თემა 10. ჭკვიანი პოინტერები - ეული პოინტერი

საკითხები:

ეული პოინტერი (unique_ptr<)

ეს თემა მომზადებულია ლიტერატურაში მითითებული წყაროების ([1],[2]) საფუძველზე.

unique_ptr<> არის C++11-ის მიერ შემოთავაზებული ჭკვიანი პოინტერების ერთ-ერთი სახეობა. ჭკვიანი პოინტერები შექმნილია მეხსიერების გაჟონვის თავიდან ასაცილებლად. `unique_ptr<>` წარმოადგენს ჩვეულებრივი, ე.წ. ნედლი პოინტერის შეფუთვას და მის ხელშია იმ ობიექტის არსებობის ხანგრძლივობა, რომელზეც მიუთითებს.

მხედველობაში უნდა ვიქონიოთ, რომ `std::unique_ptr`-ები ორი სახით შემოვიდა ენაში, ერთი ინდივიდუალური ობიექტებისთვის (`std::unique_ptr<T>`) და ერთი მასივებისთვის (`std::unique_ptr<T[]>`). თუმცა, {1} გვეუბნება, რომ `std::unique_ptr`-ების არსებობა მასივისთვის მხოლოდ ინტელექტუალურად არის საინტერესო, რადგან `std::array`, `std::vector` და `std::string` პრაქტიკულად ყოველთვის უკეთესი მონაცემთა სტრუქტურაა ვიდრე ნედლი მასივი, განსაკუთრებით დამწყებებისთვის. შესაბამისად, ჩვენ ზერელედ შევეხებით მეორე სახეს.

თუ ის ობიექტი, რომელიც დაკავშირებულია ეულ პოინტერთან, არაა რთულად დასაშლელი (ანუ ნაგულისხმევი დესტრუქტორიც კმარა მის დასაშლელად), მაშინ ეული პოინტერი იგივე ზომისაა რაც ნედლი პოინტერი, და მუშაობს ისევე სწრაფად, როგორც ნედლი პოინტერი. ეს დიდი უპირატესობაა იმის გათვალისწინებით, რაც დამატებით შეუძლია ეულ პოინტერს.

თვალსაჩინოებისთვის, ჩვენ ხშირად ვისარგებლებთ „ჭიჭყინა მთელების“ კლასით:

```
class noisyInt
{
public:
	int data;
	noisyInt() {}
	noisyInt(int k) :data(k) {}
	~noisyInt() { std::cout << "deleted " << data << std::endl; }
};
```

არაერთი მიზეზი განაპირობებდა ჭკვიანი პოინტერების შემოტანის აუცილებლობას, მაგრამ ალბათ ძირითადი იყო დინამიკური მეხსიერების მართვასთან დაკავშირებული საკითხები. მაგალითად, შემდეგ პროგრამა ისე მთავრდება, რომ „ჭიჭყინა მთელის“ დესტრუქტორი არ ამუშავდება.

```
#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()
{
        noisyInt* p = new noisyInt(77);
  //    p = nullptr;
}
```

სულ ერთია, კომენტარებშია თუ არა მეორე სტრიქონი. თუ პროგრამა დიდია და დაგვავიწყდება რომელიმე განმეორების შეტყობინებაში შექმნილი ობიექტების დაშლა მას შემდეგ რაც ისინი არ გამოიყენება, შესაძლოა მივიღოთ ე.წ. მეხსიერების გაჟონვა. თუ შევქმნით ეულ პოინტერებს ობიექტების პოინტერების შესანახად:

```
int main()
{
        unique_ptr<noisyInt> p = make_unique<noisyInt>(77);
        cout << "First object  " << p->data << endl;
        unique_ptr<noisyInt> q(new noisyInt(99));
        cout << "Second object  " << q->data << endl << endl;
}
```

მაშინ მივიღებთ შედეგს:

```
First object  77
Second object  99

deleted 99
deleted 77
Press any key to continue . . .
```

