excepția elementelor consecutive egale sau care respectă o relație dată ca funcție.

0 10 20 30 40

```
bool f(int i, int j) { return (i == j); }

int main() {
  int a[] = {10,20,20,20,30,30,40,40,30,10};
  vector<int> v(10);
  vector<int>::iterator it;

  it = unique_copy (a, a + 10, v.begin());
  for (it = v.begin(); it != v.end(); ++it)
    cout << " " << *it;
  cout << endl;
  sort (v.begin(), it);
  for (it = v.begin(); it != v.end(); ++it)
    cout << " " << *it;
  cout << endl;

  it = unique_copy(v.begin(), it, v.begin(), f);
  v.resize(it - v.begin());

  cout << "Vectorul contine:";
  for (it = v.begin(); it != v.end(); ++it)
    cout << " " << *it;

  cout << endl;
  return 0;
```

10 20 30 40 30 10 0 0 0 0

STL – reverse

- *template <class BidirectionalIterator> void reverse (BidirectionalIterator first, BidirectionalIterator last);*

- Inversează ordinea elementelor din intervalul `[first, last)`.

9 8 7 6 5 4 3 2 1

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

int main() {
  vector<int> v;
  vector<int>::iterator it;

  for (int i = 1; i < 10; ++i)
    v.push_back(i);

  reverse(v.begin(), v.end());

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

  return 0;
}
```

STL – partition

- *template <class BidirectionalIterator, class Predicate> BidirectionalIterator partition (BidirectionalIterator first, BidirectionalIterator last, Predicate pred);*

- Rearanjează elementele unei secvențe astfel încât toate elementele pentru care un predicat returnează *true* sunt aşezate înaintea celor pentru care returnează *false.*

- Returnează un pointer spre cel de-al doilea grup.

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

int main() {
  vector<int> v;
  vector<int>::iterator it, bound;

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

  bound = partition(v.begin(), v.end(),
                    impar);

  for (it = v.begin(); it != bound; ++it)
    cout << " " << *it;

  for (it = bound; it != v.end(); ++it)
    cout << " " << *it;
  cout << endl;

  return 0;
}
```

→ 1 9 3 7 5

→ 6 4 8 2

STL – Sortări

- `sort`
- `stable_sort`
- `partial_sort`
- `partial_sort_copy`
- `n_th_element`

STL – sort

- *template <class RandomAccessIterator> void sort (RandomAccessIterator first, RandomAccessIterator last);*
- *template <class RandomAccessIterator, class Compare> void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);*
- Sortează elementele dintre `first` şi `last` în ordine crescătoare sau folosind un criteriu `comp`.
- `Compare` – obiect de tipul funcție de comparare primeşte ca argumente 2 obiecte de tipul template-ului şi returnează *true* dacă primul argument < decât al doilea argument.
- `Stable_sort` asigură faptul că elementele cu valori egale rămân pe poziții.

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

struct CFunctor {
  bool operator() (int i, int j)
  { return (i < j); }
} objFunctor;

int main () {
  int a[] = {32,71,12,45,26,80,53,33};
  vector<int> v (a, a+8);
  vector<int>::iterator it;

  sort(v.begin(), v.begin()+4);

  sort(v.begin()+4, v.end(), f);

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

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

  cout << endl;
  return 0;
}
```

Tipul functie de comparatie

32 71 12 45 26 80 53 33

(12 32 45 71)26 80 53 33

12 32 45 71(26 33 53 80)

(12 26 32 33 45 53 71 80)

STL – partial sort

- *template <class RandomAccessIterator> void partial_sort (RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last);*
- *Template <class RandomAccessIterator, class Compare> void partial_sort (RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last, Compare comp);*

- Ordonează elementele din domeniul `[first,last)` astfel încât secvența `[first, middle)` conține elementele ordonate crescător, iar restul nu sunt ordonate.

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

int main() {
  int a[] = {9,8,7,6,5,4,3,2,1};
  vector<int> v (a, a + 9);
  vector<int>::iterator it;
  partial_sort(v.begin(), v.begin() + 5,
               v.end());

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

  partial_sort(v.begin(), v.begin() + 5,
               v.end(), comp);

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

  return 0;
}
```

```
1 2 3 4 5 9 8 7 6
```

```
1 2 3 4 5 9 8 7 6
```

STL – nth element

- *template <class RandomAccessIterator> void nth_element (RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);*

- *template <class RandomAccessIterator, class Compare> void nth_element (RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, Compare comp)*

- Rearanjează elementele dintre `first` şi `last` astel încât `nth` va reprezenta elementul de pe poziția n într-o ordonare crescătoare a elementelor, fără a garanta că elementele anterioare sau posterioare ar fi ordonate.

