

განხილული საკითხები:

- ფუნქციის ადაპტერები (ბაინდერები, ნეგატორები)
- ფუნქტორები >>>
- იტერატორები >>>
- input იტერატორები >>>
- output იტერატორები >>>
- forward იტერატორები >>>
- bidirectional იტერატორები >>>
- random access იტერატორები >>>
- Contiguous იტერატორები >>>
- იტერატორების მახასიათებლები (traits) >>>
- STL იტერატორთა იერარქია: ალგორითმებისა და კონტეინერების კომბინირების ეფექტურობა >>>
- კონტეინერთა ადაპტერები >>>
- ლიტერატურა >>>

STL ბიბლიოთეკა შეიცავს 6 განსხვავებული სახის კომპონენტებს: კონტეინერებს, განზოგადებულ (generic) ალგორითმებს, იტერატორებს, ფუნქტორებს, ადაპტერებს და ალოკატორებს. კონტეინერების (vector, deque, list) და ალგორითმების (sort, count, count\_is, reverse) ნაწილს ჩვენ ვიცნობთ და ამ კურსის განმავლობაში კიდევ უფრო გავიღრმავებთ ცოდნას. ალოკატორები დამწყებებისთვის არ არის განკუთვნილი. მოკლედ შევეხებით დანარჩენ სამ კომპონენტს.

### ფუნქციის ადაპტერები (ბაინდერები, ნეგატორები)

როგორც სახელი მიგვანიშნებს, ასეთი კომპონენტები გამოიყენება ერთი გარკვეული კლასის ფუნქციების გარდასაქმნელად (ადაპტირებისთვის) სხვა კლასის ფუნქციებად.

ზოგადად, bind არის ფუნქციის თარგი, რომელიც ქმნის რაიმე არსებული პარამეტრებიანი ფუნქციის შემფუთავს (wrapper). შემფუთავის გამოძახება ნიშნავს შეფუთული ფუნქციის გამოძახებას გარკვეული არგუმენტებით, რომლებიც მითითებულია შეფუთვაში. მაგალითად, იხ. <https://en.cppreference.com/w/cpp/utility/functional/bind>.

უფრო გავრცელებულია ადაპტერი bind2nd(), რომელიც ორადგილიან (ანუ ბინარულ) მიმართებას გარდაქმნის ერთადგილიან (უნარულ) მიმართებად. ბაინდერი bind2nd() დააფიქსირებს მიმართების მეორე არგუმენტს - მას არგუმენტებად გადაეცემა დამოკიდებულების სახელი და არგუმენტის დასაფიქსირებელი მნიშვნელობა.

ფუნქციის ადაპტერების შექმნა რამდენიმე მიზეზით იყო განპირობებული. ერთი მხრივ, მათი შექმნა აუცილებელი გახდა, რადგან C++ ენაში არ იყო გათვალისწინებული ისეთი ფუნქციების შექმნა რომლის დასაბრუნებელი მნიშვნელობა ისევ ფუნქციაა. მეორე მხრივ, ფუნქციის ადაპტერები გამოიყენება პრედიკატების შესაქმნელად. ასეთი პრედიკატები მხოლოდ ზოგიერთ შემთხვევაში ცვლის ჩვეულებრივ ბულის ტიპის ფუნქციებს, სამაგიეროდ მათი სპეციალიზირება (ტიპის ცხადად მითითება) შესაძლებელია. ეს მნიშვნელოვანია, რადგან ფუნქციის თარგში (ტემპლიტიან ფუნქციაში) ფუნქციის თარგის (ტემპლიტიანი ფუნქციის) გადანოდება ტექნიკურად რთულია. მაგალითად, თუ მთელი და ნამდვილი ტიპის ვექტორებში გვინდა 100-ზე მეტი რიცხვის მოძებნა, იძულებული ვართ გავაკეთოთ ორი ბულის ტიპის ფუნქცია, ერთი მთელი არგუმენტისთვის, მეორე ნამდვილისთვის. ფუნქციის ადაპტერს ასეთი რამ არ სჭირდება, რადგან ფუნქციის გამოძახების მომენტში მივუთითებთ საჭირო ტიპს.

STL იყენებს ორ ნეგატორს: `not1`, `not2`. ორივე მათგანი ცვლის არგუმენტის მნიშვნელობას სანინალმდეგოთი: თუ არგუმენტის მნიშვნელობა არის `true`, ნეგატორი მას გარდაქმნის `false`-ად და პირიქით. ამ თემას აგრეთვე მხოლოდ პრაქტიკულზე და ლაბორატორიულზე შევეხებით.

ამუამად, თანამედროვე ენები და მათ შორის C++ პრდიკატების შექმნისთვის იყენებენ ლამბდა ფუნქციებს. ამ ტიპის ფუნქციებს ჩვენც გამოვიყენებთ პრაქტიკულ და ლაბორატორიულ მეცადინეობებზე. ლამბდა ფუნქციები მრავალფეროვან საშუალებებს ქმნიან პრედიკატების შესაქმნელად და მოქნილობითაც აღემატებიან სხვა არსებულ საშუალებებს.

## ≪≪ ფუნქტორები

მოტივის ასახსნელად, ვისარგებლოთ განზოგადებული ალგორითმით `accumulate`:

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init)
{
    while (first != last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

ეს ალგორითმი აგროვებს მნიშვნელობების ჯამს. როდესაც მას ვიყენებთ, ვგულისხმობთ რომ შეკრების ოპერატორი განსაზღვრული არის მნიშვნელობების ტიპზე. მაგრამ არსებობს ამ ალგორითმის კიდევ უფრო აბსტრაქტული ვარიანტი, რომელიც საშუალებას იძლევა რომ დავაგროვოთ მნიშვნელობების ნამრავლი

```
template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation binary_op)
{
    while (first != last) {
        init = binary_op(init, *first);
        ++first;
    }
    return init;
}
```

ნაცვლად `+` მოქმედების გამოყენებისა, შემოყვანილია ახალი პარამეტრი `binary_op` - როგორც ბინარული მოქმედება მნიშვნელობების შეერთებისთვის.