აქ ჭკვიანი პოინტერი გავაკეთეთ ორნაირად. პირველი გზა უფრო საიმედოა და სწრაფი. სინამდვილეში, p და q ჭკვიანი პოინტერები კი არ ინახავენ ობიექტების მისამართებს, არამედ მათ მიერ შეფუთული ნედლი პოინტერები, რომლების ამოღებაც შეიძლება `get()` მეთოდით:

```
int main()
{
        unique_ptr<noisyInt> p = make_unique<noisyInt>(77);
        cout << "First object  " << p->data << endl;

        noisyInt* rp = p.get();
        cout << rp->data << endl;

        p = nullptr;
        cout << "Wild pointer should be set to nullptr" << endl;
        rp = nullptr;
}
```

ამ პროგრამის შედეგი არის:

```
First object  77
77
deleted 77
Wild pointer should be set to nullptr
Press any key to continue . . .
```

პირველი სტრიქონი მივიღეთ იმის გამო, რომ ჭკვიან პოინტერზე გადატვირთულია `->` ოპერატორი. მეორე სტრიქონი მოგვცა `rp->data` სიდიდემ, სადაც `rp` არის `p` ჭკვიანი პოინტერის მიერ შეფუთული ნედლი პოინტერი. ჭკვიანი პოინტერის განულება

ავტომატურად იწვევს მის მიერ მითვალყურებული რესურსის დაშლას (deleted 77). დასასრულ, როდესაც ნედლი პოინტერი მიუთითებს მისამართზე, რომელზე არსებული ობიექტი უკვე დაშლილია, მას ჰქვია ე.წ. ველური პოიტერი და წარმოადგენს პოტენციური თავისტკივილის წყაროს, ამიტომ აუცილებლად უნდა გავანულოთ.

მსგავსი სცენარები ხშირად თამაშდება პროგრამისტის ცხოვრებაში, ამიტომ შემოღებულია მეთოდი release(), რომელიც გაანულებს ჭკვიან პოინტერს და პასუხისმგებლობას ობიექტის დაშლაზე დააკისრებს პროგრამისტს. მაგალითად:

```
int main()
{
	unique_ptr<noisyInt> p = make_unique<noisyInt> (77) ;
	cout << "First object" << p->data << endl;
	unique_ptr<noisyInt> q ( new noisyInt(99));
	cout << "Second object" << q->data << endl << endl;

	noisyInt* nip = p.release();
	cout << "From pointer:" << nip->data << endl;
}
```

პროგრამის შედეგი არის:

```
First object77
Second object99

From pointer:77
deleted 99
Press any key to continue . . .
```

როგორც ვხედავთ, p.release(); შეტყობინებამ დააბრუნა ამ პოინტერის მიერ შეფუთული ნედლი პოინტერი, მაგრამ რესურსი არ დაშალა. უფრო მეტი, თუ ჩავამატებთ

```
	if (p == nullptr)
		cout << "p is nullptr" << endl;
```

შეტყობინებას, დავრწმუნდებით, რომ p.release(); -ის შემდეგ ჭკვიანი პოინტერი განულდა (გაუტოლდა nullptr -ს).

<<< ეული პოინტერი - ეული მესაკუთრე

ეული პოინტერი არის მის მიერ შეფუთული პოინტერისა და შესაბამისი რესურსის ეული (ერთპიროვნული) მფლობელი.

