OBJEKTNO PROGRAMIRANJE 1

Oznaka predmeta: OP1

Predavanje broj: 10

Nastavna jedinica: Objektno orijentisani koncepti.
Preklapanje operatora

Nastavne teme: Konstruktor: inicijalizacija članova, pozivi, konverzije tipova. Konstruktor kopije. Move konstruktor. Destruktor. Privremeni objekti. Eksplicitna inicijalizacija objekata. Preklapanje operatora. Operatorske funkcije kao članice i prijatelji. Globalne operatorske funkcije. Operatorske funkcije. Unarni i binarni operatori. Operatori ++ i --. Operator ().

Predavač: prof. dr Perica S. Štrbac, dipl. ing.

Literatura:

Dragan Milićev, “Objektno orijentisano programiranje na jeziku C++”, Mikro knjiga, Beograd, 2005.

Pojam konstruktora

- Funkcija članica koja nosi isto ime kao i klasa naziva se *konstruktor* (*constructor*). Ova funkcija poziva se automatski prilikom kreiranja objekta te klase i ima zadatak da inicijalizuje članove klase:

```
class complex {
  float real,imag;
public:
  void cAdd(complex);    void cSub(complex);
  float cRe();           float cIm();
  complex(float=0, float=0);  // konstruktor sa podrazumevanim
};                            // argumentima
complex::complex (float r, float i) { real=r; imag=i; }
```

- Konstruktor nema tip koji vraća, čak to ne može biti ni tip void. Konstruktor može da ima argumente proizvoljnog tipa. Unutar konstruktora, privatnim članovima pristupa se kao i u bilo kojoj drugoj funkciji članici.
- Konstruktor se poziva pri kreiranju objekta klase, odnosno u trenutku definisanja objekta. Ako klasa ima konstruktor, onda će svaki objekat ove klase biti inicijalizovan pozivom konstruktora, pre bilo koje upotrebe tog objekta.

```
void main () { complex c1(0.5,-3); // poziva se konstruktor
               complex c2;         // poziva se konstruktor sa
}                                  // podrazumevanim argumentima
```

Pojam konstruktora

- Konstruktor, kao i svaka funkcija članica, može biti preklopljen (*overloaded*).

- Na mestu kreiranja objekta, poziva se onaj konstruktor, koji najbolje odgovara po tipovima svojih formalnih argumenata, prema opštim pravilima o razrešavanju poziva.

- Konstruktor, kao i svaka druga funkcija članica, podleže uobičajenim pravilima kontrole prava pristupa.
  - konstruktor može biti
    - javni (najčešće),
    - zaštićen (jedino izvedene klase mogu kreirati objekte neke klase)
    - privatni.

- Konstruktor, kao i ostale funkcije članice, može biti *inline*,
  - postiže se na isti način kao i sa drugim funkcijama članicama (eksplicitnom deklaracijom inline u cpp-u, ili navođenjem definicija funkcije unutar deklaracije klase)

Eksplicitni poziv konstruktora

- Dozvoljena je implicitna konverzija:

```
class A {
  int i;
public:
  A(int);
};
void f(A a) {...}
void g()  {   A a1 = 1;  A a2 = A(2);  A a3(3);  a1 = 4;  f(5); }
```

- Dozvoljen je eksplicitni poziv konstruktora:

```
class A {
  int i;
public:
  explicit A(int);
};
void f(A) {...}
void g()  {
  A a1 = 1;          //error
  A a2 = A(2);       //OK
  A a3(3);           //OK
  a1 = 4;            //error
  f(5);              //error
}
```

Inicijalizacija članova

- Za osnovne tipove podataka, značenje dodele vrednosti isto je kao i značenje inicijalizacije.
- Kako će se inicijalizovati članovi objekta koji su, konstantni (opracija dodele nije dozvoljena) koji su reference (dodela nije moguća), ili koji su objekti neke druge klase (dodela nije definisana).

```
class Y {
          int a;
  public: Y(int i) {a=i;} // konstruktor Y::Y;
};
class X {
  const int ci;       //konstanta
  int &ri;              //referenca
  Y y;                    //objekat klase Y
public:
  X(int&);            // konstruktor X::X;
};
X::X (int &i) {
  ci=i;           // greška: ci je konstanta!
  ri=i;           // ovo nije inicijalizacija reference!
  //??? kako uopšte inicijalizovati y pozivom Y::Y?
}
```

Inicijalizacija članova

- Pojam inicijalizacije u opštem slučaju se potpuno razlikuje od pojma dodele.
- Članovi objekta se inicijalizuju navođenjem inicijalizatora u zaglavlju konstruktora, iza znaka dvotačka (:). Za prethodni primer klase X, definicija konstruktora može da bude:

```
X::X (int &i)
:
ci(i), ri(i), y(i)
{}
```

- Sada je u zaglavlju konstruktora navedeno da se članovi ci, ri i y inicijalizuju objektom i,
  član y, koji je tipa Y, inicijalizuje se zapravo pozivom konstruktora Y::Y sa argumentom i.
- U inicijalizatorima članova mogu se upotrebljavati izrazi koji sadrže imena formalnih argumenata konstruktora u kome se nalaze.
- Članovi objekta inicijalizuju se po redosledu po kome su deklarisani, bez obzira na redosled navođenja inicijalizatora u zaglavlju konstruktora. Tek kada se svi članovi inicijalizuju, izvršava se telo konstruktora.
- Na ovaj način se NE mogu eksplicitno inicijalizovati članovi koji su nizovi.

