Objektno-orijentirano programiranje

Objekti

- svijet se **sastoji od objekata**: stolova, stolica, računala, automobila, računa, utakmica...
- ljudi objekte oko sebe **klasificiraju** na način da naznačuju određena svojstva objekata
- primjeri:
  - psi i mačke su ...
  - nogomet i tenis su ...
- klasifikacija se može vršiti na (beskonačno) mnogo načina i razina
- objekte možemo slagati u hijerarhije

Objektno-orijentirani pristup

- kod objektno orijentiranog programiranja koristi se ideja stvaranja hijerarhija povezanih objekata, pa objektno-orijentirani pristup pojednostavljeno obuhvaća
  - **identificiranje** relevantnih **objekata**
  - njihovo **organiziranje u hijerarhije**
  - **dodavanje svojstava (atributa) objektima** - važnih za problemski kontekst
  - **dodavanje funkcija (metoda) - ponašanja objektima** - kako bi se na objektu izvršilo željeni zadatak
- objektno-orijentirano programiranje – korištenje objekata

Objektno-orijentirano programiranje - OOP

- **objektno-orijentirani model** programiranja nasuprot **proceduralno-strukturiranom** programiranju.

Proceduralno programiranje	**funkcije**	**podaci**
Objektno-orijentirano programiranje	**objekti**	

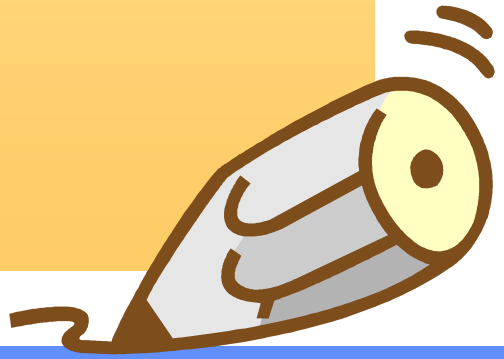
- o podacima razmišljamo preko operacija koje možemo obavljati nad njima, odnosno objekt se sastoji od **podataka** koji opisuju objekt i **operacija** (funkcija) koje na njemu mogu biti primijenjene

OOP

- prije sekvencijalno, danas programiranje **upravljano događajima (event-driven)**
- program je razbijen u **cjeline koje međusobno surađuju** u rješavanju problema
- umjesto sa procedurama koje barataju sa podacima, radimo sa objektima koji objedinjavaju operacije i podatke
- objektno-orijentirani programski jezici dizajnirani su za podršku **OO programskim metodama**
- ključno svojstvo OO programskih jezika: podržavaju prirodni proces identificiranja i klasificiranja objekata.

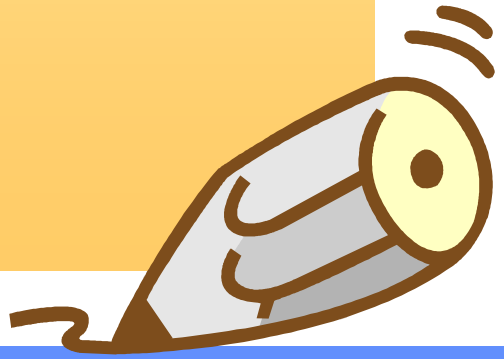
Ključna svojstva OO programskih jezika

- **encapsulation**
  - spajanje podataka i operacija
- **data hiding**
  - podaci objekta su privatni, dakle strukture podataka i detalji implementacije nekog objekta skriveni su od drugog objekta u sustavu
  - spriječava druge objekte da pristupaju detaljima koje ne moraju znati – veći programi su zato robustniji
- **inheritance**
  - nasljeđivanje svojstava
- **polymorphism**



Encapsulation

- povezivanje podataka i funkcija u objekte
- klasa – opisuje način na koji su povezani podaci i funkcije
- klasa pokazuje temeljna svojstva svojih objekata i istovremeno skriva detalje implementacije
- objekti međusobno mogu biti u interakciji samo preko javnih (public) atributa-svojstava i metoda objekta

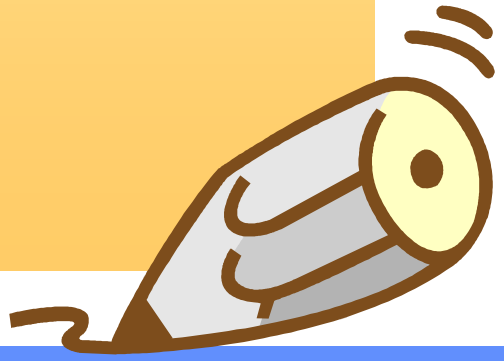


Inheritance

- omogućuje gradnju hijerarhija koje izražavaju međusobne relacije
- klase mogu nasljeđivati mogućnosti od klasa koje su više u hijerarhiji
- nove klase mogu nasljeđivati funkcionalnost postojećih klasa i prema potrebi modificirati ili proširiti tu funkcionalnost
- "roditeljska" klasa od koje se funkcionalnost nasljeđuje zove se **bazna**, a nova klasa poznata je kao **derivirana** klasa.

Polymorphism

- grčki: “mnogo oblika”
- prilično ga je teško definirati
- polimorfizam u osnovi znači da klase mogu imati isto ponašanje, ali ga implementirati na različite načine
  - ta je funkcionalnost korisna jer daje mogućnost rada sa generičkim tipovima objekata onda kada nas se ne tiče način na koji neka klasa implementira određenu funkcionalnost



Prednosti korištenja objekata i klasa

- jača razgraničenja među programskim segmentima

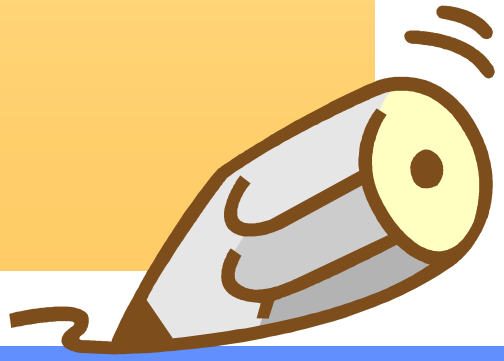
- jednostavniji tijek podataka
- olakšana koordinacija većeg broja programera, pa oni mogu raditi donekle nezavisno u razvoju programa i traženju grešaka po svom programskom segmentu
- temeljito testirane programske segmente lakše je spajati i kreirati velike pouzdane programe
- klase kreirane od strane nezavisnih autora mogu se ugraditi u novi program što smanjuje količinu programiranja koja se mora izvoditi sasvim od početka

