10.Standard Template Library (II)

- Containeri asociativi

Containerii asociativi sunt proiectați pentru a stoca şi a accesa elemente prin intermediul *cheilor* numite, uneori, şi *chei de căutare*. Cei patru containeri asociativi sunt `multiset`, `set`, `multimap` şi `map`. În fiecare container cheile sunt păstrate sortate. Iterarea printr-un container asociativ parcurge elementele în ordinea sortată. Clasele `multiset` şi `set` manipulează mulțimi de valori pentru care valorile sunt chiar chei de căutare, pentru care nu există, aşadar, o separare între valori şi chei. Clasele `multimap` şi `map` suportă operații de manipulare a valorilor asociate cu chei. Aceste valori se mai numesc şi *valori mapate*. Principala diferență dintre clasele `multiset` şi `set` este că prima permite chei duplicate față de a doua care nu permite aşa ceva. Asemănător, clasa `multimap` permite chei duplicate asociate cu valori, în timp ce `map` permite doar chei unice. Suplimentar față de funcțiile pe care le oferă toți containerii, cei asociativi dispun şi de funcțiile `find`, `lower_bound`, `upper_bound` şi `count`.

Vom prezenta în programele de mai jos vom prezenta modul în care se pot folosi containerii asociativi `multiset` şi `map`.

Containerul asociativ `multiset`

Ordinea elementelor într-un container `multiset` este determinată de un *obiect comparator*. De exemplu, într-un container `multiset` cu valori de tip întreg, elementele pot fi sortate în ordine crescătoare ordonând cheile prin comparatorul `less<int>`. Cheile containerilor asociativi sortate cu `less<int>` trebuie să suporte compararea cu `operator<`. Un container `multiset` foloseşte iteratori bidirecționali.

Programul de mai jos demonstrează folosirea containerului asociativ `multiset` pentru valori întregi sortate crescător. Pentru a putea folosi containerii `multiset` şi `set`, programul trebuie să includă fişierul header `<set>`. Containerii `multiset` şi `set` au aceleaşi funcții membre.

Exemplu

test_multiset.cpp

```
#include <iostream>

using std::cout;
using std::endl;

#include <iterator>
#include <set>
#include <algorithm>

int main()
{
  const int SIZE = 10;
  int a[SIZE] = {7,22,9,1,18,30,100,22,85,13};
  typedef std::multiset<int, std::less<int> > ims;
  ims intMultiset; //ims = integer multiset
  std::ostream_iterator<int> output(cout, " ");
```

```
  cout << "Sunt " << intMultiset.count(15)
       << " valori de 15 in multiset\n";
  intMultiset.insert(15);
  intMultiset.insert(15);
  cout << "Dupa insert, sunt "
       << intMultiset.count(15)
       << " valori de 15 in multiset\n";
  ims::const_iterator result;
  result = intMultiset.find(15);  //find returneaza un iterator

                                  //daca iteratorul nu este
  if(result != intMultiset.end()) //pe pozitia end
    cout << "A fost gasita valoarea 15\n";
                                  //a fost gasita valoarea 15

  result = intMultiset.find(20);

  if(result == intMultiset.end())
    cout << "Nu a fost gasita valoarea 20\n";

  intMultiset.insert(a, a+SIZE);//adauga elemente din tabloul a
  cout << "Dupa insert intMultiset contine:\n";
  std::copy(intMultiset.begin(), intMultiset.end(), output);

  cout << "\nVecinul lui 22 de pe pozitia din stanga : "
       << *(intMultiset.lower_bound(22));
  cout << "\nVecinul lui 22 de pe pozitia din dreapta : "
       << *(intMultiset.upper_bound(22));

  std::pair<ims::const_iterator, ims::const_iterator> p;

  p = intMultiset.equal_range(22);
  cout << "\nFolosind equal_range pentru 22"
       << "\n  Lower bound: " << *(p.first)
       << "\n  Upper bound: " << *(p.second);

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

Rulând acest program, obținem următorul rezultat:

