OBJEKTNO ORIJENTISANO PROGRAMIRANJE

Oznaka predmeta: OOP

Predavanje broj: 1

Nastavna jedinica: OOP osnovni koncepti, CPP

Nastavne teme:

Pregled osnovnih koncepata OOP u jeziku C++:klase i objekti, kreiranje i korišćenje objekata,kontrola pristupa članovima, konstruktori i destruktori, preklapanje operatora, nasleđivanje, polimorfizam.

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

Literatura:

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

OOP

- Objektno orijentisano programiranje (*Object Oriented Programming*, skraćeno OOP) je novi pristup realizaciji softvera kao modela realnog sveta.
- To je poseban način razmišljanja pri projektovanju programa.
- OOP sadrži nekoliko ključnih koncepata:
  - Klase i objekte;
  - Nasleđivanje;
  - Polimorfizam.
- Softver ima za cilj da modeluje realni svet. OOP ima za cilj da programski model bude što bliži realnom svetu.
- Klasično, struktuirano, ne-objektno orijentisano programiranje biće ovde nazivano tradicionalnim programiranjem.
- Tradicionalni, algoritamski način se sastoji od rešavanja problema po koracima, dakle, razmišljamo, po koracima, *kako* se i *kada* nešto radi.

OOP

- Posmatranje problema na objektno orijentisani način podrazumeva da se *problem (*a ne *rešenje)* posmatra po delovima.
- U problemu se uočavaju *objekti* koji *nešto rade*. Bitno je *šta* to oni rade, a ne *kako* to rade.
- Objekti imaju svoje unutrašnje stanje, čija se implementacija spolja ne vidi, i koje se spolja može menjati samo definisanim i kontrolisanim akcijama.
- Uočavajte objekte u problemu koji nešto predstavljaju ili nešto rade. Zamišljajte ih kao "crne kutije" koje skrivaju svoju unutrašnjost i interagiraju sa ostalim objektima.
- Unutrašnjost kutija je stvar konkretne realizacije, koja se ostavlja za kasnije, jer je lakši deo posla.

OOP

- OOP je smišljeno kao odgovor na tzv. softversku krizu koja nastaje krajem sedamdesetih godina, kada su mogućnosti računara naglo porasle, a time i zahtevi za kompleksnim softverom.
- Organizacija velikih softverskih projekata nije više dovoljno dobra da bi bila u stanju da izađe na kraj sa velikim zahtevima korisnika.
- Projektovanje, izrada i održavanje softvera postali su preskupi, preglomazni poslovi, koji su davali sve manje rezultata u odnosu na rad koji se u njih ulagao.
- Greške u softveru su postajale sve ozbiljnije i teže za uočavanje i otklanjanje.
- Zahtevi korisnika su se *drastično* povećali (bilo je neophodno postići veću produktivnost programera).
- Ključ uspeha velikih softverskih projekata je podela posla na delove koji imaju jasno specificirane interakcije i koji ne zalaze jedan drugom u "unutrašnjost".

OOP, C++

- Bolja proizvodnja softvera može se postići i ponovnom upotrebom već napisanih delova softvera (*software reuse*).
- Jezik C++ je objektno orijentisani jezik opšte namene.
  - C++ nije čist objektno orijentisani jezik već hibridni jezik (kao naslednik C-a zadržao je koncepte tradicionalnog programiranja).
- Pogodan je kako za sistemsko, tako i za aplikativno programiranje svih vrsta.
- Jezik C++ je sredstvo za primenu objektno orijentisanih koncepata. Treba ga shvatiti kao pogodan alat, kojim se zamisao u objektnom duhu sprovodi u delo.
- Za pisanje malih programa do nekoliko stotina linija koda za jednokratno obavljanje malih poslova učenje principa OOP i jezika C++ nije potrebno.
- OOP i C++ služe za pisanje velikih programa, pogodnih za stalno proširivanje i timski rad.

