9. Standard Template Library (STL)

Obiective

- Înțelegerea noțiunii de container şi folosirea containerilor STL
- Folosirea algoritmilor STL în programe
- Înțelegerea modului în care algoritmii folosesc iteratori pentru a accesa elementele unui container STL

- Introducere

Pe lângă avantajele oferite de posilibatea de a menține şi înțelege mai uşor aplicațiile, programarea orientată pe obiecte îl oferă pe acela al reutilizării codului. C++ standard include o bibliotecă standard care cuprinde o colecție foarte cuprinzătoare de componente reutilizabile. Vom introduce în acest capitol *Standard Template Library (STL)* şi vom prezenta cele trei componente cheie ale acestei biblioteci: *containerii* (structuri de date sub formă de template-uri), *iteratorii* şi *algoritmii*. Deoarece STL cuprinde foarte multe clase, vom prezenta doar o parte dintre acestea însoțite de câteva exemple.

În anii ’70, componentele folosite în programe erau sub forma structurilor de control şi a funcțiilor. În anii ’80, au început să se folosească componente sub formă de clase dintr-o gamă largă de biblioteci dependente de platformă. În ultima parte a anilor ’90, odată cu apariția STL se introduce un nou nivel de folosire a componentelor prin clase independente de platformă, fiind de aşteptat ca în următorii ani numărul acestor clase să crească exponențial.

Structurile de date sunt colecții de date sau containeri organizate după diverse reguli. În programarea orientată pe obiecte, structurile de date sunt obiecte care conțin colecții de obiecte. Să presupunem ca implementăm o clasă `Array` care ar reprezenta un tablou de obiecte de tip `int`. Folosind template-uri, am putea extinde acest tip de dată la `Array<T>` astfel încât să putem implementa `Array<char>`, `Array<double>`, `Array<Employee>` sau, în general, tablouri de orice tip de dată. În mod similar putem proceda pentru implementarea structurilor de date de tip stivă sau pentru alte tipuri de structuri de date. STL este o bibliotecă de clase template care conține, între altele, şi implementări ale structurilor de date.

În C şi C++ obişnuim să accesăm elementele unui tablou folosind pointeri. În C++ STL, accesăm elementele containerilor prin obiecte iterator care arată ca pointerii, dar se comportă mai inteligent. Clasele iterator sunt proiectate să poată fi folosite generic pentru orice container. Containerii implementează operații primitive, iar algoritmii STL sunt implementați independent de containeri.

Containerii STL sunt unul dintre cele mai bune exemple de reutilizare a codului şi de folosire a claselor template. Implementările structurilor de date care fac legăturile între obiecte prin pointeri pot conduce la probleme serioare legate de gestiunea dinamică a memoriei, erori care nu pot fi detectate la compilare şi care sunt uneori foarte dificil de rezolvat. Unul dintre marile avantaje ale containerilor STL este că rezolvă aceste neajunsuri.

STL evită folosirea moştenirii şi a funcțiilor virtuale din considerente de performanță. De asemenea, STL evită folosirea operatorilor `new` şi `delete` în favoarea *alocatorilor* pentru a permite o mai mare varietate de metode de control al alocării şi dealocării memoriei.

Containeri

Containerii STL sunt de trei tipuri: *containeri secvență*, *containeri asociativi* şi *adaptori container*.

Containerii secvență

- `vector` – fişierul header `<vector>` – implementarea unui tablou dinamic, cu redimensionare automată la inserarea unui nou element sau la ştergerea unui element;
- `list` – fişierul header `<list>` - listă dublu înlănțuită cu inserare şi ştergere rapidă;
- `deque` – fişierul header `<deque>` - coadă în care elementele pot fi adăugate sau şterse din ambele capete; diferă de coada obişnuită prin faptul că acolo adăugarea şi ştergerea elementelor se face la un singur capăt.

Containerii asociativi

- `map` – fişierul header `<map>` – păstrează asocieri între perechi de valori şi chei, mapare unu-la-unu;
- `multimap` – fişierul header `<map>` – este similar cu `map`, dar acceptă duplicate, mapare unu-la-mai multe;
- `set` – fişierul header `<set>` – cheile sunt chiar valorile păstrate;
- `multiset` – fişierul header `<set>` – este similar cu `set`, dar acceptă duplicate.

