# თემა 6. არითმეტიკული, ინდექსის და ლოგიკური ოპერატორების გადატვირთვა

## საკითხები:

არითმეტიკული ოპერატორების გადატვირთვა
ინდექსის ოპერატორის გადატვირთვა >>>
ოპერატორიები == და != >>>
სავარჯიშოები >>>

**არითმეტიკული ოპერატორების გადატვირთვა.** ოპერატორების გადატვირთვა უფრო ადვილად აღსაქმელს ხდის მთელ რიგ საკითხებს. საინტერესოა, რომ ოპერატორების გადატვირთვა სასარგებლოა მაშინაც, როდესაც სტანდარტული ბიბლიოთეკის კლასებთან ვმუშაობთ, რომლებსაც ღია წვდომა აქვთ თავიანთ ველებთან. ეს ეხება თითქმის ყველა სახის ოპერატორის გადატვირთვას. თუმცა ამ შემთხვევაში გადატვირთვა ხდება გარე ფუნქციის და არა მეთოდის საშუალებით. განვიხილოთ მაგალითი.

**მაგალითი 3:** კლასი pair<double, double> ჩვენს მიერ გადატვირთული შეკრების და შეტანა-გამოტანის ოპერატორებით.

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

pair<double, double> operator+(const pair<double, double>& a, const pair<double,
double>& b)
{
	pair<double, double> tmp;
	tmp.first = a.first + b.first;
	tmp.second = a.second + b.second;
	return tmp;
}

istream& operator>>(istream& is, pair<double, double>& a)
{
	std::cout << "Enter 2 reals, plz" << std::endl;
	is >> a.first >> a.second;
	return is;
}

ostream& operator<<(ostream& os, const pair<double, double>& a)
{
	os << "first=" << a.first << ",  second=" << a.second << std::endl;
	return os;
}

int main()
{
	pair<double, double> x;
	pair<double, double> a(4.4, 1.2);
	pair<double, double> b;                    // (9.9, 0.1);
	cin >> b;
	x = a + b;
	cout << x;
}
```

სინტაქსი operator+, operator*, operator/= და ა. შ. ზოგადად operator სიტყვაზე მიდგმული ოპერატორით მოქმედების ნიშნით) საშუალებას იძლევა, რომ მომხმარებელმა ოპერატორი გამოიყენოს ამ ოპერატორისთვის მიღებული სინტაქსით.

ასეთი გადატვირთვები, როდესაც უკვე არსებულ კლასს გარედან გადავუტვირთავთ ოპერატორებს, არაა ძალიან გავრცველებული პრაქტიკა. უფრო ბუნებრივი და ხშირია კლასის შექმნის პროცესში ვიზრუნოთ საჭირო ოპერატორების გადატვირთვაზე.

შემდეგ მაგალითში, ვქმნით კლასის თარგს 2-განზომილებიანი წერტილისთვის. წერტილის კოორდინატებს ვუწოდოთ პირველი და მეორე.

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

template<typename T>
class numeric_pair
{
public:
	T first;
	T second;
	numeric_pair() {}
	numeric_pair(const T&, const T&);
	template<typename S>
	friend istream& operator>>(istream& , numeric_pair<S>&);
	template<typename S>
	friend ostream& operator<<(ostream&, const numeric_pair<S>&);
	void operator+=(const numeric_pair&);
	void operator *=(const T&);

};
template<typename T>
numeric_pair<T>::numeric_pair(const T& a, const T& b) : first(a), second(b) {}

template<typename S>
istream& operator>>(istream& is, numeric_pair<S>& a)
{
	std::cout << "Enter valiues for the first and second, plz" << std::endl;
	is >> a.first >> a.second;
	return is;
}

template<typename S>
ostream& operator<<(ostream& os, const numeric_pair<S>& a)
{
	os << "first=" << a.first << ",  second=" << a.second << std::endl;
	return os;
}

template<typename T>
void numeric_pair<T>::operator+=(const numeric_pair& other)
{
	first += other.first;
	second += other.second;
}

template<typename T>
void numeric_pair<T>::operator*=(const T& scalar)
{
	first *= scalar;
	second *= scalar;
}

int main()
{
```

```
	numeric_pair<int> a;
	cin >> a;
	cout << a << endl;

	numeric_pair<int> b(1,1);
	a += b;
	cout << "After addition: " << endl;
	cout << a << endl;

	a *= 10;
	cout << "After multiplication: " << endl;
	cout << a << endl;
}
```