- Kao odgovor na softversku krizu, OOP pruža nekoliko bitnih koncepata koje podržava i jezik C++.
- **Apstrakcija tipova podataka** (*abstract data types*).
  - Kao što u jeziku C ili nekom drugom jeziku postoje ugrađeni tipovi podataka (int, float, char, ...), u jeziku C++ korisnik (programer) može proizvoljno definisati svoje tipove i potpuno ravnopravno ih koristiti.
    Na primer: definisati tipove complex, point, disk, printer, jabuka, bankovni_racun, klijent itd.
  - Korisnik može definisati proizvoljan broj promenljivih svog tipa i vršiti predviđene operacije nad njima.
    Ovo važno svojstvo definisanja proizvoljno mnogo promenljivih korisničkog tipa, naziva se svojstvo višestrukih instanci (*multiple instances*).
  - Prema tome, tipovi podataka u OOP i jeziku C++ mogu biti apstraktni pojmovi, poput navedenih primera.

- **Enkapsulacija** (*encapsulation*), sakrivanje detalja realizacije nekog tipa.
  - Realizacija nekog tipa može (i treba) da se sakrije od ostatka sistema (od onih koji ga koriste).
  - Treba korisnicima tipa precizno definisati samo *šta* se sa tipom može raditi, a način *kako* (definisanje) se to radi sakriti od korisnika.
  - Termin "enkapsulacija" se može slobodnije prevesti i kao "učaurivanje" neke celine.
- **Preklapanje operatora** (*operator overloading*). Da bi tipovi koje je definisao programer bili sasvim ravnopravni sa ugrađenim tipovima, i za njih se u jeziku C++ mogu definisati značenja operatora koji postoje u jeziku.
  - primer: ako je korisnik definisao tip kompleksnih brojeva complex (koji inače ne postoji ugrađen u jezik), može slobodno pisati c1+c2 ili c1*c2, ako su c1 i c2 promenljive tog tipa.

- **Nasleđivanje** (*inheritance*).
  - Pretpostavimo da je u programu već formiran tip štampača *Printer* koji ima operacije nalik na *print_line* (odštampaj liniju teksta), *line_feed* (pređi u novi red), *form_feed* (pređi na novu stranicu), *goto_xy* (pozicioniraj se na datu poziciju) itd. i da je njegovim korišćenjem već realizovana velika količina softvera.
  - Novost je da je preduzeće u kome se koristi ovaj program, nabavilo i štampače koji imaju bogat skup stilova pisma i želja je da se oni ubuduće iskoriste.
  - Nepotrebno je ispočetka praviti novi tip štampača ili prepravljati stari kôd.
  - Dovoljno je kreirati novi tip *PrinterWithFonts* (štampač sa stilovima) koji je "baš kao i običan" štampač, samo "još može da" menja stilove štampe. Novi tip će *naslediti sve* osobine starog, ali će još ponešto i više moći da uradi.

C++, polimorfizam

- **Polimorfizam** (*polymorphism*), ili pojavljivanje u više oblika.
  - Iz prethodnog primera:
    pošto je *PrinterWithFonts* već ionako jedna vrsta tipa *Printer*, jer je to prethodno obezbeđeno odgovarajućim mehanizmima, nema razloga da ostatak programa, koji koristi ovaj tip, ne "vidi" njega kao i običan štampač, sve dok mu nisu potrebne nove mogućnosti štampača.
  - Ranije napisani delovi programa koji koriste tip *Printer ne moraju se uopšte prepravljati*, oni će jednako dobro raditi i sa novim tipom. Štampači novog tipa će vršiti ispis na sebi svojstven način, dakle sa stilovima koje poseduju, iako ih ostatak programa smatra starim štampačima.
  - Karakteristika da se novi tip "odaziva" na novi, sebi svojstven način, iako ga je neko spolja "prozvao" kao da je stari tip, naziva se *polimorfizam*.

- Evo nekoliko napomena o tome šta se menja u razmišljanju i pisanju programa na OO način:
  - 1. OOP uvodi *drugačiji način razmišljanja* u programiranje!
  - 2. U OOP, *mnogo* više vremena se troši na *projektovanje*, a mnogo manje na samu *implementaciju* (kodiranje).
  - 3. U OOP, razmišlja se o *problemu* koji se rešava, a ne direktno o *programskom rešenju*.
  - 4. U OOP, razmišlja se o delovima sistema (*objektima*) koji *nešto* rade, a ne o *algoritmima*, tj. o tome *kako* se nešto radi.
  - 5. U OOP, pažnja se prebacuje sa *realizacije delova* programa na međusobne *veze* između tih delova. Težnja je da se te veze što više redukuju i strogo kontrolišu.
    Cilj OOP je da smanji interakciju između softverskih delova.