Adaptori container

- `stack` – fişierul header `<stack>` – implementează o stivă – last-in-first-out (LIFO);
- `queue` – fişierul header `<queue>` – implementează o coadă – first-in-first-out (FIFO);
- `priority_queue` – fişierul header `<queue>` –coadă în care elementul cu prioritatea cea mai mare este întotdeauna primul element extras.

Conținutul tuturor fişierelor header face parte din `namespace std`. Containerii secvență şi cei asociativi sunt grupați generic sub denumirea *first-class containers*. Containerii STL au fost proiectați în aşa fel încât dispun de o serie de funcționalități comune. Iată câteva dintre acestea:

- constructor implicit;
- constructor de copiere;
- destructor;
- `empty` – funcție care returnează `true` dacă sunt elemente în container şi `false` în caz contrar;
- `operator=`;
- `operator<`;
- `operator<=`;
- `operator>`;
- `operator>=`;
- `operator==`;
- `operator!=`;
- `erase` – funcție care şterge unul sau mai multe elemente din container (doar în *first-class containers*);

- `clear` – funcție care şterge toate elementele din container (doar în *first-class containers*).

Atunci când decidem să folosim un container, este important să ne asigurăm că tipul de dată elementelor care urmează a fi stocate în container dispune de un set minim de funcționalități. La inserarea unui element într-un container se face o copie a acestuia, de aceea tipul de dată al elementului trebuie să dispună de un constructor de copiere şi de un operator de asignare.

Iteratori

Iteratorii au multe caracteristici comune cu pointerii şi sunt folosiți, printre altele, pentru a pointa la elemente ale containerilor de tip first-class. Un iterator este un obiect care permite programatorului să parcurgă toate elementele unei colecții, indiferent de implementarea acesteia. Există iteratori STL a căror implementare este adaptată unor anumiți containeri. Sunt, însă, iteratori care pot opera asupra tuturor containerilor. De exemplu, operatorul de dereferențiere `*` se poate aplica unui iterator pentru a folosi elementul asupra căruia pointează. Operatorul `++` aplicat unui iterator mută poziția iteratorului asupra următorului element din container.

Folosim iteratorii împreună cu secvențele. Secvențele pot fi organizate în containeri sau pot fi secvențe de intrare ori ieşire. Programul de mai jos demonstrează citirea datelor folosind `istream_iterator` de la intrarea standard care poate fi privită ca o secvență de date de intrare în program. Programul ilustrează, de asemenea, tipărirea datelor folosind `ostream_iterator` la ieşirea standard care poate fi privită ca o secvență de date de ieşire din program. Programul citeşte de la tastatură doi întregi şi afişează suma lor.

Exemplu

test_iostream_iterators.cpp

```
#include <iostream>

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

#include <iterator>

int main()
{
  cout << "Introduceti doua numere intregi: ";
  std::istream_iterator<int> inputInt(cin);
  int number1, number2;
  number1 = *inputInt; //citeste primul int
  ++inputInt;          //muta iteratorul pe urmatoarea valoare
  number2 = *inputInt; //citeste urmatorul int

  cout << "Suma este: ";
  std::ostream_iterator<int> outputInt(cout);
  *outputInt = number1 + number2; //tiparire la cout
  cout << endl;

  return 0;
```

```
}
```

Rulând acest program, putem obține următorul rezultat:

```
Introduceti doua numere intregi: 12 25
Suma este: 37
```

Prin instrucțiunea

```
std::istream_iterator<int> inputInt(cin);
```

se creează un iterator de tip `istream_iterator` care este capabil să extragă valori de tip `int` de la dispozitivul standard de intrare `cin` într-o manieră sigură.

Prin dereferențierea iteratorului `inputInt` se citeşte prima valoare întreagă din `cin`:

```
number1 = *inputInt; //citeste primul int
```

Operația `++inputInt` mută iteratorul pe următoarea valoare din secvența de date de intrare.

Prin declarația

```
std::ostream_iterator<int> outputInt(cout);
```

se creează un iterator de tip `ostream_iterator` prin care se inserează valori de tip `int` la ieşirea standard `cout`.

Asignarea

```
*outputInt = number1 + number2;
```

permite tipărirea sumei celor doi întregi dereferențiind iteratorul `outputInt`.