Poziv konstruktora

- Konstruktor je funkcija koja pretvara "presne" memorijske lokacije, koje je sistem odvojio za novi objekat, u "pravi" objekat koji ima svoje članove i koji može da prima poruke.
- Konstruktor ima zadatak da inicijalizuje članove i postavi sve potrebne pomoćne informacije vezane za nasleđivanje i polimorfizam, kako bi se objekat "ponašao" onako kako je propisano semantikom klase.
- Konstruktor se poziva uvek kada se kreira objekat klase:
  - 1. Kada se izvršava naredba koja je definicija globalnog objekta.
  - 2. Kada se izvršava naredba koja je definiciju lokalnog objekta unutar bloka.
  - 3. Pri kreiranju objekta, pozivaju se konstruktori njegovih podataka članova.
  - 4. Kada se kreira dinamički objekat operatorom new.
  - 5. Kada se kreira privremeni objekat (npr. pri povratku iz funkcije).
  - 6. Kada se kreira objekat neke klase D, poziva se konstruktor klase B iz koje je klasa D izvedena (konstruktor osnovne klase inicijalizuje podobjekat tipa B, koga sadrži objekat tipa D).
  - 7. Kada se konstruktor eksplicitno pozove.

Poziv konstruktora

- Kada se konstruktor eksplicitno poziva, ne navodi se ime nijednog objekta, jer objekat i ne postoji kada se poziva njegov konstruktor. Pri tome se kreira privremeni, bezimeni objekat date klase neodređenog životnog veka.

```
complex c=complex(3.0,1.1);  // eksplicitni poziv konstruktora
```

- Klasa `table` realizuje neku tabelu elemenata tipa T.
  - klasa ima konstruktor bez argumenata, kojim će se kreirati prazna tabela,
  - klasa ima drugi konstruktor koji se poziva sa jednim argumentom tipa T&, kojim se kreira tabela sa jednim elementom dostavljenim kao argument.

```
class table {
  //...
public:
  table();        // konstruktor koji kreira praznu tabelu;
  table(T&);      // konstruktor koji kreira tabelu sa jednim el.;
  void put(T&);   // funkcija koja stavlja element u tabelu;
  T& get();       // funkcija koja uzima element iz tabele;
};
table::table (T &t) {
  table();    // greska: poziv konstruktora koji kreira praznu tabelu
  put(t);     // stavi element u tabelu;
}
```

Poziv konstruktora

- Poziv konstruktora table() unutar konstruktora table::table(T&) predstavlja kreiranje jednog posebnog privremenog objekta, koji nema nikakve veze sa objektom koji se kreira.
- Rešenje je u definisanju posebne funkcije članice init, koja će obavljati sav posao potreban za kreiranje prazne tabele, i koju će pozivati oba konstruktora. Ova funkcija članica treba da bude privatna, ili, još bolje, zaštićena.

```
class table {
    //...
  protected:
    void init();   // funkcija koja kreira praznu tabelu;
  public:
    table();       // konstruktor koji kreira praznu tabelu;
    table(T&);     // konstruktor koji kreira tabelu sa jednim el.;
    void put(T&);  // funkcija koja stavlja element u tabelu;
    T& get();      // funkcija koja uzima element iz tabele;
};
table::table (T& t) {
  init();        // kreiraj praznu tabelu;
  put(t);        // stavi element u tabelu;
}
```

- Sada se funkcija init unutar konstruktora poziva upravo za objekat koji se trenutno kreira, i izvršava sve potrebne radnje za kreiranje prazne tabele.

Konstruktor kopije

- Kada se objekat x1 klase X inicijalizuje drugim objektom x2 tipa X ili tipa X&, C++ podrazumevano vrši prostu inicijalizaciju redom članova objekta x1 članovima objekta x2.
- Kada objekti klase X sadrže člana koji je pokazivač ili referenca na dinamički objekat tipa T, a potrebno je da svaki objekat klase X poseduje svoj primerak dinamičkog objekta (prevodilac bi podrazumevano formirao novi pokazivač (ili referencu) koji ukazuje na isti dinamički objekat na koga je ukazivao i član objekta x2 kojim se x1 inicijalizuje).

```
class X {
  int *pi;
public:
  X(int i) : pi(new int(i)) {}  // konstruktor
};
void main () {
  X x2(1);
  X x1=x2; //x1.pi ukazuje gde i x2.pi
}
```

Konstruktor X::X(int) inicijalizuje pokazivač pi da ukazuje na dinamički objekat kreiran operatorom new. Kada se kreira x1, član x2.pi će se prepisati u člana x1.pi, pa će članovi oba objekta ukazivati na isti dinamički objekat. Ovo nije ono što se želi: da svaki objekat klase ima svoj primerak dinamičkog objekta.

