თემა 4. კლასის განსაკუთრებული წევრი-ფუნქციები. lvalue და rvalue მნიშვნელობების რეფერენსები

განხილული საკითხები:

- ასლის კონსტრუქტორი >>>
- ასლის მინიჭების ოპერატორი >>>
- ასლის ოპერაციები ერთმანეთისგან დამოუკიდებელია >>>
- lvalue -ზე და rvalue-ზე რეფერენსები >>>
- გადაადგილებასთან დაკავშირებული კონსტრუქტორები >>>
- სავარჯიშოები >>>

C++ -ის ოფიციალურ ტერმინოლოგიაში წევრი-ფუნქცია ითვლება განსაკუთრებულად, თუ C++ თვითონ წარმოქმნის მას. C++98 შეიცავს ოთხ ასეთ ფუნქციას: კონსტრუქტორი გაჩუმების პრინციპით, დესტრუქტორი, ასლის კონსტრუქტორი და ასლის მინიჭების ოპერატორი. მაგალითად, თუ `Widget` არის რაიმე კლასი, მის განსაკუთრებულ წევრებზე განაცხადი კეთდება ასე:

```
class Widget {
public:
        …
        ~Widget();                      // მომხმარებლის მიერ შექმნილი დესტრუქტორი
        …
        Widget(const Widget&);                  // ასლის კონსტრუქტორი
        Widget& operator=(const Widget&)        // ასლის მინიჭების ოპერატორი
        …
};
```

ეს ფუნქციები იქმნება მხოლოდ მაშინ, როდესაც ისინი საჭიროა. კონსტრუქტორი გაჩუმების პრინციპით წარმოიქმნება მხოლოდ იმ შემთხვევაში, თუ კლასში განაცხადი არ არის არც ერთ სხვა კონტრუქტორზე. წარმოქმნილი განსაკუთრებული წევრი-ფუნქციები არაცხადად არის `public` და `inline`. სხვა მახასიათებლებს დავუბრუნდებით ვირტუალური ფუნქციების შესწავლის პროცესში.

C++11-ში შემოვიდა ორი ახალი წევრი-ფუნქცია: გადაადგილების კონსტრუქტორი და გადაადგილებულის მინიჭების ოპერატორი. მათ ხელმოწერებს შემდეგი სახე აქვთ:

```
class Widget {
public:
        …
        Widget(Widget&& rhs);                   // გადაადგილების კონსტრუქტორი
        Widget& operator=(Widget&& rhs);        // გადაადგილებულის მინიჭების ოპერატორი
        …
};
```

განვიხილოთ ტიპური საკითხები:

- როგორ შევქმნათ განსაკუთრებული წევრი-ფუნქციები,
- ჩვენს გარეშე როგორ და როდის იქნება ისინი,
- როგორ დავეთანხმოთ ნაგულისხმევი განსაკუთრებული წევრების შექმნას,
- რას ვერ ოითვალისწინებენ ნაგულისხმევი წევრები,
- როგორ ავკრძალოთ ნაგულისხმევი წევრების შექმნა თუ არ გვინდა მათი გამოყენება.

თვალსაჩინოებისთვის მოსახერხებელია ვისარგებლოთ შემდეგი მარტივი კლასით, რომელსაც მისი შინაარის გამო ზოგჯერ მოვიხსენიებთ როგორც „ჭიჭყინა მთელი“.

```
#include<iostream>
#include<string>
```

```
using namespace std;

class noisyInt
{
private:
	static int noisyInt_count;
public:
	int	value;
	noisyInt()
	{
		++noisyInt_count;
		cout << "New noisyInt created. noisyInt_count=" << noisyInt_count << endl;
	}
	noisyInt(int itsValue) :value(itsValue)
	{
		++noisyInt_count;
		cout << "New noisyInt created. noisyInt_count=" << noisyInt_count << endl;
	}
	~noisyInt()
	{
		--noisyInt_count;
		cout << "Deleted: " << toString() << " noisyInt_count=" << noisyInt_count
			<< endl;
	}
	string toString()
	{
		return ((string)typeid(this).name() + ":  value: " + to_string(value));
	}
};

int noisyInt::noisyInt_count = 0;
int main()
{
	noisyInt n;
	n.value = 16;
	cout << n.toString() << endl;

	noisyInt* p = new noisyInt(77);
	delete p;
	p = nullptr;
}
```

ვიდრე ძირითად თემაზე გადავალთ, გვერდით ეფექტად შეგვიძლია შევამოწმოთ, რომ დინამიკურად შექმნილი ობიექტი თავისით არ წაიშლება (საკმარისია დავაკომენტაროთ ბოლო ორი სტრიქონი).