```
Sunt 0 valori de 15 in multiset
Dupa insert, sunt 2 valori de 15 in multiset
A fost gasita valoarea 15
Nu a fost gasita valoarea 20
Dupa insert intMultiset contine:
1 7 9 13 15 15 18 22 22 30 85 100
Vecinul lui 22 de pe pozitia din stanga : 22
Vecinul lui 22 de pe pozitia din dreapta : 30
Folosind equal_range pentru 22
  Lower bound: 22
  Upper bound: 30
```

Funcția `count` este comună tuturor containerilor asociativi şi întoarce numărul de apariții ale valorii care este parametru al acestei funcții:

```
intMultiset.count(15)
```

În acest exemplu, funcția `count` calculează numărul de apariții ale valorii `15` în containerul `intMultiset`.

Apelând funcția insert, se adaugă un nou element in container:

```
intMultiset.insert(15);
```

Localizarea unei valori în container se face prin funcția find. Aceasta returnează un `iterator` sa un `const_iterator` care pointează către primul element care are valoarea căutată:

```
result = intMultiset.find(15);
```

Dacă valoarea nu este găsită, atunci returnează un `iterator` sau un `const_iterator` egal cu valoarea returnată de apelul funcției `end`.

O altă variantă a funcției insert:

```
intMultiset.insert(a, a+SIZE);
```

este folosită pentru a adăuga la `intMultiset` valorile din tabloul `a`. Cei doi parametri sunt iteratori, dar pot fi folosiți şi pointeri.

Funcțiile `lower_bound` şi `upper_bound` care pot fi folosite pentru toți containerii asociativi determină poziția primei apariții a valorii dată ca parametru, respectiv poziția elementului de după ultima apariție a valorii parametru în cadrul containerului:

```
*(intMultiset.lower_bound(22))
*(intMultiset.upper_bound(22))
```

Ambele funcții returnează iteratori care pot fi dereferențiați pentru a extrage valoarea de la locația către care pointează.

Instrucțiunea

```
std::pair<ims::const_iterator, ims::const_iterator> p;
```

instanțiază obiectul `p` din clasa pair. Obiectele acestei clase sunt folosite pentru a asocia perechi de valori. În exemplul nostru, obiectul `p` conține doi iteratori de tip `const_iterator` pentru tipul `ims` care a fost definit ca un `multiset` cu elemente de tip `int`. Scopul folosirii obiectului `p` este cel de a păstra valoarea funcției `equal_range` care returnează o pereche care conține rezultatele operațiilor `lower_bound` şi `upper_bound`. Tipul de dată `pair` conține două date membre publice numite `first` şi `second`:

```
p = intMultiset.equal_range(22);
cout << "\nFolosind equal_range pentru 22"
     << "\n  Lower bound: " << *(p.first)
     << "\n  Upper bound: " << *(p.second);
```

Containerul asociativ `map`

Containerul asociativ `map` se foloseşte la păstrarea şi regăsirea asocierilor între chei şi valori. Cheile duplicate nu sunt acceptate de clasa `map`, astfel că unei chei îi poate fi asociată doar o singură valoare. Aceasta este o *asociere unu-la-unu*. Pentru o companie, de exemplu, care foloseşte numere unice de forma 100, 200, 300 pentru a păstra evidența angajaților, s-ar putea folosi clasa `map` pentru a asocia codurile numerice ale asociaților cu numere lor interioare de telefon care ar putea fi 4321, 4115, 5217 respectiv. Pentru a regăsi o valoare, se foloseşte operatorul `[]` în interiorul căruia se plasează cheia.

Multe dintre metodele folosite cu `multiset` şi `set` pot fi folosite şi cu `multimap` şi `map`. Programul de mai jos demonstrează modul în care poate fi folosit containerul `map` pentru care trebuie inclus fişierul header `<map>`.

Exemplu

test_map.cpp

```
#include<iostream>
using std::cout;
using std::endl;