ვნახოთ, თუ როგორ შეიძლება ამ აბსტრაქტული ვერსიის გამოყენება ნამრავლის გამოსათვლელად. ერთი გზა არის ფუნქციის შექმნა და გადანოდება ალგორითმში, ისე როგორც შემდეგ პროგრამაშია.

```
// განზოგადებული accumulate ალგორითმის გამოყენება ნამრავლის გამოსათვლელად
#include <iostream>
#include <vector>
#include <numeric> // For accumulate
using namespace std;

int mult(int x, int y) { return x * y; }

int main() {
    cout << "Using generic accumulate algorithm to "
         << "compute a product." << endl;
    vector<int> vector1{1, 2, 3, 4, 5};
    int product = accumulate(vector1.begin(), vector1.end(), 1, mult);
    cout << " Product: " << product << endl;
}
```

საწყის მნიშვნელობად გადავანოდეთ 1, რადგან ნამრვალს ვითვლით. აგრეთვე გადავანოდეთ ჩვეულებრივი ფუნქცია, უფრო სწორად კი ფუნქციის მისამართი &mult. მაგრამ ეს მხოლოდ ერთი გზა არის და არცთუ საუკეთესო. სხვა არჩევანი არის ფუნქტორის (ფუნქციური ობიექტის) გადანოდება, რაც განმარტების თანახმად არის ნებისმიერი ობიექტი, რომელიც შესაძლოა მიყენებულ იქნას ნულ, ერთ ან მეტ არგუმენტზე მნიშვნელობის მისაღებად, და/ან გამოთვლების მიმდინარეობის შესაცვლელად. შემდეგი პროგრამა შეიცავს ფუნქტორის შექმნისა და გამოყენების სცენარებს:

```
#include <iostream>
#include <vector>
#include <numeric> // For accumulate
using namespace std;
class multiply {
public:
    int operator()(int x, int y) const { return x * y; }
};
int main()
{
    cout << "Using generic accumulate algorithm to "
         << "compute a product." << endl;
    vector<int> vector1{ 1, 2, 3, 4, 5 };
    int product = accumulate(vector1.begin(), vector1.end(), 1, multiply());
    cout << " Product: " << product << endl;
}
```

`multiply` კლასში ფუნქციის გამოძახების ოპერატორ `operator()`-ის შექმნით ჩვენ შევქმენით ისეთი ტიპის ობიექტი, რომლის მიყენება შესაძლებელია არგუმენტების სიაზე, ისევე როგორც ფუნქცია აკეთებს. `accumulate` -ისთვის გადაცემული ობიექტი იქნება კლასის ნაგულისხმევი კონსტრუქტორის გამოძახების შედეგად. თავის მხრივ, ის იქმნება კომპილერის მიერ ავტომატურად. შევნიშნოთ, რომ ეს ობიექტი არაა შენახული მეხსიერებაში.

ასეთი ობიექტის გამოყენებას, ჩვეულებრივ ფუნქციასთან შედარებით, აქვს უპირატესობები. არგუმენტად გადანოდების შემთხვევაში, მას მიყვება დამატებითი ინფორმაცია (გარდა მისამართისა). გასათვალისწინებელია სისწრაფის და ეფექტურობის საკითხებიც.

რაც ჩვენ გავაკეთეთ იმდენად გავცვლელბულია, რომ STL-ში არის ასეთი რამ, ოღონდ უფრო ზოგადი სახის:

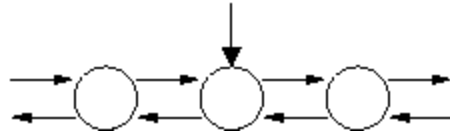
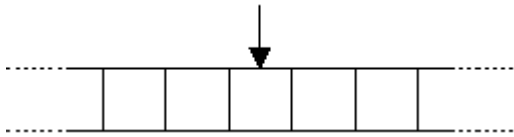
```
template <typename T>
class multiplies : public binary_function <T,T,T> {
public:
    T operator() (const T& x, const T& y) const
    {return x*y;}
};
```

ეს კლასი არის `binary_function` კლასის მემკვიდრე, რომლის მიზანს წარმოადგენს დამატებითი ინფორმაციის შენახვა ფუნქციის შესახებ. `multiplies` კლასის გამოყენებით, ბოლო პროგრამიდან შეგვიძლია ამოვიღოთ კლას `multiply`-ის განსაზღვრა, ხოლო დარჩენილ კოდში შეიცვლება ერთადერთი სტრიქონი:

```
int product = accumulate(vector1.begin(), vector1.end(), 1, multiplies<int>());
```

### ←←← იტერატორები

დანიშნულებით და დიზაინით იტერატორი არის პოინტერის მსგავსი ობიექტი. იტერატორი მიუთითებს ობიექტის პოზიციას დიაზონში:



იტერატორის საშუალებით შეიძლება კონტეინერში მოძრაობა (ყოველ იტერატორს გააჩნია ოპერატორი ++, ზოგიერთს --) და კონტეინერში შენახული ობიექტებზე წვდომა (ოპერატორი \*). იტერატორის კლასის მეთოდები პასუხობს გარკვეულ კითხვებს. მათ შორის

1. როგორ ფიქსირდება დიაპაზონის დასაწყისი? (begin())
2. როგორ გამოვიცნოთ დიაპაზონის დასასრული? (end())
3. როგორ ხდება მოძრაობა დასაწყისიდან ბოლოსკენ? (operator++)
4. როგორ მივწვდეთ დიაპაზონის ობიექტებს? (operator\*)

იტერატორების საშუალებით უკავშირდებიან ერთმანეთს განზოგადებული ალგორითმები და კონტეინერები. ალგორითმი იყენებს კონტეინერის შემოვლის მეთოდს (იტერატორს), ხოლო კონტეინერი უზრუნველყოფს შემოვლის ერთ ან რამდენიმე მეთოდს.

