Računske vježbe 2

Programabilni uređaji i objektno orijentisano programiranje

1. Projektovati klasu sa elementarnim operacijama nad kompleksnim brojevima (sabiranje, oduzimanje) pri čemu je potrebno koristiti konstruktore.

Klase su složeni tipovi podataka koji se sastoje od elemenata koji mogu biti različitih tipova i koji se nazivaju članovima klasa. Podaci klasnih tipova nazivaju se **objekti** - odatle i objektno orjentisano programiranje (OOP). Članovi klasa mogu da budu podaci ili funkcije. Vrijednosti podataka članova se nazivaju **polja** klase ili njeni **atributi**. Funkcije članovi koriste se za izvođenje operacija nad podacima članovima i nazivaju se **metode**. Članovi klasa mogu da budu **javni** ili **privatni**. Razlika je u tome što se privatnim članovima može pristupati samo iz unutrašnjosti klase odnosno privatna polja mogu da koriste ili mijenjaju samo metode date klase, a privatne metode se mogu pozivati isključivo iz drugih metoda iste klase. Javnim članovima može da se pristupa bez ograničenja. Nazive klasa pisaćemo sa prvim velikim slovom kako bismo ih razdvojili od metoda i polja. Opet treba naglasiti da konvencije imenovanja koje uvodimo nisu obavezujuće. U većini slučajeva polja su privatna, dok su neke metode privatne a neke javne. Ovim se postiže enkapsulacija.

Kako su metode koje se definišu unutar klase implicitno pretvorene u inline metode, većinu metoda ćemo samo deklarisati u tijelu klase, a definisati van. Izvan klase se ne mogu dodavati nove metode datoj klasi. U definiciji klase vidi se **šta** sve može da se izvodi nad objektima klase dok se izvan klase može samo opisati **kako** se te radnje izvode. Metode koje čitaju vrijednosti podataka članova su inspektori (engl. *getters*). Njih ćemo definisati na sljedeći način:

```
oznaka_tipa  getPodatak () const {return podatak;};
```

gdje ključna riječ `const` nije neophodna, ali označava konstantnu metodu odnosno metodu koja ne mijenja unutrašnja stanja objekta. Inspektori treba da se nađu unutar tijela klase jer zadovoljavaju sve uslove inline funkcija. Da bismo realizovali neku metodu van tijela klase koristićemo operator dosega `::` prije naziva funkcije, a nakon navođenja tipa rezultata. Na primjeru sabiranja dva kompleksna broja imamo:

```
Complex Complex::add(Complex a)
{
    Complex result;
    result.real = real + a.real;
    result.imag = imag + a.imag;
    return result;
}
```

gdje početno `Complex` označava tip vrijednosti funkcije (tip rezultata) dok sljedeće `Complex` označava naziv klase čiju metodu `add` dosežemo pomoću operatora dosega. Parametar funkcije je objekat tipa `Complex` (nazvan je sa `a`, nepametan naziv, treba smisliti bolji!) zato što ova metoda sabira dva kompleksna broja. Pošto ćemo ovu metodu pozivati za dva operanda na sljedeći način:

```
c1.add(c2);
```

to znači da ćemo pomoću `real` i `imag` imati direktan pristup atributima prvog, a za drugi (u našem primjeru se zove a) radićemo to preko `a.real` i `a.imag`. Obratite pažnju kako metodom jednog objekta pristupamo podacima članovima drugog objekta koji su privatni. Ovo je moguće zato što su metodima

