პრაქტიკული მეცადინეობა 1 **ტემპლიტიანი ფუნქციები**

სააუდიტორიო სამუშაო:

- ამოცანა 1 /ობიექტის ძებნა ვექტორში ინდექსის დაბრუნებით/
- ამოცანა 2 /გარკვეული თვისების მქონე ობიექტის ძებნა ვექტორში ინდექსის დაბრუნებით/ >>>
- ამოცანა 3 /count_if ალგორითმი ბაინდერებით შექმნილი პრედიკატებით/ >>>
- ამოცანა 4 /count_if ალგორითმი ლამბდა ფუნქციებით/ >>>
- სავარჯიშოები >>>

ამოცანა 1 /ობიექტის ძებნა ვექტორში ინდექსის დაბრუნებით/. შექმენით ფუნქცია, რომელიც დაადგენს არის თუ არა მოცემული v ვექტორის ინდექსების `[p,r)` დიაპაზონში მოცემული `value` სიდიდის ტოლი ელემენტი. დადებითი პასუხის შემთხვევაში ალგორითმმა უნდა დააბრუნოს პირველივე (p-დან დაწყებული) ასეთი ელემენტის ინდექსი, წინააღმდეგ შემთხვევაში კი r (ანუ დიაპაზონიდან გასული პირველივე ინდექსი). მოიყვანეთ ძირითად პროგრამაში ამ ფუნქციის გამოძახების მაგალითები რამდენიმე განსხვავებული ტიპის ვექტორისთვის.

ამოხსნა: ძებნის ალგორითმის ფსევდოკოდში გავითვალისწინოთ, რომ `[p,r)` დიაპაზონი ნახევრადღიაა და r ინდექსზე მდგომი მნიშვნელოვა არ გვაინტერესებს:

```
int search(ვექტორი vec, საძიებელი value, დასაწყისი p, დასასრული r )
{
    while (p != r) {
       if (vec[p] == value) break;
       ++p;
    }
   return p;
}
```

ფსევდოკოდში არაა აუცილებელი იმის დაზუსტება, თუ რა ტიპის ელემენტებისგან შედგება ვექტორი. რადგან იმპლემენტაცია დამოუკიდებელია ტიპისგან, საკმარისია მხოლოდ, რომ გადატვირთული იყოს "==" ოპერატორი. C++ საშუალებას იძლევა, რომ ფუნქციის პარამეტრის ტიპიც პარამეტრად ვაქციოთ. ეს კეთდება **template**-ების გამოყენებით, რაც ქართულად ითარგმნება როგორც ყალიბი, ან თარგი. ანალოგია სრულიად ბუნებრივია, რადგან როგორც ერთ ყალიბში შეგვიძლია ჩავასხათ სხვადასხვა მასალა, როგორც ერთი თარგით შეიძლება გამოიჭრას სხვადასხვა სახის ქსოვილისგან ერთი ფორმის ნაჭერი, ასევე, ერთი და იგივე კოდი შესაძლოა სამართლიანი იყოს რამდენიმე სხვადასხვა ტიპის ცვლადისთვის. **template**-ების მოქმედების მექანიზმი ახსნილი იქნება ობიექტზე ორიენტირებული პროგრამირების კურსში.

ალგორითმის შესაბამისი კოდს და პროგრამა დრაივერს შესაძლოა ჰქონდეს სახე:

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

using namespace std;

template<typename T>
int search(const vector<T> &vec, const T& value, int p, int r)
{
    while (p != r) {
       if (vec[p] == value) break;
       ++p;
    }
   return p;
}
int main()
{
       vector<int> v = { 21, 43, 5, 7, -23, 45, 12 }; //ვექტორი შევქმენით და შევავსეთ
       vector<string> vStr = { "George", "Nick", "Tbilisi", " Johannesburg" };
```

```
	int index(search(v, 7, 0, v.size()));  //ძებნის ფუნქციის გამოძახება მთელებისთვის
	if (index != v.size())  //თუ მოიძებნა, ანუ  თუ ინდექსი არ გავიდა დიაპაზონიდან
	    cout << "7 found at index " << index << endl;

