

# Strukture Klase i OOP

# Strukture (tip struct)

# Podatkovni tip `struct`

- **Derivirani - izvedeni podatkovni tip** – stvara ga programer
- grupa logički povezanih podataka **različitih** podatkovnih tipova (npr. `char`, `int`, `double`) sa zajedničkim imenom
- podaci (članovi strukture) su međusobno povezani identifikatorom – imenom strukture; deklaracijom članova strukture određuje se ime i tip pojedinih članova strukture
- ako je definiran izvan tijela funkcije `main()` – sve funkcije programa mogu koristiti taj tip
- između imena strukturne varijable i imena člana te strukture stavlja se operator `'.'`



- struktura je korisnički definiran složeni tip
- sastavljen je od polja ili članova koji mogu biti različitih tipova
- u C++, struktura je ista kao i klasa osim što su njeni članovi podrazumijevano javni
- varijable se kod definicije strukturnog tipa može deklarirati nizanjem jednog ili više imena varijabli odvojenih zarezima između zatvorene zagrade i ;

# Strukture - primjeri



```
struct OSOBA                // deklaracija strukture OSOBA
{int starost;               // deklaracije tipova članova
float tezina;
char ime[25]; } clan_obitelji; // definira objekt tipa OSOBA

OSOBA brat;
OSOBA sestra;               // C++ stil deklaracije strukture
sestra.starost = 13;
brat.starost = 7;           // pridružuje vrijednosti članovima

struct TOCKA                // deklaracija strukture TOCKA
{ int x;                    // definira članove x i y
  int y; } mjesto = { 20, 40 }; // varijabla mjesto ima vrijednosti x = 20, y = 40

TOCKA tamo;                 // varijabla tamo je tipa TOCKA
```

- **Deklarirati strukturu “sportski\_klub” koja će moći pamtit i podatke za neki sportski klub, i to:**
  - Naziv kluba (polje znakova ili string)
  - Grad iz kojeg je klub
  - Ime i prezime trenera
  - Broj igrača
  - Broj postignutih pogodaka
  - Broj primljenih pogodaka
  - Da li je član prve lige
  - Iznos godišnjeg “proračuna”
- **Deklarirati dvije varijable tog tipa i pridružiti im vrijednosti**

```
struct KLUB
```

```
{string NAZIV_KLUBA;  
  string GRAD;  
  char ime[100];  
  int BROJ_IGRACA;  
  int BROJ_POSTIGNUTIH_GOLOVA;  
  int BROJ_PRIMLJENIH_GOLOVA;  
  bool PRVA_LIGA;  
  float GODISNJI_PRORACUN;  
} klub1, klub2;
```

```
KLUB klubx;
```

```
klub1.NAZIV_KLUBA="PIPO IPC";  
klub1.GRAD="ČAKOVEC";  
klub1.PRVA_LIGA=true;
```

```
klub2.NAZIV_KLUBA="CROATIA ZAGREB";  
klub2.GRAD="ZAGREB";
```



# struct – definiranje, deklariranje, upotreba varijabli

```
struct Tekucina
{
    public:                                (ili private)
        char oznaka;
        double temp, gustoca;
};

int main()
{
    Tekucina voda, ulje, mineralna;
    voda.oznaka = 'V';
    voda.temp = 28.6;
    voda.gustoca = 9.81;

    mineralna = voda;

    cout << "voda: " << voda.oznaka << endl; ...}
```



# Deklariranje `struct` tipa i `struct` varijabli



- deklaracijom određujemo koji će se tipovi informacija moći spremiti u memoriju rezerviranu za **strukturnu varijablu** tog tipa (za sve podatkovne članove te strukture)
- strukturna varijabla je složena varijabla koja sadrži niz vrijednosti istog ili različitog tipa
- deklaracijom **strukture** određuje se **oblik strukture** i ne zauzima se memorijski prostor; deklaracijom **strukturne varijable rezervira se memorijski prostor**
- za spremanje vrijednosti podatkovnih (public) članova pišemo ime `struct` varijable (objekta), točku (strukturni operator) i ime podatkovnog člana



# Pridruživanje i ispis vrijednosti podatkovnih članova

- vrijednosti **svih članova** jedne `struct` varijable možemo pridružiti članovima druge istovrsne `struct` varijable jednom naredbom pridruživanja
- za ispis vrijednosti **pojedinih** podatkovnih članova `struct` varijable (kao i za sve ostale operacije nad strukturama) moramo navoditi **svaki član pojedinačno** upotrebom operatora `(.)`, npr.:

- `cout << "voda: " << voda.oznaka << endl;`
- `cout << "temperatura: " << voda.temp << endl;`
- `cout << "gustoca: " << voda.gustoca << endl;`



# Usporedba struktura i razreda

- u C++ strukture imaju ista svojstva kao razred
- način deklaracije je isti s tom razlikom što se koristi ključna riječ `struct`
- struktura može sadržavati podatkovne i funkcijske članove, konstruktore, destruktore i ključne riječi za dodjelu prava pristupa
- razlika:
  - ako se ne navede pravo pristupa, članovi imaju podrazumijevani javni pristup
  - strukture ne omogućuju nasljeđivanje
- nakon što se usvoji koncept razreda, strukture se obično izbacuju iz upotrebe



- Napraviti program koji će uz upotrebu strukture izračunati opsege dva pravokutnika, pa ispisati koji pravokutnik ima veći opseg. Duljine stranica su određene realnim brojevima. (Pravokutnike se može označiti slovima p, odnosno r.)
- **Napraviti program koji će uz upotrebu strukture izračunati površinu dva trokuta zadanih stranica čije su duljine realni brojevi. Ispisati za oba trokuta površinu. Ispisati koliko puta drugi trokut ima veću/manju površinu od prvoga.**

- Napravite program koji (uz upotrebu strukture Kompleksni) omogućuje učitavanje dvaju kompleksnih brojeva, pa računa i ispisuje njihov zbroj, razliku, produkt i kvocijent.
- Napravite program koji će izračunati udaljenost točke T od ishodišta koordinatnog sustava. Za zadavanje točke iskoristite strukturu.
- **Napravite program koji će učitati dvije točke u koordinatnom sustavu. Program treba izračunati opseg i površinu pravokutnika kojemu su učitane točke vrhovi dijagonale. Koristite se strukturama.**

- **Napravite program koji će omogućiti unošenje podataka (ime, prezime, matični, broj, prosjek, datum rođenja(struktura)) za sve učenike jednog razreda. Program treba ispisati koji učenik ima najbolji prosjek. Uputa: Učenike smjestite u polje.**
- **Napravite program koji će iz datoteke učitavati imena i prezimena učenika te njihove ocjene iz pismenog ispita. Program treba ispisati koliko učenika je dobilo odličan, koliko vrlo dobar, dobar, dovoljan i nedovoljan te kolika je prosječna ocjena razreda iz pismenog. Ispisati imena i prezimena učenika koji su dobili odličan.**

# 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, npr.: psi i mačke su..., nogomet i tenis su ...
- klasifikacija se može vršiti na (beskonačno) mnogo načina i razina
- objekte možemo slagati u hijerarhije





- 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



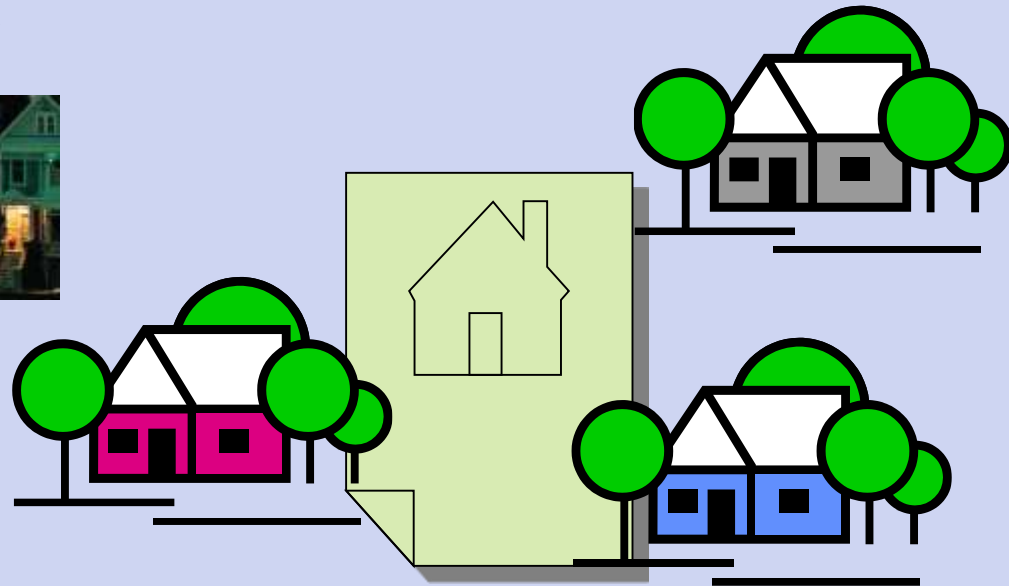
# Prednosti korištenja objekata i klasa



- 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



- osnovna ideja: učiniti kod jednostavnijim za pisanje i izvedbu
- stvaraju se objekti visoke razine koji se brinu o specifičnim zadacima
- temeljno je da dobar objektno-orijentirani pristup teži oponašati okoline i sustave realnog svijeta koji se često mogu koristiti i u složenijim sustavima



