Računske vježbe 1

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 orijentisano 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. Dobro je 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 other)
{
    Complex result;
    result.real = real + other.real;
    result.imag = imag + other.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` i nazvan je sa `other` 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 radićemo to preko `other.real` i `other.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 neposredno. To omogućava skriveni parametar koji predstavlja adresu objekta za koji je metoda pozvana (tekući objekat) i čiji je identifikator `this`. Ovaj pokazivač se implicitno koristi uvijek kada se pristupa odgovarajućem članu tekućeg objekta, ali se rijetko kada eksplicitno navodi 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`.

Stvaranje objekata podrazumijeva dodjelu memorijskog prostora i, eventualno, inicijalizaciju dodjeljivanjem nekih početnih vrijednosti. Memorijski prostor se dodjeljuje automatski za sva polja klasa dok je inicijalizacija tj. dodjeljivanje početnih vrijednosti dužnost programera. Konstruktori predstavljaju specijalne metode koje se koriste za inicijalizaciju objekta. Neinicijalizovani objekat i nije objekat već parče memorije čija sadržina nema smisao. Pravi objekat ima sadržaj koji zadovoljava osobine svoje klase. Postavljanje vrijednosti polja na osnovu izraza u njihovoj definiciji jeste vid inicijalizacije, ali nam je takav vid inicijalizacije obično besmislen osim u slučajevima kad polja imaju neke podrazumijevanje vrijednosti. Konstruktori imaju iste identifikatore kao i klase kojima pripadaju, za klasu `Complex` naziv konstruktora će takođe biti `Complex` pa je preklapanje konstruktora dozvoljeno, a mogu imati i podrazumijevane argumente. Za razliku od metoda ne daju nikakvu vrijednost. Takođe, 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 real, double imag) : real(real), imag(imag) {}
```

gdje se jasno vidi korisna strana inicijalizatora. Parametre konstruktora direktno prosljeđujemo inicijalizatoru polja. Uočite da se inicijalizator polja i parametar mogu isto znati. 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. Metoda:

```
void Complex::print()
```

nije tražena postavkom zadatka, ali je veoma korisna jer nam omogućava da odštampamo kompleksni broj u čitljivom formatu.

```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;};
    void print();
};

void Complex::print()
{
    cout << "(" << real << ", " << imag << "j)";
}

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

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

Complex Complex::subtract(Complex other)
{
    Complex result;
    result.real = real - other.real;
    result.imag = imag - other.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);
    c3.print();

    c3 = c1.subtract(c2);
    c3.print();
}
```

2. Implementirati funkcije koje određuju:
   - maksimalni elemenat niza,
   - veći od dva cijela broja,
   - maksimalni od dva stringa (leksikografski),
   - ,,maksimalni" od dva karaktera,
   - za jedan string, karakter sa najvećom ASCII vrijednošću.

Definisanje funkcija u programskom jeziku `C++` ostvaruje se naredbama za definisanje funkcije čiji je opšti oblik:

```
oznaka_tipa naziv_funkcije ( niz_parametara ) tijelo_funkcije
```

U odnosu na programski jezik `C` u kojem navođenje prototipa funkcije nije bilo obavezujuće (mada se snažno savjetovalo da se prototip ne izostavlja), u programskom jeziku `C++` prototip je obavezan. Deklarisanje funkcije postiže se naredbama oblika:

```
oznaka_tipa naziv_funkcije ( niz_parametara ) ;
```

gdje se u nizu parametara njihovi identifikatori mogu izostaviti. Obratiti pažnju da deklaracija funkcije ne sadrži njeno tijelo. Rezultat deklarisanja funkcije jeste njen prototip. Prototip mora biti naveden kada se neka funkcija poziva na mjestu na kome definicija funkcije nije dostupna kompajleru. Naravno, definicija i deklaracija se mogu spojiti.

U programskom jeziku `C++` deklaracija funkcije mora sadržati tip vrijednosti funkcije, ali i tip svakog parametra. Za razliku od programskog jezika `C` gdje smo funkcije realizovali po pravilu jedan problem - jedna funkcija odnosno više sličnih problema - više sličnih funkcija (što nikako nije dobro), `C++` nam nudi da preklopimo funkcije (može i preopteretimo od engl. *function overloading*) i napišemo daleko elegantnija i održivija rješenja. Međutim, preklapanje funkcija može dovesti do problema ukoliko se ne koristi na pravi način. Ovi problemi nastaju usljed implicitne konverzije argumenata i razrješenja poziva. Kako bi kompajler izvršio razrješenje poziva, mora znati koje su mu sve preklopljene funkcije dostupne pa mu je samim tim neophodno da u trenutku poziva funkcije na raspolaganju ima bar njihov prototip.

Kao i u programskom jeziku `C`, program počinje sa pretprocesorom. Uključićemo nekoliko zaglavlja:

```
#include <iostream>
//#include <string.h>
#include <cstring>
```

gdje se `iostream` koristi za ulaz/izlaz i ima prednost u odnosu na `stdio.h` zbog objektne orijentisanosti programskog jezika `C++`. Slična je i sudbina zaglavlja `string.h` čija se upotreba obeshrabruje, a koje sada mijenja `cstring` pomoću kojeg dobijamo pristup funkcijama za `C` stringove. Obratite pažnju da su svi identifikatori na engleskom jeziku što je programerska praksa i čega ćemo se pridržavati na ovim vježbama, ali u provjerama znanja nije obavezujuće.

U dijelu pretprocesora imamo još jedan novitet:

```
using namespace std;
```

pomoću kojeg dobijamo pristup prostoru imena (engl. *namespace*) `std` iz već uključenog `iostream`. Da pojednostavimo:

```
std::cout<<maximum(a, n)<<std::endl;
cout<<maximum(a, n)<<endl;
```

ove dvije naredbe su identične, mada nam upotrebu druge omogućava uključivanje prostora imena. U kompleksnim programima ova praksa se izbjegava zbog mogućnosti greške, ali u našem slučaju nam značajno štedi nepotrebno kucanje i povećava preglednost. Obratiti pažnju da smo u slučaju funkcije:

```
char * maximum(char *, char *);
```

vratili pokazivač na niz karaktera jer, kao i u programskom jeziku `C`, niz ne može biti vrijednost funkcije. Neke od funkcija počinju sa ključnom riječju `inline` čime se zapravo tijelo funkcije kopira na svim mjestima gdje se ona poziva tj. ugrađuje direktno u kod programa. Dobra je praksa da to budu kraće i jednostavnije funkcije jer njihovom pretjeranom upotrebom raste EXE verzija programa. U zadatku je korišćen i ternarni operator:

```
return (izraz) ? a : b; // ako je izraz tacan vrati a, u suprotnom vrati b
```

Rješenje zadatka je:

```
#include <iostream>
//#include <string.h>
#include <cstring>

using namespace std;

int maximum(int *, int);
inline int maximum(int, int);
char * maximum(char *, char *);
inline char maximum(char, char);
char maximum(char*);

int main()
{
    int a[] = {1,5,4,3,2,6}, n = 6;
    cout << maximum(a, n) << endl;
    cout << maximum(2, 5)<< endl;
    cout << maximum("string", "program")
   << endl;
    cout << maximum('a', 'A') << endl;
    cout << maximum("string");
}

int maximum(int *a, int n)
{
    int m;
    m = a[0];
    for(int i = 1; i < n; i++)
    if(a[i] > m)
        m = a[i];
    return m;
}
```

```
inline int maximum(int a, int b)
{
    return (a >= b) ? a : b;
}

char * maximum(char *a, char *b)
{
    if(strcmp(a,b) >= 0) return a;
    return b;
}

inline char maximum(char a, char b)
{
    return (a >= b) ? a : b;
}

char maximum(char *a)
{
    char m;
    m = a[0];
    for(int i = 1; i < strlen(a); i++)
        if(a[i] > m)
            m = a[i];
    return m;
}
```

3. Date su deklaracije funkcija:

```
int f(int, char='0');
int f(char, char);
int f(double);
```

Šta će se desiti nakon sljedećih poziva funkcije:

```
f('0');
f(2.34);
f('c', 3);
```

U prvom slučaju je očigledno da će se pozvati prva funkcija. Zašto? Druga funkcija otpada zato što ima dva obavezujuća argumenta. Mada je parametar prve funkcije tipa `int` on se standardnom konverzijom može dobiti implicitnim konvertovanjem karaktera u cjelobrojnu vrijednost. Funkcija će se pozvati kao:

```
f('0', '0');
```

Kod drugog poziva će se pozvati treća funkcija zato što je došlo do potpunog poklapanja argumenata tipa `double`. Međutim, u slučaju trećeg poziva imamo malo komplikovaniju situaciju. Prva funkcija ima 0 potpunih i 2 preklapanja konverzijom dok druga funkcija ima 1 potpuno preklapanje i 1 sa konverzijom podataka. Zapamtiti sva pravila razrješenja poziva i znati ih primijeniti nije lako te se uvijek savjetuje da se kada je god to moguće izbjegavaju sporne situacije.

4. Dat je niz cjelobrojnih elemenata. Kreirati funkciju koja podrazumijevano obrće elemente niza ,,naopačke". Ako je kao argument zadat pozitivan cijeli broj, potrebno je formirati novi niz koji predstavlja elemente starog niza od prvog elementa, sa datim preskokom do kraja. Ako je zadat negativan broj, potrebno je formirati novi niz koji ide od posljednjeg elementa ka početku sa odgovarajućim zadatim preskokom. Podrazumijevana vrijednost preskoka je `-1`.

U programskom jeziku `C` upoznali smo se sa naredbama `malloc` i `free` pomoću kojih smo alocirali i dealocirali memoriju. Mada ih i dalje možemo koristiti, programski jezik `C++` ima operatore ugrađene u sam jezik koji su više objektno orijentisani. To su `new` i `delete`. Pomoću njih možemo zauzeti i osloboditi memoriju. U slučaju našeg zadatka potrebno je formirati novi niz, odnosno:

```
xNew = new(int[n]);
```

nakon što smo zauzeli odgovarajuću memoriju, potrebno ga je po zahtjevima zadatka popuniti. Funkcija će imati podrazumijevanu vrijednost `-1`. Napomena, vrijednost podrazumijevanog argumenta se mora unijeti unutar deklaracije funkcije. Zašto? Kao rezultat funkcije vraćamo novu dužinu niza.

`C++` kao neki drugi jezici ne posjeduje jedinstven standard za imenovanje funkcija i promjenljivih. Ogromne kompanije kao što su Google i Microsoft definišu interne standarde koji su javno dostupni. Način na koji pišemo kod ne utiče na njegove performanse, ali programiranje je timski posao. Standardi smanjuju greške zbog bolje preglednosti i omogućavaju nam da se lakše upoznamo sa tuđim kodom. U ovom zadatku koristili smo **camelCase** za nazive funkcija i promjenljivih, a pored ovoga postoji i još nekoliko standarda o kojima se možete informisati. Poštovanje ove prakse se neće ocjenjivati, ali će vam biti od koristi!

Rješenje zadatka je:

```
#include <iostream>

using namespace std;

int reorder(int *, int, int *, int = -1);

int main()
{
    int x[10], n, nNew;
    int i, *xNew;

    cout << "Unijeti duzinu niza: ";
    cin >> n;
    cout << "Unijeti elemente niza: ";
    for(i = 0; i < n; i++)
        cin >> x[i];

    xNew = new int[n];

    cout << "Unijeti niz je: " << endl;
    for(i = 0; i < n ; i++) cout << x[i] << " ";
    cout << endl;

    nNew = reorder(x, n, xNew);
    for(i = 0; i < n; i++) cout << xNew[i] << " ";
    cout << endl;

    nNew = reorder(x, n, xNew, 0);
    for(i = 0; i < nNew; i++) cout << xNew[i] << " ";
    cout << endl;

    nNew = reorder(x, n, xNew, -2);
    for(i = 0; i < nNew; i++) cout << xNew[i] << " ";
    cout << endl;

    delete [] xNew; // dobra praksa!
}

int reorder(int *x, int n, int *xResult, int step)
{
    int j = 0;
    if(step > 0) // modifikovati da radi i za step = 0
        for(int i = 0; i < n; i += step) xResult[j++] = x[i];
    else
        for(int i = n - 1; i >= 0; i += step) xResult[j++] = x[i];
    return j;
}
```

Pažnja! Većina modernih OS-ova će osloboditi dinamički zauzetu memoriju na kraju programa iako mi to eksplicitno ne naglasimo, međutim programeri se na to ne mogu osloniti te je obavezno osloboditi dinamički zauzetu memoriju kada nam više nije potrebna. Statički zauzeta memorija se čuva u strukturi podataka koja se zove **stack** i o njoj vodi brigu OS. Dinamički zauzeta memorija se čuva u strukturi podataka koja se zove **heap**. Statička memorija se zauzima prije izvršavanja programa dok se dinamička zauzima za vrijeme njegovog trajanja. Operator `delete` dealocira memoriju i poziva destruktor (učićemo) za pojedinačni objekat napravljen pomoću operatora `new`. Operator `delete []` dealocira memoriju i poziva destruktore za niz objekata kreiranih pomoću operatora `new []`. Zamjena ovih operatora dovešće do nedefinisanog ponašanja.