თემა 7. ფუძე და წარმოებული კლასები. წარმოებული ობიექტის აგება

საკითხები:

ფუძე კლასი და წარმოებული კლასი

- ფუძე კლასი გასაყიდი პროდუქტების აღწერისთვის >>>
- კლასი კერძო სახის პროდუქტისთვის, რომლის ობიექტები ორი ნაწილისგან - ფუძისგან და წარმოებული ნაწილისგან შედგებიან >>>
- კლასი სამგანზომილებიანი სივრცის წერტილისთვის გადატვირთული ოპერატორებით >>>

წარმოებული ობიექტების აგება >>>

სავარჯიშოები >>>

ფუძე და წარმოებული კლასი

განვიხილოთ რამდენიმე მაგალითი, როდესაც უკვე არსებული კლასის საფუძველზე იქმნება ახალი კლასი რადგან ასე უფრო მოსახერხებელია. მოტივაცია იმაში მდგომარეობს, რომ წარმოებულ კლასში ფუძე (საბაზო, წინაპარი) კლასის ველებს შეგვიძლია დავუმატოთ სხვა ველები. ასევე შეგვიძლია ფუძე კლასის მეთოდების გამრავალფეროვნება. ამის გაკეთება შეიძლება მაშინ, როდესაც ორ კლასს შორის არსებობს „არის“ მიმართება. მაგალითად, ტბა არის გეოგრაფიული ობიექტი. ამიტომ, თუ უკვე არსებობს კლასი გეოგრაფიული ობიექტისთვის, შესაძლოა უმჯობესი იყოს მისი განვითარება, ვიდრე ახალი ობიექტის შექმნა ნულიდან.

ან, როგორც მომდევნო (ძალიან გამარტივებულ) მაგალითშია, თუ შექმნილი გვაქვს კლასი რომელიც მაღაზიაში შემოსული ნებისმიერი პროდუქტის აღრიცხვისთვის შეგვიძლია გამოვიყენოთ, მაშინ ყოველი ახალი სახის პროდუქტისთვის მხოლოდ განხვავებული მახასიათებლების აღრიცხვა იქნება საჭირო, რაც უფრო სწრაფად გაკეთდება და შეცდომის ალბათობაც შემცირდება (მრავალი სხვადასხვა კერძო სახის პროდუქტში არ იქნება საჭირო ერთი და იმავე ველების გათვალისწინება).

რადგან ახლა პირველად შევეხებით მემკვიდრეობის რთულ თემას, ამიტომ ნიმუშად მოყვანილ კლასებში გავიმარტივებთ საქმეს ინკაპსულაციასთან და ინფორმაციის დაფარვასთან საშუალებით, შევეცდებით რა თავის არიდებას getter და setter ფუქნციებისთვის.

<<< ფუძე კლასი გასაყიდი პროდუქტების აღწერისთვის

```
#include<iostream>
#include<string>
using namespace std;
class goods
{
public:
	string name;
	int receiptYear;
	double price;
	goods(const string& nameValue, const int& receiptYearValue, const double&
		priceValue)
		:name(nameValue), receiptYear(receiptYearValue), price(priceValue) {}
	string toString()
	{
		return ((string)typeid(this).name() + ":  name: " + name + ", receiptYear: "
			+ to_string(receiptYear) + ":  price: " + "$" + to_string(price));
	}
};
int main()
{
	goods product("Something", 1917, 100);
	cout << product.toString() << endl;
}
```

ახლა, წარმოვიდგინოთ რომ გაყიდვაში შემოვიდა ახალი მოწყობილობის პარტია. შევქმნათ კლასი მისი ობიექტების აღწერისთვის.