#include <map>

int main()
{
  typedef std::map<int, double, std::less<int> > mid;
  mid pairs;

  pairs.insert( mid::value_type(15, 2.7) );
  pairs.insert( mid::value_type(30, 111.11) );
  pairs.insert( mid::value_type(5, 1010.1) );
  pairs.insert( mid::value_type(10, 22.22) );
  pairs.insert( mid::value_type(25, 33.333) );
  pairs.insert( mid::value_type(5, 77.54) );//cheie duplicata
  pairs.insert( mid::value_type(20, 9.345) );
  pairs.insert( mid::value_type(15, 99.3) );//duplicat ignorat
  cout << "pairs contine:\nCheie\tValoare\n";

  mid::const_iterator iter;
  for(iter = pairs.begin(); iter != pairs.end(); ++iter)
  cout << iter->first << '\t'
       << iter->second << '\n';

  pairs[25] = 9999.99;//inlocuieste valoarea pentru 25
  pairs[40]    = 8765.43;//insereaza o noua valoare pentru 40
  cout << "\nDupa operatiile cu operatorul [], pairs contine:"
       << "\nCheie\tValoare\n";

  for(iter = pairs.begin(); iter != pairs.end(); ++iter)
    cout << iter->first << '\t'
         << iter->second << '\n';

  return 0;
}
```

Rulând acest program, obținem următorul rezultat:

```
pairs contine:
Cheie     Valoare
5         1010.1
10        22.22
15        2.7
20        9.345
25        33.333
30        111.11
```

```
Dupa operatiile cu operatorul [], pairs contine:
Cheie     Valoare
5         1010.1
10        22.22
15        2.7
20        9.345
25        9999.99
30        111.11
40        8765.43
```

Instrucțiunea

```
typedef std::map<int, double, std::less<int> > mid;
```

foloseşte `typedef` pentru a defini un alias cu numele `mid` pentru tipul `map` în care cheile sunt de tip `int`, valorile asociate sunt de tip `double` şi elementele sunt ordonate crescător. Linia

```
mid pairs;
```

declară obiectul `pairs` de tip `mid`.

Funcția `insert` adaugă o nouă pereche cheie/valoare obiectului pairs:

```
pairs.insert( mid::value_type(15, 2.7) );
```

Expresia `mid::value_type(15, 2.7)` creează o pereche în care `15` este cheia, iar `2.7` este valoarea asociată ei.

Bucla

```
mid::const_iterator iter;
for(iter = pairs.begin(); iter != pairs.end(); ++iter)
cout << iter->first << '\t'
     << iter->second << '\n';
```

tipăreşte conținutul obiectului `pairs`. Se foloseşte un iterator de tip `const_iterator` pentru a accesa membrii lui `pairs` prin datele membre `first` şi `second`.

Operatorul `[]` se foloseşte pentru a accesa elemente ale unui `map` prin intermediul cheii:

```
pairs[25] = 9999.99;
pairs[40] = 8765.43;
```

Atunci când cheia există deja, ca în `pairs[25]`, operatorul întoarce o referință la valoarea asociată. Când indicele este o cheie care nu este în map, ca în `pairs[40]`, operatorul inserează cheia întorcând o referință care poate fi folosită pentru a asocia o valoare cu acea cheie. Prin cele două linii de mai sus, valoarea asociată cheii `25` se înlocuieşte cu `9999.99`, iar cheia `40` se inserează în `map` cu valoarea asociată `8765.43`.

- Adaptori container

STL dispune de trei *adaptori container*: `stack`, `queue` şi `priority_queue`. Adaptorii nu sunt containeri first-class pentru că nu reprezintă implementarea unei structuri de date în sine în care elementele pot fi stocate. În plus, adaptorii nu pot lucra cu iteratori. Programatorul poate alege o structură de date pe care să o ataşeze adaptorului. Toate clasele adaptor dispun de funcțiile membre `push` şi `pop` care implementează metodele de inserare şi de ştergere a unui element din structura de date a adaptorului.

Adaptorul `stack`

Clasa `stack` are funcționalități care permit inserarea şi ştergerea din structura de date ataşată la un singur capăt – *last-in-first-out*. Un obiect de tip `stack` poate fi implementat împreună cu orice container secvență: `vector`, `list` şi `deque`.

Exemplu

test_stack.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include <stack>
#include <vector>
#include <list>

template <class T>
void popElements(T &s);

int main()
{
  std::stack<int> intDequeStack; //stiva asociata cu deque
  std::stack<int, std::vector<int> > intVectorStack;
  std::stack<int, std::list<int> > intListStack;

  for(int i=0; i<10; ++i)
  {
    intDequeStack.push(i);
    intVectorStack.push(i);
    intListStack.push(i);
  }

  cout << "Pop din intDequeStack: ";
  popElements(intDequeStack);
  cout << "\nPop din intVectorStack: ";
  popElements(intVectorStack);
  cout << "\nPop din intListStack: ";
  popElements(intListStack);

  cout << endl;
  return 0;
}

template <class T>
void popElements(T &s)
{
  while(!s.empty())
  {
    cout << s.top() << ' ';
    s.pop();
  }
}
```