C++, klase i objekti

- *Klasa* (*class*) je osnovna organizaciona jedinica programa u OO jeziku kakav je i C++.
- Klasa predstavlja tip, skup svih objekata istih svojstava.
- Klasa predstavlja neki apstraktni tip koga definiše korisnik jezika, tj. programer.
  - Klasa u jeziku C++ se naziva još i korisničkim tipom, za razliku od tipova podataka koji su već ugrađeni u jezik C++.
  - Primer: svi istovetni štampači itd.
- Klasa je opisana svojim:
  - *Atributima* (svojstvima, sadržajem) - najčešće predstavljaju unutrašnjost klase - njenu realizaciju
  - *Operacijama* (akcijama) koje se mogu vršiti nad objektima te klase, predstavljaju interfejs klase - ono što se spolja može raditi sa objektima te klase.

C++, klase i objekti

- Klasa se u jeziku C++ predstavlja strukturom u kojoj su grupisani podaci i funkcije.
- Podacima klase realizuju se atributi klase (stanje objekta), a funkcijama operacije nad klasom (ponašanje objekta).
- Članovi klase su, dakle, podaci članovi i funkcije članice.
- Na primer želimo da modeliramo klasu *Osoba*.
  - U našem problemu potrebno je da svaki pripadnik ove klase ima svoje ime i godine starosti.
  - Ime i godine starosti predstavićemo podacima klase: *ime* i *god*.
  - Svaki pripadnik ove klase treba da odgovori na akciju koja traži da se osoba predstavi.
  - Akcija će biti predstavljena funkcijom članicom *ko_si()*.

C++, klase i objekti

// Deklaracija klase: pseudo kod

```
class Osoba {
  podatak koji sadrži ime i prezime osobe ime;
  podatak koji sadrži godine osobe god;
public:
  funkcija ko_si();
};
```

- Ključna reč **`class`** jezika C++ označava da počinjemo sa *deklaracijom* klase.
- Deklaracija u jeziku C++ predstavlja iskaz kojim se neko ime uvodi u program.
- Deklaracijom se prevodiocu "objašnjava" šta predstavlja ime koje se njome uvodi.
- Prevodiocu se time omogućava da zna kako se to ime može koristiti u programu, šta se sa njim može, a šta ne može raditi. Prevodilac će tako kontrolisati upotrebu tog imena u daljem tekstu programa.

C++, klase i objekti

- Ovom deklaracijom klase nismo objasnili prevodiocu *šta* treba da uradi funkcija *ko_si()*.
  - mi smo samo deklarisali da klasa poseduje ovu funkciju i da se ona može izvršiti za svaki objekat ove klase.
  - funkcija *ko_si()* je samo deklarisana unutar deklaracije klase.
  - telo funkcije *definiše* se posebno, van deklaracije klase, iskazom koji se naziva *definicijom* funkcije.
  - definicija nekog imena u programu predstavlja iskaz koji odvaja memorijski prostor za neku promenljivu, daje telo funkcije. Definicija funkcije *ko_si()* navodi se izvan deklaracije klase i može izgledati otprilike ovako:

```
funkcija Osoba::ko_si () {
  ispiši ime i godine
}
```

- U ovoj definiciji funkcije znaci **Osoba::** označavaju da funkcija *ko_si()* pripada klasi *Osoba*.
- Telo funkcije nalazi se između para vitičastih zagrada ({}). Definicija funkcije *ne* završava se znakom tačka-zarez (;).

C++, klase i objekti

- Klasa predstavlja tip za koji se mogu kreirati instance (primerci).
- Primerci klase nazivaju se *objektima*. Kako je klasa tip, objekat se može smatrati promenljivom tog tipa u programu.
- Svaki objekat ima svoje sopstvene elemente koji su navedeni u deklaraciji klase.
  - Ovi elementi nazivaju se *članovima* klase (*class members*). Podaci se nazivaju podacima članovima (*data members*), a funkcije - funkcije članice (*member functions*).
- Podaci članovi mogu biti objekti nekih drugih klasa.
  - primer,
    - podatak član *ime* klase *Osoba* može biti objekat neke klase koja predstavlja nizove znakova maksimalne dužine npr. 30
    - podatak član *god* može biti objekat neke klase koja predstavlja cele brojeve u opsegu 0 do 120 i slično. Na ovaj se način klasa izgrađuje od manjih gradivnih elemenata, objekata drugih klasa. Ovakva relacija između klasa naziva se *ugrađivanjem*.

