Curs 12

- **STL – Iteratori**
- **STL - Algoritmi**
- **Șabloane de proiectare**

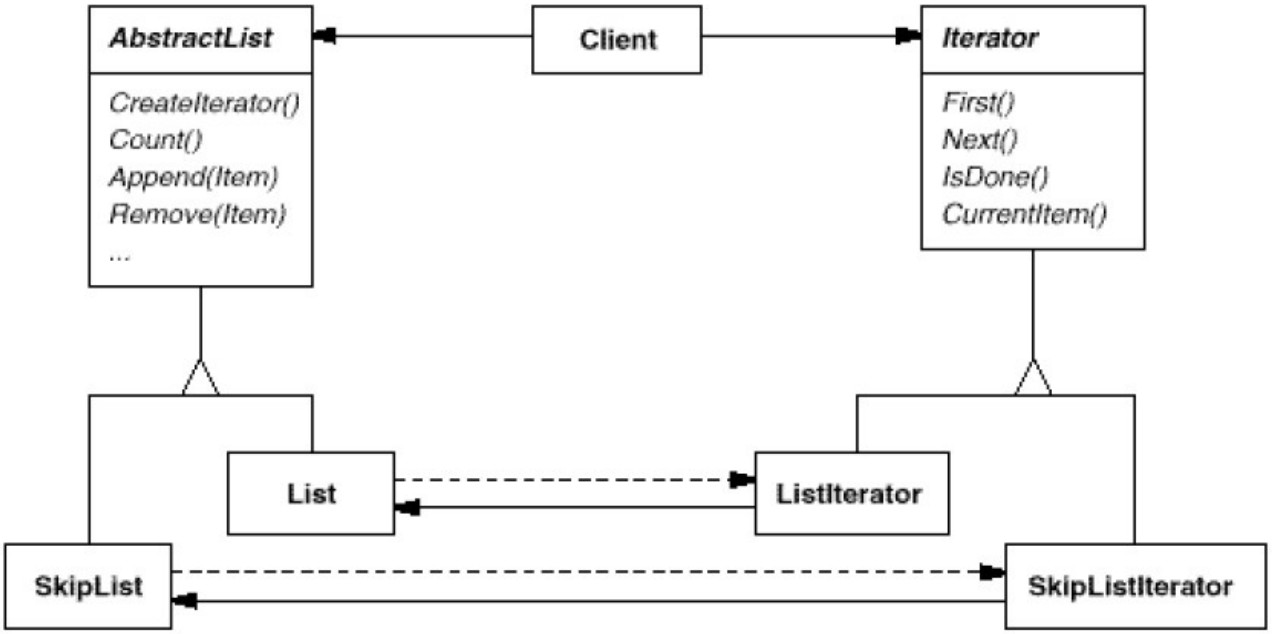
Șablonul Iterator (Cursor)

Intenție: Oferă acces secvențial la elementele unui agregat fără a expune reprezentarea internă.

Motivație:

- Un agregate (ex. listă) ar trebui sa permita accessul la elemente fără a expune reprezentarea internă
- Ar trebui sa permita traversarea elementelor în moduri diferite (inainte, înapoi, random)