# Razred (klasa), objekt

## ■ Razred (class)

- kod koji se piše da bi se koristio kao nacrt za stvaranje objekata
- opisuje karakteristike objekta (koje vrste atributa ima, kako ga se može upitati da čini razne stvari i kako odgovara na događaje.)

## ■ Objekt (object)

- instanca klase
- općenito se stvara pozivanjem konstruktora klase

## ■ Metode (method)

- funkcije definirane u razredu
- opisuju akciju koja se može zadati objektu da ju učini.

## ■ Svojstva ili atributi (property)

- varijable definirane u razredu
- tipično opisuju kvalitete (stanje) objekta

## ■ Događaji (event)

- metode u objektu koje izaziva neka vanjska akcija
- mogu se pridružiti akciji korisnika (pritisak tipke) ili sustava (npr. proteklo vrijeme)

# Konstruktor, destruktor

## ■ Konstruktor

- specijalna metoda koja se poziva kod stvaranja objekta
- u C# upotrebom ključne riječi „new“ koju slijedi ime razreda

## ■ Destruktor

- specijalna metoda koja se poziva kod uništavanja objekta
- u C#, da bi se stvorio destruktore treba “pregaziti” metodu Dispose bazne klase
- zbog automatskog „skupljanja smeća“ u CLR-u, eksplicitno pozivanje destruktora je rijetko potrebno

## ■ Overriding

- objektno-orijentirani koncept koji definira da izvedena klasa može stvoriti različitu implementaciju metode bazne klase – u potpunosti „gazi“ preko ponašanja bazne klase

## ■ Sučelje

- dogovor koji definira strukturu metoda, svojstava, događaja (i indeksiranja)
- ne može se stvoriti objekt direktno iz sučelja
- prvo treba stvoriti razred koji implementira mogućnosti sučelja

## ■ Overloading (preopterećenje)

- objektno-orijentirani koncept koji postavlja da neka metoda može imati više različitih sučelja uz zadržavanje istog imena

## ■ Nasljeđivanje

- objektno-orijentirani koncept koji definira da jedan razred može biti izveden iz drugog ili drugih (bazne klase) i nasljeđuje njihovo sučelje i kod (derivirana ili klasa dijete)

## ■ Enkapsulacija

- koncept skupljanja metoda, svojstava, događaja i atributa u povezanu klasu i micanja detalja od korisnika (npr. auto)

## ■ Polimorfizam

- objektno-orijentirani koncept koji kaže da različiti objekti mogu imati različite implementacije iste funkcije (npr. metoda Zbroji).

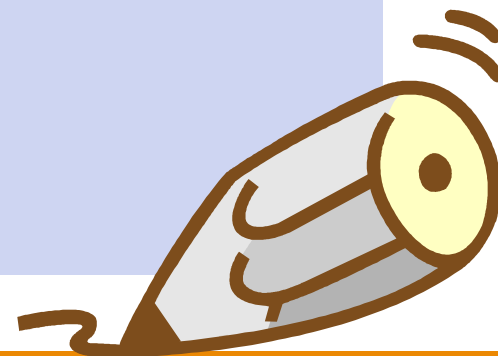


# OOP - razredi

**Primjer**

# Razumijevanje koncepta objekata

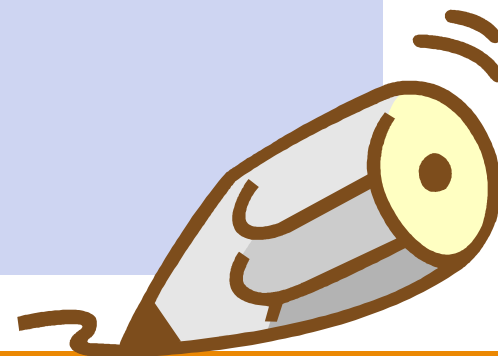
- analogija: klasa – kalup, objekt – figurica
- kalup određuje veličinu i oblik, ali figurica može biti od različitog materijala, boje..., dakle prilično jedinstvena
- ovisno o vrijednostima podatkovnih članova (a time i različitim pozivima funkcijskih članova) objekti se u programu ponašaju različito
- **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