C++, klase i objekti

// Korišćenje klase *Osoba*

// negde u programu se deklarišu promenljive tipa *Osoba*:

```
Osoba Pera, otac, direktor;
```

// a onda se one koriste:

```
Pera.ko_si();       // poziv funkcije ko_si objekta Pera
otac.ko_si();       // poziv funkcije ko_si objekta otac
direktor.ko_si();   // poziv funkcije ko_si objekta direktor
```

- Neka je izlaz:

```
Ja sam Pera Peric i imam 20 godina.
Ja sam Sima Simovic i imam 36 godina.
Ja sam Jovana Jovicic i imam 40 godina.
```

Kontrola pristupa članovima

- Jedini članovi klase koji su dostupni spolja su oni koji se u deklaraciji klase nalaze iza ključne reči **public:** (javni).
  - Ovakvi članovi nazivaju se *javnim* članovima klase (*public members*). Samo njima korisnici mogu pristupiti, dok će eventualni pokušaj pristupa spolja ostalim članovima prevodilac prijaviti kao grešku.
- "Skriveni" članovi dostupni su unutar funkcija članica klase, jer one predstavljaju sastavni deo klase. Ovakvi članovi nazivaju se *privatnim* članovima klase (*private members*).
- U datom primeru članovi *ime* i *god* su privatni (podrazumevano), a član *ko_si()* javni član klase *Osoba* (zbog bloka public:).
- Opisani mehanizam naziva se mehanizmom *kontrole pristupa članovima* klase (*member access control*).
- Kada projektant klase *Osoba* ne bi imao mogućnost da *sigurno* zabrani pristup spolja članovima *ime* i *god* moglo bi see desiti da neki korisnik klase "nehotice" ovim podacima dodelio neke nedozvoljene vrednosti.

Kontrola pristupa članovima

```
// Izvan funkcija članica klase Osoba nije moguće:
   Pera.ime="Pera Peric";      // nedozvoljeno
   otac.god=55;                // takođe nedozvoljeno

// Šta bi tek bilo da je ovo dozvoljeno:
   direktor.ime="bu...., kr...., ..";  // nešto nezgodno
   direktor.god=1000;

// a onda neko pita (što je dozvoljeno):
   direktor.ko_si();
```

- Opisani mehanizam obezbeđuje da projektant klase ima potpunu kontrolu nad tim šta se sa objektima te klase može raditi.
  - Izbegava se "podmetanje" ili promicanje greške.
  - Sve je koncentrisano u deklaraciji klase.
  - Princip enkapsulacije pomaže programerima da se zaštite od velikog broja grešaka.
    Greška se traži unutar realizacije same klase ("čudnog") objekta.

Konstruktori i destruktori

- U realnom svetu objekat predstavlja složeni sistem koji ima neko svoje unutrašnje stanje.
  - To stanje treba uspostaviti na određen način na početku života objekta. Treba u programu obezbediti mehanizam da se objekat, po nastanku, automatski dovede u ispravno početno stanje.
- Da bi se to obezbedilo, u klasi se definiše posebna funkcija članica koja se poziva automatski pri kreiranju objekta.
  - Prevodilac obezbeđuje automatsko pozivanje ove funkcije članice pri definiciji objekta (programer ne mora eksplicitno da je poziva).
  - Ova funkcija se naziva *konstruktor* (*constructor*) i nosi isto ime kao i sama klasa.
- Konstruktor u klasi *Osoba* bi trebalo da pri kreiranju objekta klase *Osoba* postavi vrednosti za podatke ime i godine starosti ovog objekta (na početku života objekta).

Konstruktori i destruktori

Konstruktor se definiše kao i svaka druga funkcija članica:

```
class Osoba {
  podatak koji sadrži ime i prezime osobe ime;
  podatak koji sadrži godine osobe god;
public:
  funkcija ko_si ();
  funkcija Osoba (argumenti za ime i godine); // konstruktor
  funkcija ~Osoba();                          // destruktor
};
```

Implementacija:

```
funkcija Osoba::ko_si () {      ispiši ime i godine       }
funckija Osoba::Osoba (argumenti za ime i godine) {
  proveri ime da li je dozvoljeno;
  proveri godine;
  smesti argumente za ime i godine u članove ime i god;
}
```

