თემა 5. იტერატორებთან დაკავშირებული რამდენიმე კერძო საკითხი:

იტერატორის შექმნა; იტერატორის, როგორც დასაბრუნებელი მნიშვნელობის, თავისებურება; იტერატორების გაუქმება

განხილული საკითხები:

- Forward იტერატორის სპეციფიკა, იტერატორების მახასიათებლები
- იტერატორის შექმნა >>>
- იტერატორის, როგორც დასაბრუნებელი მნიშვნელობის თავისებურება >>>
- იტერატორის გაუქმება >>>
- ლიტერატურა >>>

Forward იტერატორის სპეციფიკა, იტერატორების მახასიათებლები

როდესაც იტერატორს აქვს როგორც Input, ასევე Output იტერატორის თვისებები, მაშინ ცხადია რომ ის არის Forward იტერატორი:

```
using namespace std;
template < class ForwardIterator, class UnaryPredicate, class T >
void replace_if(ForwardIterator first, ForwardIterator last,
	UnaryPredicate pred, const T& new_value)
{
	while (first != last)
	{
		if (pred(*first)) *first = new_value;
		++first;
	}
}
```

ვნახოთ, თუ როგორაა შესაძლებელი თარგის პარამეტრების შემცირება:

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

template <typename T>
using
	ValT = typename std::iterator_traits<T>::value_type;
template <typename T>
using
	CValTRef = add_lvalue_reference_t<add_const_t<ValT<T>>>;
template < class ForwardIterator, class UnaryPredicate>
void replace_if (
	ForwardIterator first,
	ForwardIterator last,
	UnaryPredicate pred,
	CValTRef<ForwardIterator>  new_value
)
{
	while (first != last)	{
		if (pred(*first)) *first = new_value;
		++first;
	}
}
int main()
{
	vector<int> v{ 10, 20, 30, 30, 20, 10, 10, 20 };
	auto lm = [](int n) {return n % 3 == 0; };
	replace_if(v.begin(),v.end(),lm,-111);
	for (auto e : v)  cout << e << "   ";
	cout << endl;
}
```

შემდეგ ალგორითმში (ალგორითმი სტანდარტულია, ოღონდ სახელი არის შეცვლილი)

```
template <class ForwardIterator>
ForwardIterator my_max_element(ForwardIterator first, ForwardIterator last)
{
	if (first == last) return last;
	ForwardIterator largest = first;

	while (++first != last)
	if (*largest < *first)
		largest = first;
	return largest;
}
```

იტერატორში ჩაწერა არ ხდება, მაგრამ გვინდა იტერატორის დამახსოვრება იმ ვარაუდით, რომ დამახსოვრებული იტერატორიდან კვლავ მოვახერხებთ კონტეინერის შემოვლას (ერთი მიმართულებით მაინც). გასაგებია, რომ კონტეინერების იტერატორებს აქვთ უნარი რომ მრავალჯერ გაიარონ სჭირო დიაპაზონი. ვნახოთ რა მოხდება თუ შემავალ ნაკადში მოვინდომებთ მაქსიმალურის იტერატორის პოვნას:

```
int main()
{
	int myints[] { 3, 7, 2, 5, 6, 4, 9, 54, -11 };
	cout << "The largest element is " << *my_max_element(myints, myints + 9) << '\n';

	string s = "3 7 2 5 6 4 9 54 -2 ";
	istringstream iss(s);

	istream_iterator<int> iit(iss),eos,j;
	j = my_max_element(iit,eos);

	cout << "The largest element in string is " << *j << '\n';
	++j;
	cout << "And..." << *j << endl;
}
```

ბოლო ორი სტრიქონის გარეშე რჩება შთაბეჭდილება, თითქოს ალგორითმი კორექტულად მუშაობს Input იტერატორების კატეგორიაზეც. სინამდვილეში, ფუნქციის გამოძახების შემდეგ აღარავითარი ნაკადი არ არის, ამიტომ იტერატორი ვერსად ვერ გადადის. ლაბორატორიულ მეცადინეობაზე მოყვანილია დამატებითი მასალა ამ საკითხთან დაკავშირებით.

<<< **იტერატორის შექმნა.** იმისათვის, რომ წარმოვიდგინოთ თუ როგორ ხდება იტერატორის კლასის შექმნა სხვადასხვა კონტეინერის კლასის შიგნით, განვიხილოთ მაქსიმალურად გამარტივებული შემთხვევა. ქვემოთ მოყვანილ კოდში, სიაში გასაღებებს მთელი რიცხვები წარმოადგენს, ხოლო თვითონ სია მხოლოდ ერთ ველს (სიის პირველი კვანძის მისამართი) და რამდენიმე მეთოდს (ახალი ელემენტის დამატება სიის თავში, begin() და end()) მოიცავს წევრებად. სიის კვანძის სტრუქტურა და იტერატორის სტრუქტურა შექმნილია სიის კლასის შიგნით (ჩადგმული კლასები - გავრცელებული ტექნიკაა პროგრამირებაში). კვანძის სტრუქტურა გვიჩვენებს, რომ ცალმხრივ ბმულ სიასთან გვაქვს საქმე. იტერატორის სტრუქტურა გვიჩვენებს, რომ იგი თავის წინაპრად თვლის

