პრაქტიკული 3: Forward იტერატორები

სააუდიტორიო სამუშაო:

- ამოცანა 1 /ასლის შექმნის ფუნქცია/
- ამოცანა 2 /ორი ტოლი მეზობლის ძებნა/. >>>
- ამოცანა 3 /ერთი ობიექტის მეორით ჩანაცვლების ალგორითმი/ >>>
- სავარჯიშოები >>>

ამოცანა 1 /ასლის შექმნა/. შექმენით ფუნქცია, რომელიც შექმნის `[first, last)` ფრაგმენტის ასლს, რომელიც დაიწყება `result` იტერატორიდან. ფუნქციამ უნდა დააბრუნოს `result` -ის ბოლო მნიშვნელობა.

ამოხსნა: ამ ფუნქციას C++ -ში ეწოდება `copy()`. ჩვენ აქ შევუცვლით სახელს, რომ გატესტვის დროს არ მივიღოთ სახელების კონფლიქტი სტანდარტული ბიბლიოთეკის ფუნქციასთან. მისი ერთი შესაძლო ვარიანტი ასეთია:

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

ფუნქცია დრაივერს, რომელიც გამოიყენებს ამ კოდს, შესაძლოა ჰქონდეს სახე:

```
int main()
{
	list<int>  lst { 11, 21, 541, 100 };
	vector<int> v(7);
	myCopy(lst.begin(), lst.end(), v.begin());

	ostream_iterator<int> out(cout, "\t");
	myCopy(lst.begin(), lst.end(), out);
	cout << endl;

	myCopy(v.begin(), v.end(), out);
	cout << endl;

	myCopy(lst.begin(), lst.end(), v.begin() + 3);
	myCopy (v.begin(), v.end(), out);
	cout << endl;
}
```

როგორც ვხედავთ, ძალიან მოხერხებულია, რომ კონტეინერის ბეჭდისთვის გამოვიყენოთ იგივე ალგორითმის გადატვირთული ვერსია, რომელიც მუშაობს `ostream_iterator`-თან, თუმცა ამ დროს აუცილებელია `#include<iterator>` -ის ჩართვა.

აქ ჩვენ ორი ასლი გავაკეთეთ: ერთი მეორე კონტეინერის დასაწყისიდან, მეორე მისი შუა ნაწილიდან. შედეგს აქვს სახე:

```
11	21	541	100
11	21	541	100	0	0	0
11	21	541	11	21	541	100
Press any key to continue . . .
```

<<< ამოცანა 2 /ორი ტოლი მეზობლის ძებნა/.

ალგორითმი ეძებს `[first, last)` ფრაგმენტში პირველივე ერთმანეთის მეზობელ და ტოლ წყვილ ობიექტს. დააბრუნებს ამ ობიექტებიდან პირველის იტერატორს ან `last` -ს თუ ასეთი წყვილი არ აღმოჩნდა.

ამოხსნა: ამ ფუნქციას ბიბლიოთეკაში ეწოდება `adjacent_find`. ჩვენ ვუწოდოთ სხვა სახელი, რომ ავერიდოთ სახელების კონფლიქტს:

```
template <class ForwardIterator>
ForwardIterator my_adjacent_find(ForwardIterator first, ForwardIterator last)
{
	if (first != last)
	{
		ForwardIterator next = first; ++next;
		while (next != last) {
			if (*first == *next)  //or: if (pred(*first,*next)), for version (2)
				return first;
			++first; ++next;
		}
	}
	return last;
}
```

როგორც კოდიდან ჩანს, იტერატორის დამახსოვრება აუცილებელია. ეს არსებითია იტერატორისთვის და ნიშნავს, რომ ალგორითმს უნდა გადავაწოდოთ ForwardIterator-ის წყვილი. ასევე კოდიდან ჩანს, რომ ალგორითმი იმუშავებს იმ შემთხვევაშიც, თუ მას გადავაწვდით InputIterator -ის წყვილს (მაგალითად, შემავალ ნაკადს). მაგრამ თუ იპოვა წყვილი და დააბრუნა პირველის იტერატორი, იტერატორის ცვლილება არაპროგნოზირებად შედეგს მოგვცემს.