`class`

klasa ili razred

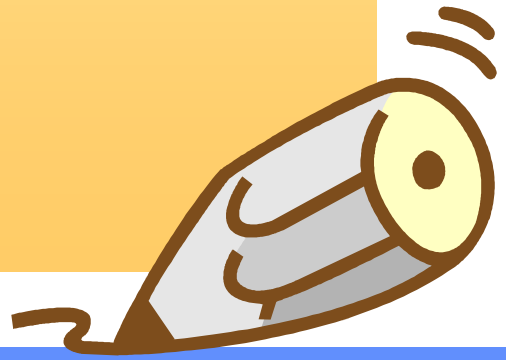
Razumijevanje koncepta objekata

- analogija: klasa – kalup, objekt – figurica
- kalup određuje veličinu i oblik, ali figurica može biti od različitog materijala, boje..., dakle prilično jedinstvena
- ovisno o vrijednostima podatkovnih članova (a time i različitim pozivima funkcijskih članova) objekti se u programu ponašaju različito
- **svaki podatkovni član mora biti karakteristika ili atribut klase, svaki funkcijski član treba izvoditi koristan zadatak namijenjen klasi**



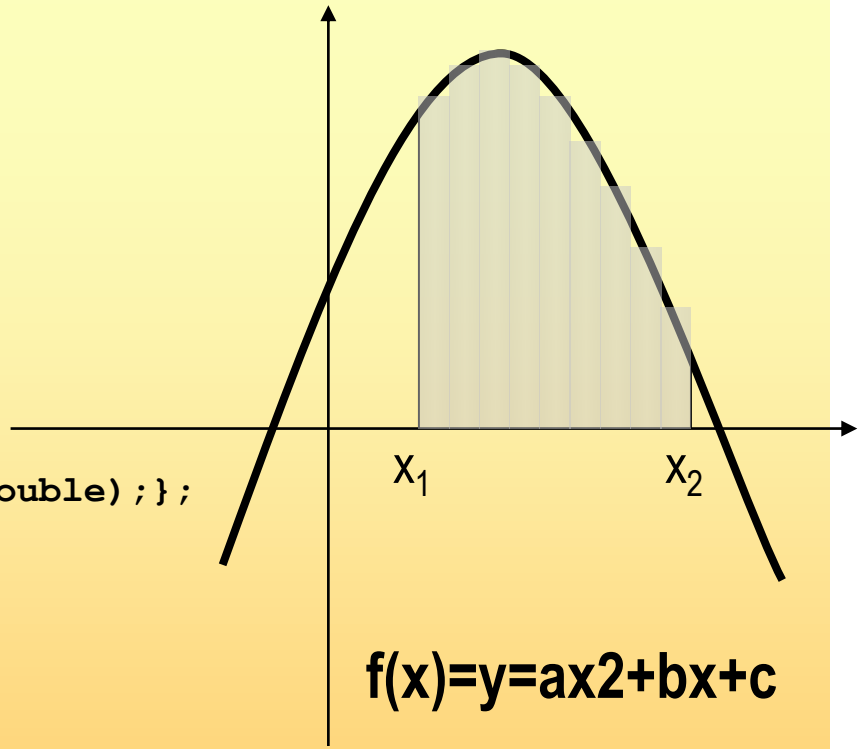
Usporedba između `struct` i `class`

- oba grupiraju podatke i funkcije (`class` se može pisati i umjesto `struct`)
- `struct` i `class` imaju isti oblik definiranja, operator (.) radi isto i imaju isti oblik deklaracije
- terminologija za `struct` i `class` je različita.
- ekvivalent `struct` **varijabli** u tipu `class` je **objekt**.



Primjer programa - površina ispod parabole

```
#include <iostream>
using namespace std;

class Parabola {
   private:
         double a, b, c;
   public:
         void citaj_koef ();
         double izr_povr (double,double);};

void Parabola::citaj_koef () {
   cout << "a, b i c: "<<endl;
   cin >> a >> b >> c;}

double Parabola::izr_povr (double x1, double x2) {
   double r;
   r=(a*x2*x2*x2/3+b*x2*x2/2+c*x2)-(a*x1*x1*x1/3+b*x1*x1/2+c*x1);
   return r;}
```



$f(x)=y=ax2+bx+c$

Primjer programa - površina ispod parabole

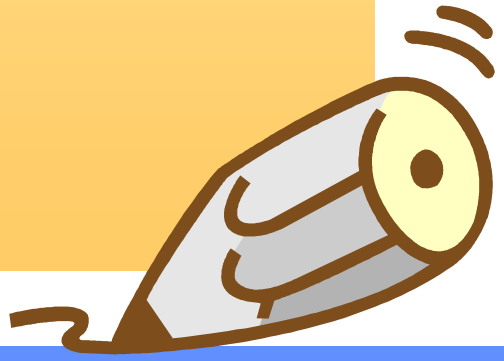
```
int main () {
   double lij_gr, des_gr, povrsina;
   Parabola p1, p2;

   p1.citaj_koef ();
   cout << "Granice integrala: "<<endl;
   cin >> lij_gr >> des_gr;
   povrsina=p1.izr_povr (lij_gr, des_gr);
   cout << "Povrsina je " << povrsina << endl << endl;

   p2.citaj_koef ();
   cout << "Granice integrala: "<<endl;
   cin >> lij_gr >> des_gr;
   povrsina=p2.izr_povr (lij_gr, des_gr);
   cout << "Povrsina je " << povrsina << endl << endl;

return 0;}
```

Vježbe

- **Dodajte klasi Parabola nove funkcijske članove**
  - za prikaz jednadžbe parabole
  - za izračun sjecišta parabole sa osima
  - za izračun vrijednosti parabole u zadanoj točki
  - …

