Računske vježbe 7

Programabilni uređaji i objektno orijentisano programiranje

1. 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 težišta 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.

```cpp
#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;
    }
}
```

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 također 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. Također, 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.

2. Realizovati šablonsku funkciju **arrange** kojom se od dva uređena neopadajuća niza formira treći, na isti način uređen niz. Realizovati glavni program koji primjenjuje ovu funkciju nad nizovima cijelih brojeva i nizovima tačaka, pri čemu koordinate tačaka mogu biti cijeli ili realni brojevi. Klasu **Point** realizovati kao šablonsku klasu kako bi koordinate mogle biti traženog tipa.

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

using namespace std;

template<typename T>
void arrange(T *first, int nFirst, T *second, int nSecond, T *result)
{   // brojaci f za first, s za second, r za result...
    for(int f = 0, s = 0, r = 0; f < nFirst || s < nSecond; r++)
        if(f == nFirst)
            result[r] = second[s++];
        else if(s == nSecond)
            result[r] = first[f++];
        else
        {
            if(first[f] < second[s])
                result[r] = first[f++];
            else
                result[r] = second[s++];
        }
}

template<typename S>
class Point
{
private:
    S x;
    S y;
public:
    Point(S = 0, S = 0);
    bool operator<(const Point&) const;
    void print()
```

```
33      {
34          cout << "(" << x << "," << y << ")" <<endl;
35      }
36  };
37
38  template<typename S>
39  Point<S>::Point(S x, S y): x(x), y(y) {}
40
41  template<typename S>
42  bool Point<S>::operator<(const Point& point) const
43  {
44      return (pow(x, 2) + pow(y, 2)) < (pow(point.x, 2) + pow(point.y, 2));
45  }
46
47  int main()
48  {
49      int arr1[]= {1, 2, 5, 7, 9};
50      int arr2[]= {3, 4, 7, 8};
51      int *arrResult = new int[9];
52      //primjer niza cjelobrojnih vrijednosti, tip se zakljucuje!
53      arrange(arr1, 5, arr2, 4, arrResult);
54
55      cout << "Uredjeni niz je:" << endl;
56      for(int i = 0; i < 9; i++) cout << " " << arrResult[i];
57      cout << endl;
58
59      delete []arrResult; //oslobadjamo memoriju
60      //primjer niza tipa Point cije su koordinate tipa int
61      cout << "Koliko ima tacaka prvi niz?" << endl;
62      int nFirst;
63      cin >> nFirst;
64      Point<int> *first;
65      first = new Point<int>[nFirst];
66
67      cout << "Unesite niz tacaka sa cjelobrojnim koordinatama" << endl;
68      int x, y;
69      for(int i = 0; i < nFirst; i++)
70      {
71          cin >> x >> y;
72          first[i] = Point<int>(x, y);
73      }
74
75      cout << "Koliko ima tacaka drugi niz?" << endl;
76      int nSecond;
77      cin >> nSecond;
78      Point<int> *second;
79      second = new Point<int>[nSecond];
80
81      cout << "Unesite niz tacaka sa cjelobrojnim koordinatama" << endl;
82      for(int i = 0; i < nSecond; i++)
83      {
84          cin >> x >> y;
85          second[i] = Point<int>(x, y);
86      }
87
88      int nResult;
89      nResult = nSecond + nFirst; //racunamo duzinu rezultujuceg niza
90
91      Point<int> *result;
```

```
    result = new Point<int>[nResult];

    arrange(first, nFirst, second, nSecond, result);

    cout << "Uredjeni niz je:" << endl;
    for(int i = 0; i < nResult; i++)
    {
        cout << " ";
        result[i].print();
    }
    cout << endl;

    delete [] first;
    delete [] second;
    delete [] result;
    //tacke cije su koordinate tipa double
    Point<double> point1(2.3, 4.6), point2(2.5, 6.8);

    cout << "Tacka ima koordinate: ";
    (point1 < point2) ? point1.print() : point2.print();
}
```

Kada smo na jednom od uvodnih časova naglašavali prednosti programskog jezika `C++` nad programskim jezikom `C` pomenuli smo pojam preklapanja funkcija. Ako za primjer uzmemo problem pronalaženja većeg od dva podatka (bilo kojeg tipa) to bi značilo da bismo za svaki tip podataka pisali po jednu funkciju. Istine radi, u programskom jeziku `C` ove funkcije nisu mogle imati isto ime te smo preklapanje funkcija smatrali značajnim iskorakom. Međutim, u primjeru pronalaženja većeg od dva podatka, razlike između ovih funkcija svode se na sistematsko zamjenjivanje oznake tipa unutar cijele definicije funkcije:

```
int max (int a, int b) { return a > b ? a: b; }
double max (double a, double b) { return a > b ? a: b; }
char max (char a, char b) { return a > b ? a: b; }
```

što predstavlja rutinski posao koji se može automatizovati odnosno generalizovati. Jezik `C++` omogućava nam definisanje šablona (engl. *template*) pomoću kojih opisujemo datu obradu u vidu opšteg slučaja bez navođenja konkretnih tipova podataka. Pomoću ovako definisanih šablona se automatski generišu konkretne funkcije za konkretne tipove podataka. Ovo generisanje se vrši po potrebi, ne prave se istovremeno sve moguće kombinacije za sve raspoložive tipove. Šabloni se mogu definisati za funkcije, klase i još ponešto. Opšti oblik šablona za generičke stvari bi bio:

```
template <parametar, parametar, ... , parametar> stvar
```

gdje `stvar` predstavlja definiciju ili deklaraciju funkcije, klase itd. čiji se šablon definiše. Šabloni nemaju oznaku kraja i završavaju se tamo gdje se završava opisivana stvar. Parametri šablona se pišu unutar `<>` zagrada i mogu da označavaju kako tipove tako i konstante. Opšti oblik parametra je:

```
class Ime
typename Ime
tip Ime
```

gdje ključne riječi `class` i `typename` označavaju tipovne parametre dok sa `tip` predstavljamo netipovne parametre koji su zapravo konstante navedenog tipa, recimo `int k`. Ključna riječ `typename` nam govori da se može raditi o bilo kojem tipu dok sa `class` naglašavamo one parametre šablona koji moraju biti klase. Važno je znati da ovo nije nešto što nameće sam programski jezik već standard kojeg ćemo se pridržavati. Naime, inicijalna verzija programskog jezika `C++` nije dodala novu ključnu riječ kojom bi se označavali tipovni parametri već je samo upotrijebila ključnu riječ `class` što je stvorilo veliku konfuziju jer tipovni parametar može biti bilo kojeg tipa, ne radi se o klasi. Zbog toga je naknadno uvedena nova ključna riječ `typename`. Kako se stari programi ne bi pokvarili ostavljena je mogućnost i da se koristi `class` kojoj smo

našli praktičnu namjenu. Imena parametara šablona obično se pišu velikim početnim slovom. Najčešće se koristi slovo T (type) ili njemu u engleskom alfabetu susjedna slova S i U. Dakle, radi se o standardu, ne i o obavezi. Pa, kako bismo napisali šablon za jednu funkciju? Sjetimo se našeg pokaznog primjera za određivanje većeg od dva podatka:

```
template<typename T> T max (T a, T b) { return a > b ? a : b; }
```

gdje sa T označavamo tipove argumenata, ali i tip rezultata funkcije! Ovaj šablon se može koristiti i za klase, potrebno je ,,samo" preklopiti operator veće za one klase od interesa. Funkcija se generiše na osnovu šablona po potrebi, odnosno:

```
int a = max<int>(1, 2);
```

gdje između <> prosljeđujemo tip od interesa. Prilikom generisanja, tip parametra se može i zaključiti:

```
int a = max(1, 2);
double b = max(1, 2.0) // greska, tipovi se ne uklapaju (int i double)!
```

Slično funkcijama, šablonima se mogu definisati i generičke klase:

```
template<parametri> class Ime { clanovi };
```

pa smo u slučaju naše klase imali:

```
template<typename S>
class Point
{
private:
    S x;
    S y;
public:
    Point(S = 0, S = 0);
};
```

i uočavamo da nam S predstavlja tip koordinata koji koristimo u njihovoj deklaraciji, ali naravno i u konstruktoru i gdje god nam to u klasi treba. Kada želimo da definišemo neku metodu ili konstruktor koji je u tijelu klase (odnosno unutar šablona) samo deklarisan, koristićemo operator dosega, ali moramo obavezno naglasiti da se radi o šablonu:

```
template<typename S>
Point<S>::Point(S x, S y): x(x), y(y) {}
```

i još jednom skrenimo pažnju da se šablon završava tamo gdje se završava opisivana stvar (u ovom slučaju konstruktor) te da slovo S koje predstavlja tip parametra možemo koristiti u prvom narednom šablonu, a da smo u deklaraciji klase mogli koristiti neko drugo.