

# Programare II

## Limbajul C/C++

CURS 11



# Curs anterior

- ❑ Tipuri de date generice
- ❑ Funcții şablon
- ❑ Clase şablon

# Cuprins

- ❑ Standard Template Library
  - ❑ Introducere în Standard Template Library
  - ❑ Containere (secvențiale + asociative)
  - ❑ Iteratori
  - ❑ Algoritmi

# Standard Template Library

- ❑ Bibliotecă generică
- ❑ Conține
  - ❑ Tipuri de date generice
    - ❑ Liste, vectori,...
  - ❑ Algoritmi
    - ❑ Utilizați împreună cu
      - ❑ Tablouri
      - ❑ Structurile de date definite în STL

# Caracteristici unei biblioteci generice

- ❑ Refolosire
  - ❑ Posibilitatea de a opera cu tipuri de date definite de utilizator
- ❑ Compoziția
  - ❑ Posibilitatea de a opera cu tipuri de date definite în altă bibliotecă
- ❑ Eficiența
  - ❑ Performanțe mai bune decât implementările non-generice (codificării personale)

# Ce oferă bibliotecile standard în C++?

- ❑ Suport pentru **proprietățile limbajului** (gestionarea memoriei, RTTI)
- ❑ Informații despre **implementarea compilatorului** (ex. cea mai mare valoare pentru numere de tip float)
- ❑ Funcții care nu pot fi implementate optimal în limbaj pentru orice sistem (sqrt, memmove, etc)
- ❑ Suport pentru lucrul cu **șiruri de caractere** și **streamuri** (include suport pentru internaționalizare și localizare)
- ❑ Framework pentru **containere** (vector, map,...) și **algoritmi generici** pentru ele (parcuregere, sortare, reuniune, ...)
- ❑ Prelucrării numerice (numere complexe, BLAS=Basic Liniar Algebra Subprograms etc.)
- ❑ Bază pentru alte biblioteci

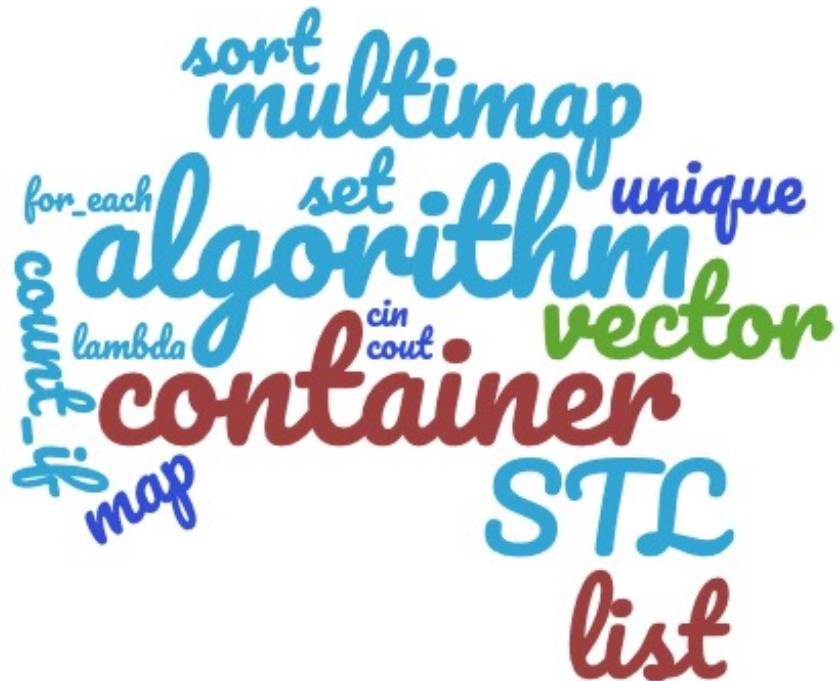
# Standard Template Library

- ❑ STL, biblioteca standard a limbajului C++ oferă cele **mai uzuale structuri de date și algoritmi fundamentali** pentru utilizarea lor
- ❑ Prima bibliotecă generică a C++ folosită pe scară largă
- ❑ Conținutul biblioteci STL
  - ❑ Containere
  - ❑ Iteratori
  - ❑ Algoritmi
- ❑ Performanță
  - ❑ Măsurată prin **benckmarks** de penalitate

# De ce să folosim STL?

- ❑ Micșorează timpul de implementare
  - ❑ Structuri de date deja implementate și testate
- ❑ Cod mai ușor de citit
- ❑ Robustetea
  - ❑ Structurile STL sunt actualizate automat
- ❑ Cod portabil
- ❑ Mantenabilitatea codului

# Componente STL



- Containere
- Iteatori
- Algoritmi

# Containere

## ❑ Definiție

❑ Un container este un **obiect care conține obiecte**

## ❑ Exemple

❑ list, vector, stack, etc

## ❑ Avantaje utilizare

❑ Simple și eficiente

❑ Fiecare container are atașat o mulțime de operații comune

❑ iteratori standard sunt disponibili pentru fiecare container

❑ Type-safe și omogene

❑ Sunt **non-intrusive** (ex. un obiect nu are nevoie de o clasă de bază pentru a fi membru a unui container )