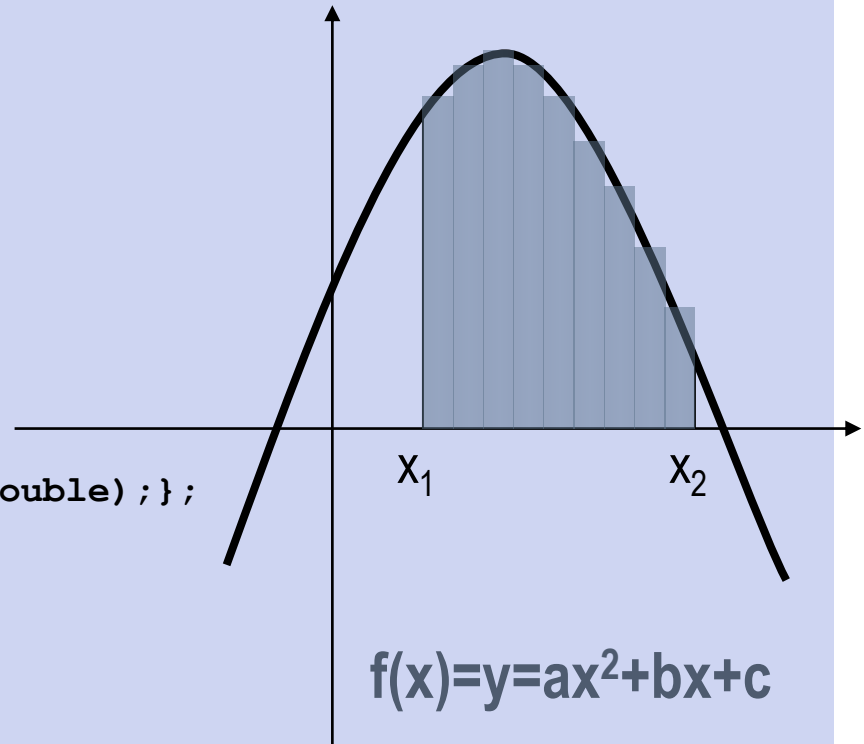


```
#include <iostream>
using namespace std;
```

```
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;}
```

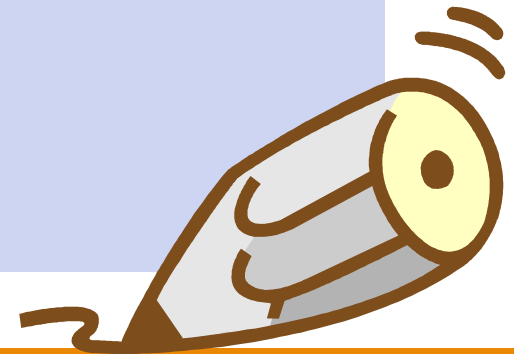


# Primjer programa - površina ispod parabole



```
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;}
```

- 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)



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

## Zadatak 2.



- **Objektno orijentirano, uz pomoć razreda, zadati koeficijente pravca, pa**
  - ispisati jednadžbu
  - izračunati presjecišta sa osima  $x$  i  $y$
  - Ispisati jednadžbu paralelnog pravca koji prolazi kroz neku zadanu točku
  - Ako pravac sječe  $x$  i  $y$  os, izračunati površinu trokuta koji s njima tvori.



## Zadatak 3.



Učitati podatke za dva strujna kruga sa po dva otpornika u svakom. Učitati iznose napona i otpora.

U jednom strujnom krugu su dva otpora spojena serijski u drugom paralelno. Zatim

- Izračunati i ispisati kolika je struja kroz svaki strujni krug.
- Ispisati koliki bi napon u strujnom krugu sa paralelno spojenim otpornicima trebao biti kako bi kroz oba strujna kruga tekla ista struja.

Zadatak riješiti objektno-orijentirano uz pomoć klasa.