ყველა ალგორითმი ყველა კონტეინერთან ვერ იქნება თავსებადი. მაგალითად, ბინარული ძებნის ალგორითმი, ზოგადად, ვერ იმუშავებს მიმდევრობის კონტეინერებთან, რადგან ესენი მონაცემებს ინახავენ დაუხარისხებლად. იმისათვის რომ გასაგები გახდეს რომელი განზოგადებული ალგორითმი რომელ კონტეინერთან არის თავსებადი (ანუ კორექტულად, ეფექტურად და სწრაფად მუშაობენ), იტერატორები იერარქიული პრინციპით იყოფა ხუთ კატეგორიად: `input`, `output`, `forward`, `bidirectional`, `random access`. დაყოფა ეფუძნება კომპონენტის ობიექტებზე წვდომის პრინციპს.

წინასწარ განვიხილოთ ერთი მნიშვნელოვანი აღნიშვნა და განმარტება რომელსაც არსებითად იყენებენ STL-ის ალგორითმები იტერატორებთან (კერძოდ, იტერატორების დიაპაზონთან, ანუ `range`-თან) სამუშაოდ.

STL-ის ყველა ალგორითმი დიაპაზონის ობიექტებს წვდება იტერატორების საშუალებით. ჩვეულებრივ, ამისათვის საკმარისია იტერატორების იმ `first` და `last` წყვილის მითითება, რომლებიც აჭდევენ ობიექტების სასურველი დიაპაზონის დასაწყისსა და დასასრულს. იმისათვის რომ ზუსტად განისაზღვროს დიაპაზონი, ყოველთვის იგულისხმება, რომ დიაპაზონი `first`-იდან `last`-ამდე შედგება ყველა იმ იტერატორისგან, რომლებიც მიიღება დაწყებული `first`-ით და შემდეგ ოპერატორ ++ -ის გამოყენებით ვიდრე არ მივაღწევთ `last`-ს. მაგრამ ეს დიაპაზონი არ შეიცავს თვითონ `last`-ს. ამ ფაქტის ჩანერისთვის გამოიყენება ისეთივე ჩანაწერი, რაც გამოიყენება მათემატიკაში ნახევრადღია შუალედისთვის:

`[first, last)`.

STL-ის ყველა ალგორითმი გულისხმობს, რომ დიაპაზონი `[first, last)`, რომელიც მათ მიეწოდება არის კორექტული (ვალიდური), რაც ნიშნავს რომ `last` მიღწევადია `first`-იდან ოპერატორ ++ -ის გამოყენებით.

მნიშვნელოვანი კერძო შემთხვევა ე.წ. ცარიელი დიაპაზონისა იქმნება, როცა `first==last` არის ჭეშმარიტი.

კონცეფცია (concept) არის მოთხოვნების სიმრავლე, რაც ახასიათებს ტიპს:

- სინტაქსური მოთხოვნები განსაზღვრავს ნებადართულ გამოსახულებებს;
- სემანტური მოთხოვნები განსაზღვრავს გამოსახულებების მიერ გამოწვეულ ეფექტებს;

## <<< input იტერატორები

იტერატორების დაყოფა კატეგორიებად ხდება სხვადასხვა ალგორითმის საჭიროებების გათვალისწინებით. მაგალითად, ბმული სიის იტერატორს არ შეუძლია ყველა იმ ოპერატორის უზრუნველყოფა, რაც განსაზღვრულია პოინტერებზე. თუმცა საკმაო რაოდენობა ალგორითმებისთვის სრულიად საკმარისია ის, რისი უზრუნველყოფაც შეუძლია სიის იტერატორს. ერთ-ერთი ასეთი არაპრეტენზიული ალგორითმი არის ძებნის განზოგადებული ალგორითმი `find`. იგი გამოიყენება სხვადასხვა მონაცემთა სტრუქტურაში (მათ შორის ვექტორში, სიაში და ა.შ.) სიდიდის მოსაძებნად. მას შეუძლია შემავალ ნაკადში სიმბოლოების მოძებნაც სპეციალური იტერატორის საშუალებით. `find` ალგორითმის იმპლემენტაციაა:

```
template<typename InputIterator, typename T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    while(first != last && *first != value)
        ++first;
    return first;
}
```

ამ კოდში, `InputIterator` -ის ობიექტები მონანილოებენ შემდეგ გამოსახულებებში:

- `first != last`
- `++first`
- `*first`

იმისათვის, რომ `find` ალგორითმმა კორექტულად იმუშავოს, ეს გამოსახულებები უნდა იყვნენ განსაზღვრული და იღებდნენ შემდეგ მნიშვნელობებს:

- `first != last` - უნდა დააბრუნოს `true` როდესაც `first` განსხვავდება `last` - ისგან, და `false` წინააღმდეგ შემთხვევაში;
- `++first` - უნდა მოძებნოს და დააბრუნოს `first` -ის მომდენო მნიშვნელობა;
- `*first` - აბრუნებს იმ ობიექტს, რომლის მისამართსაც ინახავს `first`.