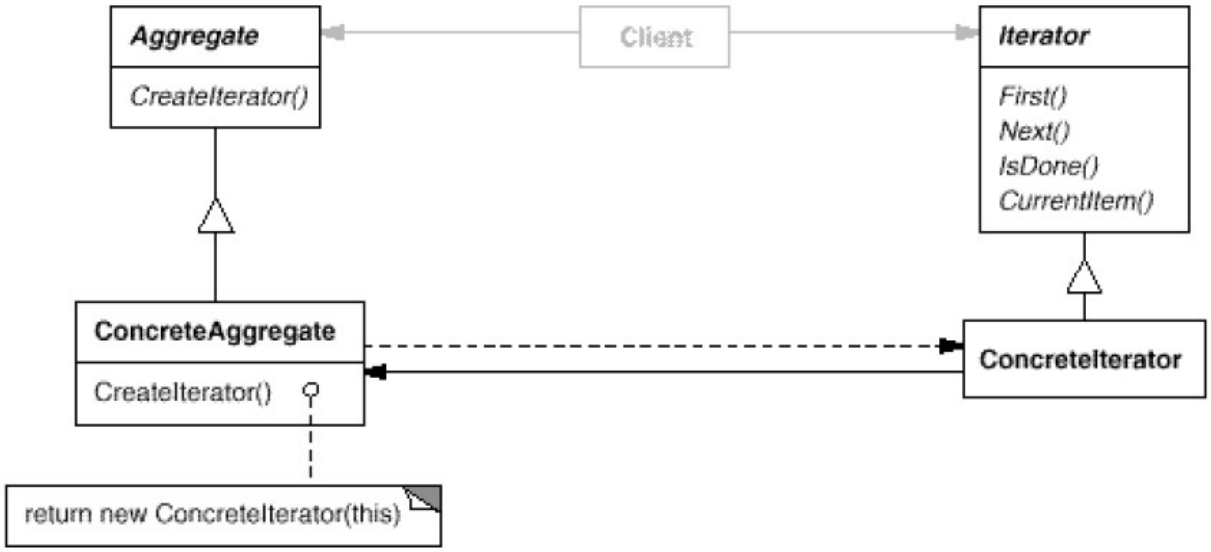
Exemplu: List, SkipList, Iterator



Aplicabilitate:

- diferite tipuri de traversari pentru un agregat
- oferă o interfață uniformă pentru accesul la elementele unui agregat

Șablonul Iterator - structură



Participanți:

Iterator: definește interfața pentru a traversa elementele

ConcreteIterator: Implementează interfață Iterator, responsabil cu gestiunea poziției curente din iterație

Aggregate: definește metode pentru a crea un obiect de tip iterator

ConcreteAggregate: Implementează interfața necesară pentru creare de Iterator crează ConcreteIterator

Consecințe:

- suportă multiple tipuri de traversari. Agregate mai complexe au nevoie de diferite metode prin care se accesează elementele
- se simplifică interfața Aggregate.
- Putem avea mai multe traversari în același moment.

Iteratori in STL

Iterator: obiect care gestionează o poziție (curentă) din containerul asociat. Oferă suport pentru traversare (++,--), dereferențiere (*it).

Iteratorul este un concept fundamental in STL, este elementul central pentru algoritmi oferiţi de STL.

Fiecare container STL include funcţii membre **begin()** and **end()**

```
void sampleIterator() {
	vector<int> v;
	v.push_back(4);
	v.push_back(8);
	v.push_back(12);
	//Obtain an the start of the iteration
	vector<int>::iterator it = v.begin();
	while (it != v.end()) {
		//dereference
		cout << (*it) << " ";
		//go to the next element
		it++;
	}
	cout << endl;
}
```

Permite decuplarea intre algoritmi si containere

Existe mai multe tipuri de iteratori:

- iterator input/output (istream_iterator, ostream_iterator)
- forward iterators, iterator bidirectional, iterator random access
- reverse iterators

Tipuri de Iterator in STL

Iterator Type	Behavioral Description	Operations Supported
random access (most powerful)	Store and retrieve values Move forward and backward Access values randomly	`* = ++ -> == != --` `+ - [] < > <= >= += -=`
bidirectional	Store and retrieve values Move forward and backward	`* = ++ -> == != --`
forward	Store and retrieve values Move forward only	`* = ++ -> == !=`
input	Retrieve but not store values Move forward only	`* = ++ -> == !=`
output (least powerful)	Store but not retrieve values Move forward only	`* = ++`

Iteratori oferiți de containere STL

Container Class	Iterator Type	Container Category
vector	random access	sequential
deque	random access	
list	bidirectional	
set	bidirectional	associative
multiset	bidirectional	
map	bidirectional	
multimap	bidirectional	

- Adaptoarele de containere (container adaptors) - stack, queue, priority_queue - nu oferă iterator

Algoritmi STL

- O colecție de funcții template care pot fi folosite cu iteratori. Funcțiile operează pe un domeniu (range) definit folosind iteratori.

- Un domeniu (range) este o secvente de obiecte care pot fi accesate folosind iteratori sau pointeri

header files: <algorithm> or <numeric>.