Atunci când containerul conține elemente ale căror valori pot fi modificate putem folosi şi iteratorul `iterator`, iar dacă elementele nu pot fi modificate folosim `const_iterator`.

Categoriile de iteratori folosiți în STL sunt următoarele:

- *input* – pentru citirea unui element dintr-un container;
- *output* – pentru scrierea unui element într-un container;
- *forward* – combină capacitățile primelor două categorii de iteratori;
- *bidirectional* - combină capacitățile unui iterator forward cu posibilitatea de a se deplasa în ambele direcții;
- *random-access* - combină capacitățile iteratorului bidirectional cu posibilitatea de a accesa direct un element din container, adică de a se deplasa inainte sau înapoi cu un număr arbitrar de elemente.

Operațiile care pot fi folosite de un iterator sunt următoarele (operațiile pentru un iterator le includ pe toate cele precedente):

Toți iteratorii

- `++p` – preincrementarea unui iterator
- `p++` – postincrementarea unui iterator

Iteratorii de tip input

- `*p` – dereferențierea unui iterator pentru a fi folosit ca *rvalue*
- `p = p1` – asignarea unui iterator altui iterator
- `p == p1` – compararea iteratorilor pentru egalitate
- `p != p1` – compararea iteratorilor pentru inegalitate

Iteratorii de tip output

- `*p` – dereferențierea unui iterator pentru a fi folosit ca *lvalue*

Iteratorii de tip forward

- `--p` – predecrementarea unui iterator
- `p--` – postdecrementarea unui iterator

Iteratorii de tip random-access

- `p += i` – incrementarea iteratorului `p` cu `i` poziții
- `p -= i` – decrementarea iteratorului `p` cu `i` poziții
- `p + i` – iteratorul este poziționat la `p` incrementat cu `i` poziții
- `p - i` – iteratorul este poziționat la `p` decrementat cu `i` poziții
- `p[i]` – returnează o referință la elementul deplasat de la `p` cu `i` poziții
- `p < p1` – returnează `true` dacă iteratorul `p` este înaintea lui `p1` în container
- `p <= p1` – returnează `true` dacă iteratorul `p` este înaintea lui `p1` sau pe aceeaşi poziție în container
- `p > p1` – returnează `true` dacă iteratorul `p` este după `p1` în container
- `p >= p1` – returnează `true` dacă iteratorul `p` este după `p1` sau pe aceeaşi poziție în container.

Algoritmi

Un aspect esențial al STL este că oferă algoritmi care pot fi folosiți generic pentru o mare varietate de containeri: inserare, ştergere, căutare, sortare etc. STL include aproximativ 70 de algoritmi standard. Algoritmii operează asupra elementelor unei secvențe doar indirect, prin intermediul iteratorilor. Deoarece STL este extensibil, se pot adăuga cu uşurință noi algoritmi fără a opera nicio modificare asupra containerilor. Algoritmii pot opera şi asupra tablourilor declarate în formatul promovat de limbajul C ca pointeri.

Algoritmii STL sunt de două tipuri. Cei de tip *mutating-sequence* fac modificări asupra containerilor pe care sunt aplicati, în timp ce algoritmii *non-mutating-sequence* se execută fără a schimba conținutul containerilor.

Câțiva dintre algoritmii care fac parte din fişierul header `<numeric>` sunt:

- `copy();`
- `fill();`
- `replace();`
- `swap();`
- `count();`
- `find();`
- `search().`

- Containeri secvență

C++ STL oferă trei containeri secvență: `vector`, `list` şi `deque`. Clasele `vector` şi `deque` se bazează pe tablouri. Clasa `list` implementează o structură de dată de tip listă înlănțuită.

Unul dintre cei mai folosiți containeri este `vector` care este o implementare a unui tablou. Spere deosebire de tablourile simple din C++, acestea pot fi asignate unui altuia, şi, suplimentar, clasa `vector` verifică încadrarea indicilor în limite.

Pe lângă operațiile comune tuturor containerilor, unii dintre aceştia implementează şi operații suplimentare. Containerii secvență dispun de operațiile:

- `front` – returnarea unei referințe la primul element din container;
- `back` – returnarea unei referințe la ultimul element din container;
- `push_back` – inserarea unui element nou pe ultima poziție din container;
- `pop_back` – ştergerea ultimului element din container.

