Programare II
Limbajul C/C++

CURS 11-12

Curs anterior

- Tipuri de date abstracte
- Tipuri de date generice
- Funcţii şablon
- Clase şablon

Cuprins

Stanard 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 (implementări 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 (parcurgere, sortare, reuniune, ...)
- Prelucrării numerice (numere complexe, BLAS=Basic Liniar Algebra Subprograms etc.)
- Bază pentru alte biblioteci

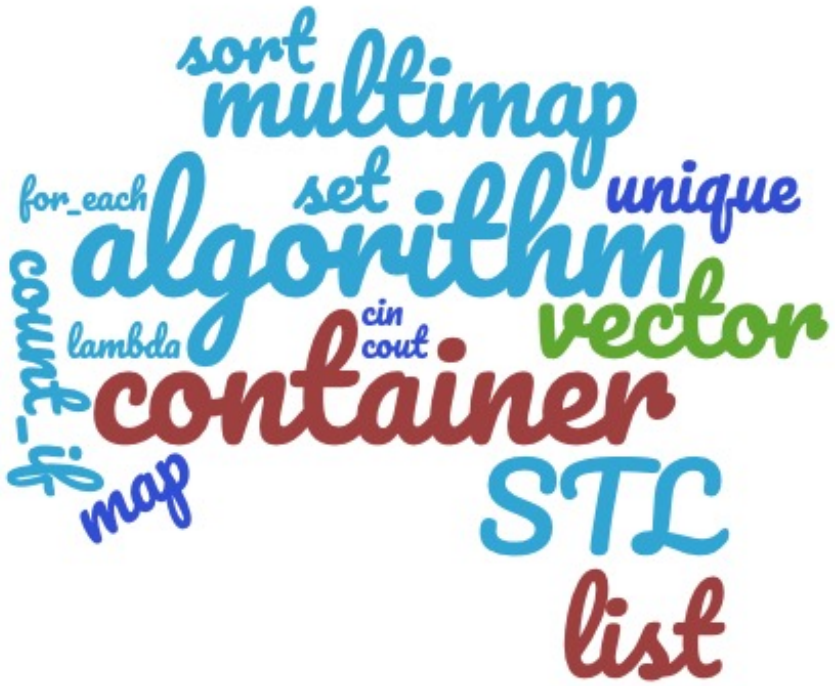
Standard Template Library

- 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
- Robustețea
  - Structurile STL sunt actualizate automat
- Cod portabil
- Mentenabilitatea 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
  - iterarori 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

```
#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 șiruri de caractere

Declararea unei structuri de date de tip map pentru a retine numele produselor și prețurile corespunzătoare

Stocare a în map a 2 produse și a prețurilor corespunzătoare

Prelucrarea: atâta 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 secvențiale (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 secvențiale (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 secvențiale

- vector
- deque
- list

Containere asociative

- set
- multiset
- map
- multimap

Adaptori ai containerelor

- stack
- queue
- priority_queue

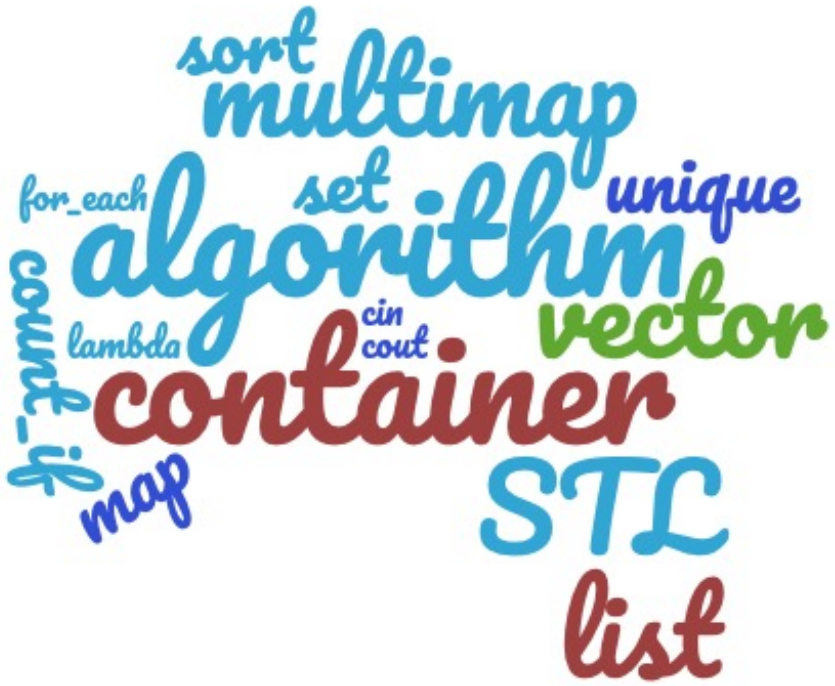