Rulând acest program, obținem următorul rezultat:

```
Pop din intDequeStack: 9 8 7 6 5 4 3 2 1 0
Pop din intVectorStack: 9 8 7 6 5 4 3 2 1 0
Pop din intListStack: 9 8 7 6 5 4 3 2 1 0
```

Prin instrucțiunile

```
std::stack<int> intDequeStack; //stiva asociata cu deque
std::stack<int, std::vector<int> > intVectorStack;
std::stack<int, std::list<int> > intListStack;
```

se instanțiază trei obiecte de tip `stack` cu elemente de tip `int`. Stiva `intDequeStack` foloseşte implicit containerul `deque`. Stiva `intVectorStack` foloseşte containerul `vector` de întregi, iar stiva `intListStack` foloseşte containerul `list` de întregi.

Funcțiile `push` şi `pop` se folosesc pentru a insera şi pentru a extrage valori din cele trei stive. Funcția `pop` nu returnează nicio valoare. Funcția `top` returnează primul element din stivă, fără a îl şterge.

Adaptorii `queue` şi `priority_queue`

Clasa `queue` permite inserarea la sfârşitul structurii de date ataşate şi ştergerea de la începutul structurii – *first-in-first-out*. Un obiect `queue` poate fi implementat cu structurile de date STL `list` şi `deque`. În mod implicit, unui obiect `queue` îi este ataşat un `deque`. Operațiile uzuale pentru queue sunt `push` care inserează un element la sfârşitul cozii (implementată prin apelul funcției `push_back` a containerului) şi `pop` pentru ştergerea unui element din capătul cozii (implementată prin apelul funcției `pop_front` a containerului).

Clasa `priority_queue` permite păstrarea sortată a elementelor inserate şi poate fi implementată cu containerii `vector` şi `deque`. Implicit, elementele sunt ordonate cu comparatorul `less<T>`, dar programatorul poate modifica acest comparator.

Exemplu

test_queue.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include <queue>

int main()
{
  std::queue<double> values;
  std::priority_queue<double> priorities;

  values.push(3.2);
  values.push(9.8);
  values.push(5.4);

  cout << "Pop din values: ";
  while(!values.empty())
  {
```

```
    cout << values.front() << ' '; //nu sterge elementul
    values.pop();                  //sterge elementul
  }

  priorities.push(3.2);
  priorities.push(9.8);
  priorities.push(5.4);

  cout << "\nPop din priorities: ";
  while(!priorities.empty())
  {
    cout << priorities.top() << ' ';
    priorities.pop();
  }

  return 0;
}
```

Rulând acest program, obținem următorul rezultat:

```
Pop din values: 3.2 9.8 5.4
Pop din priorities: 9.8 5.4 3.2
```

Prin instrucțiunile

