# Računske vježbe 5

Programiranje II

1. Realizovati klasu `Complex` koja predstavlja kompleksne brojeve. Nakon toga realizovati klasu `ComplexArray` koja će imati sljedeće članove:

   - pokazivač na niz kompleksnih brojeva (pokazivač na niz objekata klase `Complex`);
   - dužinu niza (cijeli broj);
   - Funkciju koja računa proizvod dva kompleksna broja, pri čemu je potrebno realizovati kao funkciju članicu i kao prijateljsku funkciju.

   Uzeti da je klasa `ComplexArray` prijateljska klasa klase `Complex`.

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

using namespace std;

class Complex
{
private:
    double real;
    double imag;
public:
    Complex() {}
    Complex(double real, double imag) : real(real), imag(imag) {}
    Complex multiply(Complex);
    void print()
    {
        cout << real << ((imag >= 0) ? "+" : "-") << abs(imag) << "j" << endl;
    }
    double getReal() const {return real;}
    double getImag() const {return imag;}
    friend Complex multiplication(Complex, Complex);
    friend class ComplexArray;
//koristimo kljucnu rijec class jer klasa niz jos uvijek nije deklarisana.
};

Complex Complex::multiply(Complex num)
{
    Complex result;
    result.real = real * num.real - imag * num.imag;
    result.imag = imag * num.real + real * num.imag;
    return result;
}

Complex multiplication(Complex num1, Complex num2)
{
```

```
    Complex result;
    result.real = num1.real * num2.real - num1.imag * num2.imag;
    result.imag = num1.imag * num2.real + num1.real * num2.imag;
    return result;
}

class ComplexArray
{
private:
    Complex *cArray;
    int length;
public:
    ComplexArray()
    {
        cArray = 0;
    };
    ComplexArray(int, Complex *);
    ComplexArray(const ComplexArray &);
    ~ComplexArray();
    void print()
    {
        for(int i = 0; i < length; i++)
            cArray[i].print();
    }
};

ComplexArray::ComplexArray(int _length, Complex *_cArray) : length(_length)
{
    cArray = new Complex[length];
    for(int i = 0; i < length; i++)
        cArray[i] = _cArray[i];
}

ComplexArray::ComplexArray(const ComplexArray &complexArray) : length(complexArray.
    length)
{
    cArray = new Complex[length];
    for(int i = 0; i < length; i++)
        cArray[i] = complexArray.cArray[i];
}

ComplexArray::~ComplexArray()
{
    delete [] cArray;
    cArray = 0;
}

int main()
{
    double real, imag;
    int n;
    Complex cArray[10];

    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.multiply(c2);
    c3.print();

    c3 = multiplication(c1, c2);
    c3.print();

    cout << "Unesite duzinu niza: ";
    cin >> n;

    cout << "Unesite elemente niza" << endl;

    for(int i = 0; i < n; i++)
    {
        cin >> real >> imag;
        cArray[i] = Complex(real, imag);
    }
    ComplexArray testArray(n, cArray);
    testArray.print();
}
```

Prijateljske funkcije neke klase jesu funkcije koje **nisu članovi te klase** ali imaju **pravo pristupa do privatnih članova** te klase. Prijateljske funkcije mogu da budu obične funkcije ili da budu metode drugih klasa. Da bi funkcija postala prijateljska funkcija neke klase, potrebno je u definiciji te klase dodati modifikator `friend` na sljedeći način:

```
friend tip_funkcije naziv_funkcije ( parametri ) ; // ili
friend tip_funkcije naziv funkcije ( parametri ) tijelo_funkcije
```

i nije bitno da li će se ovo proglašavanje obaviti u privatnom ili javnom dijelu klase jer ona nije član posmatrane klase. Prijateljska klasa je ona klasa čije su sve metode prijateljske funkcije date klase. Kako ne bismo navodili deklaracije svih metoda te klase možemo pisati:

```
friend identifikator_klase ; // ili
friend class identifikator_klase ;
```

gdje je druga varijanta neophodna ako prethodno nisu navedene ni definicija ni deklaracija klase čije metode želimo proglasiti prijateljskim za datu klasu. Naredba se, naravno, stavlja u definiciju klase kojoj te metode treba da budu prijateljske funkcije. Drugim riječima, prijateljstvo ne narušava enkapsulaciju zato što se ono nudi, a ne nameće. Jedna klasa drugu ne može natjerati da joj bude prijatelj, može joj samo ponuditi pristup njenim članovima. Prijateljstvo **nije komutativno**. Ukoliko želimo da samo jednu metodu proizvoljne klase učinimo prijateljskom to radimo sa:

```
friend tip_funkcije identifikator_klase::naziv_funkcije( parametri ) ;
```

Čest slučaj upotrebe prijateljskih funkcija jeste kada klasična notacija pozivanja globalnih funkcija biva prirodnija od notacije pozivanja metode. Na primjeru ovog zadatka, `multiplication(c1, c2)` je prirodnije od `c1.multiply(c2)`. Prijateljstvo nam pruža još jednu zgodnu notaciju. Definišimo sljedeći konstruktor:

```
Complex(double real) : real(real), imag(0) {}
```

te je sada moguće napisati sledeće:

```
c2 = multiplication(c1, 3.0);
c2 = multiplication(3.0, c1);
c2 = multiplication(3.0, 3.0);
```

jer će se u svim slučajevima na osnovu realnog broja pozvati odgovarajući konstruktor.

2. Realizovati klasu `Fraction` koja predstavlja racionalne brojeve. Izvršiti preklapanje operatora `+`, `+=` kao i operatora za prefiksno i postfiksno inkrementiranje. Prilikom realizacije operatora `+` uzeti u obzir i mogućnost sabiranja racionalnih brojeva sa cijelim brojem, pri čemu se cijeli broj može očekivati i kao lijevi i kao desni operand.

```
#include <iostream>

using namespace std;

class Fraction
{
private:
    int numerator;
    int denominator;
public:
    Fraction(int = 0, int = 1);
    /*
    Prijateljskoj funkciji nisu proslijedjeni podaci po referenci jer
    nam je neophodan poziv konstruktora kod sabiranja sa cijelim brojevima
    */
    friend Fraction operator+(Fraction, Fraction);
    Fraction& operator+=(Fraction);
    Fraction& operator++(); //prefiksno inkrementiranje
    Fraction operator++(int); //postfisksno inkrementiranje
    void print()
    {
        cout << numerator<< "/" << denominator << endl;
    }
};

Fraction::Fraction(int a, int b) : numerator(a), denominator(b) {}

Fraction operator+(Fraction op1, Fraction op2)
{
    Fraction result;
    result.numerator = op1.numerator * op2.denominator + op2.numerator * op1.
   denominator;
    result.denominator = op1.denominator * op2.denominator;
    return result;
}

Fraction& Fraction::operator+=(Fraction op)
{
    numerator = numerator * op.denominator + denominator * op.numerator;
    denominator = denominator * op.denominator;
    return *this;
}

Fraction& Fraction::operator++() //prefiksno inkrementiranje
{
    return (*this) += 1; //koristimo operator += jer je vec realizovan
}

Fraction Fraction::operator++(int i) //postfiksno inkrementiranje
{
    Fraction temp(*this); //kreiramo kopiju objekta kojeg inkrementiramo
    (*this) += 1; //inkrementiramo vrijednost originalnog objekta
    return temp; //vracamo vrijednost prije inkrementiranja (vrijednost kopije)
```

```
}

int main()
{
    int num, denom;

    cout << "Unesite vrijednosti brojioca i imenioca prvog razlomka: " << endl;
    cin >> num >> denom;
    Fraction f1(num, denom);

    cout << "Unesite vrijednosti brojioca i imenioca drugog razlomka: " << endl;
    cin >> num >> denom;
    Fraction f2(num, denom);

    Fraction temp;
    temp = f1 + f2;
    temp.print();

    f1++;
    f1.print();

    temp = f1++; // prvo kopiramo f1 u temp pa inkrementiramo f1
    temp.print();

    ++f1;
    f1.print();

    f1 += f2;
    f1.print();

    f1 += (f2++);
    f1.print();

    temp = f1 + f2 + 5;
    temp.print();

    (5 + f2).print();
}
```

Preklapanje operatora ostvaruje se pomoću **operatorskih funkcija** na sledeći način:

```
operator op
```

gdje `op` predstavlja simbol odgovarajućeg operatora npr. `operator+`. Imena operatorskih funkcija mogu da se preklope isto kao i kod drugih funkcija - dati operator je moguće definisati za proizvoljan broj tumačenja. Operatorske funkcije za klasne tipove mogu biti metode ili obične globalne funkcije i u tom slučaju su najčešće prijateljske. Pažnja! Unarni operatori imaju jedan operand, pa odgovarajuća operatorska funkcija treba da ima tačno jedan parametar. Zbog toga se mogu ostvarivati metodama bez parametara (metode posjeduju skriveni parametar `this`) ili globalnim funkcijama s jednim parametrom. Tip tog jedinog parametra mora da bude klasa za koju se data funkcija pravi. Kod binarnih operatora koji imaju dva operanda odgovarajuće operatorske funkcije moraju imati tačno dva parametra. U slučaju metoda mogu se realizovati sa jednim parametrom (objekat za koji se poziva je implicitni, skriveni parametar), a u slučaju globalnih funkcija sa dva parametra. Jedan operand može biti proizvoljnog tipa. Vrijednosti operatorskih funkcija mogu biti proizvoljnog tipa, čak mogu i biti tipa `void`. Ne postoji nikakvo formalno ograničenje da preklapanjem operatora promijenimo ,,smisao" simbola i da recimo operatorska funkcija `operator=` ne dodjeljuje vrijednost već recimo vrši štampu. Mada ne postoji formalno ograničenje postoji ono zdravorazumno te ovakve stvari **ne treba** raditi.

Unarni operatori `++` i `--` su specifični po tome što imaju prefiksni (`++a`) i postfiksni (`a++`) oblik. Potrebno ih je prilikom preklapanja nekako razdvojiti. Prefiksni oblik (`++a`) se preklapa metodom bez parametara (u slučaju globalne funkcije jedan parametar) dok se kod postfiksnog oblika vrši preklapanje metode sa jednim parametrom tipa `int`. Svrha ove cjelobrojne vrijednosti nije da se ona koristi, već da za dva operatora istog simbola postoje dvije funkcije s različitim potpisima. U slučaju notacije za pozivanje funkcija (`a.operator++(k)`) vrijednost `k` se prenosi u funkciju.

Za razliku od gore pomenutih operatora, operator `=` ima automatsko tumačenje. On predstavlja kopiranje izvorišnog objekta u odredišni objekat polje po polje. Ovo tumačenje, kao i kod konstruktora kopije, zadovoljava slučaj kad nemamo polja koja mogu biti pokazivači. Zbog toga se uvodi kopirajuća dodjela vrijednosti (*copy assignment*) koja je jako slična konstruktoru kopije i koja se deklariše kao:

```
T& operator=(const T&);
```

gdje je `T` proizvoljan klasni tip podatka. Za proste tipove podatka dodjeljivanje vrijednosti je izraz čija je vrijednost lijevi operand. Zbog toga je u ovom slučaju tip rezultata operatorske funkcije `operator=` referenca na tekući objekat (`*this`) s izmijenjenim sadržajem. Dobra je praksa da se u slučaju `a=b` sva dinamički zauzeta memorija u `a` prvo oslobodi pa da se zauzme nova prilikom kopiranja vrijednosti polja `b` u `a`. Ovo se radi zbog toga što je mala vjerovatnoća da će podaci kao što su npr. nizovi biti iste dužine u objektima `a` i `b`. Međutim, naredba `a=a` bi u tom slučaju dovela do katastrofe. Kako se radi o dva ista objekta, oslobađanjem memorije lijevog operanda ispraznili bismo i ovaj desni. Zbog toga je zgodno da se u slučaju ovog operatora vrši sledeća provjera:

```
T& operator=(const T& t)
{
    if(this != &t)
    {
    //kopiraj, kopiraj i kopiraj...
    }
    return *this;
}
```

čime u slučaju poziva `a=a` uz pomoć if kuliramo kopiranje :) Mada nam se postavkom zadatka nije tražilo da preklopimo ovaj operator, dato objašnjenje će nam biti od koristi kako u razumijevanju preklapanja operatora tako i u razumijevanju narednih vježbi.