# Exemplu

- Declarare, popularea și afișarea conținutului unei structuri de date de tip map, care stochează informații despre un produs și prețul lui

Stocarea în map a 2 produse și a prețurilor corespunzătoare

```
#include <map>
#include <string>

int main() {
    map<string, float> price;
    price["snapple"] = 0.75;
    price["coke"] = 0.50;

    string item;
    double total=0;
    while ( cin >> item )
        total += price[item];
}
```

Includerea bibliotecilor care conțin declararea: structuri de date map și a clasei string, care ușurează lucrul cu siruri de caractere  
Declararea unei structuri de date de tip map pentru a retine numele produselor și prețurile corespunzătoare

Prelucrarea: atât timp cât se citește de la tastatură un nume de produs, acesta se căută în map și se calculează prețul tuturor produselor a căror nume a fost citit de la tastatură

# Containere

- ❑ Tipuri de containere
  - ❑ Containere **sevențiale** (containere first – class)
    - ❑ Structuri de date liniare (vector, list, deque)
  - ❑ Containere **asociative** (containere first – class)
    - ❑ Structuri de date non-liniare, pot identifica rapid elemente (map, multimap, set, multiset)
    - ❑ Perechi cheie/valoare
  - ❑ **Adaptori** ai containerelor
    - ❑ Containere obținute prin adaptarea unor containere sevențiale (stack, queue, priority\_queue)
- ❑ Clase non-STL cu caracteristici și comportament (containere **near**)
  - ❑ Asemănătoare cu containerele, dar cu funcționalitate redusă (string, bitset, valarray)
  - ❑ Pot fi folosite cu iteratori cea ce le face accesibile manipulări algoritmilor definiți în STL

# Containere

## ❑ Containere sevențiale

- ❑ vector
- ❑ deque
- ❑ list

## ❑ Containere asociative

- ❑ set
- ❑ multiset
- ❑ map
- ❑ multimap

## ❑ Adaptori ai containerelor

- ❑ stack
- ❑ queue
- ❑ priority\_queue

## ❑ Containere near

- ❑ string
- ❑ bitset
- ❑ valarray

# Functii membre comune pentru container

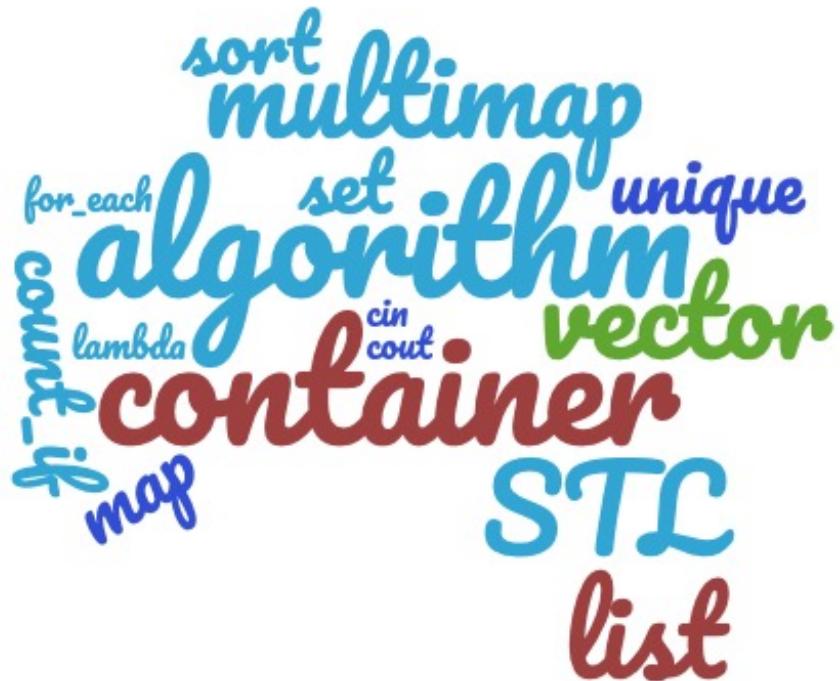
## ❑ Funcții membre pentru toate containerele

- ❑ Constructor implicit (default), constructor de copiere, destructor
- ❑ empty()
- ❑ max\_size(), size()
- ❑ <=, <, >=, >=, ==, !=

## ❑ Funcții pentru containere

- ❑ begin(), end()
- ❑ rbegin(), rend()
- ❑ erase(), clear()

# Componente STL



- Contenere
- Iteatori
- Algoritmi

# Iteratori

- ❑ Iteratori asemănători cu pointerii
  - ❑ Pointează spre primul element al unui container
  - ❑ Operatori comuni pentru toate containerele
    - ❑ \* derefențiere
    - ❑ ++ pointează spre următorul element
    - ❑ begin () întoarce un iterator spre primul element
    - ❑ end () întoarce un iterator spre ultimul element
- ❑ Utilizare iteratori cu secvențe
  - ❑ Containere
  - ❑ Secvențe de intrare: `istream_iterator`
  - ❑ Secvențe de ieșire: `ostream_iterator`

