თემა 11. ჭკვიანი პოინტერები - საზიაროს პოინტერი

საკითხები:

საზიარო პოინტერი (shared_ptr<>)

std::shared_ptr-ის საშუალებით მონიშნული ობიექტების არსებობის ხანგრძლივობა იმართება *საზიარო საკუთრების* საფუძველზე. არცერთი ცალკეული std::shared_ptr არ ფლობს ობიექტს. ამის მაგივრად, ყველა std::shared_ptr ერთად, რომლებიც მიუთითებენ ერთ ობიექტზე, თანამშრომლობენ რათა დაშალონ ობიექტი ზუსტად მაშინ როდესაც ის მეტად აღარ არის საჭირო. როდესაც ბოლო std::shared_ptr, რომელიც მიუთითებს ობიექტზე, შეწყვეტს ამ ობიექტზე მითითებას (მაგალითად std::shared_ptr განულპოინტერდა, დაიშალა ან შეიცვალა ისე რომ მიუთითებს სხვა ობიექტზე), ეს ბოლო std::shared_ptr დაშლის იმ ობიექტსაც რომელზეც მიუთითებდა.

std::shared_ptr-ს შეუძლია გვაცნობოს, არის თუ არა იგი ბოლო რომელიც მიუთითებს რესურსზე. ამას აკეთებს რესურსის *მითითებების მთვლელის (reference count)* საშუალებით, რაც არის რესურსთან დაკავშირებული სიდიდე რომელიც თვალყურს ადევნებს თუ რამდენი std::shared_ptr მიუთითებს მასზე. std::shared_ptr-ების კონსტრუქტორები ზრდიან ამ მთვლელს, დესტრუქტორები ამცირებენ მათ, ხოლო ასლის მინიჭებები აკეთებენ ორივეს. (თუ sp1 და sp2 არიან განსხვავებული ობიექტების std::shared_ptr-ები, მინიჭება "sp1 = sp2;" ცვლის sp1-ს ისე, რომ იგი უყურებს იგივე ობიექტს რასაც უყურებს sp2. მინიჭების წმინდა ეფექტი ისაა, რომ ობექტის მთვლელი (რომელზეც თავდაპირველად sp1 მიუთითებდა) მცირდება, ხოლო ობიექტისა რომელზე მიუთითებდა sp2 - იზრდება.) თუ std::shared_ptr ხედავს რომ შემცირების შემდეგ მითითებების მთვლელი გახდა 0, ეს ნიშნავს რომ არცერთი სხვა std::shared_ptr არ მიუთითებს ამ ობიექტზე და ამიტომ std::shared_ptr დაშლის მას.

მითითებების მთვლელის არსებობას აქვს გავლენა წარმადობაზე (ეფექტურობაზე):

- **std::shared_ptr არის ორჯერ დიდი ზომისა ვიდრე ნედლი პოინტერი,** რადგან ისინი შეიცავენ რესურსის ნედლ პოინტერს და პოინტერს რესურსის მითითებების მთვლელზე (სტანდარტი არ აკონკრეტებს იმპლემენტაციის გზას, მაგრამ ძირითად კომპილერებში ასეა).
- **მითითებების მთვლელის მეხსიერება დინამიკურად უნდა იქნას გამოყოფილი**. იდეურად, მითითებების მთვლელი დაკავშირებულია იმ ობიექტთან რომელიცაა მითითებული, მაგრამ მითითებულმა ობიექტმა არაფერი იცის ამის შესახებ. ამიტომ მათ არ აქვთ რაიმე ადგილი მითითებების მთვლელის შესანახად. დინამიკური განთავსების ხარჯი თავიდან არის აცილებული როდესაც std::shared_ptr იქმნება std::make_shared-ით, მაგრამ გვხვდება გარემოებები როდესაც std::make_shared-ს ვერ გამოვიყენებთ.
- **მითითებების მთვლელის გაზრდა და შემცირება ატომურია,** რადგან შესაძლებელია ერთდროული წაკითხვები და ჩაწერები განსხვავებული დინებების მიერ. მაგალითად,

