თემა 5. მუდმივი, setter, getter და მეგობარი ფუნქციები. შეტანა-გამოტანის ოპერატორების გადატვირთვა

საკითხები:

- ობიექტზე ორიენტირებული პროგრამირების (მოკლედ ოოპ) ძირითადი იდეები
- მეთოდების დამალვა. setter, getter და მუდმივი მეთოდები >>>
- მეგობარი ფუნქცია. შეტანა გამოტანის ოპერატორების გადატვირთვა >>>
- სავარჯიშოები >>>

ობიექტზე ორიენტირებული პროგრამირების (მოკლედ *ოოპ*) ძირითადი იდეები

ობიექტზე ორიენტირებული პროგრამირება (**ოოპ**) არის დაპროგრამების პარადიგმა (მოდელი, ტიპური ნიმუში), რომელიც სხვა სტილებისგან გამოირჩევა იმით, რომ მეტადაა განვითარებული და დამუშავებული მონაცემის (data) ცნება. თავის მხრივ, მონაცემის აღწერისა და დამუშავებისთვის განვითარებულია სამი ძირითადი ცნება (concept):

- ✓ მონაცემთა ინკაპსულაცია;
- ✓ მემკვიდრეობა;
- ✓ პოლიმორფიზმი.

ინკაპსულაცია. C/C++ ენების ყოველ პრიმიტიულ ტიპთან განუყრელადაა დაკავშირებული ის ოპერაციები, რისი განხორციელებაც ნებადართულია (და დაპროგრამებულია) ამ ტიპზე. პრაქტიკულად, ტიპი არის სიმრავლე მასზე განსაზღვრულ ოპერაციებთან ერთად. მაგალითად, მთელ რიცხვებზე შეკრება სხვანაირადაა დაპროგრამებული და სიმბოლოებზე სხვანაირად, თუმცა გამოიყენება ერთი და იგივე ოპერატორი (მოქმედების ნიშანი). ენა იძლევა იმის საშუალებას, რომ ერთი და იგივე სახელის მქონე რამდენიმე ოპერატორი ან ფუნქცია არსებობდეს და ისინი სხვადასხვა ტიპის არგუმენტების შემთხვევაში სხვადასხვანაირად მოქმედებდეს. ამას გადატვირთვა ჰქვია და C++ ენაში ძალიან ხშირად გამოიყენება. ნებისმიერ შემთხვევაში, პრიმიტიული ტიპი დ მასზე განსაზღვრული ოპერაციები ერთ მთლიანობას წარმოადგენს.

მათემატიკაში. მთელი რიცხვების ან მომელიმე სხვა სიმრავლის განმარტების შემდეგ, როგორც წესი, ამბობენ თუ რა მოქმედებია დაშვებული ამ სიმრავლეზე. ამ შემთხვევაში სიმრავლე და ოპერაციები ერთ მთლიანობას არ ქმნიან. უბრალოდ ბუნებრივად ჯგუფდებიან ერთად. ზოგად ალგებრაში, ტერმინი ალგებრა დაახლოებით იგივეს ნიშნავს რასაც ვხედავთ პრიმიტიული ტიპების მაგალითზე - სიმრავლე და ოპერატორები ერთი მთლიანობაა.

ინკაპსულაცია ნიშნავს, რომ მონაცემები და მონაცემებზე განსაზღვრული ფუნქციები (რომელებიც იმპელემენტაციაში ითვალისწინებენ მონაცემების კერძო თავისებურებებს) ერთ მთლიანობას ქმნის. როგორც წესი, ასეთი ერთიანობა ფორმდება კლასის საშუალებით. კლასის ყოველი წარმომადგენელი არის კონკრეტული მონაცემი, რომელსაც შეუძლია მიმართვა კლასში გაერთიანებულ ფუნქციებზე - მეთოდებზე.