# Iteratori de scriere/citire

## □ Citire

### □ istream\_iterator

#### □ Exemplu

```
□ std::istream_iterator<int>
  inputInt( cin )
    □ Poate citi intrări de la cin
  □ int a = *inputInt;
    □ Deferențiat pentru a citi primul întreg de
      la cin
  □ ++inputInt;
    □ Trece la următorul întreg din flux
      (stream)
```

## □ Scriere

### □ ostream\_iterator

#### □ Exemplu

```
□ std::ostream_iterator<int>
  outputInt( cout )
    □ Poate scrie întregi la streamul cout
  □ *outputInt = 7;
    □ Afisează 7 la cout
  □ ++outputInt;
    □ Avansează iteratorul astfel încât să se
      poată afișa următorul întreg
```

# Iteratori de scriere/citire. Exemplu

```
#include <iostream>
#include <iterator> // ostream_iterator și istream_iterator
using namespace std;
int main()
{
    cout << "Introdu 2 intregi: ";
    istream_iterator<int> inputInt (cin);
    int nr1 = *inputInt; ← Citire numere întregi de la tastatură
    ++inputInt;
    int nr2 = *inputInt; ← Citire numere întregi de la tastatură

    ostream_iterator<int> outputInt (cout);
    cout << "rezultatul este: ";
    *outputInt = nr1 + nr2; ← Trimitere rezultat la streamul cout
    cout << endl;
}
```

## Output

Introdu 2 intregi: 5 7  
rezultatul este: 12

# Iteratori pentru containere. Tipuri

- ❑ Intrare (input)
  - ❑ Citește elemente dintr-un container, se poate deplasa doar **înainte**
- ❑ Ieșire (output)
  - ❑ Scrie elemente într-un container, se poate deplasa doar **înainte**
- ❑ Înaintare (forward)
  - ❑ Combină iteratori de intrare și ieșire
  - ❑ Pot **parcurge** o secvență **de mai multe ori**
- ❑ Bidirectionali
  - ❑ La fel ca cei de **înaintare**, dar se pot deplasa și în **sens invers**
- ❑ Acces aleatoriu (random)
  - ❑ La fel ca cei bidirectionali dar **pot „ sări” la orice element**

# Tipuri de iteratori suportați de containere

- ❑ Containere secvențiale
  - ❑ vector: aleator
  - ❑ deque: aleator
  - ❑ list: bidirectional
- ❑ Containere asociative (pt. toate acces bidirectional)
  - ❑ set / multiset
  - ❑ map / multimap
- ❑ Adaptori ai containerelor (nu au iteratori atașați)
  - ❑ stack
  - ❑ queue
  - ❑ priority\_queue