<<< ასლის კონსტრუქტორი. როგორც წინა თემის ბოლოს ვნახეთ, საკმარისია პროგრამა დრაივერში გამოჩნდეს ჩვეულებრივი მთელებისთვის კარგად ცნობილი განაცხადი:

```
int noisyInt::noisyInt_count = 0;
int main()
{
	noisyInt n;
	n.value = 16;
	cout << n.toString() << endl;

	noisyInt b(n);
}
```

რომ მაშინვე აგვერევა „ჭიჭყინა მთელების“ რაოდენობის აღრიცხვა:

```
New noisyInt created. noisyInt_count=1
```

```
class noisyInt *:  value: 16
Deleted: class noisyInt *:  value: 16 noisyInt_count=0
Deleted: class noisyInt *:  value: 16 noisyInt_count=-1
Press any key to continue . . .
```

მიზეზი იმაშია, რომ ახალი ობიექტის აგება (კონსტრუირება) მოხდა ასლის საფუძველზე. ეს კონსტრუქტორი არის განსაკუთრებული წევრი-ფუნქცია. შესაბამისად, რადგან ჩვენ არ შეგვიქმნია მაგრამ დავაპირეთ გამოყენება, ამიტომ C++ -მა თვითონ შექმნა. ცხადია, მან ვერ გაითვალისწინა სხვა რამეები, რაც ჩვენ უნდა გავითვალისწინოთ ამ კონსტრუქტორის შექმნისას.

ლექციის დასაწყისში აღვნიშნეთ, რომ ასლის კონსტრუქტორს რაიმე `Widget` კლასისთვის უნდა ჰქონდეს სახე:

```
Widget(const Widget&);
```

ეს ფორმა ერთდროულად არის დაცული და ეფექტური. მის შინაარსს ჩვენ კიდევ შევეხებით რეფერენსების მონაკვეთში.

„ჭიჭყინა მთელის“ კლასში ჩავამატოთ საჭირო კონსტრუქტორი:

```
noisyInt(const noisyInt& other) :value(other.value)
{
	++noisyInt_count;
	cout << "New noisyInt created. noisyInt_count=" << noisyInt_count << endl;
}
```

ამჯერად, იგივე პროგრამა დრაივერი სწორ შედეგს გვიჩვენებს:

```
New noisyInt created. noisyInt_count=1
class noisyInt *:  value: 16
New noisyInt created. noisyInt_count=2
Deleted: class noisyInt *:  value: 16 noisyInt_count=1
Deleted: class noisyInt *:  value: 16 noisyInt_count=0
Press any key to continue . . .
```

ზემოთ შევნიშნეთ, რომ C++ -ის მიერ შექმნილმა კონსტრუქტორმა მხოლოდ მინიმალური მოქმედებები ჩაატარა, რამაც შესაძლებელი გახადა ასლის საფუძველზე ახალი ობიექტის აგება. ასეთი მინიმალისტური მიდგომა (მეტის გათვალისწინა კომპილერის შემქმნელებს არანაირად არ შეუძლათ) გარკვეულ შემთხვევებში ძალიან სახიფათოა, რადგან ასლის ნაგულისხმევ კონსტრუქტორს კლასის წევრი-მონაცემების ანუ ველების წევრ-წევრა ასლები გადააქვს მხოლოდ. თუ ეს წევრი ბმა (ლინკი) არის, მაშინ პოტენციურად სახიფათო ვითარება იქმნება, რადგან ორი ან მეტი ობიექტი საზიარო რესურსის მფლობელი აღმოჩნდება. განვიხილოთ მაგალითი, რომელიც გვიჩვენებს, რომ როდესაც კომპილერი ჩვენს მაგივრად წარმოქმნის ასლის გადაღების ოპერატორს, ის ოდნავ რთულ შემთხვევებში უკვე ვეღარ ითვალისწინებს ასეთ რამეებს:

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

class example1
{
public:
	int* a = (int*)malloc(3 * sizeof(int));
	example1(int i, int j, int k)
	{
		a[0] = i; a[1] = j; a[2] = k;
	}
```

```
        void show() { std::cout << a[0] << "  " << a[1] << "  " << a[2] << std::endl;  }
};
int main()
{
        example1 x(1, 3, 2);
        cout << "First object: " << endl;
        x.show();

        example1 y = x;
        cout << "Second object: " << endl;
        y.show();

        x.a[1] = 111;
        cout << "First object was changed: " << endl;
        x.show();
        cout << "Second object: " << endl;
        y.show();
}
```

საწყისი ობიექტი და მისი ასლი არ არიან დამოუკიდებელი. ერთის ცვლილება იწვევს მეორის ცვლილებას:

```
First object:
1  3  2
Second object:
1  3  2
First object was changed:
1  111  2
Second object:
1  111  2
Press any key to continue . . .
```

ეს მაგალითი სხვა რამითაც არის საინტერესო. ამ კლასს სჭირდება დესტრუქტორი, მაგრამ დესტრუქტორის შექმნა გააუქმებს ნაგულისხმევ ასლის ოპერატორს.

არსებობს გარკვეული წესები, რაც განსაკუთრებული წევრი-ფუნქციების შექმნასა და ურთიერთარსებობას განსაზღვრავს. ჩვენ მათ მინიმალურ დონეზე შევეხებით.

C++11-ის შემდეგ, მანამდე არსებული გამოცდილების ანალიზის საფუძველზე, პროგრამისტს მიეცა შესაძლებლობა უკეთ მართოს განსაკუთრებული წევრი-ფუნქციების შექმნის საკითხი. მაგალითად, თუ პროგრამისტი თვლის, რომ მის მიერ შექმნილი `Widget` კლასისთვის სრულიად მისაღებია ნაგულისხმევი ასლის კონსტრუქტორი, მაშინ ის ამას ასე ადასტურებს:

```
…                                       // ნაგულისხმევი ასლის გადაღების
Widget(const Widget&) = default;        // კონსტრუქტორის ყოფაქცევა სწორია
```

მეორე მხრივ, თუ რაიმე მოსაზრებების გამო პროგრამისტი არ ქმნის ასლის კონსტრუქტორს და უნდა ამის გაკეთება აუკრძალოს კომპილერსაც, მაშინ იგი წერს:

```
Widget(const Widget&) = delete;
```

ჩვენს მაგალითში, ნაგულისხმევი ასლის კონსტრუქტორის აკრძალვას ექმება სახე:

```
noisyInt(const noisyInt& other) = delete;
```

ზოგადად, არცერთი განსაკუთრებული წევრი-ფუნქცია არ გენერირდება გულისხმობის პრინციპით, თუ ჩვენ თვითონ შევქმნით შესაბამის კონსტრუქტორს.

<<< ასლის მინიჭების კონსტრუქტორი. ლექციის დასაწყისში აღვნიშნეთ, რომ ასლის მინიჭების ოპერატორს რაიმე `Widget` კლასისთვის უნდა ჰქონდეს სახე:

```
        Widget& operator=(const Widget&)
```

მისი ფორმის გამო, ამ ოპერატორს ხშირად ასლის მინიჭების ოპერატორს უწოდებენ. C++ ენა გამოირჩევა ოპერატორების გადატვირთვის მდიდარი შესაძლებლობებით (სხვა ენები, ძირითადად, ფუნქციების გადატვირთვით შემოიფარგლებიან). ოპერატორების გადატვირთვას ჩვენ საკმაო დროს დავუთმობთ მომდევნო თემებში. ამჯერად მინიჭების ოპერატორზე ვიტყვით ცოტა რამეს.

C++ ენაში როდესაც ფუნქციის (გლობალურის, ან კლასის წევრის) სახელი იწყება „operator“-ით, ეს ნიშნავს რომ ამ ფუნქციის გამოძახებისთვის არსებობს არადანი - ალტერნატიული ფორმა, რაც ემთხვევა ფუნქციის სახელში „operator“- ზე მიწებებულ ოპერატორის ნიშანს. ჩვენს შემთხვევაში ეს არის ტოლობა ანუ მინიჭება.

კვლავ შევქმნათ ხელოვნური პრობლემა ამ კონსტრუქტორის არქონათან დაკავშირებით:

```
int noisyInt::noisyInt_count = 0;
int main()
{
	noisyInt n;
	n.value = 16;
	cout << n.toString() << endl;

	noisyInt b = n;
}
```

ახლა, „ჭიჭყინა მთელების“ რაოდენობის სწორი აღრიცხვა-არაღრიცხვა დამოკიდებულია კლასში იმპლემენტირებულ კონსტრქტორებზე.

თუ კლასს აქვს სახე:

```
class noisyInt
{
private:
	static int noisyInt_count;
public:
	int	value;
	noisyInt()
	{
		++noisyInt_count;
		cout << "New noisyInt created. noisyInt_count=" << noisyInt_count << endl;
	}
	noisyInt(int itsValue) :value(itsValue)
	{
		++noisyInt_count;
		cout << "New noisyInt created. noisyInt_count=" << noisyInt_count << endl;
	}
	noisyInt(const noisyInt& other) :value(other.value)
	{
		++noisyInt_count;
		cout << "New noisyInt created. noisyInt_count=" << noisyInt_count << endl;
	}
	~noisyInt()
	{
		--noisyInt_count;
		cout << "Deleted: " << toString() << " noisyInt_count=" << noisyInt_count
<< endl;
	}

	string toString()
	{
		return ((string)typeid(this).name() + ":  value: " + to_string(value));
	}
};
```

ანუ მასში უკვე შექმნილია ასლის კონსტრუქტორი, მაგრამ არაა გაკეთებული ასლის მინიჭების ოპერატორიი, მაშინ აღრიცხვა გამართულია:

```
New noisyInt created. noisyInt_count=1
class noisyInt *:  value: 16
New noisyInt created. noisyInt_count=2
Deleted: class noisyInt *:  value: 17 noisyInt_count=1
Deleted: class noisyInt *:  value: 16 noisyInt_count=0
Press any key to continue . . .
```

საკმარისია ამოვიღოთ ასლის კონსტრუქტორი, რომ აღრიცხვა ირევა.

მაშასადამე, ამ შემთხვევაში ნაგულისხმევი ასლის მინიჭების ოპერატორი (კონსტრუქტორი) სწორად მუშაობს და ამიტომ კლასის კოდში შეგვიძლია ჩავამატოთ სტრიქონი:

```
noisyInt& operator=(const  noisyInt &) = default;
```

როდესაც კლასს აქვს სტატიკური წევრები, განსაკუთრებული წევრი ფუნქციების შექმნა და მართვა დამატებით წესებს ექვემდებარება. ამიტომ ჩვენ არ ჩავუღრმავდებით ამ საკითხს. მაგრამ ყურადღება მივაქციოოთ იმ ფაქტს, რომ იმის საფუძველზე რაც ოპერატორების შესახებ ვთვით, შემდეგ პროგრამაში

```
int noisyInt::noisyInt_count = 0;
int main()
{
	noisyInt n;
	n.value = 16;
	cout << n.toString() << endl;

	noisyInt b = n;
	b.value = 17;