dostupni atributi svih objekata klasnog tipa kojem pripadaju! Članovima tekućeg objekta pristupamo implicitno odnosno neposredno. To omogućava skriveni parametar koji predstavlja adresu prvog operanda za koji je metoda pozvana i čiji je identifikator `this`. Ovaj pokazivač se implicitno koristi uvijek kada se pristupa odgovarajućem članu tekućeg objekta, ali se rijetko kada eksplicitno koristi osim u slučajevima kada tekući objekat predstavlja vrijednost funkcije ili recimo argument neke druge uvezane funkcije itd. Ovaj pokazivač jeste nepromjenljiv, ali pomoću njega može da se promijeni vrijednost tekućeg objekta. Tekući objekat se u cjelini može dohvatiti pomoću izraza `*this`.
Konstruktori predstavljaju specijalne metode koje se koriste za inicijalizaciju objekta. Neinicijalizovani objekat i nije objekat već parče memorije čija sadržina nema smisao. Konstruktori imaju iste identifikatore kao i klase kojima pripadaju, za klasu `Complex` naziv konstruktora će također biti `Complex` pa je preklapanje konstruktora dozvoljeno, a mogu imati i podrazumijevane argumente. Za razliku od metoda ne daju nikakvu vrijednost. Također, pri definisanju konstruktora mogu se dodati i inicijalizatori za pojedina polja klase. Opšti oblik definicije konstruktora bi onda bio:

```
Klasa ( parametri ) : inicijalizator, ... , inicijalizator tijelo_konstruktora
```

gdje se uočava da inicijalizatori slijede nakon `:`. Ukoliko nema inicijalizatora treba izostaviti `:`. Tijelo se mora navesti sa vitičastim zagradama pa makar bilo prazno. Kao i slučaju metoda, ako je definicija konstruktora izvan definicije klase, koristićemo operator dosega `::`. Treba voditi računa da inicijalizator iz konstruktora ima prednost u odnosu na inicijalizator u definiciji polja odnosno ako neko polje inicijalizujemo sa recimo `int a = 2` inicijalizator konstruktora će imati prioritet. Vrijednost se, naravno, može dodijeliti i u tijelu konstruktora ali to tehnički nije inicijalizacija već dodjela vrijednosti. Opšti oblik inicijalizatora je:

```
polje (izraz, izraz, ... , izraz )
```

i u slučaju naše klase, da smanjimo apstrakciju, definicija konstruktora će biti:

```
Complex::Complex(double a, double b) : real(a), imag(b) {}
```

gdje se jasno vidi korisna strana inicijalizatora. Parametre konstruktora direktno prosljeđujemo inicijalizatoru polja. Sve klase imaju nešto što se zove podrazumijevani konstruktor koji se implicitno definiše bez našeg znanja. Međutim, kada mi eksplicitno definišemo bar jedan drugi konstruktor ovaj podrazumijevani biva obrisan. Ovo će se desiti i kada u klasi postoji nepromjenljivo `const` polje ili polje koje predstavlja referencu. U klasama koja posjeduju pokazivačka polja neophodno je eksplicitno definisati podrazumijevani konstruktor kako bi se inicijalizovala sva pokazivačka polja sa `0`. Generalno, podrazumijevani konstruktor treba uvijek definisati u slučaju eksplicitnog definisanja bar jednog drugog konstruktora zato što nam je koristan jer npr. sledeću deklaraciju:

```
Complex result;
```

iz funkcije za sabiranje dva kompleksna broja ne bismo mogli izvršiti, a da prethodno u definiciji ove klase nismo napisali:

```
Complex(){};
```

zato što se konstruktori pozivaju automatski u momentima stvaranja svih objekata. U ovom slučaju bi se pokušao pozvati podrazumijevani konstruktor bez argumenata.

```cpp
#include <iostream>

using namespace std;

class Complex
{
private:
    double real;
    double imag;
public:
    Complex(){};
    Complex(double, double);
    Complex add(Complex);
    Complex subtract(Complex);
    double getReal() const {return real;};
    double getImag() const {return imag;};
};

Complex::Complex(double a, double b) : real(a), imag(b) {}

Complex Complex::add(Complex a)
{
    Complex result;
    result.real = real + a.real;
    result.imag = imag + a.imag;
    return result;
}

Complex Complex::subtract(Complex a)
{
    Complex result;
    result.real = real - a.real;
    result.imag = imag - a.imag;
    return result;
}

int main()
{
    double real, imag;
    cout << "Unesite vrijednost za realni i imaginarni dio c1" << endl;
    cin >> real >> imag;
    Complex c1(real, imag);

    cout << "Unesite vrijednost za realni i imaginarni dio c2" << endl;
    cin >> real >> imag;
    Complex c2(real, imag);

    Complex c3;
    c3 = c1.add(c2);
    cout << "Zbir dva unijeta kompleksna broja je " << "(" << c3.getReal()<< ", " <<
   c3.getImag() << ")" << endl;

    c3 = c1.subtract(c2);
    cout << "Razlika dva unijeta kompleksna broja je " << "(" << c3.getReal() << ", "
    << c3.getImag() << ")" << endl;
}
```