# Iteratori operații

## ❑ Toți

- ❑ `++p, p++`

## ❑ Iteratori de intrare

- ❑ `*p`
- ❑ `p=p1`
- ❑ `p==p1, p!=p1`

## ❑ Iteratori de ieșire

- ❑ `*p`
- ❑ `p=p1`

## ❑ Iteratori de înaintare

- ❑ Conțin operațiile iteratorilor de intrare și ieșire

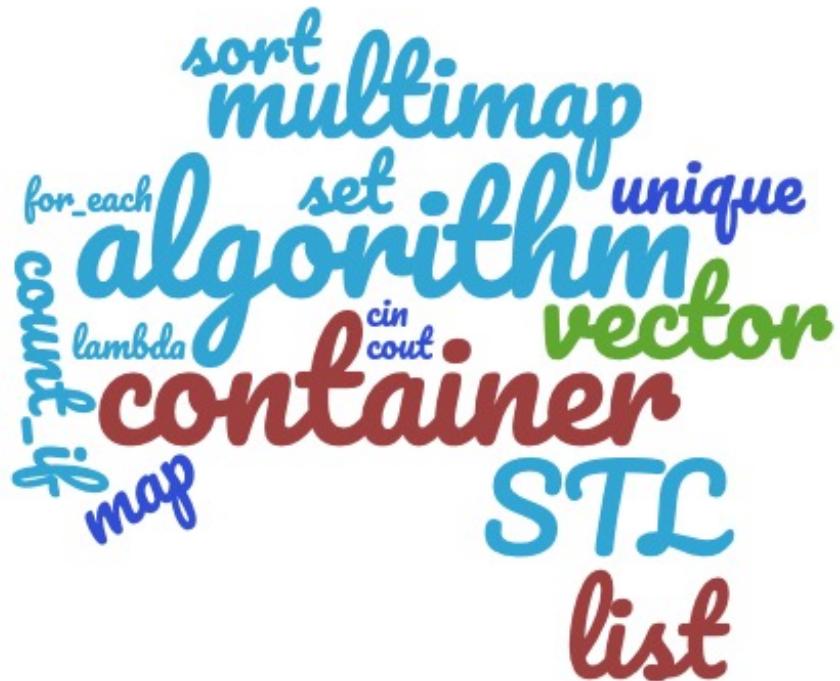
## ❑ Bidirectionali

- ❑ `--p, p--`

## ❑ Acces aleatoriu

- ❑ `p+i, p+=i`
- ❑ `p-i, p-=i`
- ❑ `p[i]`
- ❑ `p<p1, p<=p1`
- ❑ `p>p1, p>=p1`

# Componente STL



- Contenere
- Iteratori
- Algoritmi

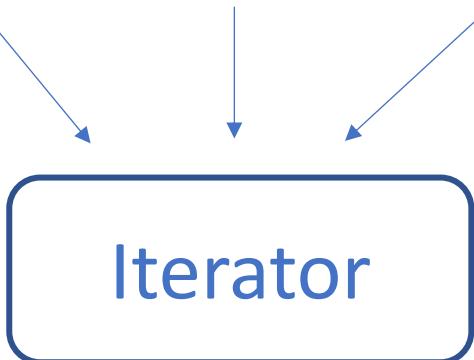
# Algoritmi

- ❑ Algoritmi STL utilizați de obicei împreună cu containerele
- ❑ Operează cu elementele containerelor indirect prin intermediul iteratorilor
- ❑ De multe ori operează pe secvențe de operatori
  - ❑ Perechi de operatori (ex. primul și ultimul element)
- ❑ De multe ori întorc iteratori
  - ❑ Ex: `find()`
    - ❑ Întoarce `end()` dacă elementul nu este găsit
- ❑ Economisesc timpul și efortul necesar implementării lor

# Algoritmi. Model

Algoritmi

sort(), find(), search(),  
copy(), ...



Containere

vector<T>, list<T>, map<T1,  
T2>, multimap<T1, T2>, ...

□ Separarea conceptelor

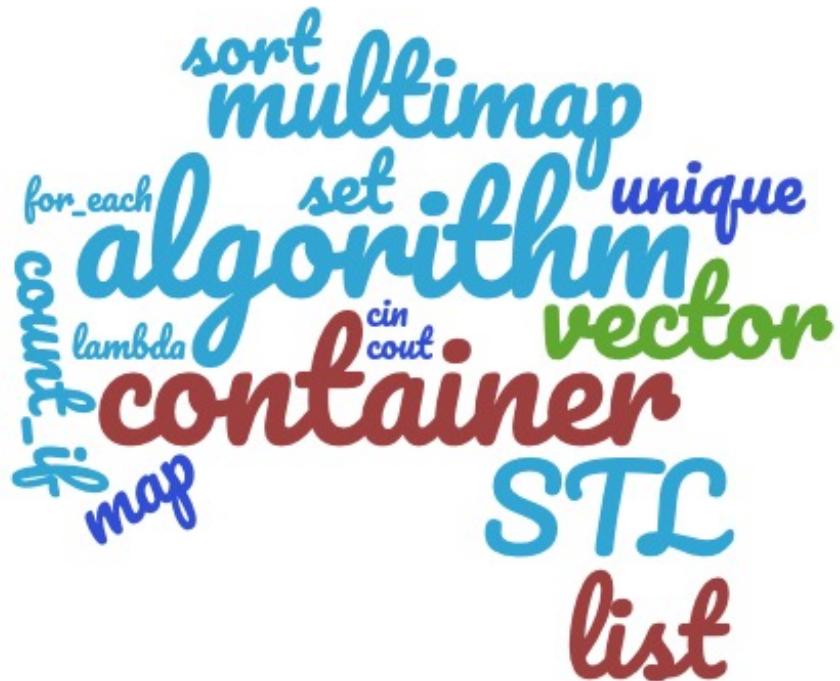
□ Algoritmi **manipulează** datele,  
dar nu știu despre containere

□ Containerele **depozitează**  
datele, dar nu știu despre  
algoritmi

□ Algoritmi și containerele  
**interactionează** prin intermediul  
iteratorilor

□ Fiecare container are un iterator atașat

# Componente STL



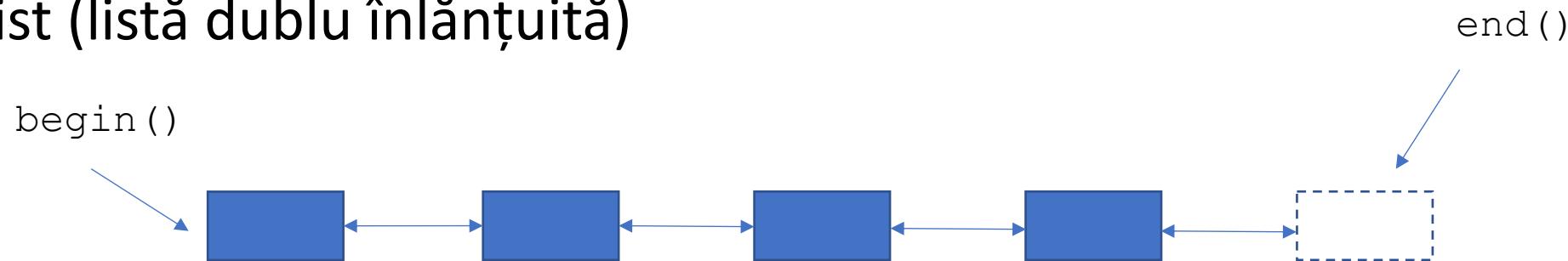
- ❑ Contenere
  - ❑ vector
  - ❑ list
  - ❑ deque
  - ❑ set/multiset
  - ❑ map/multimap
- ❑ Iteratori
- ❑ Algoritmi