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

int main() {
  vector<int> v;
  vector<int>::iterator it;

  for (int i = 1; i < 10; i++)
    v.push_back(i);

  //random_shuffle(v.begin(), v.end());
  nth_element(v.begin(), v.begin() + 5,
              v.end());

  nth_element(v.begin(), v.begin() + 5,
              v.end(), comp);

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

  return 0;
}
```

STL – Căutare binară

- `lower_bound`
- `upper_bound`
- `equal_range`
- `binary_search`

STL – lower bound / upper bound

- *template <class ForwardIterator, class T> ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last, const T& value);*

- *template <class ForwardIterator, class T, class Compare> ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last, const T& value, Compare comp);*

- Returneaza un iterator spre primul element din intervalul `[first, last)` care este `>= value` sau `>value`.

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

int main() {
  int a[] = {10,20,30,30,20,10,10,20};
  vector<int> v(a, a+8);
  vector<int>::iterator low, up;

  sort(v.begin(), v.end());
  low = lower_bound(v.begin(), v.end(), 20);
  up = upper_bound(v.begin(), v.end(), 20);

  cout << "lower_bound pe pozitia "
       << int(low - v.begin()) << endl;
  cout << "upper_bound pe pozitia "
       << int(up - v.begin()) << endl;

  return 0;
}
```

```
lower_bound pe pozitia 3
upper_bound pe pozitia 6
```

STL – binary search

- *template <class ForwardIterator, class T> bool binary_search (ForwardIterator first, ForwardIterator last, const T& value);*
- *template <class ForwardIterator, class T, class Compare> bool binary_search (ForwardIterator first, ForwardIterator last, const T& value, Compare comp);*
- Verifică dacă `value` aparține domeniului `[first, last)` ce *trebuie să fie sortat*.

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

class CFunctor {
  public:
    int operator()(const int i, const int j)
                   {return i<j;}
}myComp;

int main () {
  int a[] = {1,2,3,4,5,4,3,2,1};
  vector<int> v(a, a+9);
  sort(v.begin(), v.end());
  if (binary_search(v.begin(), v.end(), 3))
    cout << "found!\n";
  else cout << "not found.\n";

  sort(v.begin(), v.end(), comp);
  if (binary_search(v.begin(), v.end(), 6,
                    myComp))
    cout << "found!\n";
  else cout << "not found.\n";

  return 0;
```

STL – algoritmi de interclasare

- `merge`
- `inplace_merge`
- `includes`
- `set_union`
- `set_intersection`
- `set_difference`
- `set_symmetric_difference`

STL – merge

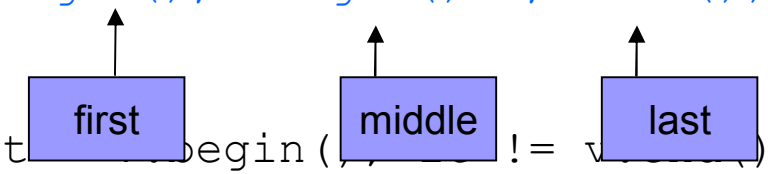
- Interclasează 2 secvențe ordonate.
- *template <class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator merge (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);*
- *template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare> OutputIterator merge (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);*

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

int main() {
  int a[] = {5,10,15,20,25};
  int b[] = {50,40,30,20,10};
  vector<int> v(10);
  vector<int>::iterator it;

  sort(a, a+5);
  sort(b, b+5);
  merge(a, a+5, b, b+5, v.begin());

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

  return 0;
}
```

STL – inplace_merge

- Interclasează 2 secvențe consecutive.
- *template <class BidirectionalIterator> void inplace_merge (BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last);*
- *template <class BidirectionalIterator, class Compare> void inplace_merge (BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last, Compare comp);*

```
int main() {
  int a[] = {5, 10, 15, 20, 25};
  int b[] = {50, 40, 30, 20, 10};
  vector<int> v(10);
  vector<int>::iterator it;

  sort(a, a + 5); sort(b, b + 5);

  copy(a, a + 5, v.begin());
  copy(b, b + 5, v.begin() +5);

  inplace_merge
   (v.begin(),v.begin()+5,v.end());

for (it                 != v.end(); ++it)
    cout << " " << *it;
  cout << endl;

  return 0;
}
```



STL – includes

- *template <class InputIterator1, class InputIterator2> bool includes (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);*
- *template <class InputIterator1, class InputIterator2, class Compare> bool includes (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp);*
- Verifică dacă toate elementele din domeniul `[first1, last1)` se regăsesc în domeniul `[first2, last2).`