	n = b;
	//n.operator=(b);
}
```

ბოლო ორი სტრიქონი ერთი და იმავე რამეს ნიშნავს, ანუ შეგვიძლია კომენტარის ნიშნის გააადგილება ზედა სტრიქონზე და ეს არაფერს შეცვლის.

<<< **ასლის კონსტრუქტორები ერთმანეთისგან დამოუკიდებელია.** ცნობილია, რომ ასლის გამოყენებასთან დაკავშირებული კონსტრუქტორები დამოუკიდებელია ერთი მეორისგან: ერთის განაცხადი არ უქმნის პრობლემას კომპილერს მეორის გენერირებისთვის. ამგვარად, თუ თქვენ განაცხადებთ ასლის კონსტრუქტორს, მაგრამ არ განაცხადებთ ასლით მინიჭების ოპერატორს, შემდეგ დაწერთ კოდს რომელიც მოითხოვს ასლით მინიჭებას, კომპილერები შექმნიან ასლით მინიჭების ოპერატორს თქვენს ნაცვლად. მსგავსი შემთხვევა არის თუ განაცხადებთ ასლით მინიჭების ოპერატორს მაგრამ არა ასლის კონსტრუქტორს - თუ თქვენი კოდი მოითხოვს ასლის კონსტრუქტორს, კომპილერები შექმნიან მას თქვენს მაგივრად. ეს წესი არსებობდა C++98-ში და ამართლებს C++11-შიც.

თუმცა, როგორც ჩვენ ვნახეთ მარტივ მაგალითზე, სავარაუდოდ C++ იყენებს მომხმარებლის მერ შექმნილ ასლის კონსტრუქტორს იმისთვის რომ თვითონ შექმნას ნაგულისხმევი კონსტრუქტორი.

<<< **lvalue -ზე და rvalue-ზე რეფერენსები.** C++ ენაში, ყოველი გამოსახულება არის ან მარცხენა სიდიდეა (lvalue) ან მარჯვენა (rvalue). მარცხენა სიდიდედ იწოდება გამოსახულება, რომელიც არსებობას აგრძელებს ცალკეული გამოსახულების მიღმა. ზუსტი განმარტება რთულია. ზოგადად, შეგიძლიათ იფიქროთ მარცხენა სიდიდეზე როგორც სახელის მქონე

გამოსახულებაზე. ამბობენ ასევე, რომ მარცხენა სიდიდე წარმოადგენს გამოსახულებას, რომელსაც უკავია განსაზღვრული ადგილი მეხსიერებაში, თუმცა ეს საკამათოა, - დასაბრუნებელი მნიშვნელობის მქონე ფუნქციის გამოძახების შედეგი ითვლება მარჯვენა გამოსახულებად, ხოლო თუ ის რეფერენსით აბრუნებს მნიშვნელობას, მაშინ აქვს ადგილი მეხსიერებაში და მისთვის შეიძლება რაღაცეების მინიჭება.

მარჯვენა სიდიდეები განისაზღვრება გამორიცხვით, რადგან გამოსახულება არის ან მარჯვენა, ან მარცხენა სიდიდე.

განვიხილოთ მაგალითი:

```
int main()
{
   int x = 3 + 4;
   cout << x << endl;
}
```

აქ, x არის lvalue, რადგან ის შენარჩუნდება მისი განმსაზღვრელი გამოსახულების შემდეგ. 3+4 არის rvalue.

შემდეგი მაგალითი გვიჩვენებს lvalue და rvalue-ების რამდენიმე სწორ და არასწორ გამოყენებას:

```
int main()
{
   int i, j, *p;

   // სწორია: i ცვლადი არის lvalue.
   i = 7;

   // არასწორია: მარცხენა ოპერანდი უნდა იყოს lvalue (C2106).
   7 = i; // C2106
   j * 4 = 7; // C2106

   // სწორია: განმისამართებული პოინტერი არის lvalue.
   *p = i;

   const int ci = 7;
   // არასწორია: ცვლადი არის არა-მოდიფიცირებადი lvalue (C3892).
   ci = 9; // C3892

   // სწორია: the პირობითი ოპერატორი აბრუნებს lvalue-ს.
   ((i < 3) ? i : j) = 7;
}
```

ამ მაგალითებში ვგულისხმობთ რომ ოპერაციები არ არის გადატვირთული. თუ გადავტვირთავთ, შეგვიძლია `j * 4` გამოსახულება ვაქციოთ lvalue-დ..

უფრო საინტერესო მაგალითები უკავშირდება ფუნქციებს. ფუნქციის გამოძახება წარმოადგენს rvalue-ს, თუ ფუნქცია აბრუნებს მნიშვნელობას. თუ რეფერენსით აბრუნებს- მაშინ lvalue-საც. მაგალითად, back() მეთოდი ვექტორში.

თავიდან C ენაში განსაზღვრეს lvalue, როგორც „სიდიდე რომელიც შესაფერისია მინიჭების მარცხენა მხარისთვის“. მოგვიანებით ISO C-ში დაამატეს const სიტყვა. თუ ობიექტის განაცხადში მონაწილეობს const, ეს ობიექტი არ არის განახლებადი (modifiable).

შესაძლებელია lvalue-ების გარდაქმნა rvalue-ებად, რასაც ხშირად შევხვდებით შემდეგში. პირიქით შეუძლებელია გარდაქმნა.

სტატიკურმა მთვლელმა ადვილად შეიძლება შექმნას გაუთვალისწინებელი სირთულეები, თუ არ ვიზრუნებთ ყველა შესაძლო სცენარის დამუშავებაზე. C++ სწრაფად ვითარდება და ახალ სტანდარტებს მოაქვს ახალი შესაძლებლობები როგორც უფრო ეფექტური კოდის შექმნისთვის,

როდესაც ობიექტი ნადგურდება პროგრამის დახურვის ან სხვა რაიმე მიზეზის გამო, პროგრამა იძახებს დესტრუქტორს. ამიტომ განადგურებული ობიექტების აღრიცხვა შედარებით ადვილად მოსავლელია. შემდეგ თემაში გავიგებთ უფრო მეტს კონსტრუქტორებთან დაკავშირებით.

lvalue-ზე რეფერენსი. ყველაზე კარგად ნაცნობი არის lvalue-ზე რეფერენსი. თუ T ტიპის lvalue ობიექტი სახელით x უკვე არის განსაზღვრული, მაშინ ჩანაწერი