Korišćenje klase *Osoba* sada je:

```
Osoba Pera("Pera Peric",20),   // poziv konstruktora Osoba
      otac("Sima Simic",35);   // poziv konstruktora Osoba
Pera.ko_si();
otac.ko_si();
```

Preklapanje operatora

```
class Complex {
  podatak koji čuva realni deo real;
  podatak koji čuva imaginarni deo imag;
public:
  funkcija Complex(argumenti za rea. i im. deo);//konstruktor
  funkcija operator+(argument tipa Complex);    // operator +
  funkcija operator-(argument tipa Complex);    // operator -
};
```

- Prva funkcija je konstruktor koji inicijalizuje kompleksni broj pomoću dva realna broja (realni i imaginarni deo).
- Druga i treća funkcija su operacije sabiranja i oduzimanja.
- Implementacija klase complex, tačnije njenih funkcija članica, nije u ovom trenutku bitna.

```
Complex c1(3,5.4),c2(0,-5.4),c3(0,0);  // definicija objekata
c3=c1+c2; //na desnoj strani znaka = poziva se c1.operator+(c2)
c1=c2-c3; //na desnoj strani znaka = poziva se c2.operator-(c3)
```

Nasleđivanje

```
class Maloletnik : public Osoba {
  podatak koji sadrži staratelja staratelj;
public:
  funkcija ko_je_odgovoran();
  funkcija Maloletnik(argumenti za ime, staratelja i godine);
};
```

- Rečju **`class`**, kao i obično, počinje opis klase.
  - Sledeća reč predstavlja ime klase (`Maloletnik`).
  - Sledeći niz znakova `:public Osoba` govori da je ova klasa izvedena (naslednik) klase *Osoba*.
  - Dalje sledi spisak članova: podataka i funkcija koje poseduje ova klasa, pored onih članova koje nasleđuje iz osnovne klase.
- Izvedena klasa `Maloletnik` ima *sve* članove kao i osnovna klasa *Osoba*, ali ima još i članove `staratelj` i `ko_je_odgovoran()`.
- Deklaracija konstruktora klase `Maloletnik` govori da se objekat ove klase kreira zadavanjem imena, staratelja i godina.

Nasleđivanje

Pokažimo i kako izgleda implementacija klase maloletnik:

```
funkcija Maloletnik::ko_je_odgovoran () {
  ispiši rečenicu "Za mene odgovara ... (staratelj)";
}
funkcija Maloletnik::Maloletnik (argumenti za ime, staratelja i
godine)
    : Osoba(preneseni argumenati za ime i godine)
{
  inicijalizuj staratelja;
}
```

- U prethodnom primeru, u zaglavlju konstruktora izvedene klase maloletnik se iskazom:
  `: Osoba(preneseni argumenati za ime i godine)`
  definiše da se konstruktor osnovne klase *Osoba* (koji inicijalizuje ime i godine) poziva sa odgovarajućim argumentima.
- U telu samog konstruktora klase maloletnik se samo inicijalizuje staratelj.
- Objekat klase *Maloletnik* kreira se pozivom konstruktora *Maloletnik*.

Nasleđivanje

- Ovom konstruktoru se prenose argumenti za zadato ime, staratelja i godine (i *Maloletnik* ima i ime i godine, jer je i *Maloletnik Osoba*).
- Klasa *Maloletnik* poseduje sve članove klase *Osoba* koje takođe treba inicijalizovati.
  - Ta inicijalizacija se vrši pozivom konstruktora osnovne klase *Osoba* pre izvršavanja samog tela konstruktora klase *Maloletnik*.
  - U zaglavlju konstruktora klase *Maloletnik* se navode argumenti poziva konstruktora klase *Osoba* , iza znaka dvotačka (:) i imena osnovne klase *Osoba*.
    - To mogu biti i argumenti prosleđeni konstruktoru izvedene klase *Maloletnik*.
    - Konstruktor klase *Osoba* inicijalizovaće onaj deo objekta klase *Maloletnik* koji je nasleđen iz klase *Osoba* .
    - Kada se završi konstruktor osnovne klase *Osoba* , koji je inicijalizovao ime i godine, izvršava se telo konstruktora izvedene klase *Maloletnik*, u kome se samo ime staratelja postavlja na zadatu vrednost.

