ლაბორატორიული 4: ისევ STL-ის კომპონენტებზე

განსახილველი საკითხები:

- ალგორითმები: `distance, find_if_not, partition`
- ფუნქტორის შექმნა მთავარ პროგრამაში >>>
- `std::iterator_traits<BidirIt>::difference_type`-ის ერთი გამოყენება >>>
- მატრიცის დახარისხება რომელიმე სვეტის მიხედვით >>>
- სკალარული ნამრავლი >>>
- სავარჯიშოები >>>

ალგორითმები: `distance, find_if_not, partition`.

STL ბიბლიოთეკაში, `<iterator>` ფაილში `distance` ალგორითმის ერთ-ერთი პროტოტიპი არის:

```
template< class InputIt >
typename std::iterator_traits<InputIt>::difference_type //დასაბრუნებელი მნიშვნელობა
    distance( InputIt first, InputIt last );
```

ეს ალგორითმი გამოძახების შემდეგ აბრუნებს `first` იტერატორზე განხორციელებული ++ ოპერატორების რაოდენობას, რის შემდეგაც იგი გახდება `last`-ის ტოლი. თუ `last` მიუღწეველია `first` -იდან, მაშინ აბრუნებს უარყოფით რიცხვს, იმ რიცხვის მოდულს, რაც აჩვენებს თუ რამდენი ++ ოპერატორის შემდეგ გახდება `last` იტერატორი `first` -ის ტოლი. მაგრამ ეს C++11 -ის სტანდარტით და ისიც იმ შემთხვევაში, თუ საქმე გვაქვს random-access იტერატორთან.

უშუალოდ ალგორითმს სჭირდება მხოლოდ შემავალი იტერატორი. მაგრამ თუ იგი სხვა ალგორითმის შიგნით გამოიყენება, ეს ნიშნავს რომ ეს ალგორითმი მრავალჯერადი გავლისაა და ამიტომ Forward იტერატორი მაინც სჭირდება. ვნახოთ მისი გამოყენების მაგალითი:

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

int main()
{
	std::vector<int> v{ 5, 3, 1, 4 };
	std::cout << "distance(first, last) = "
		<< std::distance(v.begin(), v.end()) << '\n'
		<< "distance(last, first) = "
		<< std::distance(v.end(), v.begin()) << '\n';
}
```

შედეგად დაგვიბეჭდავს რიცხვებს 4 და -4.

C++ ენის ბოლო სტანდარტებში `find_if_no` ფუნქციის რამდენიმე ვარიანტია გადატვირთული. ჩვენ მოვიყვანთ ყველაზე მარტივს, რომელიც C++11-შია იმპლემენტირებული:

```
template<class InputIt, class UnaryPredicate>
InputIt find_if_not(InputIt first, InputIt last, UnaryPredicate q)
{
	for (; first != last; ++first) {
		if (!q(*first)) {
			return first;
		}
	}
	return last;
}
```

მოვიყვანოთ მისი გამოყენების მაგალითები:

```
#include <iostream>
#include <iterator>
#include <list>
```

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

int main()
{
	list<int>  lst{ 11, 21, 541, 100, 31,23 };
	//მოვძებნოთ პირველი რომელიც არაა 100 ნაკლები
	auto lm = [](int n) {return n < 100;};
	auto i = find_if_not(lst.begin(), lst.end(), lm);
	if (i != lst.end())
		cout << "Found  " << *i << endl;
	else
		cout << "Not found." << endl;

	istream_iterator<string> it(cin), end;
	auto lam = [](string s) {return s.length() < 4; };
	auto j = find_if_not(it, end, lam);
	if (j != end)
		cout << "Found  " << *j << endl;
	else   cout << "Not found." << endl;
}
```

ძალიან ეფექტური არის გადანაწილების ალგორითმი ანუ `partition`. იგი ისეა მოწყობილი, რომ მხოლოდ `ForwardIt` სჭირდება. სხვა ჩვენთვის ცნობილი რეალიზაციები (ძალიან ცნობილ სახელმძღვანელოებში მოყვანილი) იყენებს ორმხრივ იტერატორებს.

ვნახოთ ალგორითმის კოდი

```
template<class ForwardIt, class UnaryPredicate>
ForwardIt partition(ForwardIt first, ForwardIt last, UnaryPredicate p)
{
	first = std::find_if_not(first, last, p);
	if (first == last) return first;

	for (ForwardIt i = std::next(first); i != last; ++i) {
		if (p(*i)) {
			std::iter_swap(i, first);
			++first;
		}
	}
	return first;
}
```

ალგორითმი ხსნის ამოცანას: მოცემულია `[first,last)` ფრაგმენტი და უნარული ანუ ერთადგილიანი პრედიკატი (თვისება). ფრაგმენტში ობიექტები უნდა გადანაწილდეს ორ ჯგუფად: პირველი ჯგუფის ყველა ელემენტს აქვს მოცემული თვისება, ხოლო მეორე ჯგუფის არცერთ ელემენტს არ აქვს იგივე თვისება. ალგორითმმა უნდა დააბრუნოს მეორე ჯგუფის პირველივე ელემენტის იტერატორი.