Vom prezenta în programele de mai jos câteva dintre funcționalitățile claselor `vector` şi `list`.

Containerul secvență `vector`

Un vector (tablou) este o structură de dată în care datele sunt stocate într-o zonă contiguă de memorie. Această organizare a datelor permite accesul direct la elementele vectorului prin operatorul `[]`. Atunci când memoria obiectului de tip `vector` nu mai este suficientă, se alocă o zonă mai mare de memorie contiguă, se copiază elementele originale în noua zonă de memorie şi se dealocă vechea zonă de memorie.

O caracteristică importantă a oricărui container este tipul de iterator pe care îl suportă. Acesta determină ce algoritmi pot fi aplicați containerului. Un `vector` suportă iteratorul *random-access*, adică poate fi folosit împreună cu orice operator prezentat mai devreme. Toți algoritmii STL pot opera asupra unui vector. Iteratorii pentru un `vector` sunt implementați ca pointeri la elementele vectorului. Dacă un algoritm are nevoie de un anumit tip de iterator pentru a funcționa, atunci containerul asupra căruia se aplică trebuie să poată lucra cel puțin cu acel tip de iterator.

Exemplul următor prezintă câteva dintre funcționalitățile clasei template `vector`, multe dintre aceste funcții fiind însă utilizabile de orice container STL de tip first-class.

Exemplu

test_vector.cpp

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

#include <vector>

int main()
{
  const int SIZE = 6;
  int a[SIZE] = {1, 2, 3, 4, 5, 6};
  std::vector<int> v;
  cout << "Dimensiunea initiala a lui v este: " << v.size()
       << "\nCapacitatea initiala a lui v este: "
       << v.capacity();
  v.push_back(2);  //metoda push_back se gaseste
  v.push_back(3);  //in orice colectie secventa
  v.push_back(4);
  cout << "\nDimensiunea lui v este: " << v.size()
       << "\nCapacitatea lui v este: " << v.capacity();
  cout << "\n\nContinutul tabloului folosind notatia pointer: ";

  for(int *ptr = a; ptr != a + SIZE; ++ptr)
    cout << *ptr << ' ';

  cout << "\nContinutul vectorului v folosind iteratorul: ";
  std::vector<int>::const_iterator p1;
  for(p1 = v.begin(); p1 != v.end(); p1++)
    cout << *p1 << ' ';

  cout << "\nContinutul inversat al vectorului v: ";
```

```
  std::vector<int>::reverse_iterator p2;
  for(p2 = v.rbegin(); p2 != v.rend(); ++p2)
    cout << *p2 << ' ';

  cout << endl;

  return 0;
}
```

Rulând acest program, obținem următorul rezultat:

```
Dimensiunea initiala a lui v este: 0
Capacitatea initiala a lui v este: 0
Dimensiunea lui v este: 3
Capacitatea lui v este: 4

Continutul tabloului folosind notatia pointer: 1 2 3 4 5 6
Continutul vectorului v folosind iteratorul: 2 3 4
Continutul inversat al vectorului v: 4 3 2
```

Prin instrucțiunea

```
std::vector<int> v;
```

se declară obiectul `v` din clasa `vector` care poate păstra valori `int`. Odată cu instanțierea acestui obiect, se creează un vector vid de dimensiune 0, adică numărul de elemente stocate în acel moment este 0, şi capacitate 0, adică numărul de elemente fără alocare de memorie este 0. Apelurile de funcții `v.size()` şi `v.capacity()` returnează aceste două valori.

Funcția `push_back` inserează o valoare în container. Dacă se inserează o valoare într-un vector plin, dimensiunea acestuia creşte automat. Un nou apel al funcțiilor `v.size()` şi `v.capacity()` ilustrează această modificare a dimensiunii şi a capacității vectorului.

Pentru afişarea conținutului obiectului `v` în ordine, se instanțiază obiectul `p1` de tip `const_iterator`. Funcția `v.begin()` returnează un `const_iterator` la primul element din `v`, `v.end()` returnează un `const_iterator` care indică poziția de după ultimul element din `v`, iar operația `p1++` poziționează iteratorul pe următorul element din `v`. Afişarea valorii de la poziția curentă a iteratorului se face prin dereferențierea acestuia: `*p1`.

Afişarea valorilor în ordine inversă se face cu ajutorului obiectului `p2` de tip `reverse_iterator`. Funcția `v.rbegin()` returnează un iterator la punctul de start pentru iterarea în ordine inversă , iar `v.end()` returnează un iterator la punctul de terminare a iterării în ordine inversă.

Următorul program prezintă funcții care permit diverse manipulări ale valorilor unui obiect de tip `vector`.

Exemplu

test_vector_manipulation.cpp

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

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