2. Realizovati klasu **Worker** koja ima četiri podatka člana i to:

   - koeficijent za platu (cijeli broj),
   - identifikacioni broj radnika (pokazivač na cijeli broj),
   - ime radnika (pokazivač na niz karaktera),
   - javni statički podatak koji će služiti za brojanje ukupnog broja radnika (objekata date klase).

Klasa posjeduje konstruktor, destruktor i konstruktor kopije, kao i funkcije članice za pristup podacima članovima radi očitavanja i izmjene. Potrebno je realizovati i funkciju koja od dva radnika vraća ime radnika sa većim koeficijentom za platu.

```
#include <iostream>
#include <cstring>

using namespace std;

class Worker
{
private:
    int coeff;
    int *id;
    char *name;
public:
    Worker() // podrazumijevani konstruktor
    {
        id = 0;
        name = 0;
        number++;
    }
    Worker(int, int, char *);
    Worker(const Worker &);
    ~Worker();

    int getCoeff() const {return coeff;}
    int getId() const {return *id;}
    char * getName() const {return name;}

    void setCoeff(int a) {coeff = a;}
    void setId(int a)
    {
        if (id == 0) id = new int(a);
        else *id = a;
    }
    void setName(char *a)
    {
        /*
        Memoriju oslobadjamo ukoliko je zauzeta kako ne bi doslo do njenog curenja,
        a to ce se desiti jer bismo u suprotnom naredbom new alocirali novu memoriju
        te izgubili adresu onog dijela memorije na koji je pokazivac name prije toga
        pokazivao. Ne moramo provjeravati da li je name==0 prije brisanja jer to
        radi delete za nas.
        */
        delete [] name;
        name = new char[strlen(a)];
        strcpy(name, a);
```

```
45      }
46
47      char * higherCoeff(Worker);
48
49      static int number;
50  };
51
52  int Worker::number = 0;
53
54  Worker::Worker(int _coeff, int _id, char *_name) : coeff(_coeff)
55  {
56      id = new int(_id); //*id=_id;
57      name = new char[strlen(_name) + 1];
58      strcpy(name, _name);
59      number++;
60  }
61  Worker::Worker(const Worker &worker) : coeff(worker.coeff)
62  {
63      id = new int(*worker.id); //*id=*worker.id;
64      name = new char[strlen(worker.name) + 1];
65      strcpy(name, worker.name);
66      number++;
67  }
68  Worker::~Worker()
69  {
70      delete id; // oslobadjamo dinamicki zauzetu memoriju
71      id = 0; // pokazivac sada ni na sta ne pokazuje
72      delete [] name; // niz karaktera, koristimo []!!
73      name = 0;
74      number--;
75  }
76  char * Worker::higherCoeff(Worker worker)
77  {
78      if(coeff >= worker.coeff)
79          return name;
80      else
81          return worker.name;
82  }
83  int main()
84  {
85      int coeff, id;
86      char name[20];
87
88      cout << "Unesite podatke za prvog radnika" << endl;
89      cin>> coeff >> id >> name;
90      Worker worker1(coeff, id, name);
91
92      cout << "Unesite podatke za drugog radnika" << endl;
93      cin >> coeff >> id >> name;
94      Worker worker2(coeff, id, name);
95
96      //poziv konstruktora kopije
97      Worker worker3(worker2);
98      cout << "Sada radnik 3 ima ime " << worker3.getName() << endl;
99      cout << "Vise je placen radnik " << worker1.higherCoeff(worker2) << endl;
100     cout << "Ukupno je kreirano " << Worker::number << " radnika." << endl;
101 }
```