- 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
  - **podržavaju prirodni proces identificiranja i klasificiranja objekata**



# Objektno-orijentirano programiranje - OOP



- Usporedba **objektno-orijentiranog modela** programiranja i **proceduralno-strukturiranog** programiranja

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**



# Ključna svojstva OO programskih jezika



## ■ **encapsulation**

- spajanje podataka i operacija

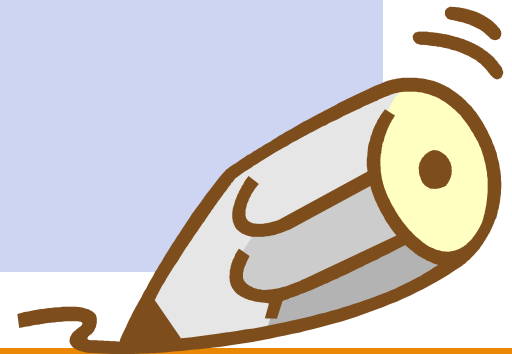
## ■ **data hiding**

- podaci objekta su privatni, dakle strukture podataka i detalji implementacije nekog objekta skriveni su od drugog objekta u sustavu
- sprječava druge objekte da pristupaju detaljima koje ne moraju znati – veći programi su zato robustniji

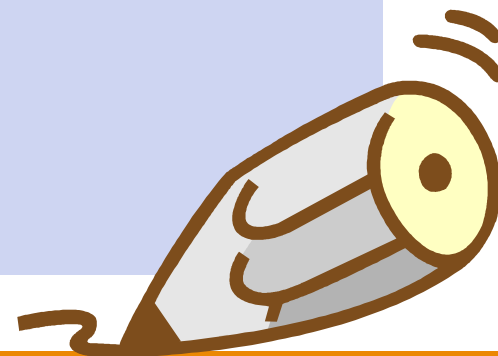
## ■ **inheritance**

- nasljeđivanje svojstava

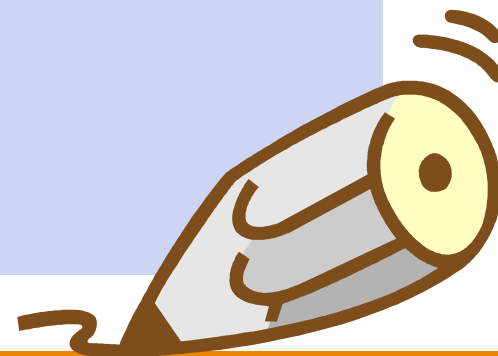
## ■ **polymorphism**



- 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



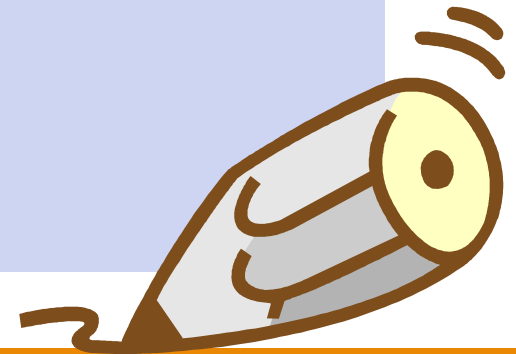
- omogućuje gradnju hijerarhija koje izražavaju međusobne odnose
- klase mogu nasljeđivati mogućnosti od klasa koje su više u hijerarhiji
- nove klase mogu nasljeđivati funkcionalnost postojećih klasa i prema potrebi modificirati ili proširiti tu funkcionalnost
- “roditeljska” klasa od koje se funkcionalnost nasljeđuje zove se **bazna** klasa **derivirane (izvedene)** klase



# Polymorphism



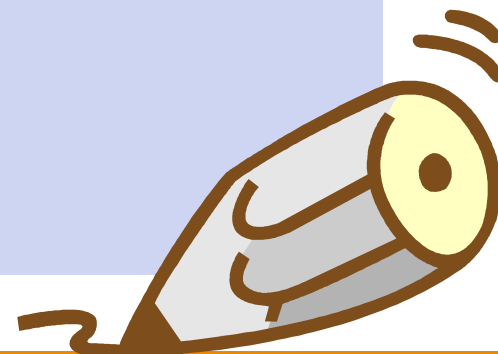
- grčki: “mnogo oblika”
- polimorfizam znači da klase mogu imati isto ponašanje, ali ga implementirati na različite načine
- ta je funkcionalnost korisna jer daje mogućnost rada sa generičkim tipovima objekata onda kada nas se ne tiče način na koji neka klasa implementira određenu funkcionalnost