Konstruktor kopije

- *Konstruktor kopije* (*copy constructor*) je konstruktor klase X koji se može pozvati sa samo jednim argumentom tipa X&.
- Taj konstruktor se poziva uvek kada se objekat inicijalizuje objektom iste klase.
- Poziv konstruktora kopije se dešava:
  - prilikom inicijalizacije objekta (pomoću znaka = ili sa zagradama) gde se navodi kopiranje drugog objekta iste klase,
  - prilikom prenosa argumenata u funkciju (kreira se lokalni automatski objekat)
  - prilikom vraćanja vrednosti iz funkcije (kreira se privremeni objekat koji prihvata vraćenu vrednost).
- Konstruktor kopije klase X može biti bilo koji konstruktor koji se može pozvati sa samo jednim argumentom tipa X&.

```
X::X(X&),
X::X(const X&)
X::X(X&, int=0)

X::X(X) //greska: konstruktor kopije ne može zahtevati
        //formalni argument tipa X
```

Konstruktor kopije

- Za prethodni primer, klasa X može biti realizovana na sledeći način:

```
class X {
  int *pi;
public:
  X(int i) : pi(new int(i)) {}  // konstruktor;
  X(const X&);                  // konstruktor kopije;
  //...
};
X::X(const X &x) :  pi(new int(*x.pi))
{}                  //kreira se novi dinamički objekat,
                    // pa pi ukazuje na njega;
X f(X x1) {
  X x2=x1;   // poziva se konstruktor kopije X(const X&) za x2;
  //...
  return x2; // poziva se konstruktor kopije za
}            // privremeni objekat u koji se smešta rezultat;

void g() {
  X xa=3, xb=1;
  xa=f(xb);     // poziva se konstruktor kopije samo za
                // formalni argument x1 funkcije f,
                // a u xa se samo prepisuje privremeni objekat
}
```

Move konstruktor

- Ideja je da se izbegne realokacija memorije tako da se samo privremeni objekat "pomeri" umesto da se kopiraju sva polja objekta.

```
class ArrayWrapper{
  private:
    int *pvals;
    int size;
  public:
    ArrayWrapper () // default konstruktor npr. za 32 elementa
        : pvals( new int[ 32 ] ), size( 32 )  {}
    ArrayWrapper (int n)
        : pvals( new int[ n ] ), size( n )    {}

    ArrayWrapper (ArrayWrapper&& other) // move constructor
        : pvals( other.pvals ), size( other.size )
    {   other.pvals = nullptr;  other.size = 0;    }

    ArrayWrapper (const ArrayWrapper& other) // copy constructor
        : pvals( new int[ other.size  ] ), size( other.size )
    {
        for( int i=0; i<size; ++i )  pvals[ i ] = other.pvals[ i ];
    }
    ~ArrayWrapper ()    {   delete [] pvals;    }
};
```

Pravila vezana za konstruktore

- Konstruktor koji se može pozvati bez argumenata naziva se *podrazumevani konstruktor* (*default constructor*), dakle, konstruktor koji nema deklarisane formalne argumente, ili čiji svi formalni argumenti imaju *default* vrednosti.
  - Prevodilac će sâm generisati podrazumevani konstruktor, ukoliko nijedan konstruktor nije deklarisan za klasu. Ovako generisani konstruktor je javni. Ovako generisani podrazumevani konstruktor će pozivati podrazumevane konstruktore za članove objekta.
    - To znači da članovi koji su objekti drugih klasa, moraju imati podrazumevane konstruktore (koji su možda opet generisani implicitno).
- Ako klasa nema nijedan deklarisani konstruktor kopije, prevodilac će implicitno generisati konstruktor kopije koji je javni.
  - Ovakav konstruktor kopije vrši redom inicijalizaciju članova klase. Ako su članovi objekti neke druge klase, pozivaće se njihovi konstruktori kopija.
- Konstruktor se može pozvati i za const i za volatile objekte, a ne može biti ni const ni volatile ni static ni virtual. Konstruktor se **NE** nasleđuje.
- Nije moguće uzimati adresu konstruktora.
- Konstruktori elemenata niza pozivaju se redom, po rastućem indeksu.
- Konstruktor poseduje pokazivač this.

Konverzije tipova pomoću konstruktora

- Ako klasa X ima konstruktor koji se može pozvati sa jednim stvarnim argumentom tipa T, gde je T ugrađeni ili korisnički definisan tip, ovim konstruktorom se definiše *korisnička konverzija* (*user-defined conversion*) iz tipa T u tip X.

```
class X {
  //...
public:
  X (T);          // ovim konstruktorom je definisana konverzija
  //...           // iz tipa T u tip X;
};

void f() {
  T t;
  X x1(t);        // isto što i: X x1=t;
  X x2=t;         // isto što i: X x2(t);
  //...
}
```

- Dovoljno je da se konstruktor može pozvati sa jednim argumentom tipa T, da bi se definisala konverzija iz tipa T u tip X.
  - ovakva konverzija je definisana i konstruktorom X::X(const T&), i konstruktorom X::X(T, int=0), i slično.