```
T& y = x;      //ან ინიციალების სხვა რაიმე ფორმა, მაგ. () ან {}
```

ნიშნავს, რომ x ცვლადს დაერქვა მეორე სახელი y, და ეს ობიექტები მეხსიერების ერთი და იმავე ნაკვეთს აღნიშნავენ. შესაძლებელია რომ უკვე არსებულ lvalue ობიექტს დავარქვათ მეორე სახელი, რომელსაც შეზღუდული ექნება ობიექტის განახლების შესაძლებლობა:

```
const T& y = x;                                                    // (1)
```

ამ განაცხადის შემდეგ, y არის მუდმივი რეფერენსი ჩვეულებრივ lvalue ობიექტზე. y სახელით მხოლოდ ამოკითხვა შეგვიძლია და არა ჩაწერა.

თუ x იქნებოდა მუდმივი, მაშინ მისი რეფერენსი მხოლოდ მუდმივი შეიძლება იყოს, ანუ მხოლოდ (1) განაცხადი არის მისაღები.

lvalue-ზე მუდმივი რეფერენსით შესაძლებელია მითითება მუდმივ ან დროებით ობიექტებზე. მაგალითად:

```
int intF()
{
	int i(55);
	return i;
}
int main()
{
	const int& cr = intF();
	const int& cr1 = 747;

}
```

შევნიშნოთ, რომ ინციალიზაციის შემდეგ lvalue-ზე რეფერენსი არის ჩვეულებრივი lvalue ობიექტი.

rvalue-ზე რეფერენსი. rvalue-ზე რეფერენსი ქმნის ახალ lvalue ობიექტს, რომელიც აუცილებლად განაცხადის გაკეთების მომენტში ინიციალდება და აუცილებლად rvalue სიდიდით.

```
T&& რეფერენსის სახელი = rvalue სიდიდე      //ან ინიციალების სხვა ფორმა, მაგ. () ან {}
```

ინიციალების შემდეგ, ეს არის ჩვეულებრივი lvalue ობიექტი, რომელსაც შეუძლია თავის მნიშვნელობის განახლება T ტიპის სხვა ობიექტების მნიშვნელობებით (მინიჭებით, თუ ეს T ტიპისთვის დაშვებულია, ან გადაადგილებით - თუ დაშვებულია, ან ორივეთი). მაგრამ იმის გამო რომ იგი არის rvalue რეფერენსი, მისი გამოყენება შესაძლებელია სხვა მიზნებითაც, კერძოდ გადაადგილებასთან და უნაკლო გადაწოდებასთან დაკავშირებით.

მაგალითად, სწორია

```
vector<int>&& b{ 1,2,3,2,1 };
```

და ამის შემდეგ შესაძლოა მოდიოდეს:

```
vector<int> a{ 8,7,5,3,2,9 };
b = a;
```

მაგრამ არასწორია

```
vector<int> a{ 8,7,5,3,2,9 };
vector<int>&& b{a};
```

რადგან rvalue-ზე რეფერენსი არ შეიძლება მიებას lvalue-ს.

<<< გადაადგილებასთან დაკავშირებული კონსტრუქტორები. გასაგებია, რომ ვიდრე არ გავერკვევით გადაადგილებასა და უნაკლო გადაწოდებაში, ვერ ვიქნებით დარწმუნებული ჩვენს მიერ შექმნილი ასეთი კონსტრუქტორების კორექტულობასა და ეფექტურობაში. ჩვენ ვნახეთ, რომ ამ და სხვა მოსაზრებების არსებობის შემთხვევაში შეგვიძლია კომპილერს ავუკრძალოთ გარკვეული წევრი-ფუნქციების გენერირება წაშლილად გამოცხადების საშუალებით.

გარდა ამისა, განსაკუთრებული წევრი-ფუნქციები, რომლებიც იყენებენ გადაადგილებას, არ გენერირდებიანი ისეთი კლასებისთვის, რომლებიც ცხადად აცხადებენ ასლის გადაღების ოპერაციებს. ამის გამართლება ისაა, რომ ასლის გადაღების ოპერატორის განაცხადი (კონსტრუქტორი ან მინიჭება) აჩვენებს რომ ობიექტის ასლის გადაღების ჩვეული გზა (წევრ-წევრად) არაა ამ კლასისთვის შესაფერისი. აედან გამომდინარე, კომპილერებიც თვლიან რომ რადგან წევრ-წევრად ასლირება არაა შესაფერისი ასლის ოპერატორისთვის, წევრ-წევრად გადაადგილებაც არ იქნება შესაფერისი გადაადგილების ოპერატორისთვის.

ჩვენი კურსის განმავლობაში ჩვენ შევქმნით პრაქტიკული ინტერესის მქონე კლასს, რომელსაც ექნება გადაადგილების კონსტრუქტორი და გადაადგილებული (ობიექტის) მინიჭების ოპერატორი.

<<< სავარჯიშოები

1. წინა მეცადინეობაზე განხილულ კლასებს (სულ ერთია, სტატიკური მთვლელითაა თუ მის გარეშე), დავამატოთ ასლის კონსტრუქტორი. პროგრამა დრაივერში გავსინჯოთ ახალი წევრის შესაძლებლობები.
2. მოიფიქრეთ (ან მოძებნეთ ინტერნეტში) სხვა მაგალითი, როდესაც ასლის კონსტრქუტორი არ ქმნის ორ ერთმანეთისგან დამოუკიდებელ (განცალკევებულ) ობიექტს.
3. ახსენით შემდეგი კოდის

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

class noisyInt
{
private:
    static int noisyInt_count;
public:
    int    value;
    noisyInt()
    {
          ++noisyInt_count;
          cout << "New noisyInt created. noisyInt_count=" << noisyInt_count << endl;
    }
    noisyInt(int itsValue) :value(itsValue)
    {
          ++noisyInt_count;
          cout << "New noisyInt created. noisyInt_count=" << noisyInt_count << endl;
    }
    noisyInt(const noisyInt& other) :value(other.value)
    {
          ++noisyInt_count;
          cout << "New noisyInt created. noisyInt_count=" << noisyInt_count << endl;
    }
    noisyInt& operator=(const  noisyInt &) = default;
```

```
    ~noisyInt()
    {
           --noisyInt_count;
           cout << "Deleted: " << toString() << " noisyInt_count="
                  << noisyInt_count << endl;
    }
    string toString()
    {
           return ((string)typeid(this).name() + ":  value: " + to_string(value));
    }
};

void printNoisyInt1(const noisyInt a)
{
    std::cout << a.value << endl;
}

int noisyInt::noisyInt_count = 0;
int main()
{
    noisyInt n(17);
    printNoisyInt1(n);
}
```

შედეგი:

```
New noisyInt created. noisyInt_count=1
New noisyInt created. noisyInt_count=2
17
Deleted: class noisyInt *:  value: 17 noisyInt_count=1
Deleted: class noisyInt *:  value: 17 noisyInt_count=0
Press any key to continue . . .
```

4. იგივე კლასისთვის, თუ ფუნქციას რეფერენსით გადავაწვდით მნიშვნელობას

```
void printNoisyInt1(const noisyInt& a)
{
    std::cout << a.value << endl;
}
```

მაშინ მივიღებთ:

```
New noisyInt created. noisyInt_count=1
17
Deleted: class noisyInt *:  value: 17 noisyInt_count=0
Press any key to continue . . .
```

ახსენით განსხვავების მიზეზი.