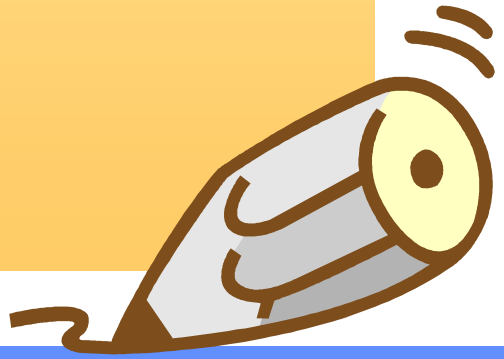
Smještaj u programu

- definicije klasa **obično se smještaju na početak programa izvan tijela bilo koje funkcije**
  - <tip povratne vrijednosti funkcije> razmak < ime klase>::(scope resolution operator) <zaglavlje funkcije (ime i lista argumenata)>
- memorija za svaki podatkovni član klase rezervira se kod deklaracije objekta iz klase (npr. p1 i p2 su objekti tipa parabola)



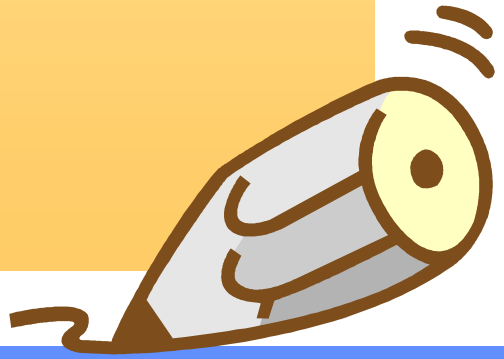
Osnove grupiranja podataka i funkcija - `class`

- definiranjem klase i deklariranjem objekta koji pripada toj klasi, određujemo koji tipovi informacija mogu biti spremljeni u području memorije rezerviranom za objekt
- klase imaju podatkovne i funkcijske članove koji su u jakoj relaciji
- funkcijski članovi koriste se za čuvanje i manipulaciju podatkovnim članovima
  - za npr. čitanje vrijednosti koeficijenata s tipkovnice, računanje površine, ispis koeficijenata...).
- **funkcijski članovi imaju zagrade**



Pristupi `private` i `public`

- `public` u definiciji klase označava da se članovima klase može pristupati iz bilo koje funkcije
- kod objektno orijentiranog programiranja obično ne želimo da svaka funkcija može pristupati podatkovnim članovima klase i ograničavamo pristup podatkovnim članovima korištenjem ključne riječi `private` u deklaraciji klase.
- `private` pristup znači da podatkovnim članovima klase **mogu pristupati samo funkcijski članovi te klase**

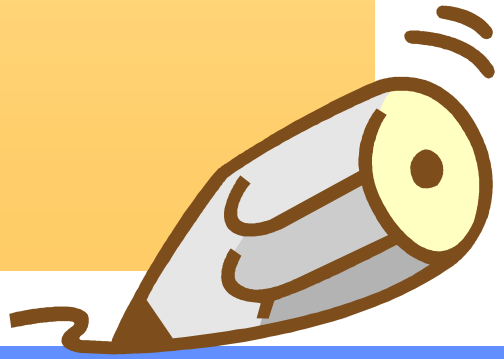


Pristup javnim (`public`) funkcijskim članovima

- svaki poziv funkcijskog člana mora biti pridružen objektu pa se poziv izvodi sa imenom (pozivanog) objekta:
  - ***ime_objekta.ime_funkcije***
- kad se funkcijski član poziva sa imenom objekta, podatkovni članovi objekta automatski se predaju funkciji (`by reference`) **bez da ih se navodi u listi argumenata**
- funkcija može koristiti svaki podatkovni član po imenu i direktno mijenjati vrijednosti podatkovnog člana objekta – **osiguran čist i uredan način rada sa podacima - temeljni koncept objektno orijentiranog programiranja**
- vrijednosti za podatke koji nisu članovi klase (npr. lokalne varijable funkcija koje nisu članovi) dostavljaju se (prenose) kroz listu argumenata pri pozivu funkcije