Konverzije tipova pomoću konstruktora

- Konverzija se vrši samo ako je neposredna.

```
class X { public: X(int);};
class Y { public: Y(X);};
Y y=1;                  // greška: Y( X(1) ) se ne pokušava!
```

- Ako se definiše konverzija iz ugrađenog tipa u korisnički tip, onda se konstante mogu upotrebljavati u izrazima.

```
class complex {
     double real,imag;
   public:
     complex (double r, double i=0) // definiše konverziju iz double
       {real=r; imag=i;}            // u complex;
     complex operator+(complex);    // operator+;
     //...
};


complex c(0);
const double pi=3.14;

c=complex(2.1,0.1)+2*pi;  // isto kao:
c=complex(2.1,0.1)+complex(2*pi);
```

Destruktor

- Posebna funkcija članica klase, koja obezbeđuje ukidanje objekta klase, naziva se *destruktorom* (*destructor*).
  - funkcija članica klase X, koja nosi ime ~X, naziva se *destruktor* (*destructor*).
- Destruktor se poziva implicitno uvek kada se ukida objekat :
  - 1. Kada se ukida statički objekat koji je kreiran, na završetku izvršavanja programa, poziva se njegov destruktor.
  - 2. Kada se napušta oblast važenja automatskog objekta poziva se njegov destruktor (npr. formalni argumenti funkcije pri napuštanju funkcije).
  - 3. Ukidanjem objekta pozivaju se destruktori njegovih podataka članova.
  - 4. Kada se dinamički objekat ukida operatorom delete, poziva se njegov destruktor, neposredno pre oslobađanja dela dinamičke memorije koju je on zauzimao.
  - 5. Kada se ukida privremeni objekat, poziva se njegov destruktor. Pri povratku iz funkcije, kreira se privremeni objekat koji se inicijalizuje vraćenom vrednošću funkcije, a ukida kada više nije potreban.
  - 6. Kada se ukida objekat neke klase D, poziva se destruktor klase B iz koje je klasa D izvedena.
  - 7. Kada se destruktor eksplicitno pozove.

Destruktor

- Data je klasa X koja sadrži član int *pi. Pri kreiranju objekta klase X, kreira se dinamički objekat koji pripada ovom objektu (konstruktorima (uključujući i konstruktor kopije)). Kada se objekat klase X ukida, potrebno je ukinuti i dinamički objekat koji mu je pridružen. Ovo se može obezbediti destruktorom:

```
class X {
  int *pi;
public:
  X(int i) : pi(new int(i)) {}           // konstruktor;
  X(const X &x) : pi(new int(*x.pi)) {}  // konstruktor kopije;
  ~X() { delete pi; }                    // destruktor;
};
void f() {
  X x(0);
}  //ovde se implicitno (automatski) poziva destruktor objekta x;
```

- Kada u ovom primeru ne bi postojao destruktor, program ne bi bio korektan.
  - Svaki put kada bi se ukidao neki objekat klase X, ukinuo bi se samo njegov član pi, pa bi dinamički objekat na koga ukazuje pi ostao da živi i ne može mu se više pristupiti.
- Telo destruktora izvršava se pre izvršavanja destruktora za članove objekta koji se izvršavaju obrnutim redosledom od redosleda konstrukcije.

Destruktor

- I za destruktor, kao i za sve članove, važe opšta pravila kontrole prava pristupa. Destruktor može biti privatan, zaštićeni ili javni (najčešće)
- Destruktor ne može imati argumente, kao ni povratni tip, čak ni tip void.
  - Ovim je jasno da za svaku klasu može postojati samo jedan destruktor.
    - Ne može se uzimati adresa destruktora.
- Destruktor se može pozvati i za const i za volatile objekat.
  Destruktor ne može biti const ni volatile, ni static
  ali može biti virtual (radimo detaljno kasnije).
- Destruktori se **NE** nasleđuju.
- Prevodilac će sâm generisati destruktor za klasu B koja nema deklarisan destruktor ako član klase B ili osnovna klasa A klase B ima destruktor.
  - Ovaj generisani destruktor će pozivati destruktor osnovne klase i destruktore članova. Ovaj generisani destruktor je javni.
- Unutar destruktora mogu se pozivati funkcije članice iste klase, na uobičajeni način.
- Destruktor se može pozivati i eksplicitno:
  - navesti konkretan objekat čiji se destruktor poziva

```
x.~X();
```

Privremeni objekti (*temporary objects*)

- Životni vek bezimenih objekata je u potpunosti pod kontrolom prevodioca.
  - Tačno mesto na kome se kreira privremeni objekat nije propisano specifikacijom jezika.
  - Tačan trenutak ukidanja (poziva destruktora, ako ga ima) privremenog objekta nije propisan tim specifikacijama.

```
class X {
  public:
    X(int);    // konstruktor;
    X(X&);     // konstruktor kopije;
    ~X();      // destruktor;
};
X f(X);                          // neka funkcija f;
void g() {   X a=f(X(1)); }
```

- U pozivu f(X(1)), prevodilac
  - može da kreira privremeni objekat, pozivom konstruktora X::X(1),
  - zatim kreira formalni argument funkcije, koji će inicijalizovati ovim privremenim objektom, pozivom konstruktora kopije X::X(X&).
  - moguće je i da prevodilac neposredno konstruiše objekat X(1) u prostoru formalnog argumenta funkcije f, bez poziva konstruktora kopije. Na ovaj drugi način prevodilac optimizuje postupak inicijalizacije.