U slučaju kada klasa ima pokazivač kao podatak član, prilikom destrukcije objekta doći će do dealokacije

podatka člana, ali ne i onoga na šta on pokazuje. Mi smo preko tog pokazivača recimo mogli da zauzmemo memoriju za cijeli broj ili niz cijelih brojeva koja neće biti oslobođena. Zbog toga je, kada god radimo sa podacima članovima koji su pokazivači, neophodno da realizujemo destruktor kako bismo oslobodili memoriju. Takođe, neophodan nam je i podrazumijevani konstruktor koji postavlja pokazivač da ni na šta ne pokazuje, odnosno da pokazuje na **0**. Još jednom naglašavamo, programer brine o dinamički zauzetoj memoriji, a OS o statički zauzetoj memoriji. Drugim riječima, upotrebom naredbe `delete` mi oslobađamo onu memoriju na koju pokazuje pokazivač, dok memoriju za sam pokazivač implicitno oslobađa destruktor bez naše kontrole. U kodu:

```
class X
{
private:
    int *p;
public:
    X()
    {
        p = 0; // ni na sta ne pokazuje
    }
    ~X() // sa ~ naglasavamo da se radi o destruktoru
    {
        delete p; // brisemo ono na sta pokazivac pokazuje
        p = 0; // ni na sta ne pokazuje
    }
};
```

U ovom zadatku neophodno je realizovati i konstruktor kopije. Namjena konstruktora kopije jeste da novostvoreni objekat inicijalizuje kopijom sadržaja drugog objekta istog tipa. Parametar konstruktora kopije mora da bude **referenca** na primjerak istog tipa zato što konstruktor ne može da ima parametar tipa svoje klase. Kao i za podrazumijevani konstruktor, tako i za konstruktor kopije postoji implicitna definicija. Ovako definisani konstruktor kopira sva polja izvorišnog objekta u novostvoreni objekat. Ukoliko su neka od polja tipa (drugih) klasa, za njihovo kopiranje pozivaće se kopirajući konstruktori tih klasa. Ako su neka od polja pokazivači, kopiraće se samo vrijednosti tih pokazivača, a neće se praviti kopije pokazivanih podobjekata. Ovakva kopija naziva se **plitka kopija** jer nije nezavisna od originala pa se obično želi izbjeći jer njome postižemo da polja dva objekta pokazuju na istu memorijsku lokaciju. Kopija koja objekte čini nezavisnim naziva se **duboka kopija**. Iz gore navedenog jasno je da implicitno definisani konstruktor kopije ne može riješiti ovaj problem pa se on mora preklopiti. Uočite kako smo, upravo radi prevazilaženja ovog problema, koristili funkciju `strcpy()`.
Takođe, uočite kako smo u konstruktoru kopije upotrijebili ključnu riječ `const`. Referenca je drugo ime za neki memorijski objekat odnosno njegov alijas. Referenca ne može da promijeni objekat na koji se odnosi, ali se pomoću nje može mijenjati sadržaj referenciranog objekta. Kako se referenca sama po sebi ne može naknadno mijenjati, konstantna referenca nam garantuje da ono na šta ona referencira postaje konstantno odnosno **read-only**. Zašto uopšte koristimo reference? Zato što kada prosljeđujemo argument po referenci ne pravi se kopija toga objekta kao u slučaju prosljeđivanja po vrijednosti. Time se u slučaju kompleksnijih klasa dobija značajna ušteda u vremenu.

3. Napraviti klasu `Point` koja sadrži:

   - koordinate tačke (dva realna broja);
   - odgovarajuće konstruktore;
   - funkciju članicu za računanje rastojanja između dvije tačke.