ჩვენ შეგვიძლია შევქმნათ ცარიელი ეული პოინტერი (საკუთრების გარეშე). მაგალითად:

```
	unique_ptr<noisyInt> u;
```

შემდეგ შეგვიძლია მას გადავცეთ სხვა ეული პოინტერის ობიექტზე საკუთრების უფლება:

```
	unique_ptr<noisyInt> p = make_unique<noisyInt>(55);
	u = move(p);
```

ადვილია იმის შემოწმება, რომ საკუთრების უფლების დათმობის შემდეგ (გადაადგილების შედეგად) ეული პოინტერი განულპოინტერდება, და პირიქით, გადაადგილებული პოინტერის მინიჭების შემდეგ ეული პოინტერი მიიღებს განუყოფელ საკუთრებაში შესაბამის ობიექტს:

```
int main()
{
	unique_ptr<noisyInt> u;
	unique_ptr<noisyInt> p = make_unique<noisyInt>(55);

	u = move(p);
	if (u != nullptr) cout << “u->data= “ <<  u->data << endl;
```

```
        if (p != nullptr)
                cout << “p->data= “ <<  p->data << endl;
}
```

ეული პოინტერის ასლის გადატანა არ შეიძლება. შესაძლებელია მხოლოდ გადაადგილება. მიღებული ტერმინოლოგიით, ესაა **მხოლოდ-გადაადგილებადი ობიექტი**.

საკუთების სემანტიკასთან არსებითადაა დაკავშირებული შემდეგი ფაქტი. ახალი ეული პოინტერის შექმნა შეგვიძლია არსებული სტატიკური ობიექტისგან (ახალი ობიექტის მისამართისგან შექმნა უკვე განვიხილეთ). მაგრამ ამ დროს ეული პოინტერი იძულებულია შექმნას არსებული ობიექტის ასლი და მის არსებობის ხანგრძლივობაზე მიიღოს სრული საკუთრების უფლება, რადგან სტატიკური ობიექტის არსებობა-არასებობის საკითხი უკვე განსაზღვრულია და პოინტერზე ვერ იქნება დამოკიდებული. მაგალითად:

```
int main()
{
        noisyInt a(999);
        unique_ptr<noisyInt> p = make_unique<noisyInt>(a);
        cout << p->data << endl;
        cout << a.data << endl;
}
```

შედეგი

```
999
999
deleted 999
deleted 999
Press any key to continue .
```

გვიჩვენებს, რომ პოინტერის განაცხადმა ახალი ობიექტი შექმნა და ამიტომ დაიშალა ორი ობიექტი.

საკუთრების უფლების დათმობა ხდება ან ამ პოინტერის გადადგილებით, როგორც ბოლო მაგალითშია, ან .release(); მეთოდის გამოყენებით (რაც შემოთ ვნახეთ), ან უბრალოდ განულპოინტერებით (რასაც შედეგად მოჰყვება ობიექტის დაშლა). არის უფრო იშვიათად გამოყენებადი შესაძლებლობაც.

<<< ეული პოინტერები და კონტეინერები

განვიხილოთ „ჭიჭყინა მთელების“ დინამიკური მასივი. შედეგი მარჯვნივაა ყუთში.

```
int main()
{
        noisyInt* p(new noisyInt[4]);
        p[0].data = 1;
        p[1].data = 11;
        p[2].data = 12;
        p[3].data = 132;
        for (int i = 0; i < 4; ++i)
                cout << p[i].data << endl;
}
```

```
1
11
12
132
Press any key to continue . . .
```

როგორც ვხედავთ, დესტრუქტორები არ ამუშავდა. ახლა იგივე მასივი შევინახოთ ეულ პოინტერში. ამისთვის საკმარისია, რომ პირველი სტრიქონი შევცვალოთ ასე:

```
unique_ptr<noisyInt[]> p(new noisyInt[4]);
```

შედეგი იქნება იგივე, და დამატებით პროგრამის დასრულებისას გამოძახებულ იქნება მასივის ობიექტების დესტრუქტორები>

```
1
11
12
132
deleted 132
deleted 12
deleted 11
deleted 1
Press any key to continue . . .
```

(1)

მასივის დინამიკურად გაუქმება შეგვეძლო პროგრამის დამთავრებამდეც. მართლაც, საკმარისია ბოლო პროგრამას დავამატოთ ორი სტრიქონი

```
	p = nullptr;
	cin.ignore(80, '\n');
```

რომ პროგრამის მუშაობის შედეგს ვნახავთ პროგრამის დასრულებამდე (`cin.ignore(80, '\n');` პროგრამას აჩერებს “Enter”-ის დაჭერამდე,).