იმისათვის რომ `find` ალგორითმმა იმუშავოს ეფექტურად, თითოეული ეს ოპერაცია უნდა შესრულდეს მუდმივ დროში.

გარდა ამ სამი ოპერაციისა, აგრეთვე იგულისხმება რომ `InputIterator` -ის ობიექტებისთვის განსაზღვრული უნდა იყოს `==` ოპერატორი.

ეს ოპერატორები განსაზღვრულია პოინტერის ტიპის ობიექტებზე. თუმცა პოინტერებს აქვთ ბევრი სხვა თვისებაც. შესაბამისად, ყველა ალგორითმი, რომელიც მუსაობს `InputIterator`-თან, იმუშავებს აგრეთვე პოინტერებთან.

ტერმინი `InputIterator` არ უკავშირდება რაიმე ერთ ტიპს. პირიქით. იგი მოიცავს ტიპების ერთობლიობას, რომელთა ობიექტები აკმაყოფილებენ ზემოთ მოყვანილ შემლუდვებს (პირობებს). `find` ფუნქციის გამოძახების დროს არგუმენტებად უნდა გადაცემული იყოს ისეთი იტერატორები, რომლებიც არიან `InputIterator` კატეგორიის.

იმის საჩვენებლად, რომ სხვადასხვა განსხვავებულ ტიპს შეუძლია ერთდროულად დააკმაყოფილოს `InputIterator`-ის მოთხოვნები, განვიხილოთ შემდეგი მაგალითი, რომელშიც `find` ალგორითმი მუშაობს სიასთან, ვექტორთან და `iostream`-თან დაკავშირებულ `InputIterator`-თან.

```
#include<iostream>
```

```

#include<iterator>
#include<iostream>
#include<algorithm>
#include<list>
#include<vector>
using namespace std;
int main()
{
    vector<int> v{12,3,26,7,11,213,7,123,-31};
    int* ptr = find(&v[0], &v[8], 7);
    if(*ptr == 7 && *(ptr+1) == 11)
        cout << "O.K., found using pointers;" << endl;

    vector<int>::iterator it = find(v.begin(), v.end(), 7);
    if(*it == 7 && *(++it) == 11)
        cout << "O.K., found using vector<int>::iterator;" << endl;

    list<int> lst(v.begin(),v.end());
    list<int>::iterator i = find(lst.begin(), lst.end(), 7);
    if(*i == 7 && *(++i) == 11)
        cout << "O.K., found using list<int>::iterator;" << endl;

    cout << "Type some characters, including an 'x' followed" << endl
         << "by at leas one nonwhite-space character: " << endl;;

    istream_iterator<char> in(cin);
    istream_iterator<char> eos;
    find(in, eos, 'x');
    cout << "The first nonwhite-space character followinfg the first 'x' was: "
         << *(++in) << "." << endl;
}

```

პროგრამის შესრულების შედეგს აქვს სახე:

```

O.K., found using pointers;
O.K., found using vector<int>::iterator;
O.K., found using list<int>::iterator;
Type some characters, including an 'x' followed
by at leas one nonwhite-space character:
as dfsa x ls xa s |
The first nonwhite-space character followinfg the first 'x' was: l.

```

### <<< output იტერატორები

InputIterator-ები გამოიყენება მონაცემების წასაკითხად, ხოლო OutputIterator-ები საშუალებას გვაძლევენ რომ ობიექტები ჩავწეროთ, თუმცა არ უზრუნველყოფენ მათ წაკითხვას.

ეს ნიშნავს შემდეგს: თუ first არის რაიმე OutputIterator, მაშინ ჩვენ შეგვიძლია ვთქვათ \*first = ..., მაგრამ არავითარი გარანტია არაა რომ შეგვეძლება \*first-ის გამოყენება იმ ობიექტის წასაკითხად, რომლის მისამართსაც ის ინახავს.

სხვა განსხვავებები InputIterator-ისგან: OutputIterator-ის ობიექტისთვის არაა სავალდებულო რომ მისთვის განსაზღვრული იყოს == და != ოპერატორები.

მაგალითად, განვიხილოთ STL-ის ალგორითმი copy, რომელიც ასლს იღებს ერთი მიმდევრობიდან მეორეში:

```

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

```

შენიშნეთ, რომ ჩვეულებრივი პოინტერების გამოყენება შეგვიძლია OutputIterator-ად.

STL-ს აქვს სპეციალური OutputIterator-ები, რომლებსაც ეწოდება ostream იტერატორები. ისინი გამოიყენება გამოტანის ნაკადში ობიექტების ჩასანერად.

მაგალითად, თუ lst არის წინა პუნქტის მაგალითში შექმნილი სია, რომელიც იგივე ელემენტებითაა შევსილი, მაშინ შემდეგი ფრაგმენტი წერს სიის ობიექტებს გამოტანის cout ნაკადში:

```

ostream_iterator<int> out(cout, "\n");
copy(lst.begin(), lst.end(), out);

```

ან

```

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

```

თუ გვინდა ერთ სტრიქონსი დაბეჭდოს და ტაბულაციის ნიშნით განაცალკევოს სიმბოლოები.

## <<< Forward იტერატორები

ძალიან ხშირად, Forward იტერატორი ერთდროულად არის როგორც input, ასევე output იტერატორი. forward იტერატორს აქვს თვისება, რომელიც არაა სავალდებულო input და output იტერატორებისთვის: შესაძლებელია forward იტერატორის შენახვა და მისი გამოყენება კონტეინერის ხელახალი შემოვლისთვის, დანყებული ერთი და იმავე პოზიციიდან. ეს თვისება მნიშვნელოვანია და გამოიყენება ე.წ. multipass ალგორითმების აგებისთვის.