Containere near

- string
- bitset
- valarray

Funcții 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



- Containere
- Iteatori
- Algoritmi

Iteratori

- Iteratori asemănători cu pointerii
  - Pointează spre primul element al unui container
  - Operatori comuni pentru toate containerele
    - `*` deferenţiere
    - `++` pointeză spre următorul element
    - `begin()` întoarce un iterator sper primul element
    - `end()` întoarce un iterator sper 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;`
      - Deferenţiat 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;`
      - Afișează 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 şi isteam_iterartor
using namespace std;
int main()
{
   cout << "Introdu 2 intregi: ";
   istream_iterator <int> inputInt (cin);
   int nr1 = *inputInt;
   ++inputInt;
   int nr2 = *inputInt;

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

Citire numere întregi de la tastatură

Trimitere rezultat la streamul `cout`

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
- Bidirecționali
  - La fel ca cei de înaintare, dar se pot deplasa și în sens invers
- Acces aleatoriu (random)
  - La fel ca cei bidirecționali dar pot „sări” la orice element

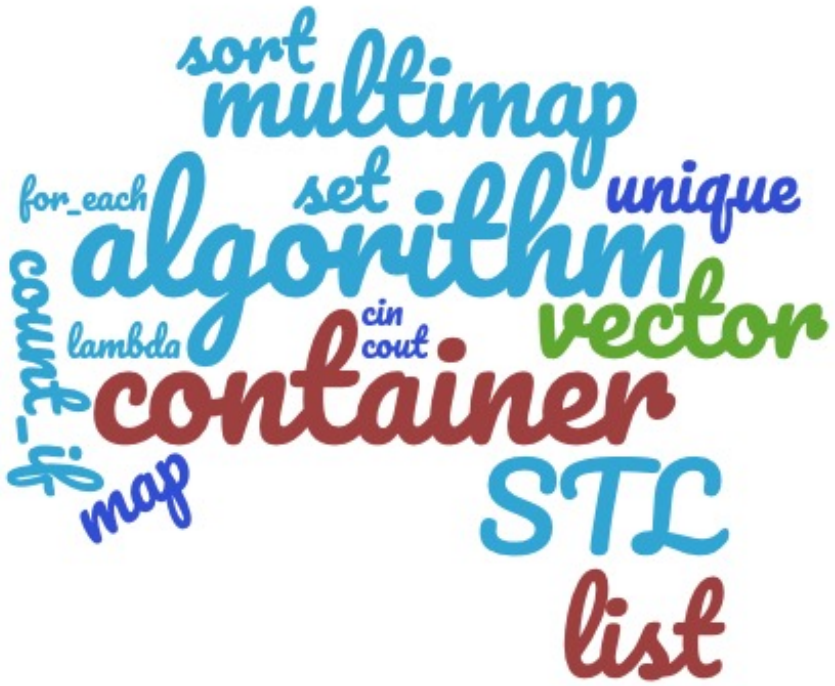
Tipuri de iteratori suportați de containere

- Containere secvențiale
  - vector: aleator
  - deque: aleator
  - list: bidirecţional