<<< წარმოებული კლასი კერძო სახის პროდუქტისთვის, რომლის ობიექტები ორი ნაწილისგან - ფუძისგან და წარმოებული ნაწილისგან შედგებიან

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

class goods
{
protected:
	string name;
	int receiptYear;
	double price;
	goods() {}
	goods(const string& nameValue, const int& receiptYearValue, const double&
	    priceValue):name(nameValue),receiptYear(receiptYearValue),price(priceValue) {}
	string toString()
	{
		return ((string)typeid(this).name() + ":  name: " + name + ", receiptYear: "
			+ to_string(receiptYear) + ":  price: " + "$" + to_string(price));
	}

};
class computer: public  goods
{
public:
	int harDisc;
	int generation;
	computer() {}
		computer(const string& nameValue, const int& receiptYearValue, const double&
				priceValue,const int& harDiscValue, const int& generationValue
		):goods(nameValue, receiptYearValue, priceValue),harDisc(harDiscValue),
		  generation(generationValue) {}
	string toString()
	{
		return ( this->goods::toString()+'\n' + (string)typeid(this).name()
		+ ":  harDisc: "+ to_string(harDisc) + ":  generation: "
		+ to_string(generation));
	}
};


int main()
{
	computer  aaa("newGeoComp", 2022, 100000, 10000,17);
	cout << aaa.toString() << endl;
}
```

როგორც ვხედავთ, ახალი კლასის კონსტრუქტორი ჯერ გამოიძახებს ფუძე-კლასის კონსტრუქტორს, რომელიც შექმნის მის ფუძეს, შემდეგ კონსტრუქტორის დანარჩენი ნაწილი ამ ფუძეზე „დააშენებს“ ობიექტის დანარჩენ ნაწილს.

ფუძე კლასის ველებიც და მეთოდებიც დახურულია ყველაფრისთვის, გარდა საკუთარი და წარმოებული კლასის მეთოდებისთვის.

წარმოებული კლასის ერთ-ერთი მეთოდი იყენებს ფუძე-კლასის (ინფორმაციის სტრინგად წარმოდგენის) მეთოდს. ამისათვის საჭირო ხდება მისი გამოძახება „სრული სახელით“, კლასის მითითებით, რადგან იგივე სახელის მქონე მეთოდი წარმოებულ კლასსაც აქვს. ეს თემა

მნიშვნელოვანია და უკავშირდება პოლიმორფიზმს. მას სხვადასხვა კუთხეებიდან შევხედავთ მომდევნო მეცადინეობებზე.

მემკვიდრეობის თვალსაზრისით C++ მდიდარ შესაძლებლობებს ქმნის რომ აიგოს კლასების მრავალფეროვანი იერარქიები. მემკვიდრეობის თემა აგრეთვე გაგრძელდება ჩვენს კურსში.

<<< კლასი სამგანზომილებიანი სივრცის წერტილისთვის გადატვირთული არითმეტიკული ოპერატორებით. განვიხილოთ კიდევ ერთი მაგალითი. ამჯერად ფუძე კლასი სტანდარტული ბიბლიოთეკიდანაა აღებული, რაც მხოლოდ ამარტივებს კოდს.

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

class triple : public pair<double, double>
{
public:
	double third;
	triple() {}
	triple(double, double, double);
	triple& operator+=(const triple&);
	triple& operator *=(double);
	string toString();
};
triple::triple(double x, double y, double z) : pair<double, double>(x, y), third(z) {}
triple& triple::operator+=(const triple& theOther)
{
	first += theOther.first;
	second += theOther.second;
	third += theOther.third;
	return *this;
}
triple& triple::operator*=(double a)
{
	first *= a;
	second *= a;
	third *= a;
	return *this;
}
string triple::toString()
{
	return ((string)typeid(this).name() + ":  first: " + to_string(first)) + ",
		 second: " + to_string(second) + ":  third: " + to_string(third);
}

int main()
{
	triple x(1.1, 4.5, 1.0);
	triple y(11.1, 11.1, 11.1);
	cout << "x is:";
	cout << x.toString() << endl;
	cout << "y is:";
	cout << y.toString() << endl;
	x += y;
	cout << "x after  x += y:" << x.toString() << endl;

	cout << "And x after x *= 10.0:" << endl;
	x *= 10.0;
	cout << x.toString() << endl;
}
```

ფუძე-კლასის ობიექტზე განაცხადის გაკეთება ავტომატურად იწვევს ფუძე-კლასის რომელიმე კონსტრუქტორის გააქტიურებას. თუ რომელი კონსტრუქტორი, - ამას განაცხადის სახე განსაზღვრავს.