ზოგიერთ შემთხვევაში, ალგორითმში Forward იტერატორის გამოცნობა შეგვიძლია მხოლოდ იმით, რომ ხდება იტერატორის დამახსოვრება დიაპაზონის განმეორებით შემოვლის მიზნით. ამ თემას ჩვენ დავუთმობთ საკმარის დროს.

ერთ-ერთი განზოგადებული ალგორითმი, რომელიც ახდენს ობიექტების წაკითხვასა და ჩანერასაც არის შემდეგი:

```

template<typename ForwardIterator, typename T>
void replace(ForwardIterator first, ForwardIterator last, const T& x, const T& y)
{
    while (first != last)
    {
        if (*first == x)
            *first = y;
        ++first;
    }
}

```

მაგალითად, ღვეკში ელემენტების შეცვლისთვის ამ ალგორითმის გამოძახებას შესაძლოა ჰქონდეს სახე:

```
deque<char> deque1;
replace(deque1.begin(), deque1.end(), 'e', 'c');
```

### ≪≪ Bidirectional იტერატორები

Bidirectional იტერატორი იმით განსხვავდება Forward იტერატორისგან, რომ შეუძლია კონტეინერის შემოვლა განახორციელოს ორივე მიმართულებით, ანუ მასზე განსაზღვრულია -- ოპერატორის ორივე ვარიანტი: პრეფიქსულიც და პოსტფიქსულიც, მაგალითად, `it--` და `--it`.

### ≪≪ Random Access იტერატორები

Random Access იტერატორები, ანუ სწრაფი წვდომის იტერატორები აკმაყოფილებენ ყველა იმ მოთხოვნას, რასაც Bidirectional იტერატორები, და დამატებით ყოველი `r`, `s` იტერატორისთვის და `n` მთელი რიცხვისთვის განსაზღვრულია:

- იტერატორის და მთელის შეკრება და გამოკლება, წარმოდგენილი ჩანაწერით: `r+n`, `n+r`, `r-n`;
- კონტეინერში `r` იტერატორის `n`-ურ მომდევნო ელემენტზე სწრაფი წვდომა ფრჩხილებიანი ოპერატორის გამოყენებით ანუ ჩანაწერით `r[n]`, რაც ნიშნავს `*(r+n)`;
- "დიდი ნახტომები" ორივე მიმართულებით, გამოხატული ჩანაწერით `r+=n` და `r-=n`;
- იტერატორების სხვაობა `r-s`, რაც აბრუნებს მთელ მნიშვნელობას;
- შედარებები: `r<s`, `r>s`, `r<=s`, `r>=s`, რაც იძლევა ბულის ტიპის მნიშვნელობებს.

ამუამად კონტეინერების შექმნის ტექნიკა იმდენად არის განვითარებული, რომ შესაძლებელია კონტეინერის ელემენტზე სწრაფი წვდომა განხორციელდეს იმ შემთხვევაშიც კი, თუ კონტეინერი მესხიერებაში არ იკავებს ერთ მთლიან მონაკვეთს.

### ≪≪ Contiguous იტერატორები

ამ კატეგორიის ტექნიკური დახასიათება მოცემულია მისამართზე <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3884.pdf>. ეს კატეგორია ახალი შექმნილია. მისი მიზანია სწრაფი წვდომის იტერატორების გარკვეული ქვესიმრავლისთვის ალგორითმების იმპლემენტაციის გაუმჯობესება და აჩქარება. კერძოდ, როდესაც ++ ოპერაციის აზრით მოსაზღვრე იტერატორებზე მოთავსებული ობიექტები მესხიერებაშიც მოსაზღვრედ (გვერდიგვერდ) არის მოთავსებული. აქედან გამომდინარეობს სახელწოდება „მოსაზღვრე იტერატორები“. სხვა სიტყვებით, როდესაც კონტეინერს უკავია მესხიერებაში ერთი მთლიანი მონაკვეთი. ასეთებია `array`, `basic_string`, `vector`.

### ≪≪ იტერატორების მახასიათებლები (traits)

იტერატორების ტერმინებში განსაზღვრულ ალგორითმებში ხშირად საჭირო იტერატორების ისეთი მახასიათებლების ამოქმედება, როგორცაა მნიშვნელობის ტიპი, კატეგორია და კიდევ ზოგი რამ. შედეგად, `Iterator`-ისთვის უნდა იყოს განსაზღვრული შემდეგი ხუთი ტიპი:

- `std::iterator_traits<Iterator>::difference_type` - გამოიყენება ორი იტერატორის სხვაობის წარმოსადგენად.
- `std::iterator_traits<Iterator>::value_type` - ტიპი, რომელიც მიიღება იტერატორის განმისამართების (ირიბი მნიშვნელობის ალების) შედეგად.



- `std::iterator_traits<Iterator>::pointer` - პოინტერი `value_type`-ზე
- `std::iterator_traits<Iterator>::reference` - რეფერენსი `value_type`-ზე
- `std::iterator_traits<Iterator>::iterator_category` - განსაზღვრავს იტერატორის კატეგორიას.