# 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 vezi
- 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





# Pristup javnim (`public`) funkcijskim članovima

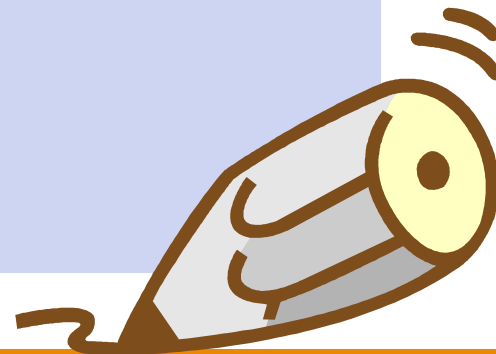
- svaki poziv funkcijskog člana mora biti pridružen objektu pa se poziv izvodi sa imenom (pozivanog) objekta:
  - ***ime\_objekta.ime\_funkcije***
- kad se funkcijski član poziva sa imenom objekta, podatkovni članovi objekta automatski se predaju funkciji (`by reference`) **bez da ih se navodi u listi 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**



# Pristupi `private` i `public`

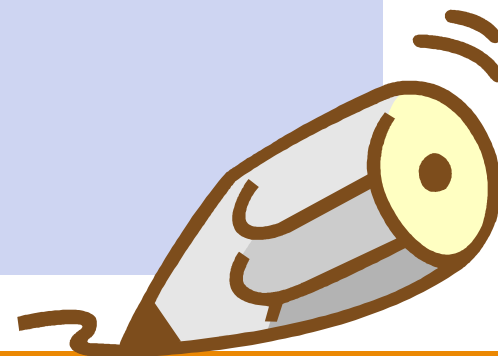


- **`public`** u definiciji klase označava da se članovima klase može pristupiti iz bilo koje funkcije
- kod objektno orijentiranog programiranja obično ne želimo da svaka funkcija može pristupiti podatkovnim članovima klase i ograničavamo pristup podatkovnim članovima korištenjem ključne riječi **`private`**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



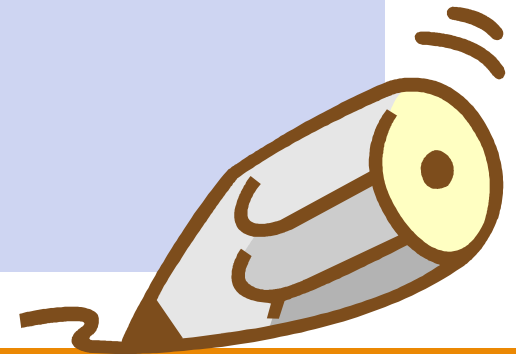
# Pristupi i enkapsulacija

- **`private` i `public` oznake određuju pristupačnost deklariranim članovima (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)