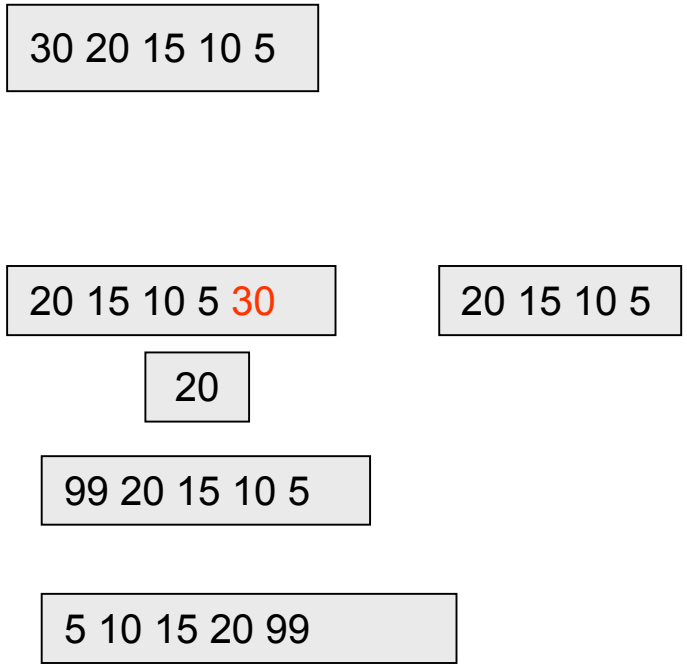
```
bool comp(int i, int j) { return i<j; }

int main() {
 int a[] = {5,10,15,20,25,30,35,40,45,50};
  int b[] = {40,30,20,10};
  int c[] = {40,30,20,10,9};

  sort(a, a+10); sort(b, b+4);
  sort(c, c+5);

  if (includes(a, a+10, b, b+4))
    cout << "a include b!" << endl;

  if (includes(a, a+10, b, b+4, comp))
    cout << "a include b!" << endl;

  if (includes(a, a+10, c, c+5))
    cout << "a include c!" << endl;
  else
    cout << "a nu include c!" << endl;
  return 0;
}
```

STL – algoritmi pentru heap-uri

- `push_heap`
- `pop_heap`
- `make_heap`
- `sort_heap`

STL – Operații pe heap-uri

```
int main() {
  int a[] = {10,20,30,5,15};
  vector<int> v(a, a+5);
  vector<int>::iterator it;

  make_heap(v.begin(), v.end());
  cout << "initial max heap   : " << v.front() << endl;

  pop_heap(v.begin(),v.end());
  v.pop_back();
  cout << "max heap after pop : " << v.front() << endl;

  v.push_back(99);
  push_heap(v.begin(),v.end());
  cout << "max heap after push: " << v.front() << endl;

  sort_heap(v.begin(),v.end());

  cout << "final sorted range :";
  for (unsigned i = 0; i < v.size(); i++)
    cout << " " << v[i];
  cout << endl;

  return 0;
}
```



STL – algoritmi de minim / maxim

- `min`
- `max`
- `min_element`
- `max_element`
- `lexicographical_compare`
- `next_permutation`
- `prev_permutation`

STL – min element / max element

- Returnează un iterator la elementul minim/maxim din domeniu.
- *template <class ForwardIterator> ForwardIterator min_element (ForwardIterator first, ForwardIterator last);*
- *template <class ForwardIterator, class Compare> ForwardIterator min_element (ForwardIterator first, ForwardIterator last, Compare comp);*

```
bool comp(int i, int j) { return i<j; }
struct CFunctor {
  bool operator() (int i, int j) { return
   i<j; }
} myComp;

int main() {
  int a[] = {3,7,2,5,6,4,9};

  cout << "The smallest element is "
       << *min_element(a, a+7) << endl;
  cout << "The largest element is "
       << *max_element(a, a+7) << endl;

  cout << "The smallest element is "
       << *min_element(a,a+7, comp) << endl;
  cout << "The largest element is "
       << *max_element(a,a+7, comp) << endl;

  cout << "The smallest element is "
       << *min_element(a,a+7,myComp) << endl;
  cout << "The largest element is "
       << *max_element(a,a+7,myComp) << endl;
```

STL – lexicographical compare

- Comparare lexicală a elementelor a două secvențe.
- *template <class InputIterator1, class InputIterator2> bool lexicographical_compare (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);*
- *template <class InputIterator1, class InputIterator2, class Compare> bool lexicographical_compare (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp);*

```
bool comp(char c1, char c2)
{ return tolower(c1)<tolower(c2); }

int main() {
  char s1[] = "Apple";
  char s2[] = "apartment";

 if (lexicographical_compare(s1, s1+5, s2, s2+9))
    cout << s1 << " is less than " << s2 << endl;
  else
    if (lexicographical_compare(s2,s2+9,s1,s1+5))
     cout << s1 << " is greater than " << s2
          << endl;
    else
     cout << s1 << " and " << s2 << " are equals\n";

  if (lexicographical_compare(s1,s1+5,s2,s2+9,comp))
    cout << s1 << " is less than " << s2 << endl;
  else
    if(lexicographical_compare(s2,s2+9,s1,s1+5,comp))
      cout << s1 << " is greater than " << s2 <<
   endl;
    else
     cout << s1 << " and " << s2 << " are equals\n";
```

STL – Bibliografie

- SGI - http://www.sgi.com/tech/stl/
- *Power up C++ with the Standard Template Library: Part I* - http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=standardTemplateLibrary
- *Power up C++ with the Standard Template Library: Part II: Advanced Uses* - http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=standardTemplateLibrary2
- http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html
- http://channel9.msdn.com/Series/C9-Lectures-Stephan-T-Lavavej-Standard-Template-Library-STL-/C9-Lectures-Introduction-to-STL-with-Stephan-T-Lavavej
- http://www.josuttis.com/libbook/toc.html
- http://www.codeguru.com/cpp/cpp/cpp_mfc/stl/article.php/c4027/C-Tutorial-A-Beginners-Guide-to-stdvector-Part-1.htm