- Containere asociative (pt. toate acces bidirecțional)
  - set / multiset
  - map / multimap
- Adaptori ai containerelor (nu au iteratori atașați)
  - stack
  - queue
  - priority_queue

Iteratori operaţii

- 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
- Bidirecționali
  - `--p, p--`
- Acces aleatoriu
  - `p+i, p+=i`
  - `p-i, p-=i`
  - `p[i]`
  - `p<p1, p<=p1`
  - `p>p1, p>=p1`

Componente STL

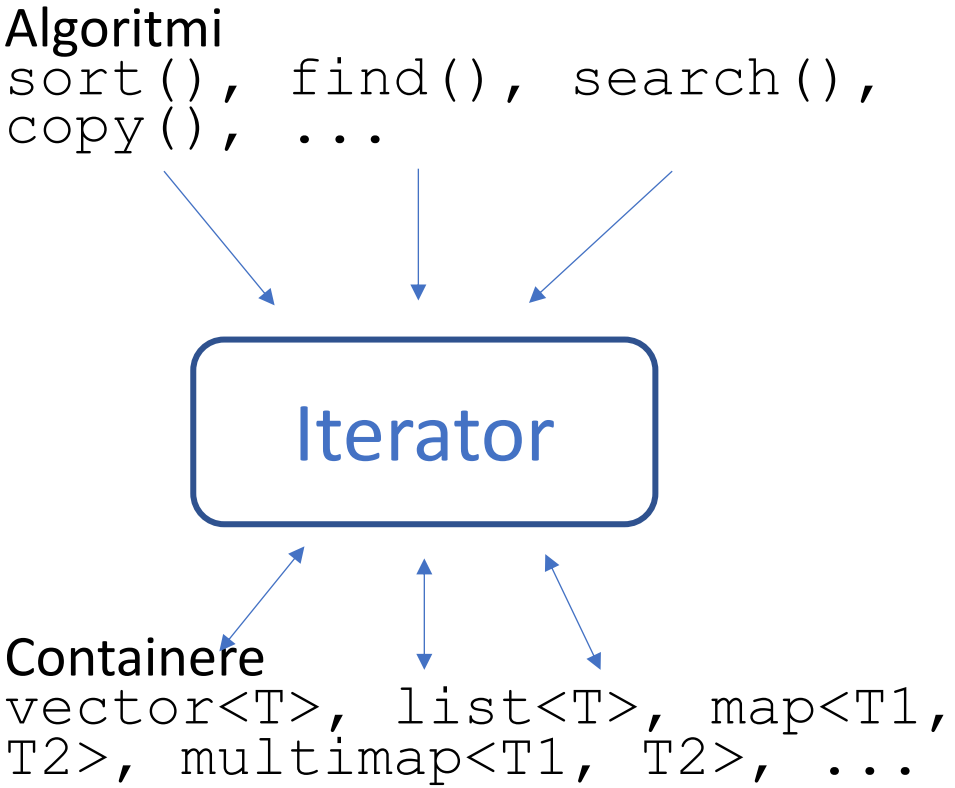


- Containere
- Iteatori
- Algoritmi

Algoritmi

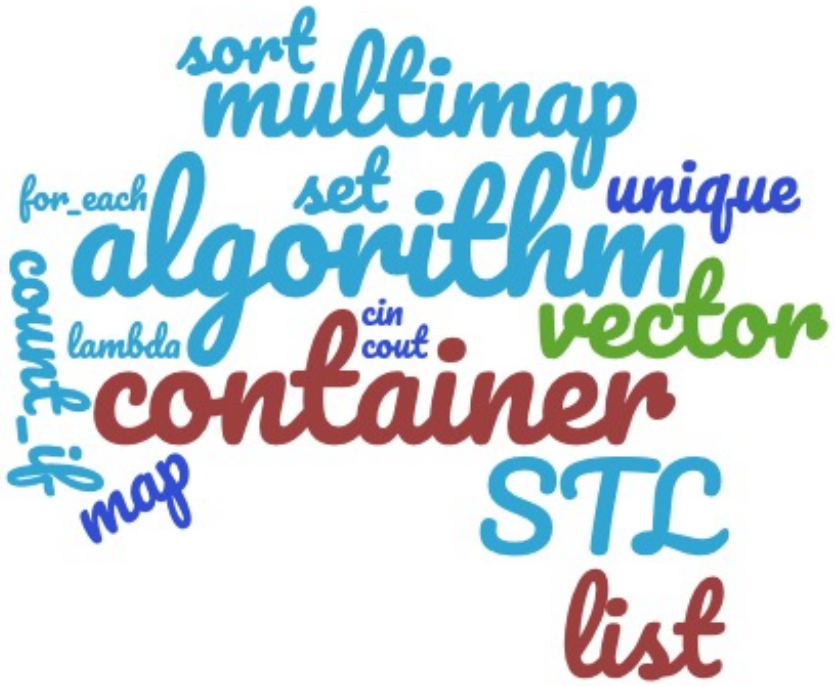
- Algoritmi STL utilizați de obicei împreună cu containerele
- Operează cu elementele containerelor indirect prin intermediul iteratorilor
- De multe ori operează pe secvențe de operatori
  - Perechi de operatori (ex. primul și ultimul element)
- De multe ori întorc iteratori
  - Ex: `find()`
    - Întoarce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

Algoritmi. Model



- Separarea conceptelor
  - Algoritmi manipulează datele, dar nu știu despre containere
  - Containerele depozitează datele, dar nu știu despre algoritmi
  - Algoritmi și containerele interacționează prin intermediul iteratorilor
    - Fiecare container are un iterator atașat

Componente STL

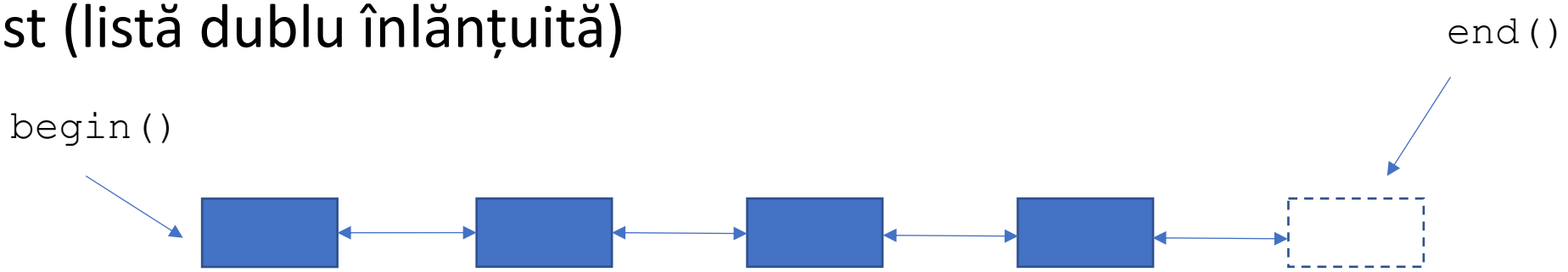


- Containere
  - vector
  - list
  - dequeue
  - set/multiset
  - map/multimap
- Iteatori
- Algoritmi

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 realocare spațiul se dublează)
    - 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 obține punctul de început se folosește funcția `rbegin()`
    - Pentru a obține 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 pană 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
- `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 in 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}

inițializare 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.remove (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 in vector - {0,0,0,0,1.4142,0,0,0,0,0,0}

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 înlănțuite
  - Iteratori bidirecționali
- Utilizare
  - `std::list < tip > nume;`

Containerul list. Funcții

- `T.sort ()`
  - Sortare în ordine crescătoare
- `T.splice (iterator, altObiect)`
  - Inserează valorile din variabila altObiect înaintea iterarorului
- `T.merge (altObiect)`
  - Șterge altObiect şi îl inserează în T sortat
- `T.unique()`
  - Șterge elementele duplicate
- `T.swap (altObiect)`
  - Interschimbă conținutul
- `T.assign (iterator1, iterator2)`
  - Înlocuiește conținutul cu elementele din domeniul iteratorului
- `T.remove (valoare)`
  - Șterge 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 șiruri 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>

- Functii
  - `push_back(), push_front()`
  - `pop_back(), pop_front()`
  - `empty()`
  - `insert()`
  - `erase()`

Deque. Exemplu