მათი გამოყენების მაგალითები მრავლადაა ინტერნეტში, ამიტომ მხოლოდ ერთ საინტერესო გამოყენებაზე შევჩერდებით - შემდეგი კოდი გვიჩვენებს, თუ როგორ შეიძლება განზოგადებული ფუნქციის შექმნა, რომელიც დააბრუნებს გარკვეული დიაპაზონის ელემენტების ჯამს (შესაბამისი ტიპის აბრით):

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

template<typename InputIt>
auto f
(
    InputIt first,
    InputIt last
)
{
    typename std::iterator_traits<InputIt>::value_type v{};
    while (first != last)
    {
        v += *first;
        ++first;
    }
    return v;
}

int main()
{
    vector<int> a{ 1,2,3,4,5 };
    auto sum = f(a.begin(), a.end());
    cout << sum << endl;

    vector<string> b{ "we ", "work ", "at ", "TSU " };
    auto conc = f(b.begin(), b.end());
    cout << conc << endl;
}
```

## <<< STL იტერატორთა იერარქია: ალგორითმებისა და კონტეინერების კომბინირების ეფექტურობა

გასაღებს იტერატორებისა და STL -ისთვის მათი როლის გაგებისთვის წარმოადგენს იმის აღქმა, თუ რატომ არის სასარგებლო იტერატორების კლასიფიცირება ზემოთ აღწერილ ხუთ კატეგორიად.

ეს კლასიფიკაცია გულისხმობს რომ იტერატორებს შორის არსებობს გარკვეული იერარქია:

- Forward იტერატორი ერთდროულად არის აგრეთვე input და output იტერატორი;
- Bidirectional იტერატორი არის აგრეთვე Forward იტერატორი, აქედან გამომდინარე, არის აგრეთვე input და output იტერატორი;

- Random Access იტერატორი (სწრაფი წვდომის იტერატორი) არის აგრეთვე Bidirectional იტერატორი, ამიტომ არის აგრეთვე Forward იტერატორი, ამიტომ არის აგრეთვე input და output იტერატორი.
- Contiguous იტერატორი არის ყველაზე მეტად დატვირთული, ყველაფერი მოეთხოვება რაც სხვა იტერატორებს, ამიტომ შედის ყველა სხვა კატეგორიაში.

იერარქიული წყობიდან გამომდინარეობს შემდეგი:

- ის ალგორითმები, რომლებიც ითხოვენ მხოლოდ input ან output იტერატორებს, მუშაობენ აგრეთვე Forward, Bidirectional, Random Access, Contiguous იტერატორებთან;
- ის ალგორითმები, რომლებიც ითხოვენ Forward იტერატორებს, მუშაობენ აგრეთვე Bidirectional, Random Access, Contiguous იტერატორებთან;
- ის ალგორითმები, რომლებიც ითხოვენ Forward იტერატორებს, მუშაობენ აგრეთვე Random Access და Contiguous იტერატორებთან;

იტერატორთა კატეგორიები შემდეგნაირად გამოიყენება კონტეინერებისა და ალგორითმების სპეციფიკაციებში:

- კონტეინერთა კლასების აღწერები შეიცავს იმ იტერატორთა კატეგორიებს, რომლებსაც ისინი უზრუნველყოფენ;
- განზოგადებული ალგორითმების აღწერები შეიცავს იმ იტერატორთა კატეგორიებს, რომლებსაც ისინი მოითხოვენ.

ვნახოთ რამდენიმე მაგალითი:

- list უზრუნველყოფს Bidirectional იტერატორებს, ხოლო find ალგორითმს მხოლოდ input იტერატორები სჭირდება, ამიტომ find ალგორითმი მუშაობს list კონტეინერთან;
- მაშინ როდესაც list უზრუნველყოფს Bidirectional იტერატორებს, sort ალგორითმს სჭირდება სწრაფი წვდომის იტერატორები, ამიტომ არ შეიძლება sort ალგორითმის გამოყენება list კონტეინერთან;
- deque-ის ობიექტები გვაძლევს სწრაფი წვდომის იტერატორებს, რასაც თხოულობს sort ალგორითმი. ამიტომ, sort ალგორითმი იმუშავებს deque კონტეინერთან და ეს იქნება ეფექტური კომბინაცია;
- set იტერატორები არიან Bidirectional კატეგორიის და merge ალგორითმი ითხოვს input იტერატორებს ან უფრო მაღალი საფეხურის იტერატორებს. რადგან Bidirectional იტერატორები იტერატორთა იერარქიაში უფრო მაღლ საფეხურზე არიან, ამიტომ merge ალგორითმი მუშაობს set კონტეინერთან.

იტერატორთა იერარქია კარგად გვიჩვენებს იმ ძირითად იდეას, რაც წარმოადგენს STL ბიბლიოთეკის აგების საფუძველს: STL კონტეინერთა და ალგორითმების ინტერფეისები ისეა აგებული, რომ წაახალისებს ეფექტურ კომბინაციებს. ეფექტური კომბინაცია ნიშნავს რომ კოდი კომპილირდება შეცდომების გარეშე, და გარკვეული დროითი სისწრაფის შეფასებები დაცულია.

[<<< კონტეინერთა ადაპტერები](#)

STL-ში კონტეინერთა ადაპტერები გამოიყენება სხვა კონტეინერის ინტერფეისის შესაცვლელად. ისინი განისაზღვრება როგორც ტემპლიტის კლასები, რომლებიც მიიღებენ კონტეინერს როგორც პარამეტრს. ჩვენ ვნახავთ თუ რა მარტივად შეიძლება მონაცემთა სტრუქტურის ადაპტირება.

როგორც არ უნდა იყოს ორგანიზებული სტეკი (და რიგიც), ინტერფეისი დამოუკიდებელია იმპლემენტაციისგან და ადაპტირებული კონტეინერისგან. ეს სასარგებლოა ბევრ შემთხვევაში. ვთქვათ, თუ ჩვენ გადავწყვიტეთ სტეკის ორგანიზება ვექტორის საშუალებით, მაგრამ გარკვეულ მომენტში მოვინდომეთ გადართვა სტეკის ოპერაციების სხვა იმპლემენტაციაზე, მაგალითად სიის საშუალებით (ამის აუცილებლობა შეიძლება შეიქმნას, მაგალითად, თუ სტეკი ძალიან გაიზარდა ზომაში და სისტემა ვერ უზრუნველყოფს ასეთი ერთიანი მესსიერების ფრაგმენტის გამოყოფას ვექტორისთვის). სხვა შემთხვევაში, სხვა ფაქტორებმა შეიძლება განაპირობონ შეზღუდული არჩევანი.

**სტეკი.** სტეკი არის ძალიან მარტივი მონაცემთა სტრუქტურა ვექტორთან, სიასთან და ღეკთან შედარებით, რადგან თითოეულ მათგანზე რეალიზებულია გაცილებით მეტი მეთოდი, ვიდრე სტეკზე. სტეკის მოქმედების პრინციპი აღინერება დევიზით: ბოლო მოვიდა - პირველი წავიდა (Last-in, first-out ანუ LIFO). მისი მოქმედები პრინციპი გამარტივებული სახით გავს თეფშების წყებას, რომელსაც ახალი თფში ემატება ზემოდან (ბოლოში), და თეფშის აღებას ასევე ზემოდან (ბოლოდან) ხდება. პრაქტიკულ ამოცანებში ხშირად გვხვდება ასეთი სტრუქტურის შემოღების აუცილებლობა.

როგორც აღვნიშნეთ, ვექტორი, სია და ღეკი შეიძლება გამოყენებულ იქნას როგორც სტეკი, რადგან თითოეულ მათგანზე არის განსაზღვრული სტეკის ოპერაციები:

- ახალი ელემენტის ჩამატება ერთ ბოლოში (`push()` მეთოდი, `push_back()`-ით);
- ელემენტის წაშლა იგივე ბოლოდან (`pop()` მეთოდი, `pop_back()`-ით);
- ბოლოში მოთავსებული ელემენტის მნიშვნელობის ამოღება (`top()`, `back()`-ით);
- სტეკის შემოწმება, - ცარიელია თუ არა იგი (`empty()`);
- სტეკის ზომა - `size()`,
- `emplace` - ამ მეთოდს არ შევეხებით, რადგან მისი ეფექტური გამოყენება უკავშირდება C++11-ის გარკვეულ საკითხებს.
- `swap` - არგუმენტად გადაცემულ ორ სტეკს შეუცვლის შიგთავსებს.

სტეკის მიღება შეგვიძლია ვექტორის, ღეკის და სიის ადაპტირებით:

- `stack<T>` არის T ობიექტებისგან შედგენილი სტეკი, რომელიც გულისხმობს პრინციპით იმპლემენტირებულია ღეკით;
- `stack< T, vector<T> >` არის T ობიექტებისგან შედგენილი სტეკი, იმპლემენტირებული ვექტორით;
- `stack< T, list<T> >` არის T ობიექტებისგან შედგენილი სტეკი, იმპლემენტირებული სიით.

`pop()` ფუნქციის დასაბრუნებელი მნიშვნელობა არის `void` ტიპის. თუ საჭიროა წასაშლელი მნიშვნელობის ნახვა, მაშინ იგი წინასწარ უნდა ამოვიღოთ `top()` მეთოდით. სტეკის მიღება შეიძლება ნებისმიერი ისეთი კონტეინერის ადაპტირებით, რომელთაც აქვთ სტეკის ოპერაციების ანალოგები. ასეთი არის სამივე მიმდევრობის კონტეინერი.

სტეკს აქვს ერთი ასლის კონსტრუქტორი, მისი გამოძახება შეიძლება ორი ფორმით. მაგალითად:

```
stack<T> s(otherStack);
```

ქმნის otherStack-ის ასლს, რომლის ელემენტებისაა T ტიპის, და მეორე ფორმა:

```
stack<T> s = otherStack;
```

სტეკისთვის განსაზღვრულია მინიჭების ოპერაცია

```
s1 = s2;
```

ამის შედეგად, მარცხნივ მდგარი სტეკი მიიღებს მარჯვენა სტეკის ზომას და მნიშვნელობებს.

ძალიან საინტერესო არის ბინარული მიმართებები სტეკებს შორის.

```
s1 == s2
```

აბრუნებს true-ს თუ s1 და s2 სტეკებს აქვთ ერთი ზომა და ერთი ტიპი (ელემენტების), და ელემენტებს ყოველ წყვილში შესაბამისი ადგილების მიხედვით აქვთ ერთი და იგივე მნიშვნელობა. წინააღმდეგ შემთხვევაში ბრუნდება false.

```
s1 != s2
```

აბრუნებს true-ს თუ s1 == s2 არის false; წინააღმდეგ შემთხვევაში არის false.

```
s1 < s2
```

აბრუნებს true-ს თუ, s1 და s2-ის მნიშვნელობების წყვილ-წყვილად შედარებისას, პირველ წყვილში რომლის ორი მნიშვნელობა განსხვავებულია, s1-ისა არის ნაკლები s2-ისაზე. წინააღმდეგ შემთხვევაში არის false.

```
s1 <= s2
```

მხოლოდ მაშინ აბრუნებს true-ს თუ ან s1<s2 ან s1 == s2, is true.

```
s1 > s2
```

მხოლოდ მაშინ აბრუნებს true-ს თუ s2<s1 არის true.