რაც std::shared_ptr-ს არ შეუძლია, არის მასივებთან მუშაობა. როგორც კიდევ ერთი განსხვავება std::unique_ptr-ისგან, std::shared_ptr-ს აქვს API რომელიც დაგეგმილია მხოლოდ ერთ ობიექტზე

მიმთითებელი პოინტერებისთვის. არაა არავითარი `std::shared_ptr<T[]>`. დროდადრო ცდილობენ ხოლმე გამოიყენონ std::shared_ptr<T> მასივის მითითებისთვის (სხვათა შორის, ასეა ჩვენს ერთ-ერთ წყაროში), შექმნან შეკვეთილი დამშლელი მასივის ამოსაღებად (ე.ი. `delete[]`). შესაძლოა მოვახერხოთ რომ ამან გაიაროს კომპილაცია, მაგრამ ასე შევქმნით სხვა, უფრო დიდ პრობლემას. მაშინ როდესაც გვაქვს C++-ში მდიდარი არჩევანი ჩაშენებულ მასივებზე (std::array, std::vector,std::string), კიდევ ჭკვიანი პოინტერის გამოცხადება მუნჯი მასივისთვის თითქმის ყოველთვის არის ცუდი დიზაინის ნიშანი.

<<< საზიარო პოინტერის შექმნა, ასლი, გადაადგილება, რეფერენსების მთვლელის ნახვა

ისევე როგორც წული პოინტერი, საზიარო პოინტერი ორნაირად შეიძლება შეიქმნას:

```
#include <iostream>
#include <memory>
using namespace std;
class noisyInt
{
public:
	int data;
	noisyInt() {}
	noisyInt(int k) :data(k) {}
	~noisyInt() { std::cout << "deleted " << data << std::endl; }
};
int main()
{
	shared_ptr<noisyInt> p = make_shared<noisyInt>(77);
	cout << "First shared object  " << p->data << endl;
	shared_ptr<noisyInt> q(new noisyInt(99));
	cout << "Second shared object  " << q->data << endl << endl;
	std::shared_ptr<int> p1(new int());
	cout << "First shared number  " << *p1 << endl;
	std::shared_ptr<double> p2(new double(3.15));
	cout << "Second shared number  " << *p2 << endl << endl;
}
```

`std::make_shared` - ის გამოყენება უკეთესია, რდგან იგი ერთდროულად აკეთებს ობიექტსაც და რეფერენსების მთვლელსაც, ანუ `new` ოპერატორს გამოიძახებს ერთხელ.

ისევე როგორც ეული პოინტერის შემთხვევაში, განსაკუთრებული შემთხვევაა როდესაც საზიაროს პოინტერს ვქმნით არსებული ობიექტისგან. ამ დროს პოინტერი შეიქმნის საკუთრებაში ამ ობიექტის ასლს:

```
int main()
{
	noisyInt a(99);
	shared_ptr<noisyInt> p = make_shared<noisyInt>(a);
	cout << "Shared object  " << p->data << endl;
}
```

მართლაც, პროგრამის შედეგად ვხედავთ რომ დესტრუქტორი იმუშავებს ორჯერ.

პოინტერების ასლების ასლების შექმნა ადვილად ხდება. ასლირების დროს იზრდება თითოეულის მითითებების მთვლელი:

```
int main()
{
	shared_ptr<noisyInt> p = make_shared<noisyInt>(77);
	shared_ptr<noisyInt> q(p);
	cout << "Shared object  " << q->data << "  ";
	cout << "and its reference counter:  " << q.use_count() << endl << endl;
```

```
	std::shared_ptr<double> p1(new double(3.15));
	std::shared_ptr<double> p2 = p1;
	cout << "Shared number  " << *p1 << "  ";
	cout << "and its reference counter:  " << p1.use_count() << endl << endl;
}
```