პრაქტიკული ინტერესის მქონე ამოცანებში, კლასში გაერთიანებული (ინკაპსულირებული) მონაცემების და ფუნქციების მნიშვნელოვან ნაწილზე წვდომა შეზღუდულია სხვა კლასების ობიექტებისთვის და გლობალური ფუნქციებისთვის. **ინფორმაციის დამალვა** - მნიშვნელოვანი თემაა ოოპ-ში და მჭიდროდაა დაკავშირებული ინკაპსულაციის იდეასთან.

მემკვიდრეობა. იდეა უკავშირდება კოდის მრავალჯერადად გამოყენების საკითხს. C++ იძლევა საშუალებას, რომ არსებული კლასების საფუძველზე აიგოს (იწარმოოს) სხვა კლასები. წარმოებული კლასების ობიექტები ორი ნაწილისგან შედგება: ერთი ნაწილი არის ფუძე (საბაზო, წინაპარი) კლასის ობიექტი, ხოლო მეორე ნაწილი აიგება წარმოებული კლასის კონსტრუქტორით. C++ იძლევა კლასების რთული იერარქიის აგების საშუალებას.

პოლიმორფიზმი. ეს სიტყვა ბერძნული წარმოშობისაა და ნიშნავს სხვადასხვა ფორმის მიღებას. კლასების იერარქიაში ერთი სახელის მქონე წევრ-ფუნქციას შეუძლია ჰქონდეს სხვადასხვანაირი ყოფაქცევა.

<<< მუდმივი, setter და getter მეთოდები

ჩვენ განვიხილეთ ერთი მაგალითი, როდესაც კლასის წევრის (სტატიკური ველის) დახურვა აუცილებელი გახდა **private** (დახურული, კერძო) *ელემენტებზე წვდომის სპეციფიკატორის* გამოყენებით. ვიდრე განვიხილავთ შედეგებს, რაც როგორც წესი მოყვება კლასის წევრების დახურვას, განვიხილოთ კიდევ ერთი გავრცელებული შემთხვევა. მაგალითად, თუ ჩვენ ვქმნით კლასს დროის რაიმე კერძო ფორმატით დამუშავებითვის, ვთქვათ 23:11:54 (ანუ day:hour:min), მაშინ მხედველობაში უნდა ვიქონიოთ, რომ რომელიმე ველის ცვლილებამ ადვილად შეიძლება ფორმატის კორექტულობა დაარღვიოს. მაგალითად, არ არსებობს 25:11:54 ან 23:11:65. ამიტომ ასეთ ამოცანებში მიზანშეწონილია გაუთვალისწინებელი შეღწევების აკრძალვა. თუ აუცილებელი იქნება რომელიმე ფუნქციის მნიშვნელობის შეცვლა, მაშინ გამოვიყენებთ სპეციალურად ამ მიზნით შექმნილ ფუნქციას, რომელიც იქნება ღია (`public`) მოხმარების.

მაგალითი 1. ტბის კლასი ღია ველებით

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

class Lake
{
public:
	string name;
	int area;
	Lake() {}
	Lake(string itsName, int itsArea) :name(itsName), area(itsArea) {}
	string toString() const;
	{
		return ((string)typeid(this).name() + ":  name: " + name + ", area: " +
			to_string(area));
	}
};
int main()
{
	Lake tba("Pali",1230);
	cout << tba.toString() << endl;
	Lake* p = new Lake("Ohoho",1000);
	cout << p->toString() << endl;
}
```

ვიდრე დავხურავთ კლასის წევრ-მონაცემებს (ველებს) და მათთან სამუშაოდ შევქმნით ღიად მიწვდომად წევრ-ფუნქციებს (მეთოდებს), შევნიშნოთ რომ C++ ენაში კლასი ხშირად იყოფა ორ ნაწილად - განაცხადად და იმპლემენტაციად. ამ მარტივი კლასის შემთხვევაში:

```
class Lake
{
public:
	string name;
	int area;
	Lake() {}
	Lake(string, int);
	string toString() const;;
};
Lake::Lake(string itsName, int itsArea) :name(itsName), area(itsArea) {}
string Lake::toString() const;
{
```