```
s1 >= s2
```

მხოლოდ მაშინ აბრუნებს true თუ ან s1>s2 ან s1 == s2 is true.

ვნახოთ მაგალითი როდესაც ერთი სტეკი ითვლება მეორეზე მეტად:

```
#include <iostream>
#include <stack>
using namespace std;
int main()
{
    stack<char> s1;
    s1.push('A');
    s1.push('B');
    s1.push('C');
    s1.push('D');
    cout << "\nThe stack s1 contains " << s1.size() << " values." << endl;

    stack<char> s2(s1);
    if (s1 == s2)
        cout << "Stacks contain the same elements" << endl;

    s2.push('E');
    s2.push('F');
    s1.push('X');
    s1.push('F');
    if (s1 > s2)
        cout << "s1 > s2" << endl;
}
```

რამდენიმე კარგი მაგალითია მისამართზე [http://cs.stmarys.ca/~porter/csc/ref/stl/cont\\_stack.html](http://cs.stmarys.ca/~porter/csc/ref/stl/cont_stack.html).

შემდეგი მარტივი კოდი გვიჩვენებს სტეკად მიმდევრობის კონტეინერების ადაპტირების გზას:

### სტეკის იმპლემენტაცია:

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

template<typename T, typename Container = deque<T> >
class myStack{
private:
    Container a;
public:
    void push(T t)    {
        a.push_back(t);
    }
    void pop(void)    {
        a.pop_back();
    }
    int size()        {
        return a.size();
    }
    T& top()          {
        return a.back();
    }
    bool empty()      {
        return (0 == a.size());
    }
};

int main()
{
    myStack<int, deque<int> > a;
    a.push(11);
    a.push(22);
    a.push(4444);
    a.push(555);
    while (!a.empty())
    {
        cout << a.top() << endl;
        a.pop();
    }
    cout << "Stack with default container:" << endl;
    myStack<double> s;
    s.push(1.1);
    s.push(22);
    s.push(44.44);
    s.push(55.5);
    while (!s.empty())
    {
        cout << s.top() << endl;
        s.pop();
    }
}
```

**რიგი.** რიგი იმით განსხვავდება სტეკიგან, რომ ელემენტების ჩამატება ხდება ერთ ბოლოში, ხოლო მნიშვნელობის ამოღება და ელემენტის წაშლა ხდება მეორე ბოლოდან. ამ პრინციპის

შესახებ ამბობენ, "პირველი მოვიდა - პირველი წავიდა" (first-in, first-out ანუ FIFO). აქედან გამომდინარეობს რომ რიგის შესაქმნელად არ შეგვიძლია ვექტორის ადაპტირება.

რიგის მისაღებად გამოყენებადია ნებისმიერი ისეთი კონტეინერი, რომელიც უზრუნველყოფს შემდეგ ოპერაციებს: `empty()`, `size()`, `front()`, `pop()`, `push()`. ასეთი კონტეინერი ჯერ-ჯერობით მხოლოდ ორია: სია და დეკი.

- `queue<T>` არის T ობიექტებისგან შედგენილი რიგი, რომელიც გულისხმობს პრინციპით იმპლემენტირებულია დეკით;
- `queue < T, list<T> >` არის T ობიექტებისგან შედგენილი რიგი, სიის საშუალებით.

რიგის ოპერაციებია `size()`, `push()`, `pop()`, `front()`, `empty()`, აგრეთვე `emplace` და `swap`.

ისევე როგორც სტეკზე, განსაზღვრულია ასლის კონსტრუქტორი ორი ფორმით და მინიჭების შეტყობინება, აგრეთვე ბინარული მიმართების ოპერატორები.

რამდენიმე საინტერესო მაგალითი და დამატებითი ცნობები შეგიძლიათ იხილოთ მისამართზე [http://cs.stmarys.ca/~porter/csc/ref/stl/cont\\_queue.html](http://cs.stmarys.ca/~porter/csc/ref/stl/cont_queue.html)

**პრიორიტეტების რიგი.** პრიორიტეტების რიგი არის ისეთი მონაცემთა სტრუქტურა, რომელშიც ამოსაღები ელემენტი, ნებისმიერ მომენტში, შესაძლოა იყოს მხოლოდ რიგის ელემენტებს შორის უდიდესი. არსებობს შესაძლებლობა რომ ელემენტების შედარების გზაც მივუთითოთ ცხადად.

პრიორიტეტების რიგის მისაღებად შესაძლებელია ნებისმიერი ისეთი კონტეინერის ადაპტირება, რომელიც უზრუნველყოფს სწრაფ წვდომას. პრიორიტეტების რიგი უზრუნველყოფს შნემდეგი ოპერაციების შესრულებას: `empty()`, `size()`, `top()`, `pop()`, `push()`, აგრეთვე `emplace` და `swap`. ომდევრო კვირებში ვნახავთ, თუ როგორ შეიძლება პრიორიტეტების რიგის გაკეთება ორობითი გროვის საშუალებით.

პრიორიტეტების რიგზე განაცხადებს აქვს სახე:

- `priority_queue<int>` არის მთელი რიცხვებისგან შედგენილი პრიორიტეტების რიგი, რომელიც გულისხმობს პრინციპით იმპლემენტირებულია ვექტორით და ასევე გულისხმობს `less<int>()` შედარებას.

მარტივი მისახვედრია თუ რას ნიშნავს:

- `priority_queue<int, vector<int>, greater<int>>`,
- `priority_queue<float, deque<float>, greater<float>>`.

ამ ადაპტერსიც გვაქვს ასლის კონსტრუქტორი ორი ფორმით და მინიჭების შეტყობინება. დამატებითი ცნობები და რამდენიმე კარგი მაგალითი არის მისამართზე [http://cs.stmarys.ca/~porter/csc/ref/stl/cont\\_priority\\_queue.html](http://cs.stmarys.ca/~porter/csc/ref/stl/cont_priority_queue.html)

## ←← ლიტერატურა

1. D. Musser, G. Derge, A. Saini. STL Tutorial and Reference Guide, 2nd Edition, Addison-Wesley, 2006
2. [http://cs.stmarys.ca/~porter/csc/ref/stl/index\\_containers.html](http://cs.stmarys.ca/~porter/csc/ref/stl/index_containers.html)