```
#include <algorithm>
#include <exception>

int main()
{
  const int SIZE = 6;
  int a[SIZE] = {1, 2, 3, 4, 5, 6};
  std::vector<int> v(a, a+SIZE);
  std::ostream_iterator<int> output(cout, " ");
  cout << "Vectorul v contine: ";
  std::copy(v.begin(), v.end(), output);

  cout << "\nPrimul element din v: " << v.front()
     << "\nUltimul element din v: " << v.back();

  v[0] = 7;
  v.at(2) = 10;
  v.insert(v.begin()+1, 22);
  cout << "\nContinutul vectorului v dupa modificari: ";
  std::copy(v.begin(), v.end(), output);

  try
  {
    v.at(100) = 725;
  }
  catch(std::exception& e)
  {
    cout << "\nExceptie: " << e.what();
  }

  v.erase(v.begin());
  cout << "\nContinutul vectorului v dupa stergere: ";
  std::copy(v.begin(), v.end(), output);
  v.erase(v.begin(), v.end());
  cout << "\nDupa stergere, vectorul v "
       << (v.empty() ? "" : "nu ") << "este vid";

  v.insert(v.begin(), a, a+SIZE);
  cout << "\nContinutul vectorului v inainte de stergere: ";
  std::copy(v.begin(), v.end(), output);
  v.clear();
  cout << "\nDupa stergere, vectorul v "
       << (v.empty() ? "" : "nu ") << "este vid";

  cout << endl;

  return 0;
}
```

Rulând acest program, obținem următorul rezultat:

```
Vectorul v contine: 1 2 3 4 5 6
Primul element din v: 1
```

```
Ultimul element din v: 6
Continutul vectorului v dupa modificari: 7 22 2 10 4 5 6
Exceptie: vector::_M_range_check
Continutul vectorului v dupa stergere: 22 2 10 4 5 6
Dupa stergere, vectorul v este vid
Continutul vectorului v inainte de stergere: 1 2 3 4 5 6
Dupa stergere, vectorul v este vid
```

Prin instrucțiunea

```
std::vector<int> v(a, a+SIZE);
```

se declară un vector `v` cu elemente de tip `int`. Se apelează un constructor supraîncărcat al acestei clase care primeşte două argumente. Pointerii la un tablou pot fi folosiți ca iteratori, astfel că prin aceasta declarație vectorul v este inițializat cu conținutul tabloului `a`, de la locația a până la `a+SIZE`, fără să o includă şi pe aceasta.

Prin

```
std::ostream_iterator<int> output(cout, " ");
```

se defineşte un `ostream_iterator` cu numele `output` care poate fi folosit pentru afişarea la `cout` a unor valori întregi separate prin spațiu.

Algoritmul `copy` din STL este invocat prin apelul

```
std::copy(v.begin(), v.end(), output);
```

pentru a trimite către output conținutul vectorul `v` dintre `v.begin()` şi `v.end()`. Primii doi parametri sunt iteratori ale căror valori sunt citite dintr-un container, iar ultimul parametru este un iterator de ieşire. Pentru a putea apela această funcție, trebuie inclus fişierul header `<algorithm>` în codul sursă.

Liniile de program

```
v[0] = 7;
v.at(2) = 10;
```

ilustrează două moduri în care se pot accesa elementele unui obiect de tip `vector`. Spre deosebire de operatorul `[]`, funcția at verifică dacă indicele de tablou care este trimis ca parametru se încadrează între limitele acestuia. În caz contrar, este generată o excepție care poate fi tratată astfel:

```
try
{
  v.at(100) = 725;
}
catch(std::exception& e)
{
  cout << "\nExceptie: " << e.what();
}
```

Pentru a putea rula corect codul de tratare a excepției, programul trebuie să includă fişierul header `<exception>`.

Funcția