ფუძიდან წარმოებული კლასის ობიექტზე განაცხადის გაკეთება ავტომატურად იწვევს ძალიან საინტერესო პროცესების გააქტიურებას. ჯერ ჩაირთვება ფუძე კლასის კონსტრუქტორი, შემდეგ წარმოებული კლასის. თუ კლასების იერარქია ორზე მეტ დონეს მოიცავს, მაშინ ჯერ აქტიურდება ყველაზე შორი წინაპრის რომელიმე კონსტრუქტორი (ვთქვათ, k_n), შემდეგ მისგან წარმოებული ერთ-ერთი კლასის ერთ-ერთი კონსტრუქტორი (ვთქვათ, k_{n-1}) და ა.შ. სულ ბოლოს აქტიურდება იმ კლასის კონსტრუქტორი, რომელზეც კეთდება განაცხადი (ვთქვათ, k_1). ეს კონსტრუქტორი ცალსახად განისაზღვრება განაცხადიდან. რაც შეეხება მისი ფუძე კლასის კონსტრუქტორს. თუ ბოლოს გამოძახებული კონსტრუქტორი თავსი იმპლემენტაციაში იძახებს თავისი ფუძე-კლასის რომელიმე კონსტრუქტორს, მაშინ სწორედ ეს გამოძახებული კონსტრუქტორი არის k_2, რომელიც გაეშვება k_1-ის წინ. თუ k_1 თავის იმპლემენტაციაში ცხადად არ იძახებს თავისი ფუძე კლასის პარამეტრიან კონსტრუქტორს, მაშინ k_1-ის წინ გაეშვება მისი ფუძე კლასის უპარამეტრო კონსტრუქტორი. და ა.შ., იგივე პრინციპით გაირკვევა თუ რომელი კონსტრქტორი გაეშვება k_2-ის წინ, k_3-ის წინ და ა.შ.

ახლა ჩვენ ვსაუბრობდით ე.წ. მარტივი მემკვიდრეობის შემთხვევაზე, როდესაც თითოეულ კლასს მხოლოდ ერთი საბაზო კლასი გააჩნია. რთული მემკვიდრეობის შემთხვევას მოგვიანებით შევეხებით,

განვიხილოთ

მაგალითი. ვთქვათ, ფუძე-კლასს ჰქვია ინტერვალი და მისი ერთადერთი ველია ინტერვალის სიგრძე. მისგან წარმოებული კლასია მართკუთხედი, რომელსაც ემატება ველი სახელად სიგანე. ორივე კლასში გადატვირთული გვაქვს შეტანა-გამოტანის ოპერატორები და თითო კონსტრუქტორი. პროგრამა დრაივერი ქმნის თითო-თითო ობიექტს. კონსტრუქტორებში დამატებულია კონსტრუქტორის სახელის ბეჭდვა, რაც გვაჩვენებს, თუ როდის რომელი კონსტრუქტორი გააქტიურდება.

განხორციელება: კლასების ფაილებს აქვს სახე (იმპლემენტაციები იგივე ფაილებშია):

```
//Interval.h
#pragma once
#include<iostream>
#include<string>
using namespace std;
class Interval
{
public:
	Interval(double itsValue);
	Interval(void);
	~Interval(void);
	double length;
	friend istream& operator>>(istream&, Interval &);
	friend ostream& operator<<(ostream&, const Interval &);
};
Interval::Interval(void) : length(0)
{
	cout << "Interval class default constructor!" << endl;
}
Interval::~Interval(void) {}
```

```
Interval::Interval(double itsValue)
{
	length = itsValue;
	cout << "Interval class constructor!" << endl;
}
istream& operator>>(istream& is, Interval & t)
{
	is >> t.length;
	return is;
}
ostream& operator<<(ostream& os, const Interval & t)
{
	os << "Interval object, length=" << t.length << endl;
	return os;

}
```