პროგრამის გაშვება გვაჩვენებს თუ რისი გაკეთება შეუძლია ამ მარტივ კლასს. სავარჯიშოებში ნაჩვენებია თუ რისი გაკეთება არ შეუძლია მას. იქვე იქნება შემოთავაზებული შესაძლო არადანები.

## <<< ინდექსის ოპერატორის გადატვირთვა

განხილული მაგალითი იმ კატეგორიას მიეკუთვნება, როდესაც ძალიან სასარგებლოა ინდექსის შექმნა, ანუ [] ოპერატორის გადატვირთვა.

**მაგალითი 2** /pair<double, double> კლასზე გადატვირთული ინდექსის ოპერატორი/. განხილულ მაგალითში გადავტვირთოთ ინდექსის ოპერატორი, რაც საშუალებას მოგვცემს ობიექტის ველებს მივმართოთ იმდექსის საშუალებით, ისევე როგორც ვექტორის ობიექტებში. სხვა სიტყვებით, თუ x წყვილია, x.first იგივე იქნება რაც x[0] და ა.შ.

**ამოხსნა:** კლასის განაცხადში დავამატოთ სტრიქონები:

```
	T& operator[](int i);
	const T&  operator[](int i) const;
```

ხოლო იმპლემენტაციაში:

```
template<typename T>
T& numeric_pair<T>::operator[](int i)
{
	return *((T*)this + i);
}

template<typename T>
const T& numeric_pair<T>::operator[](int i) const
{
	return *((T*)this + i);
}
```

this არის პოინტერი, რომელიც გვიბრუნებს გამომძახებელი ობიექტის მისამართს. რადგან თითო ობიექტი შედგება ორი T-სგან, ამიტომ *this+1 იქნება ობიექტის მეორე ველის მისამართი.

დარჩა ერთი მომენტი. ჩვენ უდა გადავტვირთოთ ამ ოპერატორის მუდმივი და მუტირებადი ვარიანტი. სინტაქსი ისეა როგორც მოცემულია მაგალითში. შედეგად, სრულიად ლეგალურია კოდი:

```
int main()
{
	numeric_pair<int> b(1,1);
	cout << "b[1]= " << b[1] << endl;
	b[2] = 222;
	cout << "b[2]= " << b[2] << endl;
}
```

თუ რომელიმე წყვილს გამოვაცხადებდით მუდმივად, კომპილერი მიგვითითებს შეცდომას ყოველთვის, როდესაც მისი მნიშვნელობის შეცვლას დავაპირებთ.

<<< ოპერატორიები == და != . ამ ორი ოპერატორის გადატვირთვას ყოველთვის აქვს აზრი, თუ მოსალოდნელია, რომ ამ კლასის ობიექტების შედარება დაგვჭირდება. განვიხილოთ მარტივი

მაგალითი: შევქმნათ ტბის კლასი. ფაილში მოთავსებული მონაცემებით ავაგოთ ტბის რამდენიმე ობიექტი, ჩავწეროთ ისინი ვექტორში, შემდეგ მოვძებნოთ ამ ვექტორში კიდევ ერთი ტბა, რომლის მონაცემებსაც შევიყვანთ კლავიატურიდან.

ვთქვათ მონაცემების ფაილს, რომელსაც ჰქვია "data.txt", აქვს სახე:

```
Paravani     37.5   2073   3.3
Paliastomi   18.2   -0.3   3.2
Tabackuri    14.2   1997   40.2
Jandari      10.6   291    7.2
Ritsa        1.49   884    101
Bazaleti     1.22   879    7
```

შევქმნათ კლასი ტბისთვის, რომელსაც ექნება ოთხი კერძო ველი შესაბამისი სახელებით, ექნება ღია წევრი-ფუნქციები (მეთოდები) ველებთან სამუშაოდ, ოპერატორები და კონსტრუქტორები.