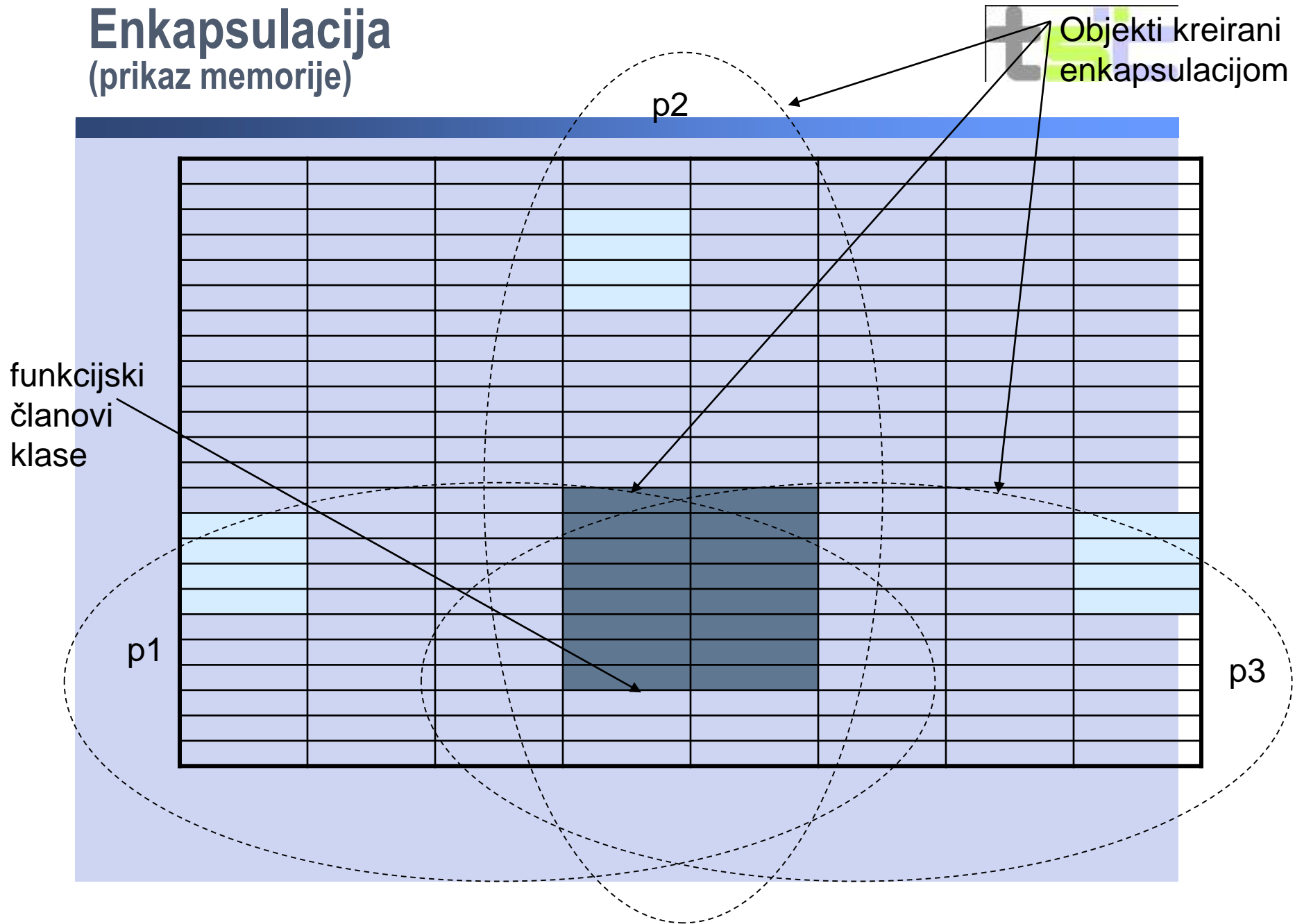


# Ideja enkapsulacije

- 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)



- 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



■ **Nabrojiti moguće podatkovne i funkcijske članove za klase:**

- A. trokut
- B. pravac
- C. strujni krug
- D. učenik

## Zadaci 4. i 5.



- Napraviti program koji će uz upotrebu klase izračunati površine dva pravokutnika (zadane su dužine stranica), pa ispisati koji pravokutnik ima veću površinu. Duljine stranica su određene realnim brojevima. Ispisati da li se površinom manji pravokutnik može prekriti većim pravokutnikom.
- Napraviti program koji će uz upotrebu klase izračunati površine dva 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_vrtanje;
    public:
        Ventilator(); //konstruktor - nema tip povratne vrijednosti, ime=ime klase
        void pisi_brzinu_vrtanje(); };

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

void Ventilator::pisi_brzinu_vrtanje() {
    cout << "Brzina vrtanje = " << brzina_vrtanje << endl;}

int main () {
    Ventilator ventilator1; //rezervira memoriju za objekt i poziva konstruktor
    ventilator1.pisi_brzinu_vrtanje();
    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 Meteostanica {
    private: int brzina_vjetra;
    public:
        Meteostanica (int); //argument odgovara tipu podatkovnog člana
        void pisi_brzinu_vjetra (); };

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

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

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

A diagram with two arrows illustrating variable initialization. One arrow points from the parameter `brvj` in the constructor's parameter list `(int brvj)` to the private member variable `brzina_vjetra` in the class definition. The second arrow points from the same `brvj` parameter to its use in the member initializer list `:brzina_vjetra (brvj)`.

# 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);    //zagrada 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);`