Privremeni objekti

- Moguće je da prevodilac kreira privremeni objekat u koji smešta povratnu vrednost poziva funkcije f, a da zatim tim objektom inicijalizuje objekat ***a***, pozivom konstruktora kopije X::X(X&).
- Moguće je i da prevodilac neposredno konstruiše povratnu vrednost poziva funkcije u prostoru objekta ***a*** bez poziva konstruktora kopije.
- Svaki privremeni objekat koji je kreiran, mora biti i ukinut, pozivom destruktora.
- Dve radnje su dozvoljene sa privremenim objektima: uzimanje (korišćenje) njihove vrednosti (implicitnim kopiranjem), i vezivanje reference za njih.
  - Ako je uzeta vrednost privremenog objekta isti se može ukinuti odmah.
  - Ako je referenca vezana za privremeni objekat, ovaj objekat se ne ukida sve dok se ne ukine referenca koja upućuje na njega, ali se svakako ukida po izlasku iz opsega važenja u kome je kreiran.
- Ako je konstruktor kopije deklarisan kao privatan, onda:

```
complex y=x;              //nedozvoljena inicijalizacija preko k.kopije
complex z=1.0;               //isti slucaj
complex c=complex(1.0,2.0);//isti slucaj
complex c(1.0,2.0);  // OK, poziv konstruktora complex(double,double)
complex d;           // OK, poziv konstruktora complex();
```

Eksplicitna inicijalizacija objekata

- Kada se inicijalizacija vrši notacijom sa znakom = semantika je sledeća:
  - Kreira se privremeni objekat tipa X, konverzijom iz tipa izraza koji predstavlja inicijalizator, a zatim se taj privremeni objekat kopira u objekat koji se inicijalizuje, pozivom konstruktora kopije.
  - Vrši se provera prava pristupa do funkcija koje specificiraju konverzije, kao i do konstruktora kopije.
- Kada se koristi inicijalizacija sa zagradama, semantika je sledeća:
  - Poziva se konstruktor za objekat koji se inicijalizuje, a poziv se razrešava u pogledu izbora iz skupa preklopljenih konstruktora.
- U sekvencama implicitnih konverzija koje se upotrebljavaju u pozivima funkcija, dozvoljava se najviše jedna korisnički definisana konverzija.

```
class Y { public: int i; Y(int);};
class X { public: X(Y);  };
X x1(1);          // u redu; konverzija (int->Y),
X x2=X(1);        // u redu; konverzija (int->Y),
X x3=1;           // greška! konverzija (int->Y->X),

typedef int* PI;
PI p1=1; PI p2(1);// greška: ne može se inicijalizovati int* sa int
PI p3=PI(1);   // ok, eksplicitna konverzija koja je dozvoljena
PI p4(PI(1));  // ok, eksplicitna konverzija koja je dozvoljena
```

Eksplicitna inicijalizacija objekata

- Nizovi objekata klasa koje imaju konstruktore, koriste konstruktore u inicijalizaciji baš kao i individualni objekti.
  - Nizovi se mogu inicijalizovati listom inicijalizatora unutar vitičastih zagrada (ranije lekcije: agregat).

```
complex a[5]={1,complex(0,1),complex(-1,0)};  //bice 5 objekata
complex c={1,2};   //greška! visak inicijalizatora, treba jedan
```

- Klasa X može imati podatak član koji je objekat klase C samo ako:
  - 1) klasa C nema deklarisan konstruktor
  - 2) klasa C ima podrazumevani konstruktor (kreiranjem objekta tipa X pri inicijalizaciji člana tipa C, pozivaće se podrazumevani konstruktor klase C)
  - 3) klasa X ima konstruktor, i svaki konstruktor klase X ima inicijalizator člana tipa C u svom zaglavlju.

```
class C {
   public: int i; C(int i){this->i=i;}
};
class X {
  public:
    C c;
    X (): c(5) {} // bez inicijalizacije c bila bi greska
};
```

Primer 1/6

```
//_____________________________Automobil.h
#pragma once
#include <iostream>
#include <string>
using namespace std;

class Automobil{
  private:
    string model;
    double cena;
  public:
    Automobil();
    void PostaviPodatke(string,double);
    void IspisiPodatke() const;
    double UzmiCenu() const;
};
//_____________________________Automobil.cpp
#include "Automobil.h"

Automobil::Automobil(){
  model = "nema naziv";
  cena = 0;
}
```