- <numeric> contine algoritmi (funcții) ce opereaza in general cu numere (calculează ceva)
- <algorithm> conține algoritmi de uz general.
- Dacă aveți erori de compilare pentru ca nu se găsește o funcție STL, includeți ambele headere.

Funcții (algoritmi) STL:

- Operații pe secvențe
  - care nu modifică sursa: **accumulate**, **count**, **find**, **count_if**, etc
  - care modifică : **copy**, **transform**, **swap**, **reverse**, **random_shuffle**, etc
- Sortări: **sort**, **stable_stort**, etc
- Pe secvențe de obiecte ordonate
  - Căutare binară : **binary_search**, etc
  - Interclasare (Merge): **merge, set_union, set_intersect,** etc
- Min/max: **min**, **max**, **min_element**, etc
- Heap: **make_heap, sort_heap**, etc

STL - accumulate

```
    vector<int> v;
    v.push_back(3);
    v.push_back(4);
    v.push_back(2);
    v.push_back(7);
    v.push_back(17);
     //compute the sum of all elements in the vector
    cout << accumulate(v.begin(), v.end(), 0) << endl;

    //compute the sum of elements from 1 inclusive, 4 exclusive [1,4)
    vector<int>::iterator start = v.begin()+1;
    vector<int>::iterator end = v.begin()+4;
    cout << accumulate(start, end, 0) << endl;
```

- accumulate : calculează suma elementelor

```
  /**
   *  @brief  Accumulate values in a range.
   *
   *  Accumulates the values in the range [first,last) using operator+().
The
   *  initial value is @a init.  The values are processed in order.
   *
   *  @param  first  Start of range.
   *  @param  last  End of range.
   *  @param  init  Starting value to add other values to.
   *  @return  The final sum.
   */
  template<typename _InputIterator, typename _Tp>
    accumulate(_InputIterator __first, _InputIterator __last, _Tp __init)
```

STL – copy

Copiează elemente

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
OutputIterator result)
{
  while (first!=last) *result++ = *first++;
  return result;
}
```

Parametrii:

- first, last – definește secvența care se copiază (Input iterator) - [first,last)
- Result – Iterator de tip Output, poziționat pe primul element în destinație. (nu poate referi element din secvența sursă)

```
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(2);
v.push_back(7);
v.push_back(17);

//make sure there are enough space in the destination
//allocate space for 5 elements
vector<int> v2(5);

//copy all from v to v2
copy(v.begin(), v.end(), v2.begin());
```

STL – sort

- Sorteaza elementele din intervalul [first,last) ordine crescătoare.
- Elementele se compară folosind operatorul **operator <**

Parametrii:

- first, last : secvența de elemente ce dorim să ordonăm [first,last)
- Necesită iterator Random-Access
- Comp: funcție de comparare, relația folosită la ordonare

```
	vector<int> v;
	v.push_back(3);
	v.push_back(4);
	v.push_back(2);
	v.push_back(7);
	v.push_back(17);

	//sort
	sort(v.begin(), v.end());
```

Folosind o funcție de comparare:

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

```
bool desc(int i, int j) {
	return (i > j);
}
```

```
void testSortCompare() {
	vector<int> v;
	v.push_back(3);
	v.push_back(4);
	v.push_back(2);
	v.push_back(7);
	v.push_back(17);

	//sort
	sort(v.begin(), v.end(), asc);

	//print elements
	for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
		cout << *it << "  ";
	}
	cout << endl;
}
```

STL – for_each, transform

Aplică o funcție pe fiecare element din secvență [first,last).

```
void testForEach() {
	vector<int> v;
	v.push_back(3);
	v.push_back(4);
	v.push_back(2);
	v.push_back(7);
	v.push_back(17);

	for_each(v.begin(), v.end(), print);
	cout << endl;
}
```

```
void print(int elem) {
	cout << elem << " ";
}
```

Transform: aplică funcția pentru fiecare element din secvență ([first1,last1)) rezultatul se depune în secvența rezultat.