კლასს, იმპლემენტაციას და პროგრამა დრაივერს შესაძლოა ჰქონდეს შემდეგი სახე:

```
#pragma once
#include<iostream>
#include<fstream>
#include<algorithm>
#include<string>
#include<vector>

using namespace std;

class lake
{
public:
	lake(void);
	~lake(void);
private:
	string name;
	double area; // km^2
	double height;	// meter
	double depth;	// meter
public:
	lake(string itsName, double itsArea, double itsHeight, double itsDepth);
	string getName(void) const;
	double getArea(void) const;
	double getHeight(void) const;
	double getDepth(void) const;
	void setName(string itsName);
	void setArea(double itsArea);
	void setHeight(double itsHeight);
	void setDepth(double itsDepth);
	bool operator==(lake other);
	bool operator!=(lake other);
	friend istream& operator>>(istream& is, lake& x);
	friend ostream& operator<<(ostream& os, lake& x);
};
```

```
lake::lake(void)
	: area(0)
	, height(0)
	, depth(0)
{
}
lake::~lake(void)
{
}
lake::lake(string itsName, double itsArea, double itsHeight, double itsDepth)
{
	name = itsName;
	area = itsArea;
	height = itsHeight;
	depth = itsDepth;
}
string lake::getName(void) const
{
	return name;
}
double lake::getArea(void) const
{
	return area;
}
double lake::getHeight(void) const
{
	return height;
}
double lake::getDepth(void) const
{
	return depth;
}
void lake::setName(string itsName)
{
	name = itsName;
}
void lake::setArea(double itsArea)
{
	area = itsArea;
}
void lake::setHeight(double itsHeight)
{
	height = itsHeight;
}
void lake::setDepth(double itsDepth)
{
	depth = itsDepth;
}
bool lake::operator==(lake other)
{
	return (name == other.name
		&& area == other.area
		&& height == other.height
		&& depth == other.depth);
}
bool lake::operator!=(lake other)
{
	return !(*this == other);
}
istream& operator>>(istream& is, lake& x)
{
```

```
	is >> x.name >> x.area >> x.height >> x.depth;
	return is;
}
ostream& operator<<(ostream& os, lake& x)
{
	os << x.name << '\t' <<
		x.area << '\t' <<
		x.height << '\t' <<
		x.depth << '\t' << endl;
	return os;
}

int main()
{
	ifstream ifs("data.txt");

	vector<lake> v;
	lake tmp;
	while (ifs >> tmp)
		v.push_back(tmp);

	for (int i = 0; i<v.size(); i++)
		cout << v[i] << endl;

	lake lk;
	cout << "Enter data of a lake, plz" << endl;
	cin >> lk;
	vector<lake>::iterator it;
	it = find(v.begin(), v.end(), lk);
	if (it != v.end())
		cout << "Found value " << *it << endl;
	else
		cout << "Not found!" << endl;
}
```

შედარების ოპერატორი `operator!=()` განიმარტება როგორც ტოლობაზე შედარების უარყოფა. ასე უფრო მოკლეა, თუმცა შეიძლებოდა წევრ-წევრად შეგვედარებინა.

რომელიც გვიჩვენებს, რომ ჩვენს მიერ შექმნილი კლასის ობიექტებთან ისევე ურთიერთობს კომპილერი, როგორც ენაში ჩაშენებულ მთელი და ნამდვილი რიცხვების ტიპებთან. კერძოდ, შეტანა-გამოტანა, ვექტორის შევსებს, ძებნის ალგორითმი სტანდარტული ბიბლიოთეკიდან, რომელიც აბრუნებს იტერატორს.

