თემა 9: მემკვიდრეობა C++ ენაში

საკითხები:

- მემკვიდრეობა
- მემკვიდრების რეჟიმები >>>
- მემკვიდრეობის ტიპები >>>
- მეგობარი კლასი >>>
- სავარჯიშოები >>>

მემკვიდრეობა

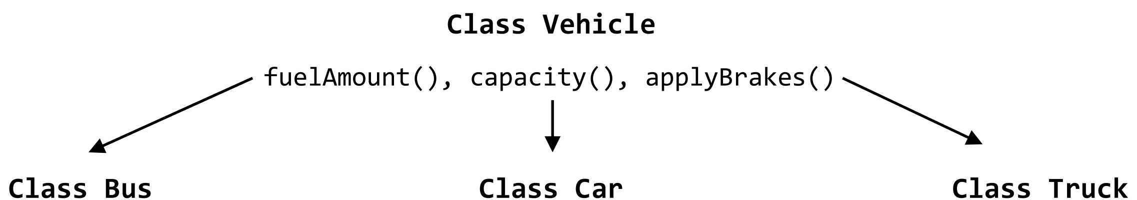
მემკვიდრეობა არის კლასის უნარი რომ სხვა კლასისგან შექმნას თვისებები და მახასიათებლები. აქ გამოიყენება ტერმინები: წარმოებული კლასი, იგივე ქვეკლასი, იგივე შვილი კლასი; ფუძე-კლასი, იგივე ზეკლასი, იგივე მშობელი კლასი.

პროგრამირებაში ხშირია ისეთი შემთხვევები, როდესაც დასამუშავებელ კომპონენტებს აქვთ მსგავსი ატრიბუტები, რომლებიც გარემოებების მიხედვით განსხვავდება დეტალებით ან ყოფაქცევით. ასეთი შემთხვევის დამუშავების ერთი გზა არის რომ ყოველი კომპონენტისთვის ცლკე კლასი ავაგოთ, და ყოველი კლასი განახორციელებს ყველა ატრიბუტს, საერთო ატრიბუტებსაც კი ყველა თვისთვის გააკეთებს. მეორე მიდგომა უფრო პრაგმატულია და იყენებს მემკვიდრეობას, რათა ერთმანეთის მსგავსმა კლასებმა ერთმანეთისგან მემკვიდრეობით მიიღონ საერთო ატრიბუტების განხორციელება. ფუძე-კლასი იძლევა საერთო ფუნქციონალობას, ხოლო წარმოებული კლასი ნაწილს უცვლელად მიიღებს, ნაწილს კი გადაწერს როდესაც საჭიროა იმ ყოფაქცევის განხორციელება, რაც მის უნიკალობას განაპირობებს.

მაგალითად, განვიხილოთ განვიხილოთ სატრანსპორტო საშუალებები. ვთქვათ, საჭიროა შეიქმნას კლასები ავტობუსების, ავტომობილისა და სატვირთო მანქანებისათვის. მეთოდები `fuelAmount()`, `capacity()`, `applyBrakes()` სამივე კლასისთვის ერთი და იგივე იქნება. თუ ჩვენ შევქმნით ამ კლასებს მემკვიდრეობის გარეშე, მაშინ ჩვენ მოგვიწევს რომ შევქმნათ ყველა ეს ფუნქცია სამივე კლასში, როგორც შემდეგ სურათზეა:

Class Bus	Class Car	Class Truck
fuelAmount(), capacity(), applyBrakes()	fuelAmount(), capacity(), applyBrakes()	fuelAmount(), capacity(), applyBrakes()

როგორც ვხედავთ, ეს გზა ერთი და იგივე კოდის გასამმაგებას იწვევს. ეს ზრდის შეცდომების დაშვების შანსს და ზრდის მეხსიერების დატვირთვას. თუ მემკვიდრეობას გამოვიყენებთ, მაშინ შევქმნით კლასს ავტომობილისთვის რომელშიც გავმართავთ ამ სამ ფუნქციას, ხოლო დარჩენილი კლასები ფუნქციებს მიიღებენ მემკვიდრეობით, როგორც შემდეგ სურათზეა:



ჩვენ ვნახეთ ერთი სცენარი როდესაც ფუძე-კლასიდან იწარმოებოდა ახალი კლასები. ახლა განვიხილოთ ზოგადი შემთხვევა:

```
class წარმოებული_კლასის_სახელი : წვდომის_რეჟიმი ფუძე_კლასის_სახელი
{
```

```
	// წარმოებული კლასის ტანი
};
```

ადრე ჩვენ შევხვდით წვდომის_რეჟიმი იყო public:. ქვემოთ ვნახავთ სხვა რეჟიმებს.