Nakon toga napraviti klasu `Circle` koja sadrži:

- tačku koja predstavlja centar kruga i tačku sa ivice kruga (objekti tipa klase tačka);
- odgovarajuće konstruktore;
- funkcije članice za izračunavanje obima i površine kruga.

Ovaj zadatak može se riješiti na dva načina. Prvi način je da u klasi `Circle` koja ima dva polja tipa `Point` prilikom inicijalizacije ipak koristimo realne brojeve pomoću kojih ćemo inicijalizovati objekte tipa `Point` pa pomoću njih inicijalizovati odgovarajuća polja, a drugi način bi bio da prilikom inicijalizacije polja direktno prosljeđujemo objekte tipa `Point`. Obratite pažnju da smo unutar klase `Circle` definisali statičko polje u kojem čuvamo vrijednost π. To smo uradili pomoću:

```
static constexpr double pi = 3.14;
```

i u ovom slučaju ključna riječ `constexpr` nam omogućava da statičko polje inicijalizujemo unutar tijela klase. Imajte na umu da se u memoriji čuva samo jedna kopija ove vrijednosti koju dijele svi objekti ovog tipa. Samim tim, ona ne pripada posebno jednom objektu pa joj, van oblasti definisanosti klase, pristupamo sa:

```
Circle::pi
```

mada joj možemo pristupiti i direktno preko objekta. Prilikom definicije konstruktora u slučaju klase `Circle` jedno polje smo inicijalizovali u inicijalizatoru dok smo drugom dodijelili vrijednost u tijelu klase što je dozvoljeno mada se upotrebi inicijalizatora obično treba dati prvenstvo jer se izvršava prije tijela konstruktora i jer direktno inicijalizuje polje.

```
1  #include <iostream>
2  #include <cmath>
3
4  using namespace std;
5
6  class Point
7  {
8  private:
9      double x, y;
10 public:
11     Point() {}
12     Point(double, double);
13     double distance(Point) const;
14 };
15
16 Point::Point(double a, double b) : x(a), y(b) {}
17
18 double Point::distance(Point a) const
19 {
20     return sqrt(pow(x - a.x, 2) + pow(y - a.y, 2));
21 }
```

```
22
23 class Circle
24 {
25 private:
26     Point center;
27     Point onTheCircle;
28 public:
29     static constexpr double pi = 3.14;
30     Circle() {}
31     //drugi nacin: Circle(): center(Point(0,0)), onTheCircle(Point(0,0)) {};
32     Circle(double, double, double, double);
33     //Drugi nacin: Circle(Point, Point);
34     double area();
35     double perimeter();
36 };
37
38 Circle::Circle(double x1, double x2, double x3, double x4) : center(Point(x1, x2))
39 {
40     onTheCircle = Point(x3, x4); // postavljanje vrijednosti
41 }
42 /*
43 Drugi nacin:
44 Circle::Circle(Point p1, Point p2) : center(p1)
45 {
46     onTheCircle = p2;
47 }
48 */
49 double Circle::area()
50 {
51     double r;
52     r = center.distance(onTheCircle);
53     return pow(r, 2) * pi;
54 }
55
56 double Circle::perimeter()
57 {
58     double r;
59     r = center.distance(onTheCircle);
60     return 2 * r * pi;
61 }
62
63 int main()
64 {
65     double x1, y1, x2, y2;
66     cout << "Unesite koordinate centra kruga i tacke sa kruga" << endl;
67     cin >> x1 >> y1 >> x2 >> y2;
68     Circle circle(x1, y1, x2, y2);
69     //Drugi nacin: Circle center(Point(x1,y1), Point(x2,y2));
70     cout << "Povrsina kruga je: " << circle.area() << endl;
71     cout << "Obim kruga je: " << circle.perimeter() << endl;
72     cout << "Pi je: " << Circle::pi << endl;
73 }
```