Nasleđivanje

- Pošto se kreiraju objekti izvedene klase, mogu se koristiti i nasleđene osobine objekata klase *Maloletnik* iz klase *Osoba*, ali su dostupna i njihova posebna svojstva kojih nije bilo u klasi *Osoba.*

```
Osoba otac("Sima Simic",35);
Maloletnik dete("Paja Simic","Sima Simic",10);

otac.ko_si();
                    //izlaz→ Ja sam Sima Simic i imam 35 godina.
dete.ko_si();
                    //izlaz→ Ja sam Paja Simic i imam 10 godina .
dete.ko_je_odgovoran();
                    //izlaz→ Za mene odgovara Sima Simic

otac.ko_je_odgovoran();     // greška: nema navedenu metodu
```

Polimorfizam

- Pretpostavimo sada da nam je potrebna nova klasa *Zena*, koja je opet "jedna vrsta" klase *Osoba*, samo što još ima i devojačko prezime.
    - Klasa *Zena* biće, tako, izvedena iz klase *Osoba*.
- I objekti klase *Zena* treba da se "odazivaju" na poruku *ko_si()*, samo što jedna dama otvoreno ne priznaje svoje godine, tako da će ispis biti bez broja godina, svojstveno izvedenoj klasi *Zena*.

```
class Osoba {
    protected:                          // dostupno naslednicima
     podatak koji sadrži ime i prezime osobe ime;
     podatak koji sadrži godine osobe god;
    public:
     virtual funkcija ko_si();   // virtuelna funkcija članica
     funkcija Osoba(argumenti za ime i godine);  // konstruktor
   };
```

Implementacija:

```
   funkcija Osoba::ko_si () {   ispiši ime i godine  }
   Osoba::Osoba (argumenti za ime i godine) {
     kao i ranije...
   }
```

Polimorfizam

```
class Zena : public Osoba {
  podatak koji sadrži devojačko prezime devojacko;
public:
  Zena(argumenti za ime, devojačko i godine);  // konstruktor
  funkcija ko_si();  // nova verzija funkcije ko_si
                     // u izvedenoj klasi ne pojavljuje se reč
                     // virtual.
};
```

Implementacija:

```
funkcija Zena::Zena (argumenti za ime, devojačko i godine)
  : Osoba (preneseni argumenti za ime i godine)
{
  inicijalizuj devojačko prezime;
}

funkcija Zena::ko_si ()
{
  ispiši ime i devojačko prezime;
}
```

Polimorfizam

- U klasu *Osoba* unesene su dve izmene.
  1. Ubačena je ključna reč protected:
     obezbeđuje da:
     - Članovi *ime* i *god* budu:
       - Dostupni izvedenim klasama (tačnije njihovim funkcijama članicama),
       - Nedostupni korisnicima spolja (npr. funkcijama članicama drugih klasa).
       - To je urađeno jer hoćemo da funkcija članica izvedene klase *Zena* direktno priđe podatku *ime* kako bi ga ispisala pri predstavljanju.
  2. Ispred deklaracije funkcije članice *ko_si()* dodata je ključna reč virtual. Ovim se najavljuje da ta funkcija može imati nove verzije u izvedenim klasama. Svaka izvedena klasa moći će da definiše svoju verziju ove funkcije, tako da operaciju predstavljanja obavlja na sebi svojstven način.

Polimorfizam

- Klasa *Ispitivac* služi za ispitivanje objekata klase *Osoba*

```
class Ispitivac {
public:
  funkcija ispitaj (argument tipa Osoba&);//referenca
};
```

- Funkcija "ispitaj()" članica klase *Ispitivac* propituje osobe i ne mora da se menja:

```
funkcija Ispitivac::ispitaj (argument X tipa Osoba&) {
  X.ko_si();  // poziv funkcije članice argumenta
}
```

- Ovoj funkciji možemo preneti kao argument i objekat klase *&Osoba*, kao i objekat klase *&Zena*, jer je klasa *Zena* izvedena iz klase *Osoba*:

```
Osoba      otac ("Laza Lazic",                40);
Zena       majka("Mara Lazic","Hadzipesic",39);
Maloletnik dete ("Mica Lazic","Laza Lazic",12);
Ispitivac  isp;

isp.ispitaj(otac);              // pozvaće se Osoba::ko_si()
isp.ispitaj(majka);             // pozvaće se Zena::ko_si()
isp.ispitaj(dete);              // pozvaće se Osoba::ko_si()
```