```
#include <iostream>
#include <deque>
int main () {
  std::deque<int> mydeque;

  mydeque.push_back (100);
  mydeque.push_back (200);
  mydeque.push_back (300);

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

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

Declararea unei cozi goale

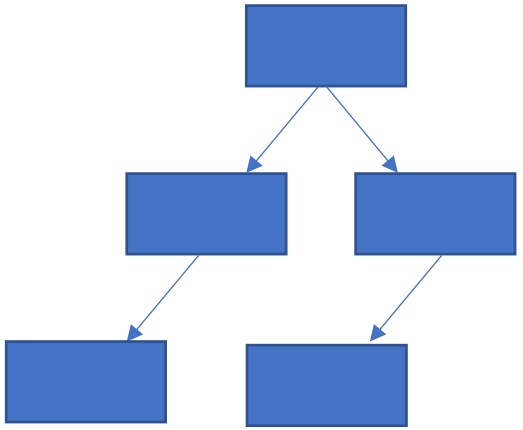
Adăugarea de elemente în coadă

Eliminarea de elemente din coadă

Afișare număr de elemente în coadă

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

Ordinea în care sunt stocate valorile în map/multimap

Definirea unui nume mai scurt pentru multimap

Inserare valori

Afișare conținut 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\tValoae\n";
  for ( mm::const_iterator iter = pairs.begin(); iter != pairs.end(); ++iter )

    cout << iter->first << '\t' << iter->second << '\n';

  cout << endl;
}
```

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 ;
  pairs.insert( std::make_pair (30, 111.11 ) );
  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 ) );
  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;
}
```

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 șir de valori distincte sortat

MULTISET

- Header <set>
- Retine un șir 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';
}
```

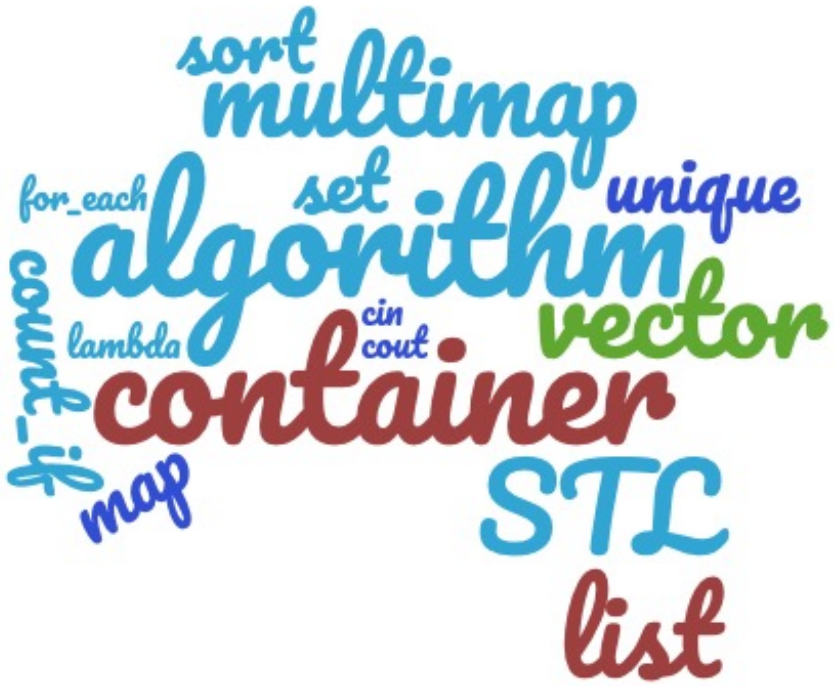
Adăugare elementelor 10,20,30,40,50 în set

Elementul 20 există in set nu va mai fi inserat

it pointează spre elementul 20

Elementul 10 există in set nu va mai fi inserat, doar elementele 5 si 15 vor fi inserate

Componente STL



- Containaere
  - vector
  - list
  - dequeue
  - set/multiset
  - map/multimap
- Iteatori
- Algoritmi

Algoritmi

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

Algoritmi

- `for_each()`
  - Invocă o funcție pentru fiecare element
- `find()`
  - Caută prima apariție a unui argument
- `find_if()`
  - Prima apariție care se potrivește cu predicatul
- `count()`
  - Numără aparițiile unui element
- `count_if()`
  - Numără potrivirile predicatului
- `replace()`
  - Înlocuiește un element cu noua valoare
- `replace_if()`
  - Înlocuiește elementul care se potrivește cu predicatul 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 afișate 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 \nVectorul de caractere dupa codificare: \n";
  std::for_each (chars.begin(), chars.end(), codificare);
}

// intoarce urmatoarea litera din alfabel (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 siftate cu 10 caracter

Pointeri la funcții

- Exemplu
  - Sortarea unui liste de persoane

- Dorim sa sortam după
  - Nume de familie
  - Vârstă
  - Studii
  - ....

- Cum implementăm aceste cerințe?
  - Definim mai multe funcţii de sortare
  - Definim o funcție de sortare care pe lângă șirul de elemente care trebuie sortat primește
  - Pointeri la funcții (funcții callback sau listener)

Pointeri la funcții. Exemplu

