პრაქტიკული მეცადინეობა 1 თვისებები. ფუნქციის თარგი

სააუდიტორიო სამუშაო:

- მაქსიმუმის განსაზღვრა std::max ალგორითმებით
- გარკვეული თვისების მქონე ობიექტის ძებნა კონტეინერებში >>>
- count_if ალგორითმი ბაინდერებით შექმნილი პრედიკატებით/ >>>
- count_if ალგორითმი ლამბდა ფუნქციებით/ >>>
- სავარჯიშოები >>>

მაქსიმუმის განსაზღვრა std::max ალგორითმებით/. სტანდარტული ბიბლიოთეკის შესაბამისი ალგორითმების შესაძლო იმპლემენტაციები https://en.cppreference.com/w/cpp/algorithm/max-ის არის:

პირველი სახე

```
template<class T>
const T& max(const T& a, const T& b)
{
    return (a < b) ? b : a;
}
```

მეორე სახე

```
template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp)
{
    return (comp(a, b)) ? b : a;
}
```

მესამე სახე

```
template< class T >
T max( std::initializer_list<T> ilist)
{
    return *std::max_element(ilist.begin(), ilist.end());
}
```

მეოთხე სახე

```
template< class T, class Compare >
T max( std::initializer_list<T> ilist, Compare comp )
{
    return *std::max_element(ilist.begin(), ilist.end(), comp);
}
```

პირველი და მესამე სახე გულისხმობს, რომ არგუმენტებზე უკვე განსაზღვრულია ორადგილიანი მიმართება, საჭირო თვისებების მქონე, რომელიც განახორციელებს ელემენტების შედარებას < ოპერატორით. სხვა ორ სახეში, მომხმარებელმა უნდა ასწავლოს ალგორითმს, თუ როგორ შეადაროს არგუმენტები ერთმანეთს.

ალგორითმის პარამეტრის ტიპი const T& ნიშნავს, რომ არგუმენტად შეიძლება გადავაწოდოთ ცვლადი (რომელსაც დაერქმევა სხვა სახელი და ალგორითმისთვის იქნება მუდმივი), და შეგვიძლია გადავაწოდოთ მუდმივი (ანუ რომელსაც თავისი ადგილი არ აქვს მეხსიერებაში). განვიხილოთ გამოყენების მაგალითები.

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

int main()
{
```

```
	int k = 5;
	auto a = max({ 21, 43, k, 7, -23, 45, 12 });  //მესამე სახე
	//ან std::max - თუ using namespace std; ჩართული არაა
	cout << "a = " << a << endl;

	double y{ 0.99 };                                        //პირველი სახე
	auto x = max( 0.55,y );
	cout << "y = " << y << endl;

	string str{ "SANGU" };
	auto s = max(str, (string)"CSPlus");
	cout << "s = " << s << endl;

	auto lm = [](const string& a, const string& b) {return a.length() < b.length();
};
	s = max(str, (string)"CSPlus",lm);
	cout << "s = " << s << endl;
}
```

<<< **ამოცანა 1** /გარკვეული თვისების მქონე ობიექტის ძებნა კონტეინერში/. მოიყვანეთ ფუნქციის კოდი, რომელიც დაადგენს არის თუ არა მოცემულ v ვექტორში მოცემული p თვისების მქონე ელემენტი. ძირითად პროგრამაში მოიყვანეთ ფუნქციის გამოძახების რამდენიმე მაგალითი სხვადასხვა ტიპის ვექტორებისთვის.

შენიშვნა: ჩვეულებრივ, ტერმინი **თვისება** (პრედიკატი) ნიშნავს ერთი ცვლადის ფუნქციას, რომლის მნიშვნელობათა სიმრავლე არის **bool**. მაგალითად:

```
bool f(int n){
    return (n>100);
}
```

ან

```
bool isNegative(int a){
    return (a<0);
}
```

ამოხსნა: ფუნქციას და პროგრამას შესაძლოა ჰქონდეს ასეთი სახე:

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

using namespace std;

bool g(int n) { return n < 0; };

int main() {
	vector<int> v = { 21, 43, 5, 7, -23, 45, 12 };//ვექტორი შევქმენით და შევავსეთ
	auto it = find_if(v.begin(),v.end(), g);
	// auto it = find_if(v.begin(),v.end(), bind2nd(less<int>(),0));
	// auto it = find_if(v.begin(),v.end(), [](int n){return n < 0; });

	if (it != v.end())
		cout << "Found the negative number  " << *it << endl;
	else
		cout << "The negative number not found " << endl;

	list<double> lst{ 0.12,0.54,1.12 };
	auto i = find_if(lst.begin(), lst.end(), bind2nd(greater<double>(), 1.));
	if (i != lst.end())
```

```
        cout << "Found " << *i << endl;
    else
        cout << "Not found" << endl;
}
```

ფუნქციის არგუმენტად თვისების გამოყენება ძალიან მნიშვნელოვანია, იგი საშუალებას იძლევა შევქმნათ განზოგადებული (generic) ალგორითმები. მაგრამ bool ტიპის ფუნქციის გადაწოდება არგუმენტად ძალიან პრიმიტიული მეთოდია. მას ბევრი ნაკლი აქვს, ძირითადი ნაკლი ისაა, რომ ფუნქციის თარგში პარამეტრად არ შეგვიძლია ფუნქციის თარგის გადაწოდება.