შედეგი:

```
Shared object  77  and its reference counter:  2

Shared number  3.15  and its reference counter:  2

deleted 77
Press any key to continue . . .
```

შესაძლებელია გადაადგილებაც. ყურადღება უნდა მივაქციოთ, თუ როგორ იცვლება პოინტერების მთვლელები:

გადაადგილების მაგალითი და შედეგი:

```
int main()
{
	std::shared_ptr<double> p1(new double(3.15));
	std::shared_ptr<double> p2 = p1;
	cout << "First shared number  " << *p1 << "  ";
	cout << "and its reference counter:  " << p1.use_count() << endl << endl;

	std::shared_ptr<double> p3(new double(11.21));
	cout << "Before move:  " << endl;
	cout << "*p3= " << *p3 << "  " << "p3.use_count()=" << p3.use_count() << endl;
	cout << "*p1= " << *p1 << "  " << "p1.use_count()="
				<< p1.use_count() << endl << endl;
	p3 = move(p1);
	cout << "After move:	p3 = move(p1);  " << endl;
	cout << "*p3= " << *p3 << "  " << "p3.use_count()=" << p3.use_count() << endl;
	cout << "p1.use_count()=" << p1.use_count() << endl << endl;
}
```

შედეგი:

```
First shared number  3.15  and its reference counter:  2

Before move:
*p3= 11.21  p3.use_count()=1
*p1= 3.15  p1.use_count()=2

After move:	p3 = move(p1);
*p3= 3.15  p3.use_count()=2
p1.use_count()=0

Press any key to continue . . .
```

p1 პოინტერმა თავის რესურსზე საკუთრების უფლება გადასცა p3-ს. ამიტომ p1-ის ყოფილ რესურზე მიტითებების რაოდენობა არ შეცვლილა, ერთი მოაკლდა, ერთი დაემატა.

<<< პოინტერის განცალკევება საზიარო რესურსისგან

შესაძლებელია რამდენიმენაირად. ერთი გზაა უპარამეტრო **reset()** ფუნქციის გამოყენება:

```
int main()
{
	shared_ptr<noisyInt> p = make_shared<noisyInt>(99);
	shared_ptr<noisyInt> q = p;
	cout << "Before reset: " << endl;
```

```
	cout << "p.use_count()=  " << p.use_count() << endl;
	cout << "q.use_count()=  " << q.use_count() << endl;
	p.reset();
	cout << "After reset: " << endl;
	cout << "p.use_count()=  " << p.use_count() << endl;
	cout << "q.use_count()=  " << q.use_count() << endl;
}
```

შედეგით:

```
Before reset:
p.use_count()=  2
q.use_count()=  2
After reset:
p.use_count()=  0
q.use_count()=  1
deleted 99
Press any key to continue . . .
```

მეორე გზაა პოინტერის განულპოინტერება, ანუ `p.reset();`-ის ნაცვლად `p = nullptr;` -ის დაწერა და პრაქტიკულად იგივე შედეგის მიღება.

(რადგან ვექტორი სტატიკურად გვაქვს გაკეთებული).