Vrste pristupa

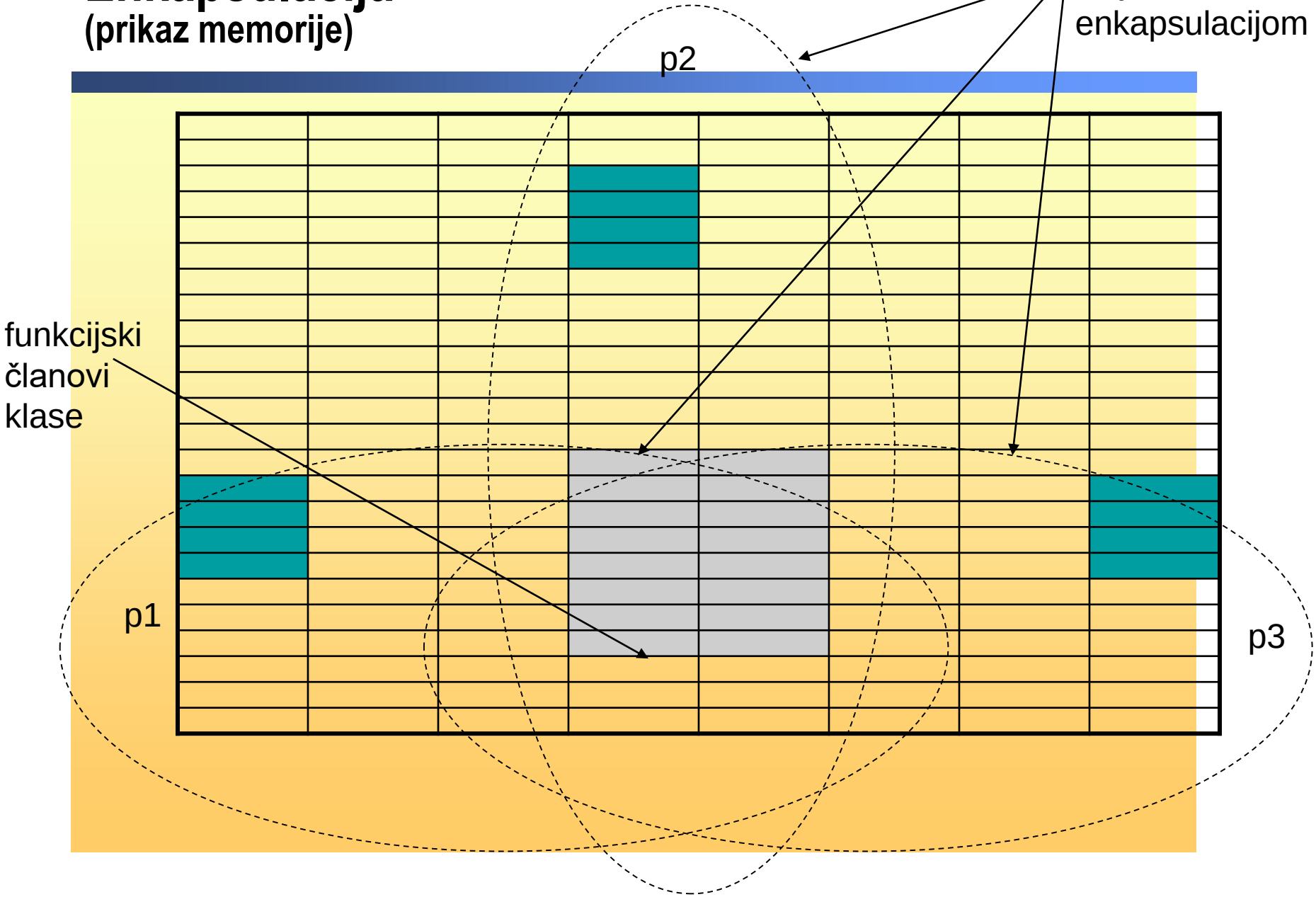
- **`private` i `public` oznake određuju pristupačnost deklariranih članova (ako se vrsta pristupa ne specificira, članovi su `private`)**
  - `private` (mogu im pristupiti samo funkcijski članovi, tipično za podatkovne članove*)
  - `public` (mogu ih pozivati sve funkcije, tipično za funkcijske članove*)
- ***osigurava enkapsulaciju**
- Specifikacija vrste pristupa se može koristiti više puta (naizmjenično)



Koncept enkapsulacije

- opisuje vezu između podataka i funkcija objekta
- rezultat enkapsulacije:
  - pristup podacima klase omogućen je samo funkcijskim članovima (*data hiding)*
- enkapsulacija spada u temelje objektno orijentiranog dizajna
- deklariran objekt neke klase smatra se instancom te klase (objekt stvaramo deklaracijom ili instanciranjem)
- Korištenje klasa olakšava izradu pouzdanog softvera

Enkapsulacija
(prikaz memorije)



Dizajniranje programa sa klasama i objektima

- Ispravno konstruirati klase i njihovo međudjelovanje
  - Klasa mora imati jasan cilj i biti relativno jednostavna (inače se cijepa) – npr. velika klasa može imati do 30-tak funkcijskih članova
  - Problem treba temeljito analizirati i odabrati odgovarajuće podatkovne i funkcijske članove
- Primjer:
  - imenice – klase
  - karakteristike imenica - podatkovni članovi
  - glagoli pridruženi imenicama - funkcijski članovi
- primjer množenja dviju matrica

Vježbe

- **Napisati podatkovne i neke funkcijske članove za klase:**
  - trokut
  - pravac
  - strujni krug
  - učenik

Primjeri

- Napraviti program koji će uz upotrebu klase izračunati opsege dva pravokutnika, pa ispisati koji pravokutnik ima veći opseg. Duljine stranica su određene realnim brojevima. Ispisati da li se manji pravokutnik može prekriti većim pravokutnikom.

- Napraviti program koji će uz upotrebu klase izračunati površinu trokuta zadanih stranica čije su duljine realni brojevi. Ispisati za oba trokuta površinu. Ispisati koliko puta drugi trokut ima veću/manju površinu od prvoga.

Konstruktori

Primjer

```
#include <iostream>
using namespace std;

class Ventilator {
   private:
          int brzina_vrtnje;
   public:
          Ventilator();//konstruktor - nema tip povratne vrijednosti, ime=ime klase
          void pisi_brzinu_vrtnje(); };

Ventilator::Ventilator() {//dva puta sadrži ime klase (ime klase i ime funkcije)
          brzina_vrtnje = 4;
          cout << " Konstruktor izveden" << endl;}

void Ventilator::pisi_brzinu_vrtnje() {
          cout << "Brzina vrtnje = " << brzina vrtnje << endl;}

int main () {
   Ventilator ventilator1;    //rezervira memoriju za objekt i poziva konstruktor
   ventilator1.pisi_brzinu_vrtnje();
   return 0; }
```

Konstruktorska funkcija – bez argumenata

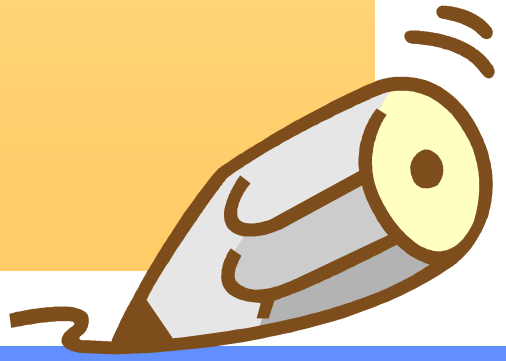
- konstruktorska funkcija (konstruktor) je funkcijski član klase koji se **izvodi automatski kod deklaracije objekta**
- **često se koristi za** inicijalizaciju vrijednosti **podatkovnih članova (inicijalizacija bez eksplicitnog poziva)**
- karakteristike **konstruktora:**
  - ime funkcije jednako je imenu klase
  - ne vraća vrijednost (ali ne piše se *void*)
  - ima *public* pristup, dakle objekt može biti deklariran u bilo kojoj funkciji
  - može i ne mora imati argument
- **može izvoditi gotovo sve operacije, ali tipično inicijalizira podatkovne članove i izvodi druge zadatke pridružene** stvaranju **objekta;**
- **kod polja objekata konstruktor se poziva za svaki član polja**

