Računske vježbe 6

Programabilni uređaji i objektno orijentisano programiranje

Projektovati klase `Circle` (krug) i `Square` (kvadrat) koje su izvedene iz klase `Figure` (figura). Klasa figura sadrži težiste kao zajedničku katakteristiku za sve figure, metodu koja omogućava pomjeraj težista za zadatu vrijednost i virtuelne metode obim, površina i čitaj. Izvedene klase treba da imaju specifične metode za računanje obima i površine kao i očitavanje odgovarajućih podataka članova.

```
#include <iostream>
#include <cmath>

using namespace std;

class Point
{
private:
    double x; // koordinate
    double y;
public:
    Point(double x = 0, double y = 0) : x(x), y(y) {}
    double getX() const
    {
        return x;
    }
    double getY() const
    {
        return y;
    }
    void print()
    {
        cout << x << "," << y;
    }
};

const Point ORIGIN; //koordinatni pocetak

class Figure
{
private:
    Point center; // teziste figure
public:
    Figure(Point center = ORIGIN) : center(center) {} // smjestamo figuru u
   koordinatni pocetak
    virtual ~Figure() { cout << "Unistavamo Figure (osnovni objekat)" << endl; }
    void shift(double x, double y) // pomjeraj tezista
    {
        center = Point(center.getX() + x, center.getY() + y);
    }
    virtual double perimeter() const = 0; // obim
    virtual double area() const = 0; // povrsina
```

```
    virtual void print()
    {
        cout << " T=";
        center.print();
    }
};

class Circle : public Figure
{
private:
    double radius;
public:
    Circle(double radius = 1, Point center = ORIGIN) : Figure(center), radius(radius)
    {}
    ~Circle() { cout << "Unistavamo Circle" << endl; }
    double perimeter() const
    {
        return 2 * radius * 3.14;
    }
    double area() const
    {
        return pow(radius, 2) * 3.14;
    }
    void print();
};

void Circle::print()
{
    cout << "U pitanju je krug: r=" << radius;
    Figure::print();
    cout << " O=" << perimeter() << ", P=" << area() << endl;
};

class Square : public Figure
{
private:
    double side; // stranica kvadrata
public:
    Square(double side = 1.0, Point center = ORIGIN) : Figure(center), side(side) {}
    ~Square() { cout << "Unistavamo Square" << endl; }
    double perimeter() const
    {
        return 4 * side;
    }
    double area() const
    {
        return pow(side, 2);
    }
    void print();
};

void Square::print()
{
    cout << "U pitanju je kvadrat a=" << side;
    Figure::print();
    cout << " O=" << perimeter() << " P=" << area() << endl;
};


```

```
int main()
{
    Figure *figures[4];
    figures[0] = new Circle; // prva figura je sada krug
    figures[1] = new Square; // druga figura je sada kvadrat
    figures[2] = new Circle(2, Point(3, 3));
    figures[3] = new Square(2.5, Point(1.3, 2));

    for(int j = 0; j < 4; j++)
        figures[j]->print(); // skraceno od (*obj).func()

    figures[0]->shift(1, 0.5);
    figures[1]->shift(0.5, 1);

    for(int j = 0; j < 2; j++)
        figures[j]->print();

    for(int j = 0; j < 4; j++)
    {
        cout << endl;
        delete figures[j]; // vrsimo dealociranje memorije
        figures[j] = 0;
    }
}
```

Klase koje opisuju podgrupu objekata s nekim dodatnim, specifičnim osobinama u odnosu na opštije grupe nazivaju se **izvedene klase**. Polazne klase, koje opisuju opštiju grupu objekata, nazivaju se **osnovne klase**. Izvođenje klase iz neke osnovne klase može biti javno, zaštićeno ili privatno, dodavanjem modifikatora `public`, `protected`, `private` ispred identifikatora osnovne klase. Kako način izvođenja utiče na vidljivost može se vidjeti u Tabeli 1. U odsustvu modifikatora podrazumijeva se privatno izvođenje. Izvedena klasa nasljeđuje sve članove svojih osnovnih klasa, ali se konstruktori, destruktor i metoda `operator=` kao i prijateljstva osnovne klase ne nasljeđuju. Jednostavan primjer izvođenja:

```
class A
{
protected: int i;
};
class B: public A
{
public: int j;
}
```

Mada se privatna polja mogu naslijediti, klasa koja ih naslijedi ih sadrži (zauzimaju memoriju), ali im ne može pristupiti. Da bi se izbjegla ova situacija, a da se ne bi narušila enkapsulacija, uvodi se modifikator `protected`. Pomoću ovog modifikatora činimo da za klasu B polje `A::i` ostane pristupačno, ali da se van klase B efektivno doživljava kao privatno polje. U praksi najveći značaj ima odnos *jeste* (objekat klase B jeste objekat klase A) između izvedenih i osnovnih klasa, pa se najčešće koristi javno izvođenje klasa, a zaštićeno i privatno samo u izuzetnim slučajevima. Izvedena klasa pored svojih članova posjeduje i jedan bezimeni primjerak svoje osnovne klase, ali se njemu (jer nema ime) ne može direktno pristupati već samo njegovim članovima pojedinačno. Prilikom definisanja konstruktora za izvedenu klasu, u listi inicijalizatora prije tijela konstruktora, mogu da se navedu i inicijalizatori za osnovne klase. Međutim, u fazi inicijalizacije primjeraka klase ona polja koja su naslijeđena još ne postoje, pa ne mogu da se navedu odvojeni inicijalizatori za njih. Moguća je samo inicijalizacija naslijeđenog podobjekta tipa osnovne klase kao cjeline. U našem zadatku smo za klasu `Square` imali:

```
Square(double side = 1.0, Point center = ORIGIN) : Figure(center), side(side) {}
```

način izvođenja	član osnovne klase		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Tabela 1: Stepeni zaštite članova osnovnih klasa u izvedenim klasama. Uočite da izvedena klasa može da zadrži vidljivost elemenata osnovne klase ili da vidljivost pogorša, ali ne može da popravlja vidljivost (recimo, sa privatne na javnu), jer bi na taj način ugrozili enkapsulaciju, odnosno potpunu kontrolu klase nad samom sobom.

