Programabilni uređaji i objektno orjentisano programiranje

Metodi članovi klase, statički djelovi interfejsa klase. Konstruktor kopije

Konstruktor kopije - Potreba

- Pored mogućnosti da se konstruktor pozove sa argumentima koji će se, recimo, iskoristiti da inicijalizuju objekat, postoji i mogućnost da se objekat inicijalizuje na vrijednost nekog drugog objekta istog klasnog tipa.

```
Student Pero(2,3);

Student Marko(Pero);
```

- Recimo, program za studentsku službu sve studente koji se upisuju inicijalizuje na neki podrazumijevani objekat (student prve godine), tako da službenici ne popunjavaju sve podatke od početka već samo dopunjuju (godina upisa studenta je odmah poznata, godina rođenja vjerovatno poznata, student upisuje prvu godinu itd.), čime se značajno štedi na vremenu.
- Postoje i druge situacije kada nam je ovakva funkcionalnost neophodna, a kojima će biti riječi kasnije.

Konstruktor kopije - Pravila

- Konstruktor kopije je konstruktorska funkcija koja inicijalizuje novonastali objekat klasnog tipa na vrijednost drugog objekta klasnog tipa.
- Ako mi ne realizujemo konstruktor kopije to će umjesto nas odraditi kompajler.
- Pored podrazumijevanih konstruktora i destruktora, podrazumijevani konstruktor kopije je treći od četiri metoda koje umjesto nas kreira kompajler.
- Po pravilu, mora se realizovati kod klasa koje imaju pokazivače članove (nije obaveza, jer kompajler ne javlja grešku ako ga nema, ali ako se koristi podrazumijevani ne radi kako treba – ne kreira ispravno kopiju), a po potrebi i kod drugih klasa.

Konstruktor kopije - Deklaracija

- Konstruktor kopije se deklariše u opštem obliku kao:

```
X(const X&); //X je naziv klase
```

- Dakle, konstruktor kopije ima isto ime kao i klasa **(kao i ostali konstruktori),** a argument je konstantna referenca na objekat istog klasnog tipa.
- Naime, objekat koji je argument konstruktora se neće prilikom konstrukcije drugog objekta mijenjati (otud const), a referenca se koristi radi efikasnosti u prenosu argumenata – nema potrebe da se pri pozivu funkcije kopira kompletan objekat koji će poslužiti za inicijalizaciju/konstrukciju.
- Npr. `Student(const Student &x){/*kod metoda*/}`
- const se ovdje koristi da bi naglasio da se objekat na osnovu kojega se vrši kontstrukcija (čiji se sadržaj kopira) ne mijenja.

Konstruktor kopije - Upotreba

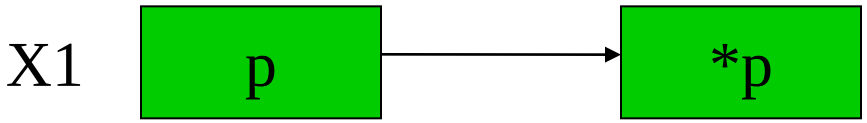
- U kodu programa moramo imati deklarisan objekat klasnog tipa:

```
Student Marko;
...
//naredbe kojima se determiniše student Marko
Student Janko(Marko); //Janko koji je
//inicijalizovan na vrijednost Marko.
```

- Janko sada ima sve podatke članove prekopirane sa objekta Marko, ali dalji život ova dva objekta je nezavisan.
- U slučaju klasa koje imaju podatke članove koji su pokazivači ovo ne bi bio slučaj. Veza bi preko pokazivača i dalje postojala.
- Kopirala bi se vrijednost podatka člana koji je pokazivač, tj. adresa u memoriji koja je zauzeta od strane objekta koji je argument konstruktora kopije.
- Oba bi objekta imala poazivač koji ukazuje na isto mjesto u memoriji.

Konstruktor kopije i pokazivači

- Pretpostavimo sljedeću generičku klasu:
  `class X{ int *p; /*još koješta*/};`
- Objekat X1 koji je tipa X (deklarisan kao `X X1;`) ima memoriju zauzetu na sljedeći način:

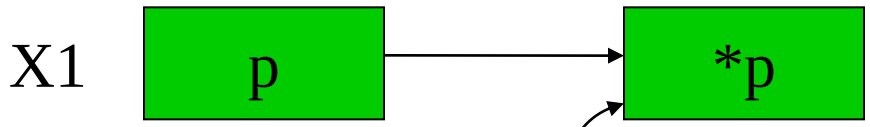


Pokazivač p pripada objektu X1 i pokazuje na neki cijeli broj *p koji se nalazi na memorijskoj lokaciji X1.p.

Ako bismo se oslonili na podrazumijevani konstruktor kopije i deklarisali objekat
`X X2(X1);`
što bi se dogodilo?

Konstruktor kopije i pokazivači Primjer - Problem

- Stanje prije poziva konstruktora kopije:



- Nakon poziva podrazumijevanog konstruktora kopije

Vrijednost p je iskopirana sa objekta X1, a kako je to adresa nekog memorijskog objekta, to onda znači da X1.p i X2.p pokazuju na istu vrijednost, odnosno, ova dva objekta nijesu nezavisna, pa promjena na jednom može da utiče na drugi. Ovo najčešće želimo **po svaku cijenu** izbjeći.