ახლა, განვიხილოთ მასივში ობიექტების მისამართების შენახვის საკითხი. თუ შევინახავთ ნედლ პოინტრებს „ხმაურიან მთელებზე“, ამ მისამართებზე მოთავსებული მონაცემების დაშლა პროგრამისტის პასუხისმგებლობაზეა, როგორც არ უნდა შევქმნათ მასივი, ეული პოინტერით, თუ ნედლით (რომელიც კომენტარებშია მოქცეული ამჟერად):

```
int main(){
	unique_ptr<noisyInt*[]> p(new noisyInt*[4]);
	//noisyInt** p(new noisyInt*[4]);
	p[0] = new noisyInt(1);
	p[1] = new noisyInt(11);
	p[2] = new noisyInt(12);
	p[3] = new noisyInt(132);
	for (int i = 0; i < 4; ++i)
		cout << p[i]->data << endl;
}
```

ცხადია, გამოსავალს წარმოადგენს მასივის შექმნა ეული პოინტერებისგან. მართლაც,

```
int main()
{
	unique_ptr<unique_ptr<noisyInt>[]> p(new unique_ptr<noisyInt>[4]);

	p[0] = make_unique<noisyInt>(1);
	p[1] = make_unique<noisyInt>(11);
	p[2] = make_unique<noisyInt>(12);
	p[3] = make_unique<noisyInt>(132);

	for (int i = 0; i < 4; ++i)
		cout << p[i]->data << endl;
}
```

პროგრამის მუშაობის შედეგს კვლავ (1) ფანჯარის სახე აქვს.

ბოლო ამოცანის გადასაჭრელად ძალიან მოსახერხებელია ვექტორის კონტეინერის გამოყენება:

```
int main()
{
	vector<unique_ptr<noisyInt>> v(4);
	v[0] = make_unique<noisyInt>(1);
	v[1] = make_unique<noisyInt>(11);
	v[2] = make_unique<noisyInt>(12);
	v[3] = make_unique<noisyInt>(132);
```

```
	for (int i = 0; i < 4; ++i)
		cout << v[i]->data << endl;
}
```

<<< ეული პოინტერები და ვირტუალური ფუნქციები

ნედლი პოინტერების შემთხვევაში, ჩვენ ვნახეთ, რომ ფუძე-კლასის პოინტერი ინიჭებს წარმოებული კლასის პოინტერს, რაზეც დაფუძნებულია პოლიმორფიზმი. გავიხსენოთ ჩვენს მიერ შესაბამის თემაში გარჩეული კოდი:

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

class Mammal
{
public:
	Mammal() { }
	Mammal(string itsName):name(itsName) { }
	virtual ~Mammal() { }
	virtual void Speak() const { cout << "Mammal " << name << " speak!\n"; }
protected:
	string name;
};

class Dog : public Mammal
{
public:
	Dog(string itsName) : Mammal(itsName) { }
	void Speak()const { cout << "Dog "  << name << ": Woof!\n"; }
};

class Cat : public Mammal
{
public:
	Cat(string itsName) : Mammal(itsName) { }
	void Speak()const { cout << "Cat " << name << ": Meow!\n"; }
};

int main()
{
	Mammal** p = new Mammal*[3];

	p[0] = new Cat("Cicqna");
	p[1] = new Dog("Cuga");
	p[2] = new Dog("Muria");

	for (int i = 0; i < 3; i++)
		p[i]->Speak();
	for (int i = 0; i<3; i++)
		delete p[i];
	delete[] p;
}
```