პრედიკატის შექმნა იმდენად მნიშვნელოვანი საკითხია, რომ C++ ენა და სტანდარტული ბიბლიოთეკა რამდენიმე გზას გვთავაზობს თვისებების ეფექტურად და საიმედოდ შექმნისთვის. პრედიკატის შექმნის ერთი გზა არის binder-ების (**დამმაგრებლები**) გამოყენება. ბაინდერი არის **ფუნქციის ადაპტერი**, ანუ იგი უკვე არსებული ფუნქციის ადაპტირებას ახდენს სხვა მიზნებისთვის. მეორე, უფრო თანამედროვე, ზოგადი და გავრცელებული გზა არის ლამბდა ფუნქციების გამოყენება.

<<< **ამოცანა 3** /count_if ალგორითმი ბაინდერებით შექმნილი პრედიკატებით/. count_if ალგორითმის გამოყენებით დათვალეთ მთელი რიცხვების v ვექტორში, რომელშიც წერია

34, 100, 783, 22, 33, 54, 92, 100, 21,

ა) 100-ზე მეტი ელემენტების რაოდენობა;
ბ) 100-ზე მეტი ან ტოლი ელემენტების რაოდენობა;
გ) 100-ზე ნაკლები ელემენტების რაოდენობა;
დ) 100-ზე ნაკლები ან ტოლი ელემენტების რაოდენობა;

პრედიკატები შექმენით ფუნქციის ადაპტერების გამოყენებით.

შენიშვნა: ვისარგებლოთ სტანდარტული ბიბლიოთეკის საშუალებებით, და ჩვენთვის ცნობილი ორადგილიანი დამოკიდებულებებიდან (less, less_equal, greater, greater_equal) შევქმნათ ერთადგილიანი პრედიკატები, ანუ თვისებები. გავითვალისწინოთ, რომ მათი გამოყენებისთვის საჭიროა #include <functional> ბიბლიოთეკის ჩართვა.

ამოხსნა: ნებისმიერი ორი მთელი a და b რიცხვებისთვის, სპეციალიზირებული დამოკიდებულება greater<int>() ნიშნავს, რომ a>b. თუ ჩვენ დავაფიქსირებთ მეორეს მნიშვნელობას, ვთქვათ b=100, მაშინ greater<int>() დაფიქსირებული მეორე არგუმენტით გადაიქცევა ერთადგილიან დამოკიდებულებად, რომლის მნიშვნელობა ყოველი a-სთვის ტოლია (a>100)-ის. ე.ი. ან ჭეშმარიტია ან მცდარი. ამიტომ, შეტყობინებების

```
int k = count_if(v.begin(),v.end(),bind2nd(greater<int>(),100));
cout <<"100-ze metia: " << k <<endl;
```

შესრულების შემდეგ k გახდება 1-ის ტოლი. აქ bind2nd ნიშნავს მეორე არგუმენტის დაფიქსირებას.

ერთადგილიანი დამოკიდებულებისთვის შეგვიძლია გამოვიყენოთ ნეგატორი not1(), რომელიც შეაბრუნებს მის მნიშვნელობას. ამ შემთხვევაში, 100-ზე მეტის შებრუნებული არის ნაკლები ან ტოლი 100-ზე. ამიტომ, შემდეგი კოდის

```
int k = count_if(v.begin(),v.end(),  not1(bind2nd(greater<int>(),100)));
cout <<"100-ze naklebia an toli: " << k <<endl;
```

შესრულების შემდეგ k გახდება 8-ის ტოლი.

იგივე ეფექტის მიღწევა შეგვიძლია უფრო იოლად, თუ პირდაპირ გამოვიყენებთ `less_equal`-ს. ანალოგიურად, 100-ზე მეტი ან ტოლის რაოდენობის გასაგებად შეგვიძლია გამოვიყენოთ ან უშუალოდ `greater_equal`, ან უფრო რთული გზა.

<<< **ამოცანა 4** /count_if ალგორითმი ლამბდა ფუნქციებით/. `count_if` ალგორითმის გამოყენებით დათვალეთ მთელ `v` ვექტორში, რომელშიც წერია რიცხვები

34, 100, 783, 22, 33, 54, 92, 100, 21,

ა) 100-ზე მეტი ელემენტების რაოდენობა;
ბ) 100-ზე მეტი ან ტოლი ელემენტების რაოდენობა;
გ) 100-ზე ნაკლები ელემენტების რაოდენობა;
დ) 100-ზე ნაკლები ან ტოლი ელემენტების რაოდენობა;

ამოხსნა: 100 –ზე მეტი რიცხვების რაოდენობას დავთვლით შემდეგნაირად:

```
int k = count_if(v.begin(), v.end(), [](int k){return k > 100; });
cout << "100-ze metebis raodenoba: " << k << endl;
```

ან

```
auto l = [](int k){return k > 100; };
int k = count_if(v.begin(), v.end(),l);
cout << "100-ze metebis raodenoba: " << k << endl;
```

<<< **სავარჯიშოები:**

1. პირველი ამოცანისთვის, სიდიდით ერთზე მეტი ნამდვილი რიცხვის გამოცნობის პრედიკატი განსაზღვრეთ ლამბდა ფუნქციით და ჩვეულებრივი bool ტიპის ფუნქციით. გამოიყენეთ ეს პრედიკატები კოდში.
2. შექმენით ბულის ტიპის ფუნქციები, რომლებიც გაარკვევს არის თუ არა მთელი რიცხვი სამის, ხუთის და 11-ის ჯერადი. გამოიყენეთ ისინი ძებნის ფუნქციებში.
3. შექმენით ლამბდა ფუნქციები, რომლებიც გაარკვევს არის თუ არა მთელი რიცხვი სამის, ხუთის და 11-ის ჯერადი. გამოიყენეთ ისინი ძებნის ფუნქციებში.

წინა სავარჯიშოებში შექმნილი პრედიკატები გამოიყენეთ `count_if()` ალგორითმში.