თემა 8: ფუძე-კლასის ფუნქციების გადაფარვა (overriding) და დამალვა (hiding). ვირტუალური ფუნქციები

საკითხები:

- ფუძე-კლასის ფუნქციების გადაფარვა (overriding) და დამალვა (hiding)
- ფუძე-კლასის მეთოდების გამოძახება >>>
- ვირტუალური მეთოდები >>>
- ვირტუალური მეთოდების გამოყენების მაგალითები >>>
- წმინდა ვირტუალური ფუნქციები და ვირტუალური დესტრუქტორები >>>
- სავარჯიშოები >>>

ფუძე-კლასის ფუნქციების გადაფარვა (overriding) და დამალვა (hiding). როდესაც წარმოებულ კლასში და ფუძე-კლასში შექმნილია ერთნაირი ხელწერის (ე.ი. ერთანირი სახელის და პარამეტრების), მაგრამ განსხვავებული იმპლემენტაციის მქონე მეთოდები, მაშინ ამბობენ, რომ **წარმოებული კლასის მეთოდი ფარავს ფუძე-კლასის მეთოდს** წარმოებულ კლასში. როდესაც იქმნება კლასის ობიექტი, იგი გამოიძახებს კორექტულ მეთოდს, - ფუძე-კლასის ობიექტი იძახებს მეთოდს საბაზო იმპლემენტაციით, ხოლო წარმოებული ობიექტი- წარმოებული იმპლემენტაციით.

თუ ფუძე-კლასში რომელიმე მეთოდი გადატვირთულია რამდენიმე ვარიანტად, და ერთ-ერთი გადატვირთული მეთოდი გადაფარულია წარმოებული კლასის მეთოდის მიერ, მაშინ ფუძე კლასში იგივე სახელის მქონე მეთოდები **იმალებიან,** ანუ მათი გამოძახება აღარ შეუძლიათ წარმოებული კლასის ობიექტებს.

განვიხილოთ მარტივი მაგალითი:

```
#include<iostream>
#include<string>
using namespace std;
class Animal
{
public:
	void animalSound()
	{
		cout << "This is a dumb animal" << endl;
	}
	void animalSound(string s)
	{
		cout << "Something like as  " << s << endl;
	}
};
class Dog : public Animal
{
public:
	void animalSound()
	{
		cout << "Woof" << endl;
	}
};
int main()
{
	Dog* obj = new Dog();
	obj->animalSound();
}
```

აქ მეორე სტრიქონი სტრიქონი გამოიძახებს წარმოებული კლასის მეთოდს (წევრ-ფუნქციას), ანუ ფუძე-კლასის მეთოდი გადაიფარება. შედეგად:

```
Woof
```

```
Press any key to continue . . .
```

ახლა, პროგრამა დრაივერში თუ დავამატებთ კიდევ ორ სტრიქონს:

```
	cout << "Dog as a very smart animal: ";
	obj->animalSound("La-La-La");
```

უკვე მივიღებთ კომპილაციის შეცდომას, რადგან ფუძე-კლასის ეს მეთოდი დამალულია და მას წარმოებული კლასიდან ვერ გამოვიძახებთ.

როგორც ვხედავთ, წევრი-ფუნქციების (მეთოდების) გადატვირთვასა და გამეორებას შორის არის შემდეგი განსხვავებები:

1) ფუნქციის გადატვირთვა ხდება ერთსა და იმავე კლასში, როდესაც აღვწერთ ერთი და იმავე ფუნქციებს სხვადასხვა არგუმენტებით. ფუნქციის გადაწერა ხდება წარმოებულ კლასში, როდესაც წარმოებული კლასი გადაწერს მშობლი კლასის ფუნქციას.

2) ფუნქციის გადატვირთვისას ფუნქციის ხელმოწერა უნდა იყოს განსხვავებული ყველა გადატვირთული ფუნქციისათვის. ფუნქციის გადაწერისას ორივე ფუნქციის ხელმოწერა გადაწერილი და გადასაწერი ფუნქციების) უნდა იყოს ერთნაირი.