	string value("Nick");
	index = search(vStr, value, 1, 3);  //ძებნის ფუნქციის გამოძახება სტრინგებისთვის
	if (index != 3)                     //თუ მოიძებნა ასეთი
	    cout << "\"Nick\" found at index " << index << endl;
}  ■
```

<<< **ამოცანა 2** /გარკვეული თვისების მქონე ობიექტის ძებნა ვექტორში ინდექსის დაბრუნებით/. მოიყვანეთ ფუნქციის კოდი, რომელიც დაადგენს არის თუ არა მოცემულ v ვექტორში მოცემული p თვისების მქონე ელემენტი. დადებითი პასუხის შემთხვევაში ალგორითმმა უნდა დააბრუნოს პირველივე (დასაწყისიდან) ასეთი ელემენტის ინდექსი, წინააღმდეგ შემთხვევაში კი `v.size()`. 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 <functional>

using namespace std;

template<typename T>
int search(const vector<T> &vec, const T& value, int p, int r)
{
	while (p != r)
	{
		if (vec[p] == value) break;
		++p;
	}
	return p;
}
template<typename T, typename Predicate>
int search_if(const vector<T> &vec, Predicate p)
{
	int i;
	for (i = 0; i < vec.size(); i++)
		if (p(vec[i])) break;
	return i;
}
bool g(int n){ return n < 0; };
int main(){
	vector<int> v = { 21, 43, 5, 7, -23, 45, 12 };//ვექტორი შევქმენით და შევავსეთ

	int index = search_if(v, g);
	//index = search_if(v, bind2nd(less<int>(),0));//ფუნქციის გამოძახება მთელებისთვის
	//index = search_if(v, [](int n){return n < 0; });
```

```
    if (index != v.size())  //თუ მოიძებნა, ანუ  თუ ინდექსი არ გავიდა დიაპაზონიდან
        cout << "Negative found at index " << index << endl;
}
```

`int index = search_if(v, g);` შეტყობინების შემდეგ, სტრიქონი

```
        cout << "Negative found at index " << index << endl;
```

დაბეჭდავს პირველივე უარყოფითი რიცხვის ინდექსს, ხოლო თუ ასეთი არაა, 7-ს. ხოლო თუ გვექნებოდა

```
bool g(int n)
{
    return (n%2 == 1);
}
```

მაშინ დაბეჭდავდა პირველივე კენტი რიცხვის ინდექსს,ან თუ ასეთი არაა მაშინ 7-ს. ■

ფუნქციის არგუმენტად თვისების გამოყენება ძალიან მნიშვნელოვანია, იგი საშუალებას იძლევა შევქმნათ განზოგადებული (`generic`) ალგორითმები. მაგრამ `bool` ტიპის ფუნქციის გადაწოდება არგუმენტად ძალიან პრიმიტიულ მეთოდია. მას ბევრი ნაკლი აქვს, ძირითადი ნაკლი ისაა, რომ ტემპლიტიან ფუნქციაში პარამეტრად არ შეგვიძლია ტემპლიტიანი ფუნქციის გადაწოდება (უფრო სწორად, შეგვიძლია მაგრამ ადვილად არა).

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. ბაინდერების გამოყენებაც ამჟამად უკვე მოძველებულ და შეზღუდულ მეთოდად ითვლება. მეორე, ბევრად უფრო ზოგადი და ეფექტური 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, გადატვირთეთ ძებნის ფუნქცია ისე, რომ მან იმუშავოს მთლიან ვექტორზე.
2. მეორე ამოცანის ფუნქცია გადატვირთეთ ისე, რომ მან მოძებნოს ვექტორის ინდექსების `[l,r)` დიაპაზონში მოცემული `p` თვისების მქონე ელემენტი. დადებითი პასუხის შემთხვევაში ალგორითმმა უნდა დააბრუნოს პირველივე (დასაწყისიდან) ასეთი ელემენტის ინდექსი, წინააღმდეგ შემთხვევაში კი `r`. მოიყვანეთ ფუნქციის გამოძახების რამდენიმე ვარიანტი სხვადასხვა ტიპის ვექტორებისთვის.
3. შექმენით ბულის ტიპის ფუნქციები, რომლებიც გაარკვევს არის თუ არა მთელი რიცხვი სამის, ხუთის და 11-ის ჯერადი. გამოიყენეთ ისინი ძებნის ფუნქციებში.
4. შექმენით ლამბდა ფუნქციები, რომლებიც გაარკვევს არის თუ არა მთელი რიცხვი სამის, ხუთის და 11-ის ჯერადი. გამოიყენეთ ისინი ძებნის ფუნქციებში.
5. დაწერეთ ფუნქცია, რომელიც მოძებნის და დააბრუნებს ამ ვექტორში ინდექსების `[l,r)` დიაპაზონში ელემენტებს შორის მაქსიმალურის ინდექსს.
6. მოცემულია ნამდვილი ან მთელი რიცხვების ვექტორი. დაწერეთ ფუნქცია, რომელიც არგუმენტად გადაცემულ ერთ-ერთ პარამეტრში დააგროვებს ამ ვექტორში ინდექსების `[l,r)` დიაპაზონში ელემენტების
   - ჯამს;
   - კვადრატების ჯამს;
   - შებრუნებული ელემენტების ჯამს (ვიგულისხმოთ რომ არანულოვანებია);
   - საშუალო არითმეტიკულს.

   პარამეტრის საწყისი მნიშვნელობა ფუნქციის ერთ-ერთ პარამეტრად უნდა იქნას გათვალისწინებული.