```
Cat Cicqna: Meow!
Dog Cuga: Woof!
Dog Muria: Woof!
Press any key to continue . . .
```

გასაგებია, რომ ჭკვიანი პოინტერების შექმნისას ნედლი პოინტერების ასეთ ძლიერ მხარეს უყურადღებოდ არავინ დატოვებდა. მართლაც, ფუძე-კლასის ეულ პოინტერს შეუძლია მიინიჭოს წარმოებული კლასის ეული პოინტერი. ვირტუალობის მექანიზმი ძალაში რჩება. შედეგად, შეგვიძლია ბოლო მაგალითის პროგრამა-დრაივერი ასე შევცვალოთ:

```
int main()
{
	vector<unique_ptr<Mammal>> v(3);

	v[0] = make_unique<Cat>(Cat("Cicqna"));
	v[1] = make_unique<Dog>(Dog("Cuga"));
	v[2] = make_unique<Dog>(Dog("Muria"));

	for (int i = 0; i < 3; i++)
		v[i]->Speak();
}
```

შედეგი იგივეა.

შევნიშნოთ, რომ ვექტორის გამოყენება შეგვეძლო ნედლი პოინტერების შემთხვევაში და შედეგად მივიღებდით ოდნავ გამარტივებულ კოდს და იგივე შედეგს:

```
int main()
{
	vector<Mammal*> p(3);

	p[0] = new Cat("Cicqna");
	p[1] = new Dog("Cuga");
	p[2] = new Dog("Muria");

	for (int i = 0; i < 3; i++)
		p[i]->Speak();
	for (int i = 0; i<3; i++)
		delete p[i];
}
```

<<< ეული პოინტერები და ვირტუალური დესტრუქტორები

ვირტუალური ფუნქციების განხილვისას ჩვენ აღვნიშნეთ რომ თუ ერთი ვირტუალური ფუნქცია მაინც არსებობს, მაშინ დესტრუქტორი უნდა იყოს აგრეთვე ვირტუალური. ახლა ჩვენ შეგვიძლია ამ წესის აუცილებლობის ახსნა.

თუ კონტეინერში შენახული გვაქვს ფუძე-კლასის ეული პოინტერები, ხოლო შემდეგ მათთვის მინიჭებული გვაქვს იერარქიის კლასების ობიექტებზე ეული პოინტერები, მაშინ პროგრამის დასრულების შემდეგ საჭიროა რომ გაეშვას ზუსტად იმ ობიექტების დესტრუქტორები, რომელთა ეული პოინტერებიც ჩავყარეთ კონტეინერში. თუ დესტრუქტორი არაა ვირტუალური, დაიშლება ობიექტების მხოლოდ ფუძე-ნაწილები, რადგან გაეშვება ფუძე-კლასის დესტრუქტორი.

თვალსაჩინოებისთვის, წინა პუნქტში განხილულ მაგალითში დესტრუქტორები „ავალაპარაკოთ“ (რომ დავინახოთ თუ რომელი კლასის ობიექტი იშლება), შემდეგ გავსინჯოთ ორივე ვარიანტი: დესტრუქტორი და ვირტუალური დესტრუქტორი.

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

class Mammal
{
```

```
public:
	Mammal() { }
	Mammal(string itsName) :name(itsName) { }
	virtual ~Mammal() { cout << "Mammal " << name << "is under destruction...\n"; }
	virtual void Speak() const { cout << "Mammal " << name << " speak!\n"; }
protected:
	string name;
};

class Dog : public Mammal
{
public:
	Dog(string itsName) : Mammal(itsName) { }
	void Speak()const { cout << "Dog " << name << ": Woof!\n"; }
	~Dog() { cout << "Dog " << name << "is under destruction...\n"; }
};

class Cat : public Mammal
{
public:
	Cat(string itsName) : Mammal(itsName) { }
	void Speak()const { cout << "Cat " << name << ": Meow!\n"; }
	~Cat() { cout << "Cat " << name << "is under destruction...\n"; }
};

int main()
{
	vector<Mammal*> p(3);

	p[0] = new Cat("Cicqna");
	p[1] = new Dog("Cuga");
	p[2] = new Dog("Muria");

	for (int i = 0; i < 3; i++)
		p[i]->Speak();
	for (int i = 0; i<3; i++)
		delete p[i];
}
```