```
v.insert(v.begin()+1, 22);
```

este o variantă de `insert` care poate fi folosită pentru containerii secvență şi care inserează valoarea 22 înainte de elementul a cărui poziție este specificată prin iteratorul din primul argument. În exemplul nostru, iteratorul pointează către al doilea element din vectorul `v`, iar valoarea 22 este inserată pe a doua poziție. Elementul care se găsea înainte pe cea de-a doua poziție devine astfel al treilea element din vector, iar toate celelalte elemente sunt deplasate cu o poziție. Instrucțiunea

```
v.insert(v.begin(), a, a+SIZE);
```

este o altă variantă a acestei funcții care inserează o mulțime de valori dintr-o secvență pentru care se precizează poziția de început şi cea de sfârşit dintr-un alt container. În exemplul nostru, se inserează elementele dintre a şi `a+SIZE` din tabloul `a`.

Funcțiile

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

indică faptul că elementul de la locația specificată sau elementele dintre cele două locații sunt şterse din colecție. Funcția

```
v.clear();
```

şterge întreg conținutul vectorului `v`.

Containerul secvență `list`

Lista înlănțuită este o structură de dată în care elementele sunt organizate sub forma unei colecții liniare de obiecte numite *noduri* care sunt interconectate prin *legături* de tip pointer. O listă înlănțuită este accesată printr-un pointer la primul nod al listei. Nodurile următoare sunt accesate prin legătura pe care fiecare nod o are către nodul următor din listă. Prin convenție, ultimul nod al listei înlănțuite nu are un element următor, iar pointerul său de legătură are valoarea `NULL`. Într-o *listă dublu înlănțuită*, fiecare nod are câte două legături, una către elementul următor şi alta către elementul precedent.

Containerul secvență `list` este o implementare a listei dublu înlănțuite. Acest container suportă, astfel, iteratori bidirecționali care permit parcurgerea sa în ambele sensuri. Orice algoritm care are nevoie de iteratori de tip *input*, *output*, *forward* sau *bidirectional* pot opera asupra unei liste.

Pe lângă funcțiile membre specifice tuturor containerilor şi cele ale containerilor secvență, clasa list dispune şi de următoarele opt funcții membre: `splice`, `push_front`, `pop_front`, `remove`, `unique`, `merge`, `reverse` şi `sort`.

Programul de mai jos prezintă câteva dintre funcționalitățile clasei `list`. Pentru a folosi această clasă, programul trebuie să includă fişierul header `<list>`.

Exemplu

test_list.cpp

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

#include <list>
#include <algorithm>
#include <iterator>

template <class T>
void printList(const std::list<T> &listRef);

int main()
{
  const int SIZE = 4;
  int a[SIZE] = {2, 4, 6, 8};
  std::list<int> values, otherValues;

  values.push_front(1);
```

```
values.push_front(2);
values.push_back(4);
values.push_back(3);

cout << "lista values contine: ";
printList(values);
values.sort();
cout << "\ndupa sort lista values contine: ";
printList(values);

otherValues.insert(otherValues.begin(), a, a+SIZE);
cout << "\ndupa insert lista otherValues contine: ";
printList(otherValues);
values.splice(values.end(), otherValues);
cout << "\ndupa splice lista values contine: ";
printList(values);

values.sort();
cout << "\ndupa sort lista values contine: ";
printList(values);
otherValues.insert(otherValues.begin(), a, a+SIZE);
otherValues.sort();
cout << "\nlista otherValues contine: ";
printList(otherValues);
values.merge(otherValues);
cout << "\ndupa merge:\n  lista values contine: ";
printList(values);
cout << "\n  lista otherValues contine: ";
printList(otherValues);

values.pop_front();
values.pop_back();
cout << "\ndupa pop_front si pop_back lista values contine: ";
printList(values);

values.unique();
cout << "\ndupa unique, lista values contine: ";
printList(values);

//metoda swap poate fi folosita de toti containerii
values.swap(otherValues);
cout << "\ndupa swap:\n  lista values contine: ";
printList(values);
cout << "\n  lista otherValues contine: ";
printList(otherValues);

values.assign(otherValues.begin(), otherValues.end());
cout << "\ndupa assign lista otherValues contine: ";
printList(values);

values.merge(otherValues);
```

```
  cout << "\ndupa merge lista values contine: ";
  printList(values);

  values.remove(4);
  cout << "\ndupa remove lista values contine: ";
  printList(values);

  cout << endl;

  return 0;
}

template <class T>
void printList(const std::list<T> &listRef)
{
  if(listRef.empty())
    cout << "Lista este goala";
  else
  {
    std::ostream_iterator<T> output(cout, " ");
    std::copy(listRef.begin(), listRef.end(), output);
  }
}
```