Polimorfizam

- U prvom pozivu *isp.ispitaj(otac)* funkcija ispitaj će dobiti kao argument objekat klase *&Osoba* i pozvaće verziju funkcije *ko_si()* iz klase *Osoba.*
- U drugom pozivu *isp.ispitaj(majka)* funkcija ispitaj će dobiti objekat klase *&Zena*. Sada nastupa mehanizam virtuelnih funkcija: iako funkcija ispitaj smatra dobijeni objekat pripadnikom klase *Osoba*, biće pozvana verzija funkcije *ko_si()* iz klase *Zena.*
- Kod poziva *isp.ispitaj(dete)* poziva se verzija *Osoba::ko_si()*, jer klasa *Maloletnik*, kojoj pripada objekat *dete*, nije definisala svoju verziju funkcije *ko_si()*, već je preuzela osnovnu verziju iz klase *Osoba*.
- Izlaz prethodnog programa bi bio:

```
Ja sam Laza Lazic i imam 40 godina.
Ja sam Mara Lazic, devojacko prezime Hadzipesic.
Ja sam Mica Lazic i imam 12 godina.
```

- Polimorfizam se aktivira prosleđivanjem pokazivača ili referenci na objekte izvedene klase kao stvarnih argumenata tamo gde je formalni agument referenca ili pokazivač na objekat osnovne klase (detaljno radimo kasnije).

Polimorfizam

- Opisani mehanizam omogućuje da program bude razvijan kao što sledi:
  - Programer A je realizovao klasu *Osoba*, a onda programer B klasu *Ispitivac* koja koristi klasu *Osoba*.
  - Posle dužeg vremena eksplotacije programa uočena je potreba da se formira nova klasa *Zena*.
    - U klasu *Osoba* su dodate reči protected i virtual na opisani način
    - Klasa *Ispitivac* ne treba da se menja. Ona će dati odgovarajući izlaz čak i ako joj se kao argument prenese referenca na objekat nove, izvedene klase.
- U složenom projektu, sa mnogo klasa koje koriste neku klasu iz koje izvodimo novu klasu, uz dobro projektovanje, najviše što ćemo morati da uradimo je da ponovo prevedemo program, ili neke njegove delove.
- Navedeno svojstvo, da se odaziva prava verzija funkcije članice klase čiji su naslednici dali nove verzije (oblike), naziva se *polimorfizam* (*polymorphism*) - pojavljivanje u više oblika.

Primer 1/3

```
//---------------------------- FAJL Osoba.h
#pragma once
class Osoba{
   protected:
      char *ime;
      int godine;
   public:
      Osoba(char* ime, int g);
      virtual void KoSi();
};

//---------------------------- FAJL Osoba.cpp
#include "Osoba.h"
#include <iostream>
using namespace std;
Osoba::Osoba(char *ime, int g): ime(ime), godine(g){}

void Osoba::KoSi(){
   cout<<ime<<", godina "<<godine<<endl;
}
```

Primer 2/3

```
//---------------------------- FAJL Zena.h
#pragma once
#include "Osoba.h"
class Zena : public Osoba {
public:
  Zena(char* ime, char* devojacko, int godine);
  void KoSi();   // nova verzija funkcije KoSi, ne mora i ovde da
                 // se navodi virtual
private:
  char* devojacko;
};
//---------------------------- FAJL Zena.cpp
#include "Zena.h"
#include <iostream>
using namespace std;
Zena::Zena (char* i, char* d, int g)
    : Osoba(i,g), devojacko(d) {}

void Zena::KoSi () {
  cout<<"Ja sam "<<ime<<", devojacko prezime "<<devojacko<<endl;
}
```

Primer 3/3

```
//-------------------------- FAJL OOPp01.cpp
#include <iostream>
#include "Osoba.h"
#include "Zena.h"
using namespace std;

void ispitaj(Osoba& hejti){    hejti.KoSi();   }

int main(void){
   Osoba osoba1("Sloboda",43);   osoba1.KoSi();

   Zena z("Sonja","Savic",30);   ispitaj(z);

   //Maloletnik m("Joe","John",10); //Za domaci rad
   //m.KoSi();                      //implementirati maloletnika
   //m.ispisi_staratelja();         //tako da rade ove 3 linije
                                    //koda
   system("PAUSE");
   return 0;
}
```