```
	return ((string)typeid(this).name() + ":  name: " + name + ", area: " +
		to_string(area));
}
```

`const;` გვეუბნება, რომ მეთოდი არ ცვლის გამომძახებელი ობიექტის ველებს.

ასეთი მიდგომის დროს კლასის წევრებს ადვილად აღვიქვამთ, სამაგიეროდ რთულდება მეთოდების სახელები, რადგან (კოლიზიების თავიდან ასაცილებლად) უნდა მივუთოთ კლასის სახელიც. მაგალითად, ტბის კლასის კონსტრუქტორი. ოთხი წერტილი არის C++ -ში ძალიან გავრცელებული ხილვადობის ოპერატორი, რომლის გამოყენების რამდენიმე განსხვავებულ მაგალითს კიდევ შევხვდებით.

ახლა დავხუროთ კლასის ორივე ველი. ამის შემდეგ იმისათვის რომ შევცვლოთ ობიექტების ველები, ან ვისარგებლოთ მათი მნიშვნელობებით, უნდა შევქმნათ ფუნქციები, რომლებიც იქნება ღია წვდომის:

მაგალითი 2. ტბის კლასი დახურული ველებით

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

class Lake
{
private:
	string name;
	int area;
public:
	Lake() {}
	Lake(string, int);
	void setName(string);
	void setArea(int);
	string getName(void) const;
	int getArea(void) const;
	string toString() const;
};
Lake::Lake(string itsName, int itsArea) :name(itsName), area(itsArea) {}
void  Lake::setName(string itsName) { name = itsName; }
void  Lake::setArea(int itsArea) { area = itsArea; }
string Lake::getName(void) const { return name; }
int Lake::getArea() const { return area; }
string Lake::toString() const
{
	return ((string)typeid(this).name() + ":  name: " + name + ", area: " +
		to_string(area));
}

int main()
{
	Lake tba("Pali",1230);
	if(tba.getName() == "Pali")
		cout << tba.toString() << endl;

	Lake* p = new Lake("Ohoho",1000);
	if(p->getArea() == 999)
		cout << p->toString() << endl;
}
```

როგორც ვხედავთ, კლასში გაიზარდა იმ წევრების რაოდენობა, რომლებსაც აქვთ `public:` (ღია) წვდომა. სამაგიეროდ, კოდი გახდა უფრო დაცული. ობიექტზე ორიენტირებულ პროგრამირებაში ძალიან ხშირად თითქმის ავტომატურ რეჟიმში ხდება ველების დახურვა. სხვა

სტილებში სხვა ვითარებაა. გვხვდება ისეთი შემთხვევებიც როდესაც საჭირო არის წევრი ფუნქციის დახურვა. ასეთ მაგალითებს მოგვიანებით შევხვდებით.

რამდენიმე ფუნქცია გამოვაცხადეთ მუდმივად. ფუნქციისათვის ატრიბურტი მუდმივი ნიშნავს, რომ იგი არაფერს არ გადააკეთებს პროგრამაში, მხოლოდ ნახულობს გარკვეულ ინფორმაციას. ჩვენს მაგალითებში `const` ფუნქციებს არგუმენტი არ სჭირდებათ. უფრო რთულ მაგალითებში, როდესაც არგუმენტი საჭიროა, ის უნდა იყოს აგრეთვე მუდმივი ობიექტი.

ფუნქციებს, რომლებიც აყენებენ ველების მნიშვნელობებს, setter-ები ეწოდებათ, ხოლო რომლებიც ამოიღებენ ველების მნიშვნელობებს, getter-ები ეწოდებათ.

არვითარი აუცილებლობა არაა, რომ setter-ს სახელად `set`რაღაცა ერქვას. ძალიან ხშირად, ამოცანის შინაარსი განსაზღვრავს თუ რა უნდა ერქვას ფუნქციას, რომელიც ველის მნიშვნელობას ცვლის. იგივეა სამართლიანი setter-ების შემთხვევაში.

<<< შეტანა-გამოტანის ოპერატორების გადატვირთვა. მეგობარი ფუნქცია.