ვნახოთ პროგრამა დრაივერი:

```
int main()
{
	list<int>  lst { 11, 21, 541, 100, 450, 21, 21, 33 };
	auto i = my_adjacent_find(lst.begin(), lst.end());

	if (i != lst.end())
		cout << "first object: " << *i << endl;
	else
		cout << "Not found!" << endl;
}
```

<<< **ამოცანა 3** /ერთი ობიექტის მეორით ჩანაცვლების ალგორითმი/ ალგორითმი ეძებს `[first, last)` ფრაგმენტში `old_value` ობიექტს. რამდენჯერაც აღმოაჩენს, იმდენჯერ შეცვლის მას `new_value` ობიექტით.

ამოხსნა: ამ ალგორითმს სტანდარტულ ბიბლიოთეკაში ჰქვია replace(). მასაც შევუცვლით სახელს. რადგან იტერატორი ერთდროულად კითხულობს და წერს, ამიტომ ეკუთვნის ForwardIterator კატეგორიას.

მისი გამოყენების მაგალითი:

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

template <class ForwardIterator, class T>
void my_replace(ForwardIterator first, ForwardIterator last,
const T& old_value, const T& new_value)
{
	while (first != last) {
		if (*first == old_value)
			*first = new_value;
		++first;
	}
}
int main() {
	vector<int> myvector { 10, 20, 30, 30, 20, 10, 10, 20 };
```

```
	my_replace(myvector.begin(), myvector.end(), 20, 99); // 10 99 30 30 99 10 10 99
	cout << "myvector contains:";
	for (auto it = myvector.begin(); it != myvector.end(); ++it)
		std::cout << ' ' << *it;
	cout << '\n';
}
```

<<< სავარჯიშოები:

1. შექმენით ასლის შექმნის ალგორითმის გადატვირთული ვერსია, რომელიც მოცემული დიაპაზონიდან შექმნის მხოლოდ გარკვეული თვისებების მქონე ობიექტების ასლს.
2. მოიყვანეთ ალგორითმის კოდი, რომელიც `[first, last)` ფრაგმენტის ობიექტებს გაამრავლებს ერთი და იმავე მუდმივზე, რომელიც აგრეთვე არგუმენტად გადაეცემა ალგორითმს.
3. შექმენით `<algorithm>` ბიბლიოთკის `count` ალგორითმის ანალოგიური ფუნქცია.
4. შექმენით `<algorithm>` ბიბლიოთკის `count_if` ალგორითმის ანალოგიური ფუნქცია.
5. გადატვირთეთ `my_adjacent_find` ფუნქცია ისე, რომ მას შეეძლოს იგივე ამოცანის გადაჭრა იტერატორებისთვის, რომლებიც მიუთითებენ კლასების ობიექტების პოზიციას. გაითვალისწინეთ მესამე არგუმენტი - ბინარული პრედიკატი, რომელიც შეადარებს ნებისმიერ ორ ობიექტს ტოლობაზე.
6. გაარჩიეთ შემდეგი პროგრამა, რომელიც საშუალებას იძლევა შევამციროთ თარგის პარამეტრი-ტიპების რაოდენობა:

```
#include <iostream>     // std::cout
#include <vector>       // std::vector

template <class ForwardIterator>
void my_replace
(
    ForwardIterator first,
    ForwardIterator last,
    const typename std::iterator_traits<ForwardIterator>::value_type  old_value,
    const typename std::iterator_traits<ForwardIterator>::value_type   new_value
)
{
    while (first != last) {
        if (*first == old_value) *first = new_value;
        ++first;
    }
}
int main() {
    int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
    std::vector<int> myvector(myints, myints + 8);        // 10 20 30 30 20 10 10 20

    my_replace(myvector.begin(), myvector.end(), 20, 9999); //10 99 30 30 99 10 10 99

    std::cout << "myvector contains:";
    for (auto m:myvector)       std::cout << ' ' << m;
    std::cout << '\n';

}
```

7. მეცადინეობაზე განხილულ მესამე მაგალითში, თარგის პარამეტრების რაოდენობა შემაცირეთ სავარჯიშო 6-ში შემთავაზებული ტექნიკის გამოყენებით.
8. როგორ შეფასდება განხილული ალგორითმების ხანგრძლივობა ასიმპტოტურ აღნიშვნებში და რატომ?