<<< საზიაროს პოინტერები შეკვეთილი დამშლელით

std::unique_ptr-ის მსგავსად, std::shared_ptr იყენებს delete-ს როგორც ნაგულისხმევ რესურსის დამშლელ მექანიზმს, თუმცა მხარს უჭერს მომხმარებლის მიერ (შექმნილ) დამშლელებსაც. ამ მხარდაჭერის დიზაინი განსხვავდება std::unique_ptr-ების მხარდაჭერის დიზაინისგან. std::unique_ptr-ისთვის, დამშლელის ტიპი არის ჭკვიანი პოინტერის ტიპის ნაწილი. std::shared_ptr-ისთვის ასე არაა. იგივე მაგალითზე, რაც განვიხილეთ ეული პოინტერებისთვის, საზიარო ჭიჭყინა მთელის პოინტერი დამშლელით ასე გაკეთდება:

```
#include <iostream>
#include <memory>
using namespace std;
class noisyInt
{
public:
	int data;
	noisyInt() {}
	noisyInt(int k) :data(k) {}
	~noisyInt() { std::cout << "deleted " << data << std::endl; }
};
class complexClass
{
public:
	noisyInt* dataPointer;
	complexClass(noisyInt* dataPointerValue) :dataPointer(dataPointerValue) {}
	~complexClass() { std::cout << "Complex object deleted " << dataPointer->data <<
std::endl; }
};

int main()
{
	auto delComplexClassObject = [](complexClass* p)
	{
		delete p->dataPointer;
		delete p;
	};
```

```
	shared_ptr<complexClass> shp(new complexClass(new noisyInt(77)),
delComplexClassObject);
	cout << shp->dataPointer->data << endl;
}
```

<<< საზიაროს პოინტერების შექმნასთან დაკავშირებული საფრთხეები

განვიხილოთ პირველი მაგალითი [4]-იდან:

```
#include<memory>
#include<iostream>
typedef class Sample
{
public:
	int internalValue;
	Sample() {
		internalValue = 0;
		std::cout << "Constructor" << std::endl;
	}
	~Sample() {
		std::cout << "Destructor" << std::endl;
	}
};
int main()
{
	{
		Sample * rawPtr = new Sample();
		std::shared_ptr<Sample> ptr_1(rawPtr);

		{
			std::shared_ptr<Sample> ptr_2(rawPtr);
		}
		// As ptr_2 dont know that the same raw pointer is used by another
		// shared_ptr i.e. ptr_1, therefore  here when ptr_2 goes out of scope and
		// it deletes the raw pointer associated with it.

		// Now ptr_1 is internally containing a dangling pointer. When ptr_1 goes
		// out of scope then it will again try to delete the associated memory,
		// which is actually a dangling pointer. Hence program will crash.
	}
}
```

ეს მაგალიტი კიდევ ერთხელ ამტკიცებს იმას, რომ განაცხადი საზიაროს პოინტერზე უნდა გაკეთდეს `make_shared< >`-ის საშუალებით.

მეორე საფრთხე ჩნდება როდესაც საზიარო ობიექტი მოთავსებულია სტეკის მეხსიერებაში და არა გროვის მეხსიერებაში. ამ ტიპის საფრთხის თვალსაჩინოებისთვის [4]-ში მოყვანილია შემდეგ მაგალითში:

```
#include<memory>
#include<iostream>

int main()
{
	int x = 12;
	std::shared_ptr<int> ptr(&x);
}
```

აქაც, განაცხადი არ კეთდება ზემოთ მოყვანილი რეკომენდაციის მიხედვით.

<<< სავარჯიშოები:

1. მესამე გვერდზე, მაგალითში **გადაადგილების მაგალითი და შედეგი** ჩაატარეთ შემდეგი ცდები: ა) იგივე კოდი გასინჯეთ ნამდვილი რიცხვების ნაცვლად ჭიჭყინა მთელებზე. გაანალიზეთ დესტრუქტორების მუშაობის შედეგები; ბ) კოდის ბოლოში შეამოწმეთ, გახდა თუ არა გადაადგილებული პოინტერი ნულოვანი.
2. გაარჩიეთ https://thispointer.com//learning-shared_ptr-part-1-usage-details/ გვერდის ბოლოში მოთავსებული კოდი.
3. გაარჩიეთ შეკვეთილი დამშლელის გათვალსაჩინოებისთვის [3]-ში მოყვანილი კოდი.

<<< ლიტერატურა

1. Scott Meyers, Effective modern C++. O'reilly 2014.
2. https://thispointer.com//learning-shared_ptr-part-1-usage-details/
3. https://thispointer.com//shared_ptr-and-custom-deletor/
4. https://thispointer.com//create-shared_ptr-objects-carefully/