```
void my_int_func(int x) {cout<<x; }

int main() {

  void (*foo)(int); //declare pointer de tip functie

  foo = &my_int_func; //initializare

  foo( 2 ); // apel functie
  (*foo)( 2 ); // apel functie, dar nu este obligatoriu

  return 0;
}
```

Functori

- Functor (function-like object / function object) este o instanță a unei clase pentru care sa supraîncărcat operatorul funcție ()

- Avantaje
  - Suportă operații mai complexe decât funcțiile obișnuite

- Mecanism de customizare a comportamentului algoritmilor standard

- Header <functor>

Functori. Exemplu

```
template <class T> class SumMe {
  public :
      SumMe(T i=0) : sum(i) { }
      void operator() (T x) { sum += x; }
      T result() const { return sum; }
  private :
     T sum ;
};
void f(vector<int> v) {
   SumMe<int> s;
   s = for_each (v.begin(), v.end(), s);
   cout << "Sum is " << s.result();
}
```

Supraîncărcarea operatorului funcție ()

Funcții Lambda

- Funcții lambda (C++11)
  - Sunt funcții fără nume
- Definesc funcționalitatea în locul în care sunt definite
- Funcțiile lambda ar trebui să fie
  - Concise
  - Ușor de înțeles (self explaining)

Funcții Lambda

- Sintaxă
  - [] () -> { ...}

- Unde
  - [] – utilizat pentru a captura variabile referință sau copii de variabile
  -
  - () – se specifică parametrii

  - -> - specifică tipul de return pentru expresii lambda mai sofisticate

  - {} – include expresii și bucăți de cod

Funcții Lambda

- [] – capturarea variabilelor
  - [a, &b] – variabila a este capturată prin copiere și variabila b prin referință
  - [this] – captează obiectul curent (*this) prin referință
  - [&] – capturează toate variabile automatice folosite în corpul expresiei lambda prin referința de asemenea captează și obiectul this prin referință dacă este cazul
  - [=] – capturează toate variabile automatice folosite în corpul expresiei lambda prin copiere de asemenea captează și obiectul this prin referință dacă este cazul
  - [] nu captează nimic

Funcții Lambda

- Sortează elementele unui vector
  - `vector<int> vec={3,2,1,5,4};`

- C++98

```
class MySort{
public:
   bool operator()(int v, int w){ return v > w;}
};
sort(vec.begin(),vec.end(),MySort());
```

- C++11

```
sort(vec.begin(),vec.end(),[](int v,int w){
                                    return v>w;});
```

Funcții Lambda

- Numără câte elemente ale unui vector sunt egale cu o valoare
- C++98

```
class Functor {
  public:
  int &a;
  Functor(int &_a):a(_a){}
  bool operator()(int x) const { return a==x; }
};
int a=45;
count_if(v.begin(), v.end(), Functor(a));
```

- C++11

```
int a=45;
count_if(v.begin(), v.end(), [&a](int b){return a==b;});
```

- C++14