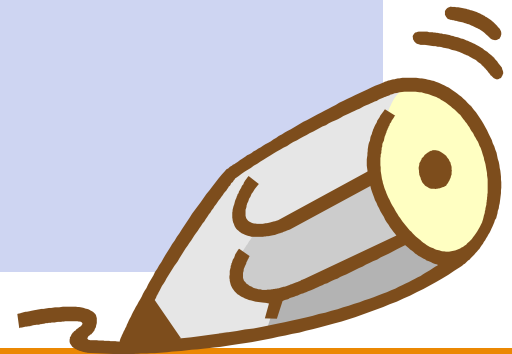


# Preopterećenje konstruktorske funkcije

## 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; }
```



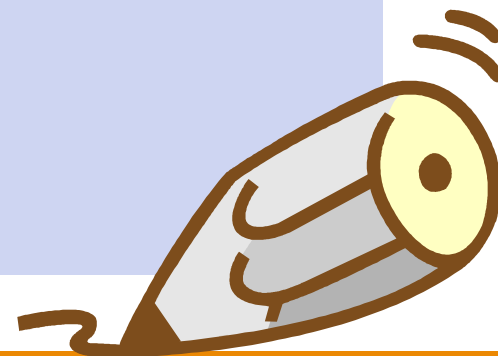
- Objekt koji poziva konstruktor bez argumenata (*program1*) nema zagrade, ali ako se konstruktor bez argumenata poziva eksplicitno, prazne zagrade moraju biti navedene, npr.

- `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;`

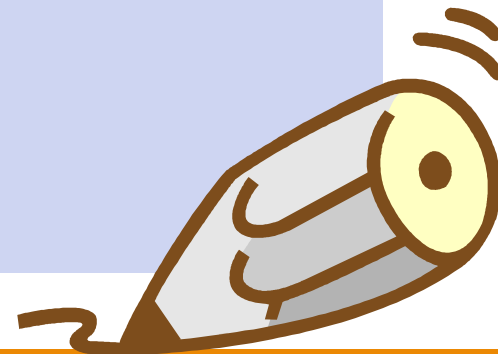
ili

- `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)



# Destruktori

# Preopterećenje operatora

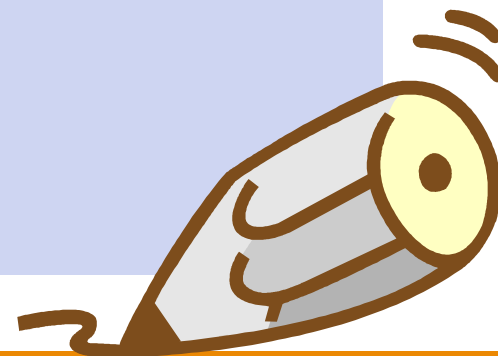
**Operator overloading**

# 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+`



- 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;}

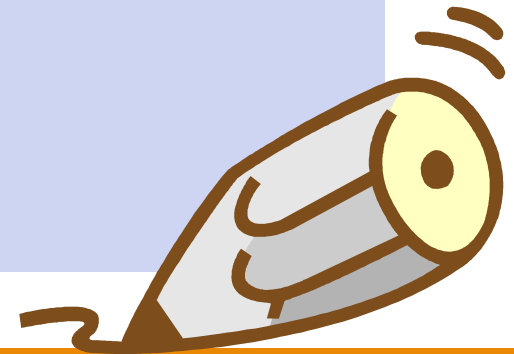
Tocka Tocka :: operator= (const Tocka& tockab) {
    x = tockab.x;
    y = tockab.y;
    cout << "Pozvana operatorska funkcija pridruzivanja." << endl;
    return *this;
}

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

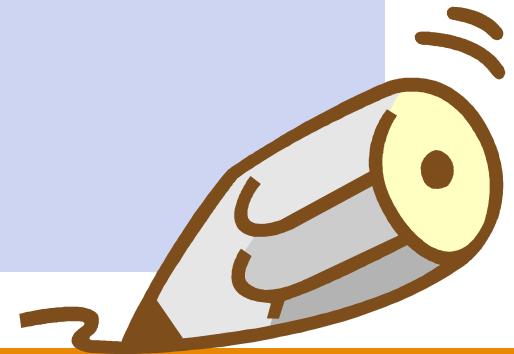
int main () {
    Tocka p1 (10, 3), p2 (4, 8), p3;
    p3 = p1;
    ++p2;
    p3.prikazi_podatke ();
    p2.prikazi_podatke ();
    return 0;
}
```



- 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)

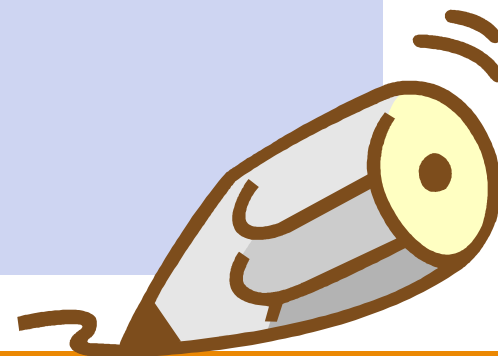


- ključna riječ `this` predstavlja pokazivač na objekt (ili adresu objekta) koji poziva
- Zato je sa `*this` predstavljen sam objekt, pa funkcija `operator++ ()` vraća izmijenjeni objekt (sa povećanim vrijednostima vlastitih podatkovnih članova)
  - kako je `operator=` funkcijski član, `x` i `y` se odnose 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



# 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:

- Napisati program za zbrajanje vektora. Iskoristiti klasu sa operatorskom funkcijom `operator+`. Napraviti i funkcijski član za određivanje duljine vektora.
- Npr., 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;
    z.display();
    return 0; }
```



```
#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;
}
```