# Containere secvențiale

## ❑ Vector (tablou dinamic)



## ❑ List (listă dublu înlănțuită)



# Containerul vector

- ❑ Header <vector>
- ❑ Inserare / ștergere rapidă de la sfârșitul tabloului
- ❑ Reprezentare: blocuri continue de memorie
  - ❑ Accesarea elementelor folosind []
  - ❑ Tablou dinamic
    - ❑ Crește cu un anumit factor atunci când mărimea (size()) depășește capacitatea (capacity())
    - ❑ Alocă o zonă de memorie continuă mai mare (la reallocare spațiul se doublează)
    - ❑ Face o copie a lui însuși
    - ❑ Dealocă vechea memorie
- ❑ Iteratori de acces aleatorii

# Vector

## □ Declarare

□ `std::vector < tip > variabila;`

## □ Iteratori

□ `std::vector ::iterator iterVar;`

□ Poate modifica valorile elementelor

□ `std::vector ::const_iterator iterVar;`

□ Nu poate modifica valorile elementelor

□ `std::vector ::reverse_iterator iterVar;`

□ Parcurge elementele in **ordine inversă**

□ Pentru a obtine punctul de început se folosește funcția `rbegin()`

□ Pentru a obtine punctul de sfârșit se folosește funcția `rend()`

□ `std::ostream_iterator nume (outputStream, separator)`

□ Separator – caracter care separă ieșirile

# Vector

## ❑ Funcții

❑ `v.push_back(valoare)`

❑ Adaugă un element la sfârșitul vectorului

❑ `v.size()`

❑ Dimensiune actuală a vectorului

❑ `v.capacity()`

❑ Cât de multe elemente poate să conțină vectorul înainte de a fi redimensionat

❑ `vector v(array, array+SIZE)`

❑ Constructor creează un vector `v` cu elementele tabloului a până la `array+SIZE`

# Vector

## ❑ Funcții

- ❑ v.insert (iterator, valoare)
  - ❑ Inserează valoarea înaintea locației iteratorului
- ❑ v.insert (iterator, array, array+SIZE)
  - ❑ Inserează un tablou de elemente înaintea poziției iteratorului
- ❑ v.erase( iterator)
  - ❑ Șterge elementul din container
- ❑ v.erase( iterator1, iterator2)
  - ❑ Șterge elementele începând cu iterator1 până la iterator2
- ❑ v.clear ()
  - ❑ Șterge întreg containerul

# Vector

## ❑ Funcții

❑ `v.front()`, `v.back()`

❑ Returnează primul, ultimul element al vectorului

❑ `v[element_number] = valoare`

❑ Atribuie valoarea elementului (supraîncărcarea operatorului de indexare)

❑ `v.at(element_number) = valoare`

❑ Atribuie valoarea elementului, verifică corectitudinea indexului

❑ Excepție `out_of_bounds`

# Parcurgere vector

- ❑ `for(int i = 0; i<v.size(); ++i)`
  - ❑ // de ce int?
  - ❑ // utilizare v[i]
- ❑ `for(vector<int>::size_type i = 0; i<v.size(); ++i)`
  - ❑ // mai lungă dar întotdeauna corectă
  - ❑ // utilizare v[i]
- ❑ `for(vector<int >::iterator p = v.begin(); p!=v.end(); ++p)`
  - ❑ // utilizare \*p
- ❑ Toate variantele sunt corecte
  - ❑ Nu există avantaje fundamentale ale unei abordări
  - ❑ Abordarea care folosește iterator este comună pentru toate containerele
  - ❑ Se preferă size\_type în loc de int pentru a împiedica erori rare

# Vector. Exemplu

```
#include <iostream>
#include <vector>
int main () {
    std::vector<int> a;
    std::vector<int> b(4, 100);
    std::vector<int> c(b.begin(), b.begin() + 2);
    int myints[] = {16, 2, 77, 29};
    std::vector<int> d(myints, myints + sizeof(myints) / sizeof(int));
    std::cout << "Afisare continut vector d:";
    for (std::vector<int>::iterator it = d.begin(); it != d.end(); ++it)
        std::cout << ' ' << *it; std::cout << '\n';
    return 0;
}
```

Definirea unui vector fără elemente - {}

Definirea unui vector care conține 4 elemente egale cu 100 - {100, 100, 100, 100}