Jedna od osnova objektno-orijentisanog programiranja je polimorfizam, pod kojim se podrazumijeva mogućnost ponašanja programa shodno tipovima obrađivanih objekata. U našem zadatku, prilikom obrade skupa geometrijskih figura površine figura se izračunavaju pomoću različitih formula u zavisnosti od vrste figura. U jeziku `C++` to omogućavaju virtuelne metode. Klase koje posjeduju bar jednu virtuelnu metodu nazivaju se polimorfne klase. Virtuelne metode deklarišu se dodavanjem modifikatora `virtual` na početku njihove deklaracije u osnovnoj klasi. Prilikom nadjačavanja, redefinisanja (engl. *override*) u izvedenim klasama, metodama s istim potpisom modifikator `virtual` se podrazumijeva, ne mora se eksplicitno navoditi. Deklaracije date virtuelne metode u osnovnoj i u svim izvedenim klasama moraju da budu istovjetne! Jedino u čemu se razlikuju jeste tip tekućeg objekta, tj. tip skrivenog parametra. Virtuelna metoda koja nije definisana, već samo deklarisana, u osnovnoj klasi naziva se **apstraktna metoda** ili **čista virtuelna metoda** i označava se sa `=0`. Primjer takve metode iz zadatka:

```
virtual double perimeter() const = 0;
```

kod klase `Figure` što je i očigledno jer je figura sama po sebi nedovoljno definisana da bi imala obim. Klasa koja sadrži bar jednu apstraktnu metodu naziva se **apstraktna klasa** i nije moguće stvarati objekte njenog tipa! Virtuelni mehanizam omogućava da se izvrši ona verzija virtuelne metode koja odgovara konkretnoj izvedenoj klasi. Pokazivači na osnovnu klasu mogu da pokazuju na objekte izvedenih klasa. Dakle, prilikom pozivanja virtuelne metode pomoću pokazivača ili reference na objekte osnovne klase, pozivaće se istoimena metoda one izvedene klase na čiji primjerak u tom momentu ukazuje dati pokazivač ili referenca. Ponašanje programa zavisi od tipa obrađivanog objekta, a ne od tipa identifikatora pomoću kojeg se dolazi do objekta. Dakle, korišćenje virtuelnih metoda čini programiranje na jeziku `C++` objektno-orijentisanim. Ovo je veoma zgodno koristiti u slučaju nizova čiji će elementi pokazivati na objekte različitih, ali srodnih tipova. Dakle, moguće je napraviti niz vozila u kojem ćemo smjestiti avione, brodove i bicikle. U našem zadatku ovu pogodnost koristimo da napravimo niz pokazivača na osnovnu klasu `Figure` koji će nam zapravo koristiti da u njega smjestimo izvedene klase `Circle` i `Square`:

```
Figure *figures[4];
figures[0] = new Circle;
figures[1] = new Square;
```

i gdje će sada poziv:

```
figures[n]->.perimeter()
```

pozvati odgovarajuću virtuelnu metodu za računanje obima. Konstruktor ne može biti virtuelna funkcija, ali zato destruktor može. Ako smo pozvali destruktor objekta izvedene klase preko pokazivača na osnovnu klasu, i ako destruktor osnovne klase nije virtuelna funkcija, došlo bi do pozivanja destruktora za osnovnu klasu, a ovaj bi dealocirao samo dio objekta koji potiče od osnovne klase ostavljajući da visi dio objekta koji je nadodala izvedena klasa. Stoga, destruktor (ako je bilo koja funkcija u klasi virtuelna) bi morao biti virtuelna funkcija! Ako osnovna klasa ima virtuelni destruktor, izvedena klasa će takođe imati virtuelni destruktor. Ukoliko nam eksplicitno definisani destruktor nije neophodan, ne moramo ga definisati, implicitni svakako postoji. Ukoliko se odlučimo da ga realizujemo, dodavanje riječi `virtual` je suvišno kao i kod

metoda, on će svakako biti virtuelan. U datom zadatku smo realizovali ove destruktore radi demonstracije njihovih poziva. Napominjemo još jednom, riječ `virtual` nije neophodna u izvedenim klasama, ali je dobra praksa navesti je. U pokaznim primjerima kao što je ovaj, uvidom u kod lako je zaključiti da li je metoda virtuelna ili ne. U složenim aplikacijama to bi bilo gotovo pa nemoguće. Takođe, izostavljanjem ove riječi možemo zaboraviti ili pogrešno shvatiti pojam virtuelnih metoda. Konačno, nismo stigli da se osvrnemo na modifikatore `override` i `final` koji su važni i nezanemarljivi za razumijevanje nasljeđivanja te je na vama da ih izučite.