<<< მემკვიდრეობის რეჟიმები

მემკვიდრეობის რეჟიმები ძალიან მჭიდროდაა დაკავშირებული კლასის წვდომის განმსაზღვრელებთან (public:, protected:, private:). დასაწყისისთვის, ვნახოთ თუ რას ნიშნავს protected: განმსაზღვრელი. თუ იგი არის გამოყენებული, მაგრამ არ გვაქვს წარმოებული კლასი, მაშინ, პრაქტიკულად არაა განსხვავება protected: და private: განმსაზღვრელებს შორის.

ახლა დავუშვათ, რომ ეს განმსაზღვრელი გამოყენებულია ფუძე-კლასში და ამავე დროს გვაქვს მისგან წარმოებული კლასი. შემდეგი მარტივი მაგალითი, სადაც წვდომის_რეჟიმი შესაძლოა იყოს ნებისმიერი შემდეგი სამიდან - public:, protected:, private:, გვიჩვენებს რომ ფუძე-კლასის კერძო ველს ვერ ხედავს წარმოებული კლასის მეთოდები ვერავითარ შემთხვევაში:

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

class A
{
public:
	int x;
	A() { x = 1; y = 2; z = 3; }
protected:
	int y;
private:
	int z;
};

class B : წვდომის_რეჟიმი  A
{
public:
	void publicAccessX() {std:cout << "Access from derived class: x=" << x <<
std::endl;}
	void publicAccessY() {std:cout << "Access from derived class: y=" << y <<
std::endl;}
//	void publicAccessZ() { std:cout <<"Access from derived class: z=" << z <<
std::endl;}
};

int main()
{
	B b;
	b.publicAccessX();
	b.publicAccessY();
//	b.publicAccessZ();
}
```

რადგან დაკომენტარებული სტრიქონებიდან კომენტარის ახსნის შემთხვევაში ვიღებთ კომპილირების შეცდომას.

ზოგადი წესი, რომელიც გვიხსნის განსხვავებას public:, protected:, private: - შორის, არის ასეთი:

- კლასის private: წვდომის წევრი (წევრი მონაცემი ან წევრი ფუნქცია, ანუ ველი ან მეთოდი), ანუ კლასის private: ნაკვეთში განსაზღვრული წევრი წვდომადია მხოლოდ ამ კლასის წევრი ფუნქციებიდან და მეგობარი კლასების წევრი ფუნქციებიდან, აგრეთვე ამ კლასის მეგობარი ფუნქციებიდან.
- კლასის protected: ნაკვეთში განსაზღვრული წევრი (წევრი მონაცემი ან წევრი ფუნქცია) წვდომადია მხოლოდ ამ კლასის, მეგობარი კლასების და წარმოებული კლასების წევრი ფუნქციებიდან, , აგრეთვე ამ კლასის მეგობარი ფუნქციებიდან.
- კლასის public: ნაკვეთში განსაზღვრული წევრი (წევრი მონაცემი ან წევრი ფუნქცია) წვდომადია მხოლოდ ამ კლასის ფუნქციებიდან და მეგობარი ფუნქციებიდან; წარმოებული კლასების წევრი ფუნქციებიდან და მეგობარი ფუნქციებიდან, ობიექტების სახელიდან და პოინტერებიდან; ამ კლასის და წარმოებული კლასის მეგობარი კლასების წევრი ფუნქციებიდან.

ამავე დროს, ფუძის წვდომის სტატუსი იცვლება წარმოებულ კლასში და ესა დამოკიდებულია წვდომის_რეჟიმზე. იმისათვის რომ გავარკვიოთ, თუ როგორი წვდომა აქვს წარმოებული კლასიდან ფუძე-კლასის სხვადასხვა წვდომის ცვლადს (და ზოგადად წევრს), განვიხილოთ მარტივი მაგალითები უფრო ღრმა იერარქიით.

ა. ღია წვდომა წარმოებულ კლასზე:

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

class A
{
public:
	int x;
	A() { x = 1; y = 2; z = 3; }
protected:
	int y;
private:
	int z;
};

class B : public A
{};

class BPub : public B
{
public:
	void publicAccessX() {
		std:cout << "Access from sub-sub-class: x=" << x << std::endl;
	}
	void publicAccessY() {
		std:cout << "Access from sub-sub-class: y=" << y << std::endl;
	}
};

int main()
{
	BPub b;
	b.publicAccessX();
	b.publicAccessY();
	cout << b.x << endl;
```

```
//	cout << b.y << endl;
}
```

B კლასი წარმოებულის ფუძე-კლასზე ღია წვდომით.