Creează un vector cu primele 2 elemente ale vectorului b - {100, 100}

initializare vector cu un tablou de întregi - {16,2,77,29}

# Vector. Exemplu

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

int main () {
    vector<double> values(10);
    values.insert (values.begin() + 5, 1.4142);
    values.erase (values.begin() + 3);
}
```

Creează un vector cu 10 elemente inițializate cu 0.0 - {0,0,0,0,0,0,0,0,0,0}

Inserează 1.4142 pe a cincea poziție în vector - {0,0,0,0,1.4142,0,0,0,0,0}

Șterge elementul de pe poziția a treia din vector - {0,0,0,1.4142,0,0,0,0,0,0}

# Containerul list

- ❑ Header <list>
- ❑ Adăugare/ștergere eficientă oriunde în container
- ❑ Reprezentare internă: liste dublu înlántuite
  - ❑ Iteratori bidirecționali
- ❑ Utilizare
  - ❑ `std::list < tip > nume;`

# Containerul list

## ❑ Funcții atașate unui obiect, T, de tip list

❑ T.sort ()

❑ Sortare în ordine crescătoare

❑ T.splice (iterator, altObiect)

❑ Inserează valorile din variabila altObiect înaintea iterarorului

❑ T.merge (altObiect)

❑ Șterge altObiect și îl inserează în T sortat

❑ T.unique ()

❑ Șterge elementele duplicate

# Containerul list

- ❑ Funcții atașate unui obiect, T, de tip list
  - ❑ T.swap (altObiect)
    - ❑ Interschimbă conținutul
  - ❑ T.assign (iterator1, iterator2)
    - ❑ Înlocuiește conținutul cu elementele din domeniul iteratorului
  - ❑ T.remove (valoare)
    - ❑ Sterge toate instanțele valorii ‘valoare’

# Containerul list. Exemplu

```
#include <list>
#include <string>
int main () {
    std::list<std::string> mylist;
    std::list<std::string>::const_iterator it;

    mylist.push_back ("one");
    mylist.push_back ("two");
    mylist.push_back ("Three");

    mylist.sort(); // sortare lista

    std::cout << "lista contine:";
    for (it=mylist.begin(); it!=mylist.end(); ++it)
        std::cout << ' ' << *it;
    std::cout<<'\n';
}.
```

Declararea unei liste care conține siruri de caractere

Declararea unui iterator pentru a parcurge elementele listei

Adăugarea de elemente la listă

Afișarea listei folosind un iterator

# Deque

- ❑ Coadă cu intrări în ambele părți
  - ❑ Inserare rapidă la începutul și sfârșitul cozi
  - ❑ Header <deque>

- ❑ Funcții
  - ❑ push\_back(), push\_front()
  - ❑ pop\_back(), pop\_front()
  - ❑ empty()
  - ❑ insert()
  - ❑ erase()

# Deque. Exemplu

```
#include <iostream>
#include <deque>
int main () {
    std::deque<int> mydeque; Declararea unei cozi goale
    mydeque.push_back (100); Adăugarea de elemente în coadă
    mydeque.push_back (200);
    mydeque.push_back (300);

    std::cout << "Scoatere elemente din coada:";
    while (!mydeque.empty ()) {
        std::cout << ' ' << mydeque.front (); Eliminarea de elemente din coadă
        mydeque.pop_front ();
    }

    std::cout << "\n Dimensiunea finala a cozi este:" << int(mydeque.size ()) << '\n';
    return 0;
}
```

# Containere asociative

- ❑ Au acces direct la stocarea/ștergerea de elemente

- ❑ Folosesc chei de căutare

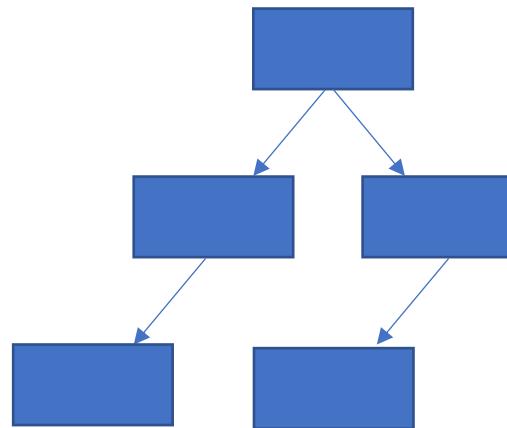
- ❑ 4 tipuri: multiset, set, multimap, map

- ❑ cheile sunt sortate

- ❑ multiset și multimap permit chei duplicate

- ❑ multimap și map conțin valori asociate

- ❑ multiset și set conțin doar valori



# Map/Multimap

## MAP

- Header <map>
- Nu permite chei duplicate
- Relația one-to-one
  - Ex. O țară poate avea o capitală
- Se poate folosi operatorul[]

## MULTIMAP

- Header <map>
- Permite chei duplicate
- Relația one-to-many
  - Ex. Un student poate participa la mai multe cursuri
- Iteratori bidirectionali

# Map/Multimap