ეს არის კოდი ნედლი პოინტერებით, ოდნავ გამრავალფეროვნებული, რადგან ფუძე-კლასის ობიექტიცაა ჩასმული. მისი შედეგია:

```
Cat Cicqna: Meow!
Dog Cuga: Woof!
Dog Muria: Woof!
Mammal Unknown Mammal speak!

Cat Cicqnais under destruction...
Mammal Cicqnais under destruction...
Dog Cugais under destruction...
Mammal Cugais under destruction...
Dog Muriais under destruction...
Mammal Muriais under destruction...
Mammal Unknown Mammalis under destruction...
Oops...
Press any key to continue . . .
```

როგორც ვხედავთ, ჯერ იმუშავა ვირტუალურმა ფუნქციებმა, შემდეგ - ვირტუალურმა დესტრუქტორებმა

```
	for (int i = 0; i<4; i++)
		delete p[i];
```

შეტყობინების ძალით. სტრიქონი

```
	cout << "Oops..." << endl;
```

მხოლოდ იმ მიზნითაა ჩასმული, რომ დავაფიქსიროთ ობიექტების დინამიკური დაშლა.

ფუძე-კლასის ობიექტს ერთი დესტრუქტორი ყოფნის. დანარჩენებს - ორ-ორი. კლასების იერარქია უფრო ღრმა რომ ყოფილიყო, მაშინ წარმოებულ კლასს იმდენი დესტრუქტორის გაშვება დასჭირდებოდა, რამდენი ნაწილისგანაც აიგება იგი.

თუ დესტრუქტორს არ გამოვაცხადებთ ვირტუალურად, მაშინ მხოლოდ ფუძე-ნაწილები დაიშლება.

ახლა, იგივე ამოცანაში გამოვიყენოთ ეული პოინტერები:

```
int main()
{
	vector<unique_ptr<Mammal>> v(4);

	v[0] = make_unique<Cat>(Cat("Cicqna"));
	v[1] = make_unique<Dog>(Dog("Cuga"));
	v[2] = make_unique<Dog>(Dog("Muria"));
	v[3] = make_unique<Mammal>(Mammal("Unknown"));
	for (int i = 0; i < 4; i++)
		v[i]->Speak();
	cout << "Oops..." << endl;
}
```

და გავარჩიოთ მისი შედეგი:

```
Cat Cicqna is under destruction...
Mammal Cicqna is under destruction...
Dog Cuga is under destruction...
Mammal Cuga is under destruction...
Dog Muria is under destruction...
Mammal Muria is under destruction...
Mammal Unknown is under destruction...
Cat Cicqna: Meow!
Dog Cuga: Woof!
Dog Muria: Woof!
Mammal Unknown speak!
Oops...
Mammal Cicqna is under destruction...
Mammal Cuga is under destruction...
Mammal Muria is under destruction...
Mammal Unknown is under destruction...
Press any key to continue . . .
```

შედეგებს დაემატა პირველი შვიდი სტრიქონი.

```
	v[0] = make_unique<Cat>(Cat("Cicqna"));
```

სტრიქონმა ჯერ შექმნა `Cat` კლასის დროებითი ობიექტი (`Cat("Cicqna")` ნაწილი), შემდეგ აიგო `Mammal` კლასის ობიექტი რომელშიც ვირტუალური ცხრილის საშუალებით ფუძე-კლასის ვირტუალური ფუნქციის ნაცვლად ჩაჯდა წარმოებული კლასის შესაბამისი მეთოდი. ბოლოს, დროებითი ობიექტი გაუქმდა.

იგივე გააკეთა ორმა მომდევნო სტრიქონმა, ხოლო

```
	v[3] = make_unique<Mammal>(Mammal("Unknown"));
```

სტრიქონის დროებითი ობიექტიც ფუძე-კლასისაა.