```
int multiply3(int el) {
	return 3 * el;
}

void testTransform() {
	vector<int> v;
	v.push_back(3);
	v.push_back(4);
	v.push_back(2);
	v.push_back(7);
	v.push_back(17);

	vector<int> v2(5);
	transform(v.begin(), v.end(), v2.begin(), multiply3);

	//print the elements
	for_each(v2.begin(), v2.end(), print);
}
```

Functor - STL Function Objects (Functors)

• Functor – Orice clasă ce are definit **operator ()**
• Față de pointer la funcție, functorul poate avea informații adiționale (variabile membre)

Dacă avem un obiect **f,** putem folosi obiectul similar ca și o funcție

Exemplu
someValue = f(arg1, arg2);

este același cu:
someValue = f.operator()(arg1, arg2);

```
class MyClass {
public:
	bool operator()(int i, int j) {
		return (i > j);
	}
};

sort(v.begin(), v.end(), MyClass());
```

Adaptor Iterator pentru inserție (insert iterator, inserters)

- Permite algoritmilor STL să opereze în modul de inserare (în loc de suprascriere)
- Rezolvă problema ce apare daca dorim să scriem (modificăm) elemente în secvența destinație dar nu mai este suficient spațiu.

Tipuri de iteratori de inserție:

- **back_inserter**, poate fi folosit pentru containere care oferă operația **push_back()**.
- **front_inserter**, poate fi folosit pentru containere care oferă operația **push_front()**.
- **inserter**, poate fi folosit pentru containere care oferă operația **insert()**.

```
deque<int> v;
v.push_back(4);
v.push_back(8);
v.push_back(12);

deque<int> v2;
//    back_insert_iterator<deque<int> > dest(v2);
front_insert_iterator<deque<int> > dest(v2);

//copy elements from v to v2
copy(v.begin(), v.end(), dest);
```

Adaptor de iterator pentru Input/Output

poate itera peste streamuri (citeşte sau scrie)

Un stream are funcţionalitatea potrivită pentru un iterator (se pot accesa elementele secvenţial) dar nu are interfaţa potrivită

Ex. Interfaţa (metodele oferite) de Cin nu este potrivit pentru a fi folosit cu algoritmi STL => STL oferă adaptor (implementarea foloseşte şablonul de proiectare adapter)

istream_iterator – transformă, adaptează intergaţa istream la interfaţa iterator

```
//create a istream iterator using the standard input
istream_iterator<int> start(cin);
istream_iterator<int> end;

//the vector where the values are stored
vector<int> v;
back_insert_iterator<vector<int> > dest(v);

//copy elements from the standard input into the vector
copy(start, end, dest);
```

Exemplu

```
const int size = 1000; // array of 1000 integers
int array[size];

cout << "Elements (Nr elements <1000):";

int elem;
int n = 0;
while (cin >> elem) {
      array[n++] = elem;
}

//sort the array
qsort(array, n, sizeof(int), cmpInt);

cout << endl;
//print the array
for (int i = 0; i < n; i++)
      cout << array[i] << " ";
cout << endl;
```

```
ifstream inFile("in.txt");
//create a istream iterator using the file
istream_iterator<int> start(inFile);
istream_iterator<int> end;

//the vector where the values are stored
vector<int> v;
back_insert_iterator<vector<int> > dest(v);

//copy from the standard input into the vector
copy(start, end, dest);
inFile.close();

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

copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
cout << endl;

//copy to file
ofstream outFile("out.txt");
copy(v.begin(), v.end(), ostream_iterator<int>(outFile, " "));
outFile.close();
```

Algoritmi STL

- simplitate: se poate folosi codul existent, nu e nevoie de implementare (sortare, copiere, etc)
- corectitudine: algoritmi din STL au fost testaţi, funcţionează corect
- performanţă: în general are performanţe mai bune decăt codul scris adhoc
  - Foloseşte algoritmi sifisticaţi (e mai performant decăt codul scris de un programator C)
  - Foloseşte technici mecanisme avansate (template specialization, template metaprogramming), algoritmi STL sunt optimizați.
  - Algoritmul STL poate exploata detalii interne de implementare de la containere.

- Claritate : codul este mai scurt, citind o linie se poate înțelege algoritmul
- Cod ușor de întreținut/modificat