3) გადატვირთვა მოხდება კომპილაციის დროს, ხოლო გადაწერა ხდება ფუნქციის შესრულების დროს.

4) ფუნქციის გადატვირთვისას შეგვიძლია ვიქონიოთ გადატვირთული ფუნქციების ნებისმიერი რაოდენობა. ფუნქციის გადაწერისას გვაქვს მხოლოდ ერთი გადამწერი ფუნქცია წარმოებულ კლასში.

<<< ფუძე-კლასის მეთოდების გამოძახება

ცხადია ეს არ ნიშნავს, რომ წარმოებული კლასიდან ვერ მივწვდებით ფუძე-კლასის მეთოდებს. საკმარისია გამოვიყენოთ ხილვადობის „::“ ოპერატორი. მაგალითად თუ პროგრამა დრაივერს ასეთ სახეს მივცემთ:

```
int main()
{
	Dog* obj = new Dog();
	obj->animalSound();

	cout << "Dog as a primitive animal: ";
	obj->Animal::animalSound();

	cout << "Dog as a very smart animal: ";
	obj->Animal::animalSound("La-La-La");
}
```

შედეგი იქნება ასეთი:

```
Woof
Dog as a primitive animal: This is a dumb animal
Dog as a very smart animal: Something like as  La-La-La
Press any key to continue . . .
```

<<< ვირტუალური მეთოდები

როგორც ვიცით კონსტრუქტორების შესახებ ლექციიდან, ფუძე-კლასის პოინტერს შეუძლია მიინიჭოს new ოპერატორით აგებული წარმოებული კლასის მისამართი. მაგალითად თუ პროგრამა დრაივერს მივცემთ სახეს:

```
int main()
{
   Animal *obj;
```

```
    obj = new Dog();
    obj->animalSound();
}
```

ის კორექტულად იმუშავებს და მოგვცემს შემდეგ შედეგს:

```
This is a dumb animal
Press any key to continue . . .
```

რაც ნიშნავს, რომ ფუძე-კლასის ობიექტის მისამართიდან მხოლოდ ფუძე კლასის წევრების გამოძახება შეგვიძლია. თუმცა ეს არ ნიშნავს, რომ წარმოებული კლასი მთლიანად არაა აგებული.

შევნიშნოთ, რომ ზოგადად, ფუძე-კლასის პოინტერს **არ** შეუძლია მიინიჭოს წარმოებული კლასის მისამართი. მაგალითად, შემდეგი კოდი **არ** კომპილირდება:

```
int main()
{
    Animal *obj;
    Dog muria();
    obj = &muria;
}
```

C++ საშუალებას გვაძლევს, რომ ფუძე კლასის ობიექტის მისამართიდან მივწვდეთ წარმოებული კლასის ზოგიერთ მეთოდს, რომელიც მონიშნულია როგორც ვირტუალური. მექანიზმი მარტივია და ეფექტური: ფუძე კლასი იმახსოვრებს ვირტუალური ფუნქციის მონაცემებს, ხოლო წარმოებული კლასის ობიექტის კონსტრუირების მომენტში ჯერ აიგება ფუძე ნაწილი, შემდეგ წარმოებული. წარმოებული ობიექტის აგების დროს, როდესაც კომპილერს შეხვდება ვირტუალური ფუნქცია, მის მონაცემებს ჩაწერს უკვე აგებული ფუძე ნაწილის იგივე სახელის მქონე ვირტუალური ფუნქციის მონაცემებად. ანუ, ისევე როგორც განხილულ მაგალითში, ფუძე კლასის ობიექტის მისამართიდან გამოიძახება მეთოდი, რომელიც ფუძე კლასშია აღწერილი. ოღონდ, თუ ეს მეთოდი ვირტუალურია, იგი უკვე ჩანაცვლებულია წარმოებული კლასის იგივე სახელის მქონე მეთოდით. ტექნიკურად, ეს კეთდება ე.წ. V-ცხრილების საშუალებით, რომლებზეც ახლა არ გავამახვილებთ ყურადღებას.