```
std::queue<double> values;
std::priority_queue<double> priorities;
```

se declară două cozi numite `values` şi `priorities`. Prima este de tip `queue`, iar cea de-a doua este de tip `priority_queue`.

Obiectele de tip `queue` folosesc funcția `front` pentru a citi valoare primului element din stivă, iar pentru cele de tip `priority_queue` se foloseşte funcția `top`.

- Algoritmi

Până la introducerea STL, bibliotecile de clase de containeri şi algoritmi ale diverşilor dezvoltatori erau incompatibile. Inițial, containerii conțineau algoritmii ca funcționalități ale claselor. Ulterior, STL a separat algoritmii de containeri făcând mult mai uşoară adăugarea unor algoritmi noi. Accesul la elementele contaierilor se face, în aceste condiții, prin iteratori. Marele avantaj al acestei abordări este că algoritmii nu mai depind de detaliile de implementare ale containerilor asupra cărora operează. Atâta timp cât satisfac cerințele algoritmilor, algoritmii STL pot lucra asupra oricăror tablouri scrise în stil C bazate pe pointeri, dar şi asupra containerilor sau asupra altor structuri de date definite de programatori.

STL dispune de o colecție de câteva zeci de algoritmi. Vom prezenta în acest capitol doar câțiva dintre ei.

Algoritmii `fill`, `fill_n`, `generate` şi `generate_n`

Programul prezentat în această secțiune demonstrează folosirea algoritmilor STL `fill`, `fill_n`, `generate` şi `generate_n`. Funcțiile `fill` şi `fill_n` setează fiecare element dintr-un domeniu de elemente ale unui container cu o valoare. Funcțiile `generate` şi `generate_n` folosesc o *funcție generator* pentru a crea valori pentru fiecare element dintr-un domeniu de elemente ale unui container. Funcția generator nu are niciun argument şi returnează o valoare care poate fi plasată într-un element al containerului.

Exemplu

test_fill_generate.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include <algorithm>
#include <vector>
#include <iterator>

char nextLetter();

int main()
{
  std::vector<char> chars(10);
  std::ostream_iterator<char> output(cout, " ");

  std::fill(chars.begin(), chars.end(), '5');
  cout << "Vectorul chars dupa fill cu valori 5:\n";
  std::copy(chars.begin(), chars.end(), output);

  std::fill_n(chars.begin(), 5, 'A');
  cout << "\nVectorul chars dupa fill_n "
       << "cu cinci elemente A:\n";
  std::copy(chars.begin(), chars.end(), output);

  std::generate(chars.begin(), chars.end(), nextLetter);
  cout << "\nVectorul chars dupa generate cu valorile A-J:\n";
  std::copy(chars.begin(), chars.end(), output);

  std::generate_n(chars.begin(), 5, nextLetter);
  cout << "\nVectorul chars dupa generate_n cu valorile K-O"
       << " pentru primele cinci elemente:\n";
  std::copy(chars.begin(), chars.end(), output);

  cout << endl;
  return 0;
}

char nextLetter()
{
  static char letter = 'A';
  return letter++;
}
```

Rulând acest program, obținem următorul rezultat:

```
Vectorul chars dupa fill cu valori 5:
5 5 5 5 5 5 5 5 5 5
Vectorul chars dupa fill_n cu cinci elemente A:
A A A A A 5 5 5 5 5
Vectorul chars dupa generate cu valorile A-J:
A B C D E F G H I J
```

```
Vectorul chars dupa generate_n cu valorile K-O pentru primele
cinci elemente:
K L M N O F G H I J
```

Funcția

```
std::fill(chars.begin(), chars.end(), '5');
```

plasează caracterul `'5'` în fiecare element al vectorului `chars`, de la `chars.begin()` până la `chars.end()` exclusiv.

Funcția

```
std::fill_n(chars.begin(), 5, 'A');
```

plasează caracterul `'A'` în primele cinci elemente ale vectorului chars. Al doilea argument specifică numărul de elemente care vor fi completate.

Funcția

```
std::generate(chars.begin(), chars.end(), nextLetter);
```

plasează rezultatul apelului funcției generator `nextLetter` în fiecare element din vectorul `chars`, conform primilor doi parametri care sunt iteratori.

Funcția

```
std::generate_n(chars.begin(), 5, nextLetter);
```

plasează rezultatul apelului funcției generator `nextLetter` în cinci elemente ale vectorului `chars`.

Algoritmi matematici

Programul de mai jos prezintă câțiva dintre algoritmii matematici din STL: `random_shuffle`, `count`, `count_if`, `min_element`, `max_element`, `accumulate`, `for_each` şi `transform`.

Exemplu

test_mathematical.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include <algorithm>
#include <numeric> //accumulate este definit aici
#include <vector>
#include <iterator>

bool greater9(int);
void outputSquare(int);
int calculateCube(int);

int main()
{
  const int SIZE = 10;
  int a1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  std::vector<int> v(a1, a1+SIZE);
  std::ostream_iterator<int> output(cout, " ");

  cout << "Vectorul v inainte de random_shuffle:\n";
  std::copy(v.begin(), v.end(), output);
  std::random_shuffle(v.begin(), v.end());
```