Primjer

```
#include <iostream>
using namespace std;

class Meteo_stanica {
   private: int brzina_vjetra;
   public:
          Meteo_stanica (int);//argument odgovara tipu podatkovnog člana
          void pisi_brzinu_vjetra (); };

Meteo_stanica  ::Meteo_stanica (int br_vj)//inicij. podatkovnog člana u konstr.
               :brzina_vjetra (br_vj) {//inicijalizacijska lista
          cout << "Konstruktor izveden" << endl; }//ne radi inicijalizaciju

void Meteo_stanica::pisi_brzinu_vjetra () {
          cout << "brzina vjetra = " << brzina_vjetra << endl; }

int main() {
   Meteo_stanica stanica1 (5), stanica2 (20);
   stanica1.pisi_brzinu_vjetra (); stanica2.pisi_brzinu_vjetra ();
   return 0; }
```

Konstruktorska funkcija – sa argumentima

- konstruktor može prihvatiti vrijednost kroz svoju listu argumenata i pridružiti takvu vrijednost podatkovnom članu objekta iz pripadajuće klase
  - kako konstruktor zovemo preko deklaracije, deklaracija mora predati početne vrijednosti
- **podatkovne članove može se inicijalizirati u tijelu konstruktora ili u inicijalizacijskoj listi koja je izvan tijela konstruktora**



Inicijalizacija većeg broja podatkovnih članova

```
class Meteo_stanica {
      private:
              double brzina_vjetra, temperatura;
      public:
              Meteo_stanica (double, double);
              void pisi_podatke (); };

Meteo_stanica::Meteo_stanica (double br_vj, double te)
               : brzina_vjetra (br_vj), temperatura (te) {
cout << "Konstruktor izveden" << endl; }

…
Meteo_stanica stanica1 (5, 27);   //zagrade iza imena objekta
…
```

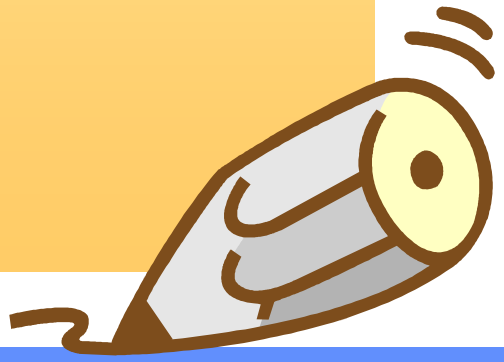
Ovaj oblik inicijalizacije preporuča se uvijek kada se podatkovnom članu pridružuje jednostavna vrijednost, inače se koristi primjer 2.

Poziv konstruktora u naredbi pridruživanja
(eksplicitni poziv konstruktora)

- Ako je objekt deklariran i konstruktor pozvan, možemo pozvati konstruktor drugi put za taj objekt upotrebom naredbe pridruživanja
  - `stanica1 = Meteo_stanica (10);`

- **Kreiran je bezimeni objekt i vrijednosti njegovih podatkovnih članova pridruženi su objektu stanica1.**

- Ovaj se oblik može koristiti i za prvo deklariranje objekta
  - `Meteo_stanica stanica1 = Meteo_stanica (5), stanica2 = Meteo_stanica (27);`

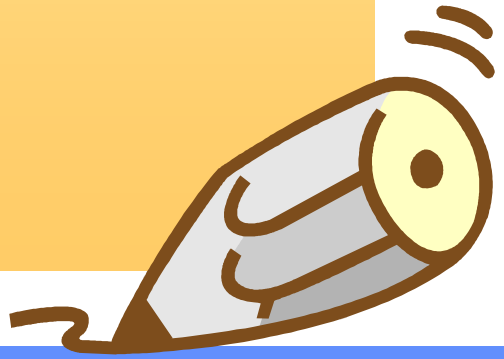


Preopterećenje konstruktorske funkcije
Podrazumijevani copy konstruktor

- C++ dozvoljava **preopterećenje konstruktorske funkcije**, što znači da možemo imati dvije ili više konstruktorskih funkcija u definiciji klase – dizajnirane tako da udovoljavaju različitim objektima

- C++ dozvoljava podrazumijevane argumente za konstruktorske funkcije, ali bolje je koristiti samo funkcijsko preopterećenje

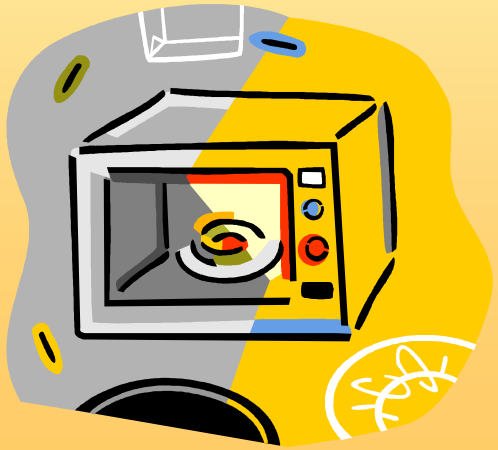
- C++ ima ugrađeni copy konstruktor nazvan ***default copy constructor*** **– automatski se izvodi kad je jedan objekt deklariran da bude identičan drugom objektu**



Primjer

```
class MiVal_naredba {
        private:
                int vrijeme, snaga;
        public:
                MiVal_naredba ();//tri konstruktora
                MiVal_naredba (int);
                MiVal_naredba (int, int);
                void pokazi_podatke (); };

MiVal_naredba :: MiVal_naredba ()//podrazumijevani konstruktor
                 : vrijeme (60), snaga (10) {
        cout << "izveden konstruktor bez argumenata" << endl;}

MiVal_naredba :: MiVal_naredba (int vr)
                : vrijeme (vr), snaga (10)  {
        cout << "izveden konstruktor s jednim argumentom" << endl;}

MiVal_naredba :: MiVal_naredba (int v, int s)
                 : vrijeme (v), snaga (s)  {
        cout << "izveden konstruktor s dva argumenta" << endl;}

void MiVal_naredba :: pokazi_podatke () {
        cout << "vrijeme = " << vrijeme << "snaga = " << snaga << endl; }
```

```
int main () {

MiVal_naredba program1, program2 (30), program3 (45,6),
program4=program3;              //poziva default copy
program1. pokazi_podatke ();
program2. pokazi_podatke ();
program3. pokazi_podatke ();
program4. pokazi_podatke ();
return 0;}
```