```
#include <map>
typedef std::multimap< int, double, std::less< int > > mm;

int main() {
    mm pairs;

    cout << "Acum sunt " << pairs.count(15) << " perechi cu cheia 15 in multimap\n";
    pairs.insert( mm::value_type( 15, 2.7 ) ); pairs.insert( mm::value_type( 15, 99.3 ) );
    cout << " Dupa insert sunt " << pairs.count( 15 ) << " perechi cu cheia 15 \n\n";

    pairs.insert( mm::value_type( 30, 111.11 ) ); pairs.insert( mm::value_type( 10, 22.22 ) );
    pairs.insert( mm::value_type( 25, 33.333 ) ); pairs.insert( mm::value_type( 20, 9.345 ) );
    pairs.insert( mm::value_type( 5, 77.54 ) );

    cout << "Multimap-ul contine :\nCheie\tValoare\n";
    for ( mm::const_iterator iter = pairs.begin(); iter != pairs.end(); ++iter )
        cout << iter->first << '\t' << iter->second << '\n';
    cout << endl;
}
```

Ordinea în care sunt stocate  
valorile în map/multimap

Definirea unui nume mai  
scurt pentru multimap

Inserare valori

Afișare conținut  
map/multimap

# Map/Multimap

```
#include <iostream>
using std::cout; using std::endl;
#include <map>
typedef std::map< int, double, std::greater< int > > mid;
int main() {
    std::map< int, double, std::greater< int > > pairs;
    pairs[15] = 2.7 ;           ← Inserare valori folosirea operatorului de indexare
    pairs.insert( std::make_pair (30, 111.11 ) ); ← Inserare valori folosind templateul pair
    pairs.insert(std::map< int, double, std::greater< int > > ::value_type( 5, 1010.1 ) );
    pairs.insert(std::map< int, double, std::greater< int > > ::value_type( 10, 22.22 ) );
    pairs.insert( mid::value_type( 25, 33.333 ) );
    pairs.insert( mid::value_type( 5, 77.54 ) ); ← Duplicat, va fi ignorat
    pairs.insert( mid::value_type( 20, 9.345 ) );
    pairs.insert( mid::value_type( 15, 99.3 ) );
    cout << "perechi:\nCheie\tValoare\n";
    for (std::map< int, double, std::greater< int > > ::const_iterator iter = pairs.begin(); iter
!= pairs.end(); ++iter )
        cout << iter->first << '\t' << iter->second << '\n';
    pairs[ 25 ] = 9999.99;           ← folosirea operatorului de indexare pentru modificarea valorii
}
```

Definirea unui nume mai scurt pentru multimap

Inserare valori folosirea operatorului de indexare

Inserare valori folosind templateul pair

Duplicat, va fi ignorat

folosirea operatorului de indexare pentru modificarea valorii

# Set/Multiset

## SET

Header <set>

Retine un sir de valori **distinrete**  
**sortat**

## MULTISET

Header <set>

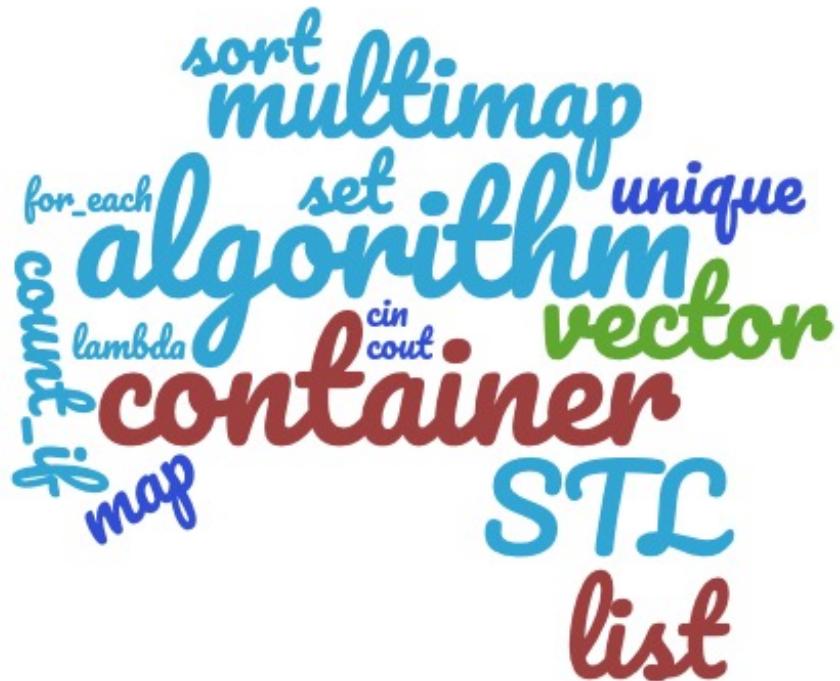
Retine un sir de valori **sortat**

# Set/Multiset