```
  cout << "\nVectorul v dupa random_shuffle:\n";
  std::copy(v.begin(), v.end(), output);

  int a2[] = {100, 2, 8, 1, 50, 3, 8, 8, 9, 10};
  std::vector<int> v2(a2, a2+SIZE);
  cout << "\n\nVectorul v2 contine: ";
  std::copy(v2.begin(), v2.end(), output);
  int result = std::count(v2.begin(), v2.end(), 8);
  std::cout << "\nNumarul de elemente cu valoarea 8: "
            << result;

  result = std::count_if(v2.begin(), v2.end(), greater9);
  std::cout << "\nNumarul de elemente mai mari decat 9: "
            << result;

  std::cout << "\n\nElementul cu valoarea minima din v2: "
            << *(std::min_element(v2.begin(), v2.end()));

  std::cout << "\n\nElementul cu valoarea maxima din v2: "
            << *(std::max_element(v2.begin(), v2.end()));

  std::cout << "\n\nSuma elementelor din vectorul v2: "
            << std::accumulate(v.begin(), v.end(), 0);

  std::cout << "\n\nPatratul fiecarul element din v2:\n";
  std::for_each(v.begin(), v.end(), outputSquare);

  std::vector<int> cubes(SIZE);
  std::transform(v.begin(), v.end(), cubes.begin(),
                 calculateCube);
  cout << "\n\nCubul fiecarul element din vectorul v este: \n";
  std::copy(cubes.begin(), cubes.end(), output);

  cout << endl;
  return 0;
}

bool greater9(int value)
  {return value > 9;}

void outputSquare(int value)
  {cout << value * value << ' ';}

int calculateCube(int value)
  {return value * value * value;}
```

Rulând acest program, obținem următorul rezultat:

```
Vectorul v inainte de random_shuffle:
1 2 3 4 5 6 7 8 9 10
Vectorul v dupa random_shuffle:
9 2 10 3 1 6 8 4 5 7
```

```
Vectorul v2 contine: 100 2 8 1 50 3 8 8 9 10
Numarul de elemente cu valoarea 8: 3
Numarul de elemente mai mari decat 9: 3

Elementul cu valoarea minima din v2: 1

Elementul cu valoarea maxima din v2: 100

Suma elementelor din vectorul v2: 55

Patratul fiecarul element din v2:
81 4 100 9 1 36 64 16 25 49

Cubul fiecarul element din vectorul v este:
729 8 1000 27 1 216 512 64 125 343
```

Instrucțiunea

```
std::random_shuffle(v.begin(), v.end());
```

reordonează aleator elementele din vectorul `v` începând cu `v.begin()` până la `v.end()` exclusiv. Funcția are ca argumente doi iteratori de tip *random-access*.

Instrucțiunea

```
int result = std::count(v2.begin(), v2.end(), 8);
```

foloseşte funcția `count` pentru a număra elementele cu valoarea `8` din vectorul `v2` cuprinse între `v2.begin()` şi `v2.end()`.

Instrucțiunea

```
result = std::count_if(v2.begin(), v2.end(), greater9);
```

foloseşte funcția `count_if` pentru a număra elementele din vectorul `v2` cuprinse între `v2.begin()` şi `v2.end()` pentru care funcția predicat `greater9` returnează valoarea `true`.

Funcția

```
std::min_element(v2.begin(), v2.end())
```

localizează cel mai mic element din vectorul `v2` cuprins între `v2.begin()` şi `v2.end()`. Funcția returnează un iterator localizat pe cel mai mic element. Pentru a afişa valoarea pe care este localizat iteratorul, acesta trebuie derefențiat. Funcția `max_element` se poate folosi asemănător funcției `min_element`.

Funcția

```
std::accumulate(v.begin(), v.end(), 0)
```

al cărei template este declarat în fişierul header `<numeric>` însumează valorile din vectorul `v` cuprinse între `v.begin()` şi `v.end()`.

Instrucțiunea

```
std::for_each(v.begin(), v.end(), outputSquare);
```

foloseşte funcția `for_each` pentru a aplica o funcție fiecărui element al vectorului `v` din domeniul cuprins între `v.begin()` şi `v.end()`.

Funcția

```
std::transform(v.begin(), v.end(), cubes.begin(),
                  calculateCube);
```

aplică o funcție tuturor elementelor din vectorul v, cuprinse între `v.begin()` şi `v.end()` şi le plasează în destinația specificată de cel de-al treilea parametru.