მაგრამ ამ კლასის ფუნქციონალობა მაინც შეზღუდულია. კერძოდ, არ შეგვიძლია გამოვიყენოთ დახარისხების და ზოგიერთი კიდევ სხვა ალგორითმი. მიზეზი იმასი მდგომარეობს, რომ ჩვენს კლასზე განსაზღვრული არაა შედარების ოპერატორები (მეტობა-ნაკლებობის ტიპის). ეს საკითხი შესაძლოა გადაიჭრას ოპერატორების გადატვირთვით, მაგრამ ბევრად უფრო თანამედროვე და მარტივი არის ლამბდა ფუნქციების გამოყენება ამ მიზნით.

მაგალითად, წარმოვიდგინოთ რომ გვინდა ამ ვექტორში ჩაწერილი ტბების დახარისხება ჯერ ფართობის, ხოლო შემდეგ სიღრმის მიხედვით და ორივე შემთხვევაში გამობეჭდვა.

ამის მიღწევა შეიძლება დახარისხების ალგორითმში მესამე არგუმენტად ლამბდა ფუნქციის გადაწოდებით, რომელიც იქმნება უშუალოდ გამოძახების წინ, როგორც შემდეგ პროგრამაშია:

```
int main()
{
	ifstream ifs("data.txt");

	vector<lake> v;
	lake tmp;
```

```
    while (ifs >> tmp)
        v.push_back(tmp);

    auto lm = [](const lake& a, const lake& b) {return a.getArea() < b.getArea(); };
    sort(v.begin(), v.end(), lm);

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

    sort(v.begin(), v.end(), [](const lake& a, const lake& b) {return a.getDepth() <
        b.getDepth(); });

    for (auto m : v)
        cout << m;
}
```

<<< სავარჯიშოები:

1. cplusplus.com -ზე და google-ში ნახეთ გადაჭრილი ამოცანები ოპერატორების გადატვირთვის შესახებ.
2. კლასში `pair<double, double>` გადატვირთეთ წყვილის ნამდვილ რიცხვზე გამრავლების ოპერატორი.
3. `numeric_pair` კლასის მაგალითში, ვერ ჩავწერთ ასეთ რამეს: `(a *= 10) *=11;` ანალოგიური მდგომარეობაა += ოპერატორებთან დაკავშირებით. ეს იმის გამო, რომ ოპერატორის დასაბრუნებელი მნიშვნელობა void არის. გასინჯეთ ასეთი პროტოტიპი
   `numeric_pair& operator *=(const T&);`
   და იმპლემენტაცია:

```
template<typename T>
numeric_pair<T>& numeric_pair<T>::operator*=(const T& scalar)
{
    first *= scalar;
    second *= scalar;
    return *this;
}
```

   ანალოგიური ექსპერიმენტიჩაატარეთ += ოპერაციასთან დაკავშირებით.
4. ასეთივე კლასისთვის, კლასის მეთოდები განახორციელეთ კლასის შიგნით. შეადარეთ თუ რამდენად გამარტივდა კოდი.
5. შექმენით ასეთივე კლასი, ოღონდ კერძოდ ნამდვილი რიცხვების წყვილებისთვის. შეადარეთ თუ რამდენად მარტივია კოდი კლასის თარგთან შედარებით.
6. შექმენით კლასი tripleInt, რომელიც შედგება მთელი ველებისგან. მაგალითი 3-ის cplusplus.com-ის და დახმარებით შეეცადეთ გადატვირთოთ შეტანა-გამოტანის ოპერატორები.
7. შექმენით კლასი ზღვისთვის, ქალაქისთვის და მთისთვის, გადატვირთეთ შეტანა-გამოტანის და ლოგიკური ოპერატორები. ამ ტიპის ობიექტებზე ჩაატარეთ განხილული ამოცანების მსგავსი მანიპულაციები.
8. განსაზღვრეთ კლასი ნამდვილი რიცხვებისგან შედგენილი სამეულისთვის. გადატვირთეთ შეტანა-გამოტანის და ლოგიკური ოპერატორები. ამ ტიპის ობიექტებზე ჩაატარეთ განხილული ამოცანების მსგავსი მანიპულაციები.
9. შექმენით კლასი ზღვისთვის, ქალაქისთვის და მთისთვის, წინაპარი კლასით და მის გარეშე. შეტანა გამოტანისთვის გაითვალისწინეთ ოპერატორები.