```
int a=45;
count_if(v.begin(), v.end(), [&a](auto b){return a==b;});
```

Clasa string

- string
  - este o secvență de caractere
- Definită în biblioteca `<string>` prin clasa `std::string`
- Permite operații comune cu șirurile de caractere
  - concatenare, inserare, atribuire, comparare, adăugare, căutarea unui sub șir, extragerea unui șir
- 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

}
```

Concepte >=C++11

- Deducerea tipului
  - `auto`
  - `decltype`
- Cicluri Range-Based
- Inițializarea obiectelor – (), {}
- Specificatori `overrride` și `final`

Deducerea tipului- auto

- Compilatorul determina tipul

```
auto myString = “my string”;
auto myInt = 6;
auto p1 = 3.14;
```

- Obținerea unui iterator la primul element al unui vector

```
vector<int> v;
vector<int>::iterator it1= v.begin(); // C++98
auto it2= v.begin(); //C++11
```

Deducerea tipului folosing- decltype

- Compilatorul determină tipul unei expresii

```
decltype("str") myString= "str"; // C++11
decltype(5) myInt= 5; // C++11
decltype(3.14) myFloat= 3.14; // C++11
decltype(myInt) myNewInt= 2011; // C++11
int add(int a,int b){ return a+b; };
decltype(add) myAdd= add; // (int)(*)(int, int)
myAdd(2,3)
```

Deducerea tipului de return a unei funcții

- Declarera unei funcții

```
template auto add(T1 first, T2 second) ->
decltype(first  + second){ return first + second; }
add(1,1);
add(1,1.1);
add(1000LL,5);
```

- Rezultatul este de tip
  - int
  - double
  - long int

Cicluri Range-Based

- Iterarea unui container se poate realiza astfel

```
std::vector<int> v;
...
for (int i : v) std::cout << i;
```

- Se pot folosi și variabile referință

```
for (int& i : v) std::cout << ++i;
```

- Se poate folosi și auto

```
for (auto i : v) std::cout << i; // same as above
for (auto& i : v) std::cout << ++i;
```

Inițializarea obiectelor – (), {}

- Inițializarea obiectelor se poate face cu paranteze rotunde, acolade sau egal

```
int x(0);

int x=0;

int x{0};

int x={0}
```

Inițializarea obiectelor – (), { }

- Inițializarea containărelor

```
std::vector<int> v{ 1, 3, 5 };
vector<string> vec= {"Scott",st,"Sutter"};
unordered_map<string, int> um= {{"C++98",1998}, {"C++11", i}};
```

- Inițializarea membrilor clasei

```
class Foo{
  ...
private:
    int x{ 0 }; // OK, valoare implicita a lui x este zero
    int y = 0; // OK
    int z(0); // eroare!
};
```

Inițializarea obiectelor – (), { }

```
class Foo{
public:
      Foo(int i, bool b);
      Foo(int i, double d);
};
```

- Creare obiectelor

```
Foo w1; //apel constructor default
Foo w2=w1; //apel constructor de copiere
w1 = w2; //apel operator egal
Foo f1(10, true); //apel primul constructor
Foo f2{10, true}; //apel primul constructor
Foo f3{10, 5.5}; //apel al doilea constructor
```

Specificatori overrride și final

- Override
  - specifica că o metodă suprascrie o metodă a supraclasei
  - Este o indicație care spune compilatorului să verifice dacă există în supraclasă o metodă cu același prototip

- Exemplu

```
class B {
public:
  virtual void f(int) {
      std::cout << "B::f" << std::endl;
  }
};
class D : public B {
public:
    virtual void f(int) const override {
      std::cout << "D::f" << std::endl;
    }
};
```

Specificatori override si final

- Final
  - face metoda imposibil de suprascris
- Exemplu

```
class B {
public:
  virtual void f(int) {std::cout << "B::f" << std::endl;}
};
class D : public B {
public:
  virtual void f(int) override final {std::cout << "D::f" << std::endl;}
};
class F : public D {
  virtual void f(int) override {std::cout << ”F::f" << std::endl;}
};
```