Primer 3/6

```
void Automobil::PostaviPodatke(
                      string nazivmodela ,double cenaautomobila){
  model = nazivmodela;   cena = cenaautomobila;
}
void Automobil::IspisiPodatke() const {
  cout<<"Model:"<<model<<endl;   cout<<"Cena :"<<cena <<endl;
}
double Automobil::UzmiCenu() const{   return cena; }
//_____________________________KKopije.h
#pragma once
class Automobil;
class KKopije {
    int *pi;
    Automobil *pa;
  public:
    KKopije(int i, Automobil a);    // konstruktor;
    KKopije(const KKopije&); // konstruktor kopije;
    int        getINT() const;
    Automobil* getAUTOMOBIL() const;
    void set(int i);
    ~KKopije();
};
```

Primer 4/6

```
//_____________________________KKopije.cpp
#include "KKopije.h"
#include "Automobil.h"

KKopije::KKopije(int i, Automobil a)
:
pi(new int(i)),
pa(new Automobil(a))
{
}
KKopije::KKopije(const KKopije& kk){
  pi=new int(*kk.pi);
  pa=new Automobil(*kk.pa);
}
int KKopije::getINT() const { return *pi; }
Automobil* KKopije::getAUTOMOBIL() const { return pa; }
void KKopije::set(int i) { *pi=i; }
KKopije::~KKopije()
{
  if(pi!=0) delete pi;
  if(pa!=0) delete pa;
}
```

Primer 5/6

```
#include <iostream>
using namespace std;
#include "KKopije.h"
#include "Automobil.h"

void Ispisi(KKopije kk_a, KKopije kk_b){ //moglo je i sa 1. f. argum.
  cout<<endl<<"Ispisi:"<<endl;
  cout<<"________ kk_a _____________________"<<endl;
  cout<<" *pi         = "<<kk_a.getINT()<<endl;
  cout<<" *Automobil = "<<endl; kk_a.getAUTOMOBIL()->IspisiPodatke();
  cout<<endl;
  cout<<"________ kk_b _____________________"<<endl;
  cout<<" *pi         = "<<kk_b.getINT()<<endl;
  cout<<" *Automobil = "<<endl; kk_b.getAUTOMOBIL()->IspisiPodatke();
  cout<<endl;
}
void main(){
  Automobil alfa;
  alfa.PostaviPodatke("Alfa Romeo GT",30000);

  KKopije kk_a(10,alfa);
  alfa.PostaviPodatke("Alfa 166",28000);
  KKopije kk_b(kk_a);
```

Primer 6/6

```
  Ispisi(kk_a, kk_b);
  kk_a.set(5);
  kk_a.getAUTOMOBIL()->PostaviPodatke("Ford Mustang", 60000);
  Ispisi(kk_a, kk_b);
  system("pause");
}
//____________________IZLAZ (plava pa crvena kolona)_____________________
```

```
Ispisi:
________ kk_a ___________________
 *pi         = 10
 *Automobil =
Model:Alfa Romeo GT
Cena :30000

________ kk_b ___________________
 *pi         = 10
 *Automobil =
Model:Alfa Romeo GT
Cena :30000
```

```
Ispisi:
________ kk_a ___________________
 *pi         = 5
 *Automobil =
Model:Ford Mustang
Cena :60000

________ kk_b ___________________
 *pi         = 10
 *Automobil =
Model:Alfa Romeo GT
Cena :30000

Press any key to continue . . .
```

Preklapanje operatora (*operator overloading*)

- Ovaj princip omogućuje da se definišu značenja operatora za korisničke tipove i formiraju izrazi sa objektima ovih tipova,
  na primer operacije nad kompleksnim brojevima (ca*cb+cc-cd),
  matricama (ma*mb+mc-md)
- Operatorske funkcije imaju sledeća posebna svojstva:
  - imaju posebna imena: one se nazivaju operator@, gde je @ neki operator ugrađen u jezik
  - osim eksplicitnog poziva preko svog imena, na način kao što se pozivaju i sve druge funkcije, operatorske funkcije mogu se pozivati notacijom koja odgovara pozivu operacija za ugrađene tipove.

- Ne mogu se preklopiti operatori:

  `.  .*  ::  ?:  sizeof`
- Mogu se preklopiti i unarne i binarne forme operatora:

  `+  -  *  &`
- Operatorske funkcije mogu se pozivati:

```
complex c=a.operator+(b); //isto kao
complex c=a+b;            // poziv a.operator+(b);
```

Preklapanje operatora

- Operatorska funkcija mora biti ili članica neke klase, ili imati bar jedan argument tipa klase, nabrajanja, reference na klasu, ili reference na nabrajanje.

```
inline complex operator+ (const complex &c, double d) {
  return complex(c.real+d,c.imag);
}  // ovo je globalna prijateljska funkcija
```

- Nije moguće promeniti prioritet, način grupisanja ili broj operanada koje operatori prihvataju, u odnosu na svojstva koja su ugrađena u jezik.
- Nije moguće uvoditi nove operatore, koji ne postoje u jeziku.
  - Na primer, nije dozvoljeno definisati operator **, (”stepenovanje”).
- Npr. za operatore dodele (=), uzimanja adrese (&) i sekvence (,), moguće je predefinisati podrazumevano značenje.
- Ne mogu se predefinisati značenja operatora za ugrađene tipove podataka. Na primer, nije dozvoljeno sledeće:

```
complex operator+(double,double);  // greška!
```

- Od operatorskih funkcija samo se operatorska funkcija dodele (=) **NE** nasleđuje.
- Operatorske funkcije NE mogu imati podrazumevane argumente (rekosmo ranije).