```
#include <iostream>
#include <set>
int main () {
    std::set<int> myset;
    std::set<int>::iterator it;
    std::pair<std::set<int>::iterator,bool> ret;
    for (int i=1; i<=5; ++i) myset.insert(i*10);
    ret = myset.insert(20);
    if (ret.second==false) it=ret.first;
    int myints[] = {5,10,15};
    myset.insert (myints,myints+3);
    std::cout << "myset contine:";
    for (it=myset.begin(); it!=myset.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
}
```

Adăugare elementelor 10,20,30,40,50 în set  
Elementul 20 există în set nu va mai fi inserat  
it pointează spre elementul 20  
Elementul 10 există în set nu va mai fi inserat, doar  
elementele 5 și 15 vor fi inserate

# Componente STL



- Containare
  - ❑ vector
  - ❑ list
  - ❑ deque
  - ❑ set/multiset
  - ❑ map/multimap
- ❑ Iteatori
- ❑ Algoritmi

# Algoritmi

- ❑ Există 60 de algoritmi definiți în biblioteca standard
- ❑ Header <algorithm>
- ❑ Pot fi aplicăți containerelor standard, string și built-in array
- ❑ Definiți ca funcții template
- ❑ Tipuri
  - ❑ Nu modifică secvența
  - ❑ Modifică secvența
  - ❑ Secvențe sortate
  - ❑ Altele: multimi, heap, minimum, maximum, permutări

# Algoritmi

- ❑ `for_each()`
  - ❑ Invocă o funcție pentru fiecare element
- ❑ `find()`
  - ❑ Caută prima apariție a unui argument
- ❑ `find_if()`
  - ❑ Prină apariție care se potrivește cu predicatul
- ❑ `count()`
  - ❑ Numără aparițiile unui element
- ❑ `count_if()`
  - ❑ Numără potrivirile predicatorului
- ❑ `replace()`
  - ❑ Înlocuiește un element cu noua valoare
- ❑ `replace_if()`
  - ❑ Înlocuiește elementul care se potrivește cu predicatorul cu noua valoare
- ❑ `copy()`
  - ❑ Copiază elementele
- ❑ `unique_copy()`
  - ❑ Copiază elementele care nu sunt duplicate
- ❑ `sort()`
  - ❑ Sortează elementele
- ❑ `equal_range()`
  - ❑ Găsește elemente cu valori echivalente
- ❑ `merge()`
  - ❑ Combină secvențe sortate

# Algoritmi

```
char nextLetter() ;  
bool equal_a(char c);  
void codificare(char);  
int main () {  
    std::vector< char > chars(10);  
    std::ostream_iterator< char > output( cout, " " );  
  
    std::generate_n( chars.begin(), 5, nextLetter );  
    cout << "\n\nVectorul de caractere dupa umplerea primelor 5 pozitii:\n";  
    std::copy( chars.begin(), chars.end(), output );  
  
    std::fill(chars.begin()+5, chars.end(), 'a');  
    cout << "\n\nVectorul de caractere dupa umplerea ultimelor 5 pozitii:\n";  
    std::copy(     chars.begin(), chars.end(), output );  
  
    std::replace_if(chars.begin(), chars.end(), equal_a, '-');  
    cout << "\n\nVectorul de caractere dupa inlocuirea lui 'a' cu '-':\n";  
    std::copy( chars.begin(), chars.end(), output );
```

Aloca spațiu pentru un vector cu 10 elemente

Definește un stream care se leagă cu ecranul și separă elementele afisate prin spații

Populează primele 5 elemente cu literele generate de funcția nextLetter() – {A, B, C, D, E}

Populează următoarele 5 elemente cu litera 'a' – {A, B, C, D, E, a, a, a, a, a}

Inlocuiește caracterul 'a' cu caracterul '-' – {A, B, C, D, E, -, -, -, -}

# Algoritmi

```
cout << " \n Vectorul de caractere dupa codificare: \n";
std::for_each (chars.begin(), chars.end(), codificare);
}

// intoarce urmatoarea litera din alfabet (incepand cu A)
char nextLetter() {
    static char letter = 'A';
    return letter++;
}

//verifica daca un caracter este egal cu litera a
bool equal_a( char c){ return c=='a'; }

//sifteaza cu 10 caracterele din sir
void codificare( char c){ cout << (char)(c+10) << '\t'; }
```

Afișează elementele  
vectorului sifrate cu 10  
caracter

# Clasa string

- ❑ **string**
  - ❑ este o secvență de caractere
- ❑ Definită în biblioteca <string> prin clasa std::string
- ❑ Permite operații comune cu sirurile de caractere
  - ❑ concatenare, inserare, atribuire, comparare, adăugare, căutarea unui sub sir, extragerea unui sir
- ❑ Suportă orice set de caractere

# Clasa string. Exemplu

```
string foo() {
    string s1 = "First string";
    string s2 = s1, s3(s1, 6, 3);
    wstring ws(s1.begin(), s1.end()); // string of wchar_t
    s3 = s2;
    s3[0] = 'A';

    const char* p = s3.data(); // conversion to C
    delete p; // ERROR: the array is owned by string object

    if(s1==s2) cout << "Strings have same content" ;
    s1 += " and some more." ;
    s2.insert( 5, "smth ");

    string::size_type i1 = s1.find("string"); // i1=6
    string::size_type i2 = s1.find_first_of("string"); // i2=3
    s1.replace(s1.find ("string"), 3, "STR");
    cout << s.substr ( 0, 5);

    cin >> s3;
    return s1 + ' ' + s2; // concatenation
}
```



ÎNTREBĂRI