დროებითი ობიექტები არის ვექტორის კომფორტის გამოყენების საფასური.

მეორე განსხვავება ისაა, რომ ჯერ პროგრამა მთავრდება (სტრიქონი `Oops...`), ხოლო შემდეგ იწყება ობიექტების დაშლა (რადგან ვექტორი სტატიკურად გვაქვს გაკეთებული).

<<< ეული პოინტერები შეკვეთილი დამშლელით

წარმოვიდგინოთ, რომ გვაქვს კლასი, რომლის ერთადერთ წევრ მონაცემს (ველს) წარმოადგენს პოინტერი „ჭიჭყინა მთელზე“. თუ ასეთ კლასს მხოლოდ ნაგულისხმევი დამშლელი (დესტრუქტორი) აქვს, მაშინ ეული პოინტერის განულპოინტერება დაშლის რთული კლასის ობიექტს, მაგრამ არ დაშლის ჭიჭყინა კლასის ობიექტს, რომელიც წარმოადგენს ობიექტის ნაწილს. მაგალითად:

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

class noisyInt
{
public:
	int data;
	noisyInt() {}
	noisyInt(int k) :data(k) {}
	~noisyInt() { std::cout << "One noisyInt 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()
{
	unique_ptr<complexClass> ccp = make_unique<complexClass>(new noisyInt(77));
	cout << ccp->dataPointer->data << endl;
}
```

შედეგი

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

გვიჩვენებს, რომ ჭიჭყინა მთელი არაა დაშლილი ეული პოინტერის მიერ.

სასურველი შედეგის მისაღებად უნდა შევქმნათ ე.წ. **შეკვეთილი დამშლელი,** რომელიც გაითვალისწინებს კლასის სტრუქტურას, და ეული პოინტერის განაცხადში უნდა მივუთითოთ დამშლელის ტიპი, ხოლო პოინტერის მნიშვნელობის განსაზღვრაში უნდა მონაწილეობდეს ეს დამშლელი:

```
int main()
{
	auto delComplexClassObject = [](complexClass* p)
	{
		delete p->dataPointer;
		delete p;
	};
	unique_ptr<complexClass, decltype(delComplexClassObject)>
		ccp(new complexClass(new noisyInt(77)), delComplexClassObject);
	cout << ccp->dataPointer->data << endl;
}
```

ამის შედეგი ისაა, რაც გვჭირდება, თუმცა მიღებულია კოდის საგრძნობი გართულების ფასად:

```
77
One noisyInt deleted 77
Complex object deleted
Press any key to continue . . .
```

ასეთ შემთხვევებში გასათვალისწინებელია ერთი მნიშვნელოვანი გარემოება: კერძო დამშლელის შემცველი ეული პოინტერის ზომა აღარაა მუდმივი, იგი მეტია ვიდრე ნედლი პოინტერის ზომა.

<<< სავარჯიშოები:

1. გაარჩიეთ ეული პოინტერების შესახებ მასალა შემდეგი მისამართიდან: https://en.cppreference.com/w/cpp/memory/unique_ptr
2. განხილული საკითხები გაარჩიეთ მსგავს მაგალითებზე, სადაც ეული პოინტერები მიუთითებს განხილულისგან განსხვავებული კლასების ობიექტებზე.
3. ეული პოინტერების გამოყენებით შექმენით პოლიმორფიზმის ვითარება კლასების ორდონიანი იერარქიისთვის (შეგიზლიათ გამოიყენოთ წინა თემებში განხილული მაგალითები).
4. მოძებნეთ ან მოიფიქრეთ ეული პოინტერების გამოყენების მაგალითები, როდესაც ისინი მიუტითებენ მარტივი ტიპის ცვლადებზე.

<<< ლიტერატურა

1. Scott Meyers, Effective modern C++. O'reilly 2014.
2. https://thispointer.com/c11-unique_ptr-tutorial-and-examples/