Operatorske funkcije kao članice i prijatelji

- Operatorske funkcije mogu da budu članice klasa ili globalne funkcije (najčešće prijatelji klasa).

  – Ako je @ neki binarni operator (na primer binarni +), on može da se realizuje kao funkcija članica klase X na sledeći način:

```
T operator@ (X);            //ima jedan argument.
```

  Tada se prvi operand operacije @ smatra objektom čija se funkcija članica operator@ poziva, a drugi operand se dostavlja kao (jedini) argument.

  – Može da se realizuje i kao globalna funkcija na sledeći način:

```
T operator@ (X,X); //ima dva argumenta.
```

  Tada se prvi operand operacije @ dostavlja kao prvi stvarni argument poziva operatorske funkcije operator@, a drugi operand se dostavlja kao drugi stvarni argument tog poziva.

  - Za razliku od funkcija članica, globalne operatorske funkcije dozvoljavaju implicitne konverzije svojih stvarnih argumenata u tipove formalnih argumenata.

Globalne operatorske funkcije

- Definisanjem konstruktora klase kompleksnih brojeva, koji može da se pozove samo sa jednim argumentom tipa double, u stvari definiše korisnička konverzija iz tipa double u tip complex.
  - ako definišemo globalnu operatorsku funkciju operator+(complex,complex), onda će ova operatorska funkcija moći da se pozove i sa prvim stvarnim argumentom tipa double (ili čak int, jer postoji standardna konverzija iz int u double), dok je samo drugi operand tipa complex

```
class complex { double real,imag;
public:
  complex(double r=0, double i=0) : real(r),imag(i) {}
  friend complex operator+ (complex,complex);
};
complex operator+(complex c1, complex c2) {
  return complex(c1.real+c2.real,c1.imag+c2.imag);
}
void main () {
  complex c = 3.14 + complex(1.0,-1.0);
     // poziva se operator+( complex(3.14) , complex(1.0,-1.0) );
     // levi operand → u tip complex prvog formalnog argumenta.
}    // operacija sabiranja racionalnog broja sa kompleksnim
```

Operatorske funkcije

- Za razliku od prethodne realizacije, sledeća realizacija to ne obezbeđuje:

```
class complex {
  double real,imag;
public:
  complex(double r=0, double i=0) : real(r),imag(i) {}
  complex operator+ (complex) const;
};

complex complex::operator+(complex c2) const {
  return complex(real+c2.real,imag+c2.imag);
}

void main () {
  complex c= 3.14 + complex(1.0,-1.0);  // greška:

  // ne vrši se: complex(3.14).operator+(complex(1.0,-1.0))!
  // jer je pozvana funkcija clanica
}
```

- Sada poziv operatorske funkcije nije korektan, jer se ne vrši nikakva konverzija levog operanda u tip complex.

Globalne operatorske funkcije

- Za neke operatore potrebno je obezbediti da menjaju vrednost nekog od svojih operanada (npr. operatori dodele (=, += itd.)) menjaju vrednost svog levog operanda ovim je potrebno da njihov levi operand bude lvrednost.
- Ovakav koncept može se realizovati i pomoću globalne prijateljske funkcije.
- Da bi se promenio prvi argument, potrebno je da prvi formalni argument bude referenca na nekonstantni objekat (stvarni argument ne može biti privremeni objekat dobijen konverzijom, jer se tada referenca na nekonstantu ne može vezati na njega). Zato se ova funkcija može pozvati samo kada je levi operand baš objekat datog tipa.

```
complex& operator+= (complex &cl, complex cr) {
  cl.real+=cr.real; cl.imag+=cr.imag;   return cl;
}
void main () {
  complex c;  double d=3.14;
  d+=c;    // privremeni objekat complex(d) ne može inicijalizovati
           // referencu u pozivu operator+=(complex&,complex)
  c+=d;  } // u redu:  poziva se operator+=(c,    complex(d));
```

- Ako operator zahteva kao operand lvrednost onda je bolje koristiti operatorsku funkciju članicu (operacija se vrši upravo nad objektom koji predstavlja levi operand).