Rulând acest program, obținem următorul rezultat:

```
lista values contine: 2 1 4 3
dupa sort lista values contine: 1 2 3 4
dupa insert lista otherValues contine: 2 4 6 8
dupa splice lista values contine: 1 2 3 4 2 4 6 8
dupa sort lista values contine: 1 2 2 3 4 4 6 8
lista otherValues contine: 2 4 6 8
dupa merge:
  lista values contine: 1 2 2 2 3 4 4 4 6 6 8 8
  lista otherValues contine: Lista este goala
dupa pop_front si pop_back lista values contine: 2 2 2 3 4 4 4
6 6 8
dupa unique, lista values contine: 2 3 4 6 8
dupa swap:
  lista values contine: Lista este goala
  lista otherValues contine: 2 3 4 6 8 2 3 4 6 8
dupa assign lista otherValues contine: 2 3 4 6 8
dupa merge lista values contine: 2 2 3 3 4 4 6 6 8 8
dupa remove lista values contine: 2 2 3 3 6 6 8 8
```

În acest program se instanțiază două liste cu elemente de tip `int`, denumite `values` şi `otherValues`:

```
std::list<int> values, otherValues;
```

Instrucțiunea `push_front` inserează o valoare la începutul listei, iar instrucțiunea `push_back` adaugă valori la sfârşitul său:

```
values.push_front(1);
values.push_front(2);
values.push_back(4);
```

```
values.push_back(3);
```

După aceste patru operații de inserare, lista `values` conține patru valori ordonate astfel: `2 1 4 3`.

Funcția membră `sort` fără niciun argument realizează o ordonare crescătoare a valorilor din listă:

```
values.sort();
```

După apelul acestei funcții, elementele din listă se vor regăsi în ordinea `1 2 3 4`.

Funcția

```
values.splice(values.end(), otherValues);
```

şterge elementele din `otherValues` şi le inserează în values înainte de poziția specificată de iteratorul care este primul argument său. În exemplul nostru, iteratorul este `values.end()`, iar valorile din `otherValues` sunt adăugate la sfârşitul listei `values`. După acestă operație, lista values va conține valorile `1 2 3 4 2 4 6 8`.

Funcția

```
values.merge(otherValues);
```

şterge toate elementele din `otherValues` şi le inserează sortate în lista `values`. Ambele liste trebuie sortate în aceeaşi ordine înainte de a realiza această operație. În exemplul nostru, inaintea operației `merge` lista `values` conține valorile `1 2 2 3 4 4 6 8`, iar lista `otherValues` conține valorile `2 4 6 8`. După apelul funcției, lista `values` va conține valorile `1 2 2 2 3 4 4 4 6 6 8 8`, iar `otherValues` va fi vidă.

Funcția `pop_front` şterge primul element din listă, iar `pop_back` şterge ultimul element din listă.

Instrucțiunea

```
values.unique();
```

şterge elementele duplicate din listă. Pentru a garanta ştergerea tuturor duplicatelor din listă, aceasta trebuie sortată în prealabil pentru ca valorile care se repetă să se găsească pe poziții alăturate.

Linia

```
values.swap(otherValues);
```

foloseşte funcția `swap` pentru a interschimba conținutul listei `values` cu conținutul listei `otherValues`.

Instrucțiunea

```
values.assign(otherValues.begin(), otherValues.end());
```

foloseşte funcția `assign` pentru a înlocui conținutul vectorului `values` cu conținutul din `otherValues` din domeniul specificat de cei doi iteratori care sunt argumente ale funcției.

Prin apelul funcției `remove` cu argumentul 4:

```
values.remove(4);
```

sunt şterse toate aparițiile lui 4 din lista `values`.