განვიხილოთ იგივე კლასები ვირტუალური ფუნქციების გამოყენებით:

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

class Animal
{
public:
	virtual void animalSound()
	{
		cout << "This is a dumb animal" << endl;
	}
	void animalSound(string s)
	{
		cout << "Something like as  " << s << endl;
	}
};

class Dog : public Animal
{
public:
	void animalSound()
	{
		cout << "Woof" << endl;
	}
};
```

```
int main()
{
	Animal *obj;
	obj = new Dog();
	obj->animalSound();
}
```

ამის შედეგს უკვე აქვს სახე:

```
Woof
Press any key to continue . . .
```

<<< ვირტუალური მეთოდების გამოყენების მაგალითები

ვირტუალური ფუნქციების გამოყენება განსაკუთრებით ეფექტურია, როდესაც რაიმე კონტეინერში ვინახავთ ფუძე-კლასის ობიექტების მისამართებს, რომლებიც სინამდვილეში ინახავენ სხვადასხვა წარმოებული კლასების ობიექტების მისამართებს. რეალურად, კონტეინერში ვინახავთ ინფორმაციას სხვადასხვა ტიპის ობიექტების შესახებ. შემდეგ, ამ მისამართებიდან ჩვენ შეგვიძლია მივწვდეთ წარმოებული ობიექტების ვირტუალურ მეთოდებს.

მაგალითის სიმარტივისთვის, განხილულ კლასებში დავტოვოთ მხოლოდ თითო ვირტუალური მეთოდი, რომელიც სტრინგად შეინახავს მონაცემებს გამომძახებელი ობიექტის შესახებ:

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

class Animal
{
public:
	virtual string toString()
	{
		return (string)typeid(this).name();
	}
};

class Dog : public Animal
{
public:
	string toString()
	{
		return "Base " + this->Animal::toString() + ": " +
(string)typeid(this).name();
	}
};

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

	p[0] = new Animal;
	p[1] = new Dog;
	p[2] = new Animal;

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

რომლის შედეგი არის:

```
class Animal *
Base class Animal *: class Dog *
class Animal *
Press any key to continue . . .
```

განვიხილოთ კიდევ ერთი მარტივი მაგალითი კლასების იერარქიისა, სადაც უკვე ორი კლასი წარმოებულია საერთო წინაპრისგან.

```
#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 . . .
```

<<< წმინდა ვირტუალური ფუნქციები და ვირტუალური დესტრუქტორები

როგორ მოვიქცეთ, თუ ფუძე-ობიექტის მისამართიდან გვინდა მივწვდეთ ისეთ ფუნქციებს, რომლებიც ვერ განიმარტება ფუძე კლასისთვის? მაგალითად, წარმოვიდგინოთ, რომ გვინდა სამი

კლასის შექმნა, ერთი აღწერს კვადრატს, მეორე წრეს, მესამე მართკუთხედს. შემდეგ, გვინდა მათი რამდენიმე ობიექტის შექმნა და შენახვა ვექტორში, იმ მიზნით რომ პერიოდულად გამოვბეჭდოთ მათი ფართობები (ან პერიმეტრები). რადგან ვექტორში მხოლოდ ერთი და იმავე კლასის ობიექტები უნდა მოთავსდეს, ამიტომ, წინა მაგალითების გამოცდილების გათვალისწინებით, სასურველი იქნება ისეთი კლასის მოფიქრება, რომელიც სამივესთვის საერთო წინაპარი იქნება და ექნება ისეთი ვირტულაური ფუნქცია, როგორიც ჩვენ გვჭირდება.

C++ საშუალებას იძლევა რომ განიმარტოს აბსტრაქტული კლასი, რომელშიც ჩამოთვლილია წმინდა ვირტუალური ფუნქციები. თუ წინასწარ ვიცით, რა მეთოდები დაემატება წარმოებულ კლასებში, რომლებზე წვდომაც აუცილებელი იქნება, მაშინ აბსტრაქტულ კლასში ისინი უბრალოდ ჩამოითვლება როგორც ვიტუალური და მიეთითება, რომ მათი იმპლემენტირება ამ კლასში არაა აუცილებელი. ეს ხდება = 0; -ის დამატებით. მაგალითად:

```
virtual double area() = 0;
```

წარმოებულ კლასში ჩვეულებრივად მოხდება ამ მეთოდების იმპლემენტირება.

ვირტუალური დესტრუქტორი ყოველთვის უნდა გამოვიყენოთ, როდესაც კლასში გამოიყენება ერთი ვირტულური ფუნქცია მაინც. ეს ზოგადი და გავრცელებული წესია.

ვნახოთ რა როგორ გადაიჭრება ბოლო ამოცანა წმინდა ვირტუალური ფუნქციების გამოყენებით. სიმარტივისთვის, კლასების ველებზე წვდომა იყოს ღია.

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

class Figure
{
public:
	Figure() { }
	virtual ~Figure() { }
	virtual double area() = 0;
};

class Circle : public Figure
{
public:
	double radius;
	Circle(double itsRadius) : radius(itsRadius) { }
	virtual ~Circle() { }
	virtual double area() { return 3.14*radius*radius; }
};

class Square : public Figure
{
public:
	double side;
	Square(double itsSide) : side(itsSide) { }
	virtual ~Square() { }
	virtual double area() { return side*side; }
};

class Rectangle : public Figure
{
public:
	double length;
	double width;
	Rectangle(double itsLength, double itsWidth) : length(itsLength), width(itsWidth)
{ }
```

```
        virtual ~Rectangle() { }
        virtual double area() { return length*width; }
};
int main()
{
        vector<Figure*> v;
        v.push_back(new Square(7));
        v.push_back(new Rectangle(5,4.01));
        v.push_back(new Circle(7));

        for (int i = 0; i < 3; i++)
                cout << v[i]->area() << endl;
        for (int i = 0; i<3; i++)
                delete v[i];
}
```

პროგრამის შესრულების შედეგად იბეჭდება ვექტორში ჩაწერილი ფიგურების ფართობები. კონტეინერი ერთგვაროვანია, თუმცა ვირტუალობის დახმარებით სინამდვილეში ვმუშაობთ არაერთგვაროვან წარმოებულ ობიექტებთან.

<<< სავარჯიშოები:

1. Cplusplus.com –ზე მოიძიეთ მასალა ვირტუალური ფუნქციების შესახებ.
2. შექმენით კლასი დაქირავებული (`Employee`) ველებით სახელი და ხელფასი (name, salary). შექმენით მისგან წარმოებული კლასი მენეჯერი (Manager), რომელსაც დაემატება ველი დანამატი (Bonus). სიმარტივისთვის, ველებზე წვდომა იყოს ღია. ორივე კლასში გააკეთეთ მეთოდი (წევრი-ფუნქცია) `double getSalary()`, რომელიც ფუძე-კლასსი დააბრუნებს ხელფასს, ხოლო წარმოებულში - ხელფასის და დანამატის ჯამს. როგორ მოვახერხოთ, რომ რამდენიმე სხვადასხვა რანგის თანამშრომლის მისამართი შევინახოთ კონტეინერში (მაგალითად ვექტორში) და საჭიროების შემთხვევაში გამოვბეჭდოთ ყველა მათგანის ხელფასი?
3. იგივე კლაში, ფუძე-კლასის ველებისთვის გამოიყენეთ protected, ხოლო წარმოებულის ველებისთვის private განმსაზღვრელები და შეიტანეთ საჭირო ცვლილებები კოდში.
4. აარჩიეთ შემდეგ გვერდზე გარჩეული კოდები:
   https://www.includehelp.com/cpp-programs/cpp-inheritance-programs-to-demonstrate-example-of-simple-inheritance.aspx