```
iterator<forward_iterator_tag, int, ptrdiff_t, const int*, const int&>
```

კლასს, რაც ნიშნავს რომ ბევრი რამ (კონსტანტები, მეთოდები, ტიპები) უკვე განმარტებულია. თუ წინაპარზე მიმართვას მოვხსნით, მაშინ ჩვენს მიერ შექმნილი იტერატორის ფუნქციონალობა მკვეთრად შეიზღუდება.

განვიხილოთ შემდეგი კოდი, რომელშიც გაერთიანებულია კლასი, მისი იმპლემენტაცია და პროგრამა-დრაივერი:

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

class List
{
public:
    struct Node
    {
        int val;
        Node* next;
        Node(){ next = NULL; }
    };
    struct iterator : std::iterator<forward_iterator_tag, int, ptrdiff_t,
        const int*, const int&>
    {
        Node *ptr;
        iterator(Node* p = 0) : ptr(p) {}
        int& operator*(void){  return ptr->val;}
        iterator& operator++(void)
        {
            ptr = ptr->next;
            return *this;
        }
        iterator operator++ (int) {
            iterator tmp = *this; ++*this; return tmp;
        }
        bool operator== (const iterator& other) const
        {
            return ptr == other.ptr;
        }
        bool operator!=(const iterator& other) const
        {
            return ptr != other.ptr;
        }
    };
    Node* head;
    iterator begin() { return iterator(this->head); }
    iterator end() { return iterator(); }

    List(){ head = NULL; }
    void push_front(int k)
    {
        Node* tmp = new Node;
        tmp->val = k;
        tmp->next = head;
        head = tmp;
    }
};

int main()
{
    List lst;
    lst.push_front(55);
    lst.push_front(111);
    lst.push_front(434);
    lst.push_front(121);

    List::iterator i;
    for (i = lst.begin(); i !=lst.end(); ++i)
        cout << *i << "  ";
    cout << endl;
```

```
        List::iterator j = find(lst.begin(), lst.end(), 111);
        cout << *j << endl;
    }
```

ჩვენი მაგალითი ფაქტიურად მხოლოდ იდეას გვიჩვენებს, თუმცა გარკვეულ წარმოდგენას ქმნის თუ რა მიმართულებით უნდა ვიფიქროთ ასეთ შემთხვევებში. შედარებით სრულად შეგიძლიათ იხილოთ [1]-ში. კიდევ უფრო სრულად [2]-ში.

<<< იტერატორის, როგორც დასაბრუნებელი მნიშვნელობის, თავისებურება. დასაბრუნებელი მნიშვნელობის, თავისებურებების გააზრება გვეხმარება დაცული კოდის შექმნაში (უფრო ვრცლად, იხ. [3]-ში). განვიხილოთ შემდეგი კოდი და ავხსნათ თუ რას წარმოადგენს ფუნქციისთვის მნიშვნელობის მინიჭების საფუძველს.

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

template<typename Iterator>
Iterator secondIterator(Iterator first)
{
        return  ++first;
}
int main()
{
        vector<int> v{ -11, 913, 21, 77, 9012 };

        for (auto m:v)
                cout << m << "  ";
        cout << endl;

        cout << "v.back()=" << v.back() << endl;
        v.back() = -10;
        cout << "v.back()=" << v.back() << endl;

        cout << "v.front()=" << v.front() << endl;
        v.front() = 1001;
        cout << "v.front()=" << v.front() << endl;

        cout << "second: " << v[1] << endl;
        *secondIterator(v.begin()) = 3232;
        cout << "second: " << v[1] << endl;
}
```

პროგრამის შედეგს წარმოადგენს:

```
-11  913  21  77  9012
v.back()=9012
v.back()=-10
v.front()=-11
v.front()=1001
second: 913
second: 3232
Press any key to continue . .
```

რადგან ფუნქცია აბრუნებს იტერატორს, ხოლო იტერატორის მნიშვნელობა არის მისამართი, ამიტომ სრულიად ბუნებრივია რომ დაბრუნებულ იტერატორს, რომელიც ამ მომენტში გაიგივებულია ფუნქციის გამოძახებასთან (მაგ. `secondIterator(v.begin())` მივანიჭოთ ირიბი მნიშვნელობა.

ზოგჯერ ასეთი შესაძლებლობა გამიზნულად არის დატოვებული კომპილერის შემქმნელების მიერ (იხ. `v.back() = -10; , v.front() = 1001;`). თუ ეს გვერდითი ეფექტია რომელიც შეიძლება მოყვეს ჩვენს კოდს, მაშინ უნდა ვიზრუნოთ მის გაუქმებაზე.

განხილულის მსგავსი სიტუაციები ხშირად შეიძლება შეიქმნას, განსაკუთრებით ოპერატორების გადატვირთვისას, ამიტომ მნიშვნელოვანია ავითვისოთ აგრეთვე გვერდითი ეფექტების გაუქმების მეთოდები.

<<< იტერატორების გაუქმება. ახლა განვიხილოთ ერთი გავრცელებული სიტუაცია, როდესაც კომპილერი გარკვეულ იტერატორებს აუქმებს. გაუქმება შეიძლება სხვა შემთხვევებშიც მოხდეს, ამიტომ იტერატორებთან დაკავშირებული მანიპულაციების ჩატარებისას ეს ასპექტი ყოველთვის უნდა გვქონდეს მხედველობაში, ხოლო საჭირო ინფორმაცია დროულად უნდა მოვიძიოთ.

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

int main()
{
	vector<char> v{ 'a', 'b', 'c', 'd', 'e' };
	for (auto m:v)
		cout << m << "  ";
	cout << endl;
	vector<char>::iterator i, j, k;
	i = find(v.begin(), v.end(), 'b');
	j = find(v.begin(), v.end(), 'c');
	k = find(v.begin(), v.end(), 'd');
	v.erase(j);
	for (auto m:v)
		cout << m << "  ";
	cout << endl;
	// *i dabewdavda 'b'-s, da axlac dabeWdavs am simbolos;
	cout << *i << endl;
	// *k dabewdavda 'd'-s, magram ar dabeWdavs am simbolos;
	cout << *k << endl;
}
```

აქ გვაქვს სამი იტერატორი: `i, j, k`. ვექტორში, იტერატორზე მოთავსებული ობიექტის წაშლის შემდეგ **უქმდება** ეს იტერატორი და მის მარჯვნივ ყველა იტერატორი, **რადგან** ისინი აღარ მიუთითებენ მათზე დამაგრებულ მნიშვნელობებს. მაგალითად, ვექტორში იტერატორზე მოთვსებული ობიექტის წაშლის შემდეგ უქმდება თვითონ ეს იტერატორი და მისი მომდევნო (რაც ++ ოპერატორით მიიღება მისგან) ყველა იტერატორი. ამავე დროს, ვექტორში, წაშლილი იტერატორის წინ ყველა იტერატორი ძალაში რჩება და კვლავ ინახავს მათზე დამაგრებული მნიშვნელობების მისამართებს.

ვექტორთან (მაგალითად v) მუშაობისას, აუცილებლად გასათვალისწინებელია, რომ სტრიქონი

`v.erase(j--);` მუშაობს კორექტულად, რადგან `j` გადმოინაცვლებს წაშლილი იტერატორის მარცხნივ, სადაც ყველაფერი წესრიგშია, ხოლო სტრიქონი `v.erase(j++);` არაკორექტულია, რადგან `j` გადაინაცვლებს წაშლილი იტერატორის მარჯვნივ და მითუმეტეს მისი მნიშვნელობა (თუ მოხერხდა მისი ამოღება) არ იქნება ის, რაც იყო იტერატორის გაუქმებამდე.

ტერმინი **"გაუქმება"** ნიშნავს, რომ კომპილერი არავითარ პასუხისმგებლობას არ იღებს და არავითარ პროგნოზს არ აკეთებს გაუქმებული იტერატორის მნიშვნელობაზე.

სიის კონტეინერი უფრო მდგრადია წაშლის ოპერაციასთნ დაკავშირებით იმ აზრით, რომ მხოლოდ წაშლილი ელემენტის იტერატორები და რეფერენსები უქმდება. ამიტომ, სიის შემთხვევაში, კორექტულია ჩანაწერი:

```
list1.erase(j++);
```

მაშინ როდესაც კორექტული არაა

```
list1.erase(j); j++;
```

რადგან წაშლის შემდეგ იტერატორი უკვე გაუქმებულია.

<<< ლიტერატურა

1. http://www.cplusplus.com/reference/algorithm/max_element/?kw=max_element
2. http://www.cs.helsinki.fi/u/tpkarkka/alglib/k06/lectures/iterators.html
3. http://www.cs.rpi.edu//~musser/gp/List/
4. D. Musser, G. Derge, A. Saini. STL Tutorial and Reference Guide, Second Edition, Addison-Wesley, 2006.