ალგორითმი პირველ რიგში ეძებს პირველივე ელემენტს, რომელსაც არ აქვს მოცემული თვისება. თუ ასეთი არაა, ყველაფერი დამთავრდება, მეორე ჯგუფი აღარაა და ამიტომ დაბრუნდება პირველი და ერთადერთი ჯგუფის დასასრული.

თუ ასეთი მოიძებნა, იგი აღინიშნება ისევ `first`-ით, ხოლო იტერატორი `i` უტოლდება მის მომდევნოს. ამის შემდეგ `i` გაირბენს დიაპაზონის ბოლომდე და რამდენჯერაც შეხვდება მოცემული თვისების მქონე, იმდენჯერ გაუცვლის ადგილს ყველაზე მარცხენა ამ თვისების არმქონეს ანუ `first`-ზე მდგომს. მნიშვნელობების გაცვლის შემდეგ `first` გაიზრდება და კვლავ გახდება ყველაზე მარცხენა ასეთი თვისების არმქონეთა შორის.

ვნახოთ პროგრამა დრაივერი:

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include <forward_list>

int main()
{
	std::vector<int> v = { 0,1,2,3,4,5,6,7,8,9 };
	std::cout << "Original vector:\n    ";
	for (int elem : v) std::cout << elem << ' ';

	auto it = std::partition(v.begin(), v.end(), [](int i) {return i % 2 == 0; });

	std::cout << "\nPartitioned vector:\n    ";
	std::copy(std::begin(v), it, std::ostream_iterator<int>(std::cout, " "));
	std::cout << " * ";
	std::copy(it, std::end(v), std::ostream_iterator<int>(std::cout, " "));

	std::forward_list<int> fl = { 1, 30, -4, 3, 5, -4, 1, 6, -8, 2, -5, 64, 1, 92 };
	std::cout << "\nUnsorted list:\n    ";
	for (int n : fl) std::cout << n << ' ';

	auto i = std::partition(fl.begin(), fl.end(), [](int i) {return i> 0; });

	std::cout << "\nPartitioned list:\n    ";
	std::copy(fl.begin(), i, std::ostream_iterator<int>(std::cout, " "));
	std::cout << " | ";
	std::copy(i, std::end(fl), std::ostream_iterator<int>(std::cout, " "));
	std::cout << "\n";
}
```

```
Original vector:
  0 1 2 3 4 5 6 7 8 9
Partitioned vector:
  0 8 2 6 4  * 5 3 7 1 9
Unsorted list:
  1 30 -4 3 5 -4 1 6 -8 2 -5 64 1 92
Partitioned list:
  1 30 3 5 1 6 2 64 1 92  | -5 -4 -8 -4
Press any key to continue . . .
```

აქ, `std::forward_list<>` არის C++11-ის კიდევ ერთი სიახლე და აღნიშნავს ცალმხრივ ბმულ სიას. ცხადია, მისი იტერატორი მხოლოდ ცალმხრივია.

პროგრამის შედეგს აქვს სახე:

<<< ფუნქტორის შექმნა მთავარ პროგრამაში. განვიხილოთ მარტივი პროგრამა, რომელიც გვიჩვენებს, რომ გარკვეული სახის ფუნქტორის შექმნა შესაძლებელია სხვა ფუნქციების შიგნით. სხვა სიტყვებით, ფუნქტორი არის ფუნქციისმაგვარი ობიექტი, ან გამოძახებადი ობიექტი.

```cpp
#include <iostream>
#include <string>
#include <list>
#include <numeric>

using namespace std;

int main()
{
	cout << "ფუნქტორამდე ბევრი რამე შეიძლება შესრულდეს..." << endl;

	struct summ  {
	    string operator() (const string& a, const string &b) {
			return a+' '+b;
		}
	};

	list<string> lst{ "Home","Island","England","Back"};
```

```
        string s{ "" };
        s= accumulate(lst.begin(), lst.end(), s, summ());
        cout << s << endl;
}
```

როგორც ვხედავთ, გარკვეული (არაშემთხვევითი) მსგავსება ლამბდა ფუნქციებთან არის.

<<< std::iterator_traits<BidirIt>::difference_type-ის ერთი გამოყენება. რაიმე დიაპაზონში ელემენტების მნიშვნელობები შევაბრუნოთ: პირველ პოზიციაზე მდგომი ადგილს უცვლის ბოლოზე მდგომს, მეორეზე მდგომი ადგილს უცვლის ბოლოდან მეორეზე მდგომს და ა.შ. ალგორითმი ავაგოთ იტერატორების ერთ-ერთი მახასიათებლის გამოყენებით.

ალგორითმს და მთავარ პროგრამას აქვს სახე:

```
#include <iostream>
#include <list>
using namespace std;

template<class BidirIt>
void my_reverse(BidirIt first, BidirIt last)
{
        typename
            std::iterator_traits<BidirIt>::difference_type n = std::distance(first,last);
        --n;
        while (n > 0) {
                typename std::iterator_traits<BidirIt>::value_type tmp = *first;
                *first++ = *--last;
                *last = tmp;
                n -= 2;
        }
}

int main()
{
        list<int> lst{ 1,2,3,4,5,6,7 };
        cout << "Original list: ";
        for (auto m : lst) cout << m << "  ";
        cout << endl;

        my_reverse(lst.begin(), lst.end());
        cout << "Reversed list: ";
        for (auto m : lst) cout << m << "  ";
        cout << endl;
}
```

`std::distance(first,last)` გვიჩვენებს თუ რამდენი ელემენტია `[first,last)` დიაპაზონში. მაგალითად, თუ მთავარ პროგამაში სიის განაცხადის შემდეგ ჩავამატებთ სტრიქონს:

```
cout << std::distance(lst.begin(), lst.end()) << endl;
```

დაიბეჭდება ელემენტების როდენობა ანუ 7.

რადგან ერთელემენტიან დიაპაზონს შებრუნება არ სჭირდება, ამიტომ ეს რაოდენობა მცირდება ერთით. შედეგად, `while (n > 0)` პირობა მხოლოდ აუცილებლობის შემთხვევასი გამოიყენება (ტუ ორზე მეტი ელემენტი იყო დიაპაზონში).

შემდეგ ადგილები ეცვლება პირველ და ბოლოო ადგილზე მდგომებს, თან ისე რომ იტერატორებიც გადანაცვლდება ერთმანეთისკენ. ამ ამოცანაში ელემენტების რაოდენობით მუშაობა ბუნებრივია, რადგან მომდევნო განმეორებაზე გადასასვლელად ელემენტების რაოდენობა უნდა შემცირდეს ორით, რასაც ჩვეული `while (first != last)` პირობით ძნელად გავადევნებთ თვალს. განმეორება ტრიალებს, ვიდრე გადასანაცვლებელი ელემენტების

რაოდენობა დადებითია. ყურადღება მივაქციოთ, რომ ამოცანაში ვიყენებთ ჩრდილოვან ეფექტებს. იტერატორებთან დაკავშირებით ეს გავრცელებული პრაქტიკაა, თუმცა დიდი სიფრთხილეა საჭირო.

უნდა შევნიშნოთ აგრეთვე, რომ `std::distance(first,last)`-ის გამოყენება ნიშნავს, რომ იტერატორის კატეგორია საკმაოდ მაღალია.

<<< მატრიცის დახარისხება მისი რომელიმე სვეტის მიხედვით. წარმოვიდგინოთ, რომ გვინდა "data.txt" ფაილში მოთავსებული ცხრილის დახარისხება მისი რომელიმე სვეტის მიხედვით.

შემდეგ როგრამასი ეს ამოცანა გადაჭრილია შემდეგი თანმიმდევრობით. იქმნება ახალი ტიპი სახელად `Matrix`, რომელიც წარმოადგენს მტელი ვექტორების ვექტორს. პირველი ორი სტრიქონი არის C++11-ის შემოთავაზება. მის ნაცვლად შეგვძლო დაგვეწერა

```
typedef vector<vector<int>> Matrix;
```

შემდეგ, სტრინგების შემომტანი ნაკადით ფაილიდან ვკითხულობთ სტრიქონ-სტრიქონ. თითოეულ სტრიქონს - სტრინგიდან კითხვის ნაკადით ვწერთ დროებით ვექტორში რიცხვ-რიცხვ, ხოლო სტრიქონის დასრულების შემდეგ დროებით ვექტორს ვაგდებთ მატრიცაში.

ამის შემდეგ, ვქმნით შედარების ფუნქტორს-პრედიკატს. იგი ხედავს გლობალურ ცვლადს, რომელიც მიუთითებს დასახარისხებელ სვეტზე. დახარისხების ალგორითმში მესამე არგუმენტად ვაწვდით ამ ფუნქტორის ანონიმურ ობიექტს - კონსტრუქტორის გამოძახების სახით. მთლიან კოდს აქვს სახე:

```
#include <iostream>
#include <fstream>
#include<sstream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

int col;

int main()
{
	using
		Matrix = vector<vector<int>>;
	Matrix m;
	string s;
	ifstream ifs("data.txt");

	while (getline(ifs,s))
	{
		vector<int> tmp;
		int n;
		istringstream iss(s);
		while (iss >> n)
			tmp.push_back(n);
		m.push_back(tmp);
	}
	cout << "Matrix:" << endl;
	for (auto v : m)
	{
		for (auto val : v) cout << val << "  ";
		cout << endl;
	}
	//Comparison

	cout << "Which column to sort?" << endl;
```

```
	cin >> col;
	struct less {
		bool operator() (const vector<int>& a, const vector<int> &b) {
			return a[col] < b[col];
		}
	};
	cout << "Matrix, sorted by the " << col <<"-nd column" << endl;
	sort(m.begin(), m.end(), less());
	for (auto v : m)
	{
		for (auto val : v) cout << val << "  ";
		cout << endl;
	}
	cout << endl;
}
```

<<< სკალარული ნამრავლი. ვთქვათ, მოცემული გვაქვს ორი მიმდევრობა. გვინდა მათი წევრ-წევრად გადამრავლება და ნამრავლების შეკრება, ანუ სკალარული ნამრავლის მოძებნა.

ამოხსნა. ამისთვის არსებობს უფრო ზოგადი `inner_product` ალგორითმები მაგრან ჩვენ ზუსტად სკალარული ნამრავლის განსაზღვრის ამოცანას დავაპროგრამებთ. შესაძლოა ელემენტების რაოდენობა განსხვავდებოდეს, მაგრამ პირველ მიმდევრობაში ელემენტების რაოდენობა ნაკლები ან ტოლი უნდა იყოს ვიდრე მეორეში. პროგრამას შესაძლოა ჰქონდეს სახე:

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

template<class InputIt1, class InputIt2>
auto scalar_product(InputIt1 first1, InputIt1 last1,
	InputIt2 first2)
{
	using
		T = iterator_traits<InputIt1>::value_type;
	T value{};
	while (first1 != last1) {
		value += *first1 * *first2;
		++first1;
		++first2;
	}
	return value;
}

int main()
{
	vector<int> v{ 1,1,1 };
	list<int> lst{ 2,2,2,1,1 };

	int res = scalar_product(v.begin(), v.end(), lst.begin());
	cout << "Product: " << res << endl;
}
```

<<< სავარჯიშოები:

1. შეაფასეთ მეცადინეობაზე განხილული ალგორითმების ხანგრძლივობა (სისწრაფე, დროითი სირთულე).
2. `distance` ალგორითმის გამოყენების მაგალითში ვექტორი ჩაანაცვლეთ ცალმხრივი სიით. ახსენით მიღებული შედეგი.

3. გააკეთეთ რამდენიმე პრედიკატი მთავარ ფუნქციაში უშუალოდ ალგორითმის გამოძახების წინ და გადააწოდეთ იგი ალგორითმს არგუმენტად.
4. მოიყვანეთ reverse ალგორითმის იმპლემნტაცია ჩრდილოვანი ეფექტების გარეშე.
5. დააპროგრამეთ პრაქტიკულ მეცადინეობაზე მოყვანილი ალგორითმები.
6. გაარჩიეთ შემდეგი კოდი:

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include <list>
#include <forward_list>
#include <functional>

using namespace std;

template<typename ForwaedIt>
void qSort(ForwaedIt first, ForwaedIt last)
{
    if ( distance(first, last) > 1 )
    {
           auto x = *first;
           auto lm = [=](decltype(x) y) {return (y < x); };
           auto q = partition(first, last, lm);
           if (q == first) q = std::next(q);

           qSort(first, q);
           qSort(q, last);
    }
}
int main()
{
    std::vector<int> v = { 90,41,2,13,34,5,6,27,8,9 };
    std::cout << "Original vector:\n    ";
    for (int elem : v) std::cout << elem << ' ';
    qSort(v.begin(), v.end());
    std::cout << "\nSorted vector:\n    ";
    for (int elem : v) std::cout << elem << ' ';

    std::list<int> lst = { -2,-30,1,5,2,7,3,6,4 };
    std::cout << "\nOriginal list:\n    ";
    for (int elem : lst) std::cout << elem << ' ';
    qSort(lst.begin(), lst.end());
    std::cout << "\nSorted list:\n    ";
    for (int elem : lst) std::cout << elem << ' ';

    std::forward_list<int> fl = { -2,-30,1,5,2,7,3,6,4 };
    std::cout << "\nOriginal forward_list:\n    ";
    for (int elem : fl) std::cout << elem << ' ';
    qSort(fl.begin(), fl.end());
    std::cout << "\nSorted forward_list:\n    ";
    for (int elem : fl) std::cout << elem << ' ';
}
```