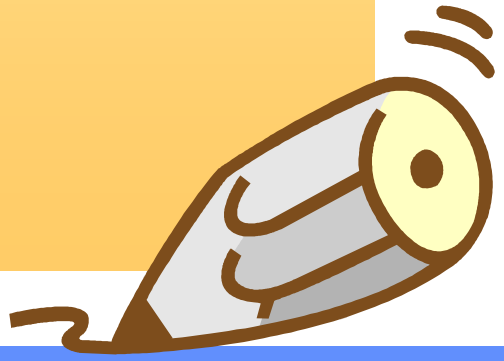


Upotreba

- Objekt koji poziva konstruktor bez argumenata (*program1*) nema zagrade, ali ako se konstruktor bez argumenata poziva eksplicitno, prazne zagrade moraju biti navedene, npr.
  - `program1 = MiVal_naredba ();`

  kreira bezimeni objekt upotrebom konstruktora bez argumenata i pridružuje ga objektu *program1*.
- Ako konstruktor bez argumenata nije definiran, ali drugi konstruktori jesu, uz objekte kod deklaracije moraju biti argumenti

- Preporuka je kreirati konstruktor bez argumenata (*postaje podrazumijevani konstruktor*)



Podrazumijevani copy konstruktor

- koristi se za inicijalizaciju objekata sa drugim objektima (kopiraju se sve vrijednosti podatkovnih članova)
  - `MiVal_naredba program4 = program3;`

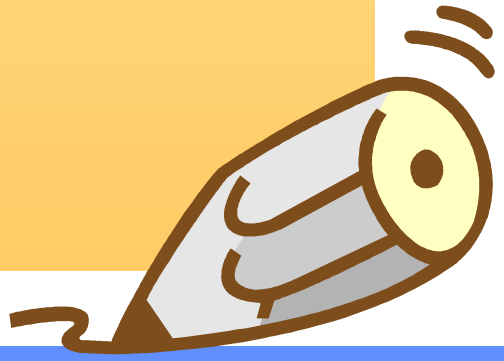
ili

  - `MiVal_naredba program4 (program3);`

Kod obje deklaracije ne poziva se niti jedna konstruktorska funkcija koju bi definirao programer

- C++ prevoditelj prepoznaje da se jedan objekt inicijalizira drugim i poziva njegov vlastiti konstruktor

- **Radi dobro za klase sa ugrađenim podatkovnim tipovima (ne za pokazivače ili apstraktne)**



Preopterećenje operatora

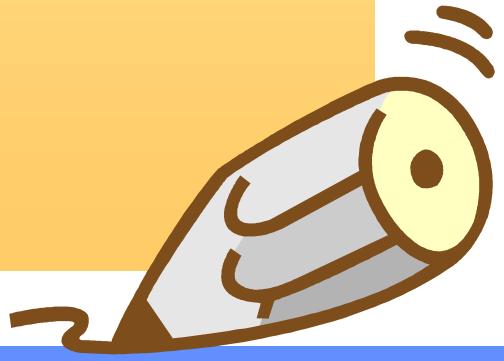
Operator overloading

Preopterećenje operatora

- kreiranje novih definicija operatora (za upotrebu sa objektima ili različitim vrstama podataka)
- odgovara stvaranju nove funkcije za neku klasu npr. operator+() (*zbrajanje*)
- koristi se ključna riječ *operator* (ime funkcije je *npr. operator+*)
- unutar tijela funkcije pišemo kôd za izvođenje zbrajanja
- kada npr. neka druga funkcija zbraja objekte te klase, automatski se poziva odgovarajuća funkcija operator+

Mogućnosti

- Najbolje je da **operatorske funkcije izvode akcije slične značenju originalnog operatora** (iako u nekim slučajevima postoje različite mogućnosti, npr. oduzimanje stringova)
- Preopterećenje se može izvesti za unarne i za binarne operatore (i oni moraju ostati takvi)
- Kod korištenja preopterećenja operatora u klasama, klase postaju sličnije generičkim tipovima podataka jer operatori definiraju akcije na objektima iz klase (takvu klasu obično zovemo abstract data type (ADT))

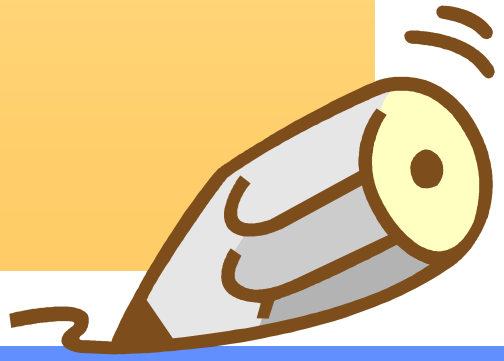


Primjer programa

```
#include <iostream>
using namespace std;

class Tocka {
        private:
                double x, y;
        public:
                Tocka ();
                Tocka (double,double);
                Tocka operator++ ();
                Tocka operator= (const Tocka&);
                void prikazi_podatke (); };

Tocka :: Tocka ()
         : x(0), y(0) { }

Tocka :: Tocka (double x_var, double y_var)
          : x (x_var), y (y_var) {}

Tocka Tocka :: operator++ () {
        ++x;
        ++y;
        return *this;}

Tocka Tocka :: operator= (const Tocka& tockab) {
        x = tockab.x;
        y = tockab.y;
        cout << "Pozvana operatorska funkcija pridruzivanja." << endl;
        return *this;
}

void Tocka :: prikazi_podatke () {
        cout << "x = "<< x <<" y = " << y << endl;
}

int main () {
        Tocka p1 (10, 3), p2 (4, 8), p3;
        p3 = p1;
        ++p2;
        p3.prikazi_podatke ();
        p2.prikazi_podatke ();
        return 0;
}
```