თუ კომენტარს ავხსნით, მივიღებთ კომპილირების შეცდომას. ასე მუშაობს. პროგრამა დრაივერის პირველი ორი სტრიქონი ნიშნავს, რომ წარმოებული B კლასი ხედავს ფუძის ღია და დაცულ წევრებს, ხოლო მესამე სტრიქონი გვეუბნება, რომ ფუძის დაცული წევრი წარმოებული კლასისთვის გახდა კერძო, ანუ წარმოებულის მემკვიდრე უკვე ველარ ხედავს მას.

ა. დაცული წვდომა წარმოებულ კლასზე:

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

class A
{
public:
	int x;
	A() { x = 1; y = 2; z = 3; }
protected:
	int y;
private:
	int z;
};

class B : protected A
{ };

class BPub : public B
{
public:
	void publicAccessX() {
		std:cout << "Access from sub-sub-class: x=" << x << std::endl;
	}
	void publicAccessY() {
		std:cout << "Access from sub-sub-class: y=" << y << std::endl;
	}
};

int main()
{
	BPub b;
	b.publicAccessX();
	b.publicAccessY();
//	cout << b.x << endl;
}
```

B კლასი წარმოებულის ფუძე-კლასზე დაცული წვდომით.

თუ კომენტარს ავხსნით, მივიღებთ კომპილირების შეცდომას. კომენტარით მუშაობს. პროგრამა დრაივერის პირველი ორი სტრიქონი ნიშნავს, რომ წარმოებული B კლასი ხედავს ფუძის ღია და დაცულ წევრებს, ხოლო მესამე სტრიქონი გვეუბნება, რომ ფუძის ღია წვდომის წევრი წარმოებული კლასისთვის გახდა დაცული, რადგან წარმოებულის მემკვიდრე უკვე ველარ ხედავს მას.

ა. დახურული წვდომა წარმოებულ კლასზე:

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

class A
{
public:
	int x;
	A() { x = 1; y = 2; z = 3; }
protected:
	int y;
private:
	int z;
};
class B : private A
{ };
class BPub : public B
{
public:
	void publicAccessX() {
		std:cout << "Access from sub-sub-class: x=" << x << std::endl;
	}
	void publicAccessY() {
		std:cout << "Access from sub-sub-class: y=" << y << std::endl;
	}
};

int main()
{
	BPub b;
	b.publicAccessX();
	b.publicAccessY();
}
```

B კლასი წარმოებულის ფუძე-კლასზე დახურული წვდომით.

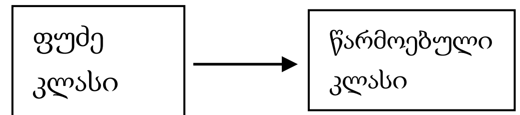
ამჯერად პროგრამა-დრაივერ ვერ იძახებს ვერცერთ მეთოდს. შეცდომის განმარტებაში წერია, რომ არ გვაქვს წვდომა შესაბამის ცვლადებზე. ცხადია, ეს ნიშნავს რომ რომ ფუძის ღია და დაცული წვდომის წევრი წარმოებული კლასისთვის გახდა კერძო, რადგან წარმოებულის მემკვიდრემ დაკარგა მათზე წვდომა.

ეს დასკვნები შეგვიძლია განზოგადებულია შემდეგ ცხრილში, სადაც სტრიქონები ასახავს ფუძე-კლასის რომელიღაც მოცემული წევრის წვდომის განმსაზღვრელს, სვეტები ასახავს ფუძიდან კლასის წარმოების რეჟიმს, ხოლო მათ თანაკვეთაზე წერია, წარმოებული კლასიდან ფუძის მოცემულ წევრზე წვდომის განმსაზღვრელს:

		მემკვიდრეობის რეჟიმი		
		public	protected	private
ფუძე-კლასში წევრზე წვდომის განმსაზღვრელი	public	public	protected	private
	protected	protected	protected	private
	private	არ გვაქვს წვდომა	არ გვაქვს წვდომა	არ გვაქვს წვდომა

<<< მემკვიდრეობის ტიპები

1. მხოლობითი მემკვიდრეობა - ერთი კლასია ნაწარმოები ერთი ფუძიდან.




```
class derivedClassName : accessMode baseClass
{
    //წარმოებული კლასის ტანი
};
```

მარტივი მაგალითი:

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

// ფუძე კლასი
class Vehicle {
public:
    Vehicle() {  cout << "This is a Vehicle" << endl; }
};
// წარმოებული კლასი
class Car : public Vehicle {

};

int main()
{
    // ობიექტის შექმნა გამოაცოცხლებს კონსტრუქტორს
    Car obj;
}
```

შედეგი:

```
This is a Vehicle
Press any key to continue . . .
```

2. მრავლობითი მემკვიდრეობა. ერთი კლასი შესაძლოა ერთზე მეტი კლასისგან იყოს ნაწარმოები. შემდეგი მარტივი მაგალითი აღებულია წიგნიდან: Jesse Liberty, Siddhartha Rao, Bradley Jones. Sams Teach Yourself C++ in One Hour a Day, 2009 გვ. 279.

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

class Mammal
{
public:
    void FeedBabyMilk() {cout << "Mammal : Baby says glug!" << endl;}
};

class Reptile
{
public:
    void SpitVenom(){   cout << "Reptile : Shoo enemy!Spits venom!" << endl;}
};

class Bird
{
public:
    void LayEggs(){     cout << "Bird : Laid my eggs, am lighter now!" << endl;     }
};

class Platypus : public Mammal, public Bird, public Reptile
{
```

```
public:
    void Swim()  {      cout << "Platypus : Voila, I can swim!" << endl;     }
};

int main()
{
    Platypus realFreak;
    realFreak.LayEggs();
    realFreak.FeedBabyMilk();
    realFreak.SpitVenom();
    realFreak.Swim();
}
```

შედეგი:

```
Bird : Laid my eggs, am lighter now!
Mammal : Baby says glug!
Reptile : Shoo enemy!Spits venom!
Platypus : Voila, I can swim!
Press any key to continue . . .
```

კლასებისთვის მრავლობითი მემკვიდრეობითობა C++ ენაში შედარებით განვითარებულია და რთული სხვა ენებთან შედარებით.

3. **მრავალდონიანი მემკვიდრება.** ამ დროს, წარმოებული კლასიდან იწარმოება სხვა კლასი. განვიხილოთ მარტივი მაგალითი.

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

class Vehicle
{
public:
    Vehicle()    {      cout << "This is a Vehicle" << endl;    }
};
class FourWheeler : public Vehicle
{
public:
    FourWheeler(){      cout << "Objects with 4 wheels are vehicles" << endl;      }
};
// sub class derived from two base classes
class Car : public FourWheeler
{
public:
    Car()  {      cout << "Car has 4 Wheels" << endl;     }
};

int main()
{
    Car obj;
}
```

შედეგი:

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
Press any key to continue . . .
```

ცხადია, განხილული ტიპების კომბინაციებით შესაძლებელია უფრო რთული იერარქიების მოფიქრება და განხორციელება.

<<< მეგობარი კლასი

მემკვიდრეობის რეჟიმების განხილვისას ჩვენ ვახსენეთ მეგობარი კლასები და მეგობარი ფუნქციები. მეგობარი ფუნქციები ჩვენ განვიხილეთ. ასეთები იყო შეტანისა და გამოტანის ოპერატორები განვიხილოთ სხვა, უფრო მარტივი და შედარებით ხელოვნური მაგალითი ჩენი კურსის ერთ-ერთი წყაროდან.

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

class Human
{
private:
	string Name;
	int Age;

	friend void DisplayAge(const Human& Person);

public:
	Human(string InputName, int InputAge)
	{
		Name = InputName;
		Age = InputAge;
	}
};

void DisplayAge(const Human& Person)
{
	cout << Person.Age << endl;
}

int main()
{
	Human FirstMan("Adam", 25);
	cout << "Accessing private member Age via friend : ";
	DisplayAge(FirstMan);
}
```

შედეგით:

```
Accessing private member Age via friend : 25
Press any key to continue . . .
```

ახლა, იგივე სტილში განვიხილოთ მეგობარი კლასის მაგალითი:

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

class Human
{
private:
	string Name;
	int Age;
```

```
	friend class Utility;

public:
	Human(string InputName, int InputAge)
	{
		Name = InputName;
		Age = InputAge;
	}
};

class Utility
 {
public:
	static void DisplayAge(const Human& Person)
	{
		cout << Person.Age << endl;
	}
};

int main()
 {
	Human FirstMan("Adam", 25);
	cout << "Accessing private member Age via friend class : ";
	Utility::DisplayAge(FirstMan);
}
```

შედეგით:

```
Accessing private member Age via friend class : 25
Press any key to continue . . .
```

<<< სავარჯიშოები

გაარჩიეთ შემდეგ მისამართებზე

- http://www.tutorialdost.com/Cpp-Programming-Tutorial/40-CPP-Friend-Class.aspx
- https://www.geeksforgeeks.org/friend-class-function-cpp/
- https://www.programiz.com/cpp-programming/friend-function-class
- https://beginnersbook.com/2017/09/friend-class-and-friend-functions/

მოყვანილი მეგობარი კლასების მაგალითები