C++ ენაში ოპერატორების გადატვირთვა ძლიერად არის მხარდაჭერილი სხვა ენებთან შედარებით. ფუნქციების გადატვირთვა უკვე განვიხილეთ. ოპერატორების გადატვირთვაც რამდენჯერმე ვახსენეთ + ოპერატორის მაგალითზე: მონაცემთა რამდენიმე ტიპზე (მთელები, ნამდვილები, სტრინგები, სიმბოლოები) არის ეს ოპერატორი განსაზღვრული, მაგრამ სხვადასხვა ტიპზე სხვადასხვანაირად არის დაპროგრამებული (იმპლემენტირებული). უფრო მეტი, ჩვენ შეგვიძლია შეკრება განვსაზღვროთ (გადავტვირთოთ) მომხმარებლის მიერ შექმნილ კლასებზეც (მაგალითად, ნამდვილი რიცხვების წყვილები).

C++ საშუალებას გვაძლევს გადავტვირთოთ თითქმის ყველა ოპერატორი, რაც ამ ენაში გამოიყენება სტანდარტული ტიპებისთვის ან ბიტური ოპერაციებისთვის. შესაძლებელია, თუმცა ზოგიერთი მათგანი თითქმის არასოდეს გამოიყენება გადატვირთვისთვის. ჩვენ გადავტვირთავთ მხოლოდ ყველაზე ხშირად გამოყენებად ოპერატორებს, თუმცა მოვიყვანთ უნარული და ბინარული ოპერატორების ცხრილს, რომლების გადატვირთვაც შეგვიძლია. :

ერთადგილიანი (უნარული ოპერატორები):

Operator	Name
++	Increment
--	Decrement
*	Pointer dereference
->	Member selection
!	Logical NOT
&	Address-of
~	One's complement
+	Unary plus
–	Unary negation
Conversion operators	Conversion operators

ორადგილიანი ოპერატორები:

Operator	Name
,	Comma
!=	Inequality
%	Modulus
%=	Modulus/assignment
&	Bitwise AND

&&	Logical AND
&=	Bitwise AND/assignment
*	Multiplication
*=	Multiplication/assignment
+	Addition
+=	Addition/assignment
-	Subtraction
-=	Subtraction/assignment
->*	Pointer-to-member selection
/	Division
/=	Division/assignment
<	Less than
<<	Left shift
<<=	Left shift/assignment
<=	Less than or equal to
=	Assignment
==	Equality
>	Greater than
>=	Greater than or equal to
>>	Right shift
>>=	Right shift/assignment
^	Exclusive OR
^=	Exclusive OR/assignment
|	Bitwise inclusive OR
|=	Bitwise inclusive OR/assignment
||	Logical OR
[]	Subscript operator

იმისათვის, რომ გადავტვირთოთ ოპერატორი, უნდა შევქმნათ მისი შესაბამისი კერძო სახის ფუნქცია, რომლის სახელი იწყება სიტყვით operator და მთავრდება გადასატვირთი ოპერატორის სახელით. ეს ფუნქციები უმეტეს შემთხვევაში არის კლასის წევრები, ხოლო უფრო იშვიათად, - **მეგობარი** ფუნქციები - რომლებიც წევრები არაა, მაგრამ აქვთ წვდომა კლასის ობიექტების წევრებზე.

კიდევ ერთხელ განვიხილოთ „ტბის“ კლასი, ამჯერად გამდიდრებული შეტანა-გამოტანის ოპერატორებით:

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

class Lake
{
private:
	string name;
	int area;
public:
```

```
	Lake() {}
	Lake(string itsName, int itsArea) :name(itsName), area(itsArea) {}
	void setName(string itsName) { name = itsName; }
	void setArea(int itsArea) { area = itsArea; }
	string getName(void) const { return name; }
	int getArea() const { return area; }
	string toString() const
	{
		return ((string)typeid(this).name() + ":  name: " + name + ", area: " +
		    to_string(area));
	}
	friend istream& operator>>(istream& is, Lake& a)
	{
		cout << "Enter name and area, plz" << endl;
		is >> a.name >> a.area;
		return is;
	}
	friend ostream& operator<<(ostream& os, const Lake& a)
	{
		os << a.name << "    " << a.area << endl;
		return os;
	}
};

int main()
{
	Lake tba("Pali", 1230);
	if (tba.getName() == "Pali")
		cout << tba.toString() << endl;