Opis programa

- definirani su konstruktor bez argumenata (inicijalizira x i y na 0) i konstruktor sa dva argumenta (inicijalizira x i y na vrijednosti argumenata)
- funkcije `operator++` i `operator=` su funkcijski članovi koji vraćaju objekt iz klase `Tocka`
- funkciji `operator=` objekt se predaje "*by reference*"
- kako je `operator++` funkcijski član, x i y se odnose na podatkovne članove objekta (funkcija povećava vrijednosti podatkovnih članova objekta)

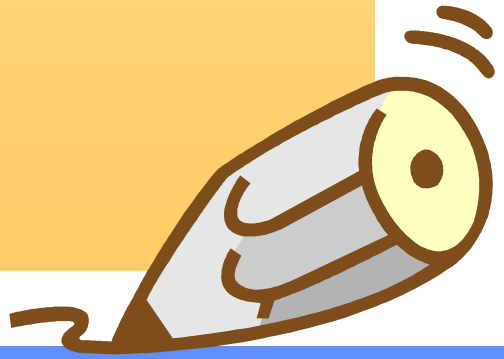


Opis programa - nastavak

- ključna riječ "`this`" predstavlja pokazivač na objekt (ili adresu objekta) koji poziva
- kao posljedica, sam objekt je predstavljen sa `*this`. Zato, funkcija `operator++ ()` vraća izmijenjeni objekt (sa povećanim vrijednostima vlastitih podatkovnih članova)
- kako je `operator=` funkcijski član, x i y se odnose na podatkovne članove objekta
- `tockab.x` i `tockab.y` odnose se na podatkovne članove objekta u listi argumenata pa ih funkcija `operator=` pridružuje podatkovnim članovima objekta

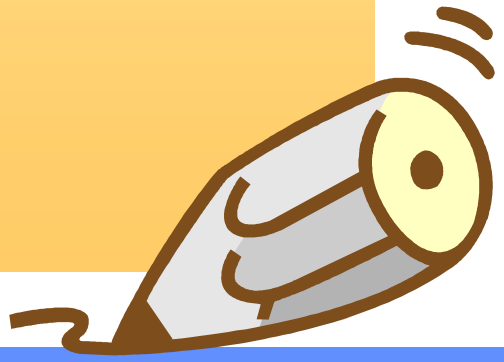
Binarne operatorske funkcije

- Opći oblik izraza i ekvivalentni funkcijski poziv su
  - objekt1 = objekt2
  - objekt1.operator= (objekt2);
- objekt1 i objekt2 su objekti iz klase, a na mjestu = može stajati bilo koji binarni operator (+,-,*,/)
- objekt prije operatora u izrazu je objekt koji poziva, a objekt poslije operatora predaje se kroz listu argumenata



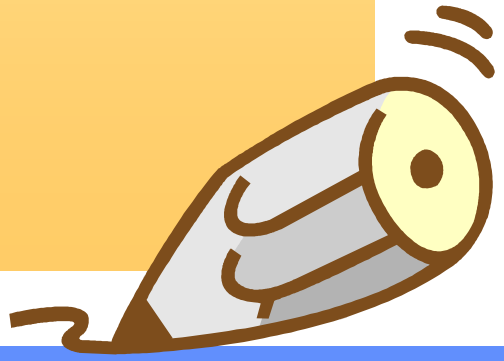
Pisanje definicija funkcija

- odgovornost programera
- funkcija mora biti usklađena sa odgovarajućim C++ funkcijskim pozivom (C++ određuje koji je objekt prvi u listi argumenata, a koji je objekt koji poziva)



Ključna riječ `this`

- podatkovni članovi objekta koji poziva operator izmijenjeni su akcijama te operatorske funkcije i mi trebamo vratiti taj objekt
- C++ osigurava posebnu ključnu riječ da izvede tu akciju (`this`)
- `this` označava pokazivač na (*adresu od*) objekt, a `*this` označava sam objekt
- kad vraćamo `*this`, funkcija vraća izmijenjeni objekt.
- vraćanje `*this` uobičajeno je za operatorske funkcije koje mijenjaju pozivajući objekt

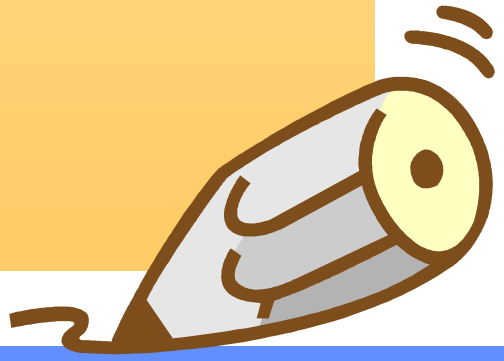


Pravila za preopterećenje operatora

- Operatori koji se mogu koristiti u definiranju operatorskih funkcija:
  - ++ -- + - * / % = += -= *=
  - /= %= ! < > <= >= == != >>
  - << && & || new delete....
- Operatori koji se ne mogu koristiti u definiranju operatorskih funkcija:
  - sizeof :: . itd.
- Nije dozvoljeno raditi svoje vlastite operatore
- Ne možemo mijenjati klasifikaciju operatora (unarni, binarni)
- Prioriteti operatora ostaju isti

Primjeri:

- zbrajanje vektora, primjer koji uključuje operator +
- zbrajanje dvodimenzionalnog vektora a(3,1) i b(1,2) daje (3+1,1+2) = (4,3)
- Kreirati opterećene operatore za zbrajanje, oduzimanje, množenje i dijeljenje razlomaka