// ფაილი rectangle.h

```
#pragma once
#include "Interval.h"
class rectangle : public Interval
{
public:
	double breadth;
	rectangle(void);
	~rectangle(void);
	rectangle(double itsLenght, double itsBreadth);
	friend istream& operator>>(istream&, rectangle &);
	friend ostream& operator<<(ostream&, const rectangle &);
};

rectangle::rectangle(void) : breadth(0)
{
	cout << "rectangle class default constructor!" << endl;
}
rectangle::~rectangle(void)
{
}
rectangle::rectangle(double itsLenght, double itsBreadth) :
	Interval(itsLenght), breadth(itsBreadth)
{
	cout << "Rectangle class constructor!" << endl;
}
istream& operator>>(istream& is, rectangle & r)
{
	is >> r.length >> r.breadth;
	return is;
}
ostream& operator<<(ostream& os, const rectangle & r)
{
	os << "Rectangle object, Length=" << r.length << ";  breadth=" << r.breadth <<
		endl;
	return os;
}
```

ხოლო პროგრამა დრაივერი არის:

```
#include "rectangle.h"
int main()
{
	Interval a;
	cout << "Enter double" << endl;
	cin >> a;
```

```
	cout << a;

	rectangle r;
	cout << "Enter double+double" << endl;
	cin >> r;
	cout << r << endl;

	Interval a1(13.21);
	cout << a1;
	rectangle r1(11, 22);
	cout << r1 << endl;
}
```

პროგრამის შესრულების შედეგად მივიღებთ:

```
Interval class default constructor!
Enter double
11.1
Interval object, length=11.1
Interval class default constructor!
rectangle class default constructor!
Enter double+double
100.99
3.14
Rectangle object, length=100.99;  breadth=3.14

Interval class constructor!
Interval object, length=13.21
Interval class constructor!
Rectangle class constructor!
Rectangle object, length=11;  breadth=22

Press any key to continue . . .
```

იგივე შედეგს მივიღებთ, თუ დინამიკურ ობიექტებს შევქმნით, ოღონდ იგივე კონსტრუქტორების გამოყენებით.

თუ მართკუთხედის კლასის პარამეტრიანი კლასის იმპლემენტაციად ავიღებთ:

```
rectangle::rectangle(double itsLenght, double itsBreadth)
{
	length = itsLenght;   breadth = itsBreadth;
	cout << "rectangle class constructor!" << endl;
}
```

მაშინ საბაზო კლასისთვის მოხდება უპარამეტრო კონსტრუქტორის გამოძახება.

```
Interval class default constructor!
Enter double
11.1
Interval object, length=11.1
Interval class default constructor!
rectangle class default constructor!
Enter double+double
321.23
2201.99
Rectangle object, length=321.23;  breadth=2201.99

Interval class constructor!
Interval object, length=13.21
Interval class default constructor!
Rectangle class constructor!
Rectangle object, length=11;  breadth=22

Press any key to continue . . .
```

<<< სავარჯიშოები:

1. cplusplus.com -ზე ნახეთ ინფორმაცია მემკვიდრეობითობის (inheritence) და ოპერატორების გადატვირთვის შესახებ.
2. კლასში `pair<double, double>` გადატვირთეთ წყვილის ნამდვილ რიცხვზე გამრავლების ოპერატორი.
3. შექმენით კლასი tripleInt, რომელიც შედგება მთელი ველებისგან. მაგალითი 3-ის cplusplus.com-ის და დახმარებით შეეცადეთ გადატვირთოთ შეტანა-გამოტანის ოპერატორები.
4. შექმენით კლასი ზღვისთვის და მთისთვის, რომლის წინაპარი კლასი იქნება `GaographicObject`.
5. შექმენით კლასები: მონაკვეთი, მართკუთხედი, ყუთი. გაითვალისწინეთ მემკვიდრეობითობა და ინფორმაციის დამალვა.
6. მეორე მაგალითის `goods` კლასში გაითვალისწინეთ `public:` წვდომის წევრები, ხოლო კლასში `computer – private:` წვდომის წევრები.
7. შექმენით კლასის თარგი სამეულისთვის (სამივე სხვადასხვა ტიპის), ფუძე კლასად გამოიყენეთ `pair<, >` კლასი. შემდეგ, გადატვირთეთ ინდექსის ოპერატორი.
8. ბოლო მაგალითის ანალოგიით, გაარკვიეთ თუ როგორი თანმიმდევრობით იმუშავებენ დესტრუქტორები წარმოებული ობიექტის ამოღების (delete) პროცესში.