Operatorske funkcije

- Preporuka je da se operatori koji zahtevaju lvrednost kao levi operand (operatori dodele, inkrementiranja/dekrementiranja i slično), realizuju kao funkcije članice.

```
complex& complex::operator+= (complex cr) {
  real+=cr.real; imag+=cr.imag;
  return *this;
}
```

- Operatore koji imaju simetrično dejstvo nad operandima, treba realizovati kao globalne, najčešće prijateljske operatorske funkcije, kako bi se dozvolila implicitna konverzija operanada.
- Ako neka operatorska funkcija realizuje operaciju nad operandima koji su pripadnici dve klase, onda ona verovatno treba da bude globalna funkcija, prijatelj obe klase.
    - Time se izražava njena podjednaka "privrženost" obema klasama.
  - Takva funkcija može biti realizovana i kao članica jedne klase i istovremeno prijatelj druge klase.

Na primer,
klase koje realizuju matrice (klasa matrix) i vektore (klasa vector).

Globalna operatorska funkcija

- Operaciju množenja matrice vektorom najbolje je realizovati globalnom operatorskom funkcijom, koja je prijatelj obe ove klase, jer pristupa i elementima matrice, i elementima vektora:

```
class vector;

class matrix {
  friend vector operator* (const matrix&, const vector&);
  //...
};

class vector {
  friend vector operator* (const matrix&, const vector&);
  //...
};

vector operator* (const matrix &m, const vector &v) {
  //... množenje matrice vektorom
}
```

Unarni i binarni operatori

- Neki operatori jezika C++ mogu da budu i unarni i binarni (npr. unarni i binarni -, unarni &-adresa i binarni &-logičko I po bitovima, itd.)
- Prefiksni unarni operator može se deklarisati kao nestatička funkcija članica koja nema argumente, ili kao funkcija nečlanica koja ima jedan argument.
  - ako je @ unarni prefiksni operator, izraz @x znači poziv x.operator@(),
  - ako je globalna operatorska funkcija članica onda je poziv operator@(x)
- Binarni operator može se deklarisati kao nestatička funkcija članica koja ima jedan argument, ili kao funkcija nečlanica koja ima dva argumenta.
  - ako je @ binarni operator, izraz x@y
    - znači x.operator@(y), ako je operatorska funkcija članica,
    - znači poziv operator@(x,y), ako je funkcija globalna.

```
class complex {
  double real,imag;
public:
  complex (double r=0, double i=0) : real(r), imag(i) {}
  friend complex operator+(complex,complex);
  complex operator!()  // unarni operator!, npr. konjugovani broj
    { return complex(real,-imag); }
  //...
};
```

Operatori ++ i --

- Operatorska funkcija sa imenom operator++() definiše prefiksni operator inkrementiranja (++) za objekte neke klase.
- Operatorska funkcija sa imenom operator++(int /*dummy*/) definiše postfiksni operator inkrementiranja (++) za objekte neke klase.
    - za postfiksni operator ++ argument mora biti tipa int.

```
class X {
  public:
    X& operator++();    // prefiksni operator++;
    X  operator++(int); // postfiksni operator++;
};

void main () {
  X x;
  ++x;  // poziv x.operator++();
  x++;  // poziv x.operator++(0);
  x.operator++();  // eksplicitni poziv: kao ++x;
  x.operator++(0); // eksplicitni poziv: kao x++;
  x.operator++(3); // može i ovako, ali ne može: x++3;
}
```

- Prefiksni i postfiksni operator dekrementiranja (--) tretiraju se potpuno analogno.

Operator ()

- Poziv funkcije oblika: *izraz*(*lista_izraza*) smatra se binarnom operacijom. Operator je (), prvi operand je *izraz*, a drugi operand je *lista_izraza*, koja može biti prazna.
- Ova operatorska funkcija mora biti nestatička funkcija članica neke klase X. Ako je x objekat klase X, poziv oblika x(arg1,arg2,arg3) se tumači kao poziv operatorske funkcije članice: x.operator()(arg1,arg2,arg3).

```
class X {
  //...
public:
  //...
  int operator()(int,int) {/*...*/}
};
void main () {
  X x1,x2,x3;
  int i=x1(2,2);  // poziv: x1.operator()(2,2);
  i=x2(1,2);      // poziv: x2.operator()(1,2);
  i=x3(2,i);      // poziv: x3.operator()(2,i);
}
```

- U sledećem primeru klasa matrix koja omogućuje kreiranje dvodimenzionalne matrice realizovane dinamički korišćenjem pokazivača.

Operator ()

```
class matrix
{
    int row;
    int col;
    int **m;
  public:
    matrix(int r, int c): row(r), col(c)
    {
      m = new int*[row];
      for(int i=0;i<row;i++) m[i] = new int[col];
    }
    int& operator()(int r, int c){return m[r][c];} //vraca referencu
    int  JustGetValue( int r, int c ){return *(*(m+r)+c); } //mala igra
    ~matrix() // oslobađa dinamički alociranu memoriju
    {
      for(int i=0;i<row;i++) delete [] m[i];
      delete [] m;
    }
};

//napisati novu klasu MYmatrix u kojoj bi se u funkciji
//JustGetValue koristio return *(m+r*col+c);
```