Primjer preopterećenja operatora: zbrajanje vektora

```
#include <iostream>
using namespace std;

class CVector
{ public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator+ (CVector); };

CVector::CVector (int a, int b)
    { x = a; y = b; }

CVector CVector::operator+ (CVector vek)
        { CVector v;
        v.x = x + vek.x;
        v.y = y + vek.y;
        return v; }

int main () {
        CVector a (3,1);
        CVector b (1,2);
        CVector c;
        c = a + b;
        cout << c.x << "," << c.y;
return 0; }
```

```
#include <iostream>
using namespace std;
class threenums {
    private: int a, b, c;
    public:  threenums() {a=b=c=0;}
             threenums(int x, int y, int z) {a=x; b=y; c=z;}
             threenums operator+ (threenums o2);
             threenums operator= (threenums o2);
             void display(); };
threenums threenums::operator+(threenums o2) {
    threenums num;
    num.a = a + o2.a; num.b = b + o2.b; num.c = c + o2.c;
    return num; }
threenums threenums::operator=(threenums o2) {
    a = o2.a; b = o2.b;  c = o2.c;  return *this; }
void threenums::display() {
             cout << a << ", ";
             cout << b << ", ";
             cout << c << '\n'; }
int main() {
             threenums x(5, 10, 15),
y(10, 15, 20), z;
             x.display(); y.display();
             z = x + y;
             z.display();
             z = x + y + z;
             z.display();
             z = x = y;
             x.display();
             y.display();
             z.display();
return 0; }
```

```
#include <iostream>
using namespace std;
class threenums {
        private:
                int a, b, c;
        public:
                threenums() {a=b=c=0;}
        threenums(int x, int y, int z) {a=x; b=y; c=z;}
        threenums operator+(threenums o2);
        threenums operator=(threenums o2);
        threenums operator++();
        void display(); };
threenums threenums::operator+(threenums o2) {
        threenums num;
        num.a = a + o2.a;
        num.b = b + o2.b;
        num.c = c + o2.c;
        return num; }
threenums threenums::operator=(threenums o2) {
        a = o2.a; b = o2.b; c = o2.c; return *this; }
threenums threenums::operator++() {
         a++; b++; c++; return *this; }
void threenums::display() {
        cout << a << ", "; cout << b << ", "; cout << c << '\n'; }
int main() {
        threenums x(5, 10, 15), y(10, 15, 20), z;
        x.display(); y.display();
        z = x + y;
        z.display();
        z = x + y + z;
        z.display();
        z = x = y;
        x.display(); y.display(); z.display();
        ++z;
```

```
#include <iostream>
        using namespace std;

        class Fraction {
        public:
           Fraction(int num = 0, int den = 1)
           {
              this->num = num;
              this->den = den;
           }
           Fraction operator+(const Fraction &rhs)
           {
              Fraction temp;
              temp.den = this->den * rhs.den;
              temp.num = rhs.den * this->num +
                 this->den * rhs.num;
              return temp;
           }
           Fraction operator+(const int i)
           {
              Fraction temp;
              temp.den = this->den;
              temp.num = this->num +
                 this->den * i;
              return temp;
           }
           void print()
           {
              cout << num << "/" << den << endl;
           }
        private:
           int num;
           int den;
        };

        int main() {
           Fraction a(1,2);    // a = 1/2
           int b = 5;
           Fraction c;

           c = a + b;
           c.print();

           return 0;
        }
```

Predaja objekata funkciji i vraćanje objekata

Procedure to Pass Object to Function

```
..... .... .....
class class_name
 {
    .... ... ....
  return_type function_name(class_name para1, class_name para2)
   {
     ..... ... .....
   }
   ..... ..... .......
 }

main()
{
 class_name obj1, obj2, obj3;

  obj1.function_name(obj2,obj3 )
  ..... ... ....
}
```

Figure: Passing Object to Function

Primjer za predaju

```
#include <iostream>
using namespace std;
class Complex
{
    private:
       int real;
       int imag;
    public:
       Complex(): real(0), imag(0) { }
       void Read()
        {
           cout<<"Enter real and imaginary number respectively:"<<endl;
           cin>>real>>imag;
        }
        void Add(Complex comp1,Complex comp2)
        {
            real=comp1.real+comp2.real;
 /* Here, real represents the real data of object c3 because this function is called using code
c3.add(c1,c2); */
            imag=comp1.imag+comp2.imag;
 /* Here, imag represents the imag data of object c3 because this function is called using code
c3.add(c1,c2); */
        }
        void Display()
        {
            cout<<"Sum="<<real<<"+"<<imag<<"i";
        }
};
int main()
```

Returning Object from Function

The syntax and procedure to return object is similar to that of returning structure from function.

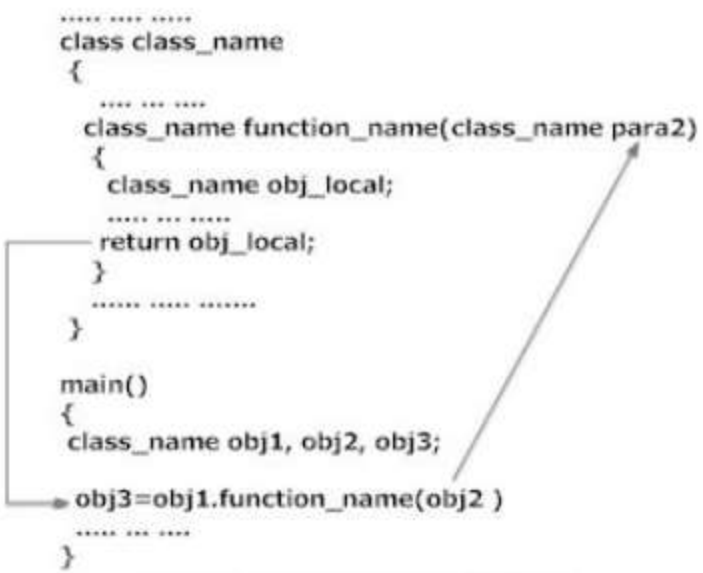


Figure: Returning Object from Function

Primjer za vraćanje

```
#include <iostream>
using namespace std;
class Complex
{
    private:
       int real;
       int imag;
    public:
       Complex(): real(0), imag(0) { }
       void Read()
        {
          cout<<"Enter real and imaginary number respectively:"<<endl;
          cin>>real>>imag;
        }
        Complex Add(Complex comp2)
        {
            Complex temp;
            temp.real=real+comp2.real;
/* Here, real represents the real data of object c1 because this function is called using code c1.Add(c2) */
            temp.imag=imag+comp2.imag;
/* Here, imag represents the imag data of object c1 because this function is called using code c1.Add(c2) */
            return temp;
        }
        void Display()
        {
            cout<<"Sum="<<real<<"+"<<imag<<"i";
        }
};
int main()
{
    Complex c1,c2,c3;
    c1.Read();
    c2.Read();
```