	Lake* p = new Lake("Ohoho", 1000);
	if (p->getArea() == 999)
		cout << p->toString() << endl;

	Lake a;
	//operator>>(cin, a);
	cin >> a;
	cout << a;
}
```

უპირველეს ყოვლისა, შეტყობინება `cin >> a;` იგივეა რაც მის წინა სტრიქონი, რომელიც დაკომენტარებულია. უბრალოდ, C++ საშუალებას გვაძლევს, რომ ფუნქციის გამოძახების ნაცვლად გამოვიყენოთ უფრო მარტივი და შეჩვეული ჩანაწერი.

შეტანა-გამოტანის ოპერატორები კლასის წევრები არაა, მაგრამ იმისთვის რომ მიირონ წვდომა კლასის წევრებზე, მათზე განაცხადს ემატება განმსაზღვრელი ტერმინი `friend`.

აქ ორი მომენტია ყურადსაღები.

იმისათვის, რასაც ამ ამოცანაში აკეთებს, ეს ოპერატორები შეგვეძლო `void` ტიპის გაგვეკეთებინა. მაგრამ მაშინ ერთი ნაკადით ვერ შევიტანდით ორ ნა მეტ მონაცემს. მაგალითად ასეთს:

```
	cin >> a >> b >> c
```

აქ, პირველი შეტანა `cin >> a` არის იგივე

```
	operator>>(cin, a)
```

რომელიც რეფერენსით აბრუნებს ნაკადს. ეს ნიშნავს, რომ მთლი ეს გამოსახულება `operator>>(cin, a)` (ანუ `cin >> a`) გაიგივდება `cin` ნაკადთან, ანუ `cin >> a >> b` იგივეა რაც `cin >> b` და ა.შ..

მეორე მომენტი ისაა, რომ შეგვიძლია ეს ოპერატორი გავაკეთოთ კლასის წევრადაც. მაგალითად,

```
istream& operator>>(istream& is)
{
	cout << "Enter name and area, plz" << endl;
	is >> name >> area;
	return is;
}
```

ამ ჩანაწერში არ ჩანს ობიექტის ველები, რადგან ობიექტი არგუმენტი არაა. ჩანს კლასის ველების სახელები, რომელიც ოპერატორის გამოძახების დროს ჩანაცვლდება გამომძახებელი ობიექტის ველებით.

საქმე ისაა, რომ ამ შემთხვევაში მოგვიწევს მონაცემების ძალიან უცნაური სახით შეყვანა:

```
a >> cin;
```

სხვანაირად არ შეიძლება, რადგან წევრი ოპერატორი უნდა გამოიძახოს ობიექტმა და ეს ბოლო ჩანაწერი იგივეა რაც:

```
a.operator>>(cin);
```

ანალოგიური მოსაზრებები არის ძალაში გამოტანის ოპერატორთან დაკავშირებით.

<<< სავარჯიშოები

1. წინა მეცადინობზე განხილული სავარჯიშოების კლასებისთვის დახურეთ ველები `private`: -ის გამოყენებით. შექმენით ღია წვდომის ფუნქციები მათთან სამუშაოდ.
2. იგივე ამოცანებში გადატვირთეთ შეტანა-გამოტანის ოპერატორები. შექმენით ობიექტების ვექტორები და ფაილიდან შეავსეთ ისინი გადატვირთული შეტანის ოპერატორის გამოყენებით.
3. შეეცადეთ დამოუკიდებლად ჩამოაყალიბოთ ობიექტზე ორიენტირებული პროგრამირების ძირითადი იდეები. მოძებნეთ შესაბამისი მასალა ინტერნეტში.