Konstruktor kopije – Pokazivači Prevazilaženje problema

- Kod pokazivača članova po pravilu moramo napraviti konstruktor kopije koji bi u slučaju naše klase mogao da ima sljedeći oblik:

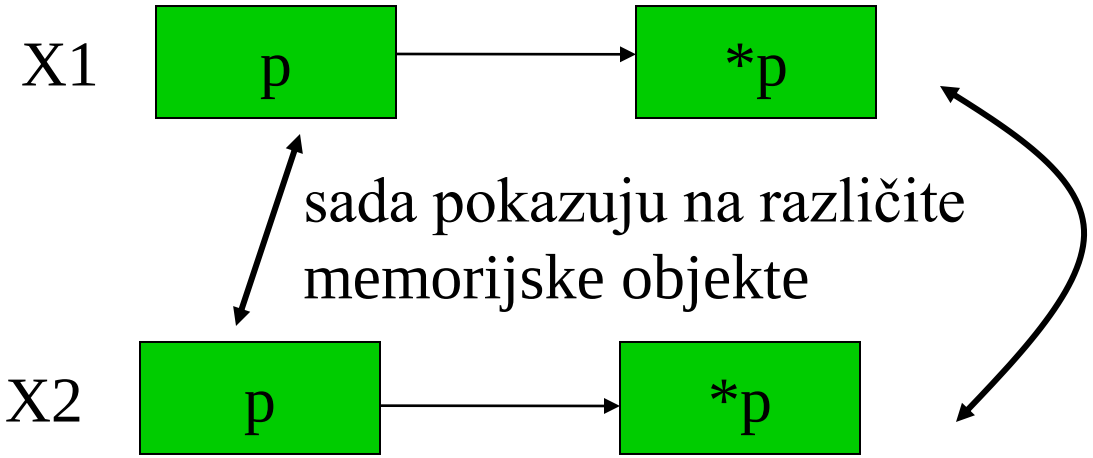
```
class X{
    int *p; /*još koješta*/
public:
    X(const X&);
    //ostali metodi uključujući konstruktore
};
```

- Realizacija konstruktora kopije:

```
X::X(const X& y){
p=new int(*y.p);
}
```

→ Alocira se nova memorija za podatak na koji pokazuje p za objekat kopiju, a ta memorija se inicijalizuje na vrijednost na koju pokazuje p iz objekta originala.

Konstruktor kopije – Pokazivači Ilustracija



Na početku, nakon konstrukcije kopije, podaci na koje se pokazuje pokazivačem *p* imaju iste vrijednosti, ali sada se ne može dogoditi da promjena vrijednosti jednog memorijskog objekta utiče na drugi!

- Još jednom da se ogradimo: Ovo je uobičajeni, ali ne i obavezni način rada sa konstruktorom kopije za klase sa pokazivačima kao članovima.
- Zavisnost objekata direktno ugrožava temelje OO koncepata!

```cpp
#include <iostream>

using namespace std;
class X{
    int *p; /*još koješta*/
public:
    X(){p=0;}
    X(int a):p(new int(a)){}
    X(const X&);
    ~X(){delete p; p=0;}
    int Daj(){return *p;}
    void Postavi(int i){*p = i;}
};
X::X(const X& r){
    p = new int;
    *p = *r.p;
    }


int main()
{
    X a(3),b(a);
    cout<<"a je iskopirano u b. *a.p = "<<a.Daj()<<" *b.p = "<< b.Daj()<<endl;
    b.Postavi(5);
    cout<<"Promjena objekta b, ne utice na objekat a: *a.p = "<<a.Daj()<<" *b.p = "<< b.Daj()<<endl;
    return 0;
}
```

```
a je iskopirano u b. *a.p = 3 *b.p = 3
Promjena objekta b, ne utice na objekat a: *a.p = 3 *b.p = 5

Process returned 0 (0x0)   execution time : 0.073 s
Press any key to continue.
```

Druge primjene konstruktora

- Po pravilu, konstruktor provjerava vrijednosti na koje se vrši inicijalizacija objekata i preduzima korektivne akcije.
- Npr. posmatrajmo klasu Vrijeme koja ima tri podatka člana hh, mm i ss, koji redom označavaju vrijeme u satima, minutama i sekundama. Konstruktor ove klase treba da bude spreman da koriguje pogrešno zadato vrijeme:

```
class Vrijeme{
private:
    int hh,mm,ss;
public://...
Vrijeme(int a=0,int b=0,int c=0){
if(a>23 || a<0) hh=0; else hh=a;
if(b>59 || b<0) mm=0; else mm=b;
if(c>59 || c<0) ss=0; else ss=c;}
//i još koješta
};
```

Dio koda u kojem se vrijednosti podataka članova "vraćaju" u normalu u slučaju da se pozove konstrukcija objekta ovog klasnog tipa sa vrijednostima van domena.

Konstruktor za provjeru domena

- Napominjemo da teorija obično ne odobrava ovakvu primjenu konstruktora, već savjetuje dizajniranje posebnog metoda koji provjerava vrijednosti podataka članova u svakoj operaciji gdje se podaci članovi mogu postaviti van domena.
- Ovakvi metodi za provjeru su po pravilu privatni i vide ih samo drugi metodi te klase.
- Praksa, međutim, često provjeru domena za podatke članove vrši u konstruktoru.
- Sa konstruktorom koji ima podrazumijevane vrijednosti za sva tri argumenta, datim u klasi Vrijeme, mi smo realizovali i podrazumijevani konstruktor i ne smijemo ga dodatno realizovati, jer bi došlo do dvosmislenosti prilikom njegovog poziva. Ovim konstruktorom smo omogućili sve naredne deklaracije

```
Vrijeme v1, v2(8), v3(8,15), v4(8,15,35);

//protumačite kolike bi vrijednosti imali
//podaci članovi ovako deklarisanih objekata
```

Konstruktor pri konverziji tipova

- Supervažna osobina konstruktora je da može da posluži i za konverziju standardnih (ugrađenih) tipova podataka u klasne tipove.

```
class complex{
private:
    float re,im;
public://...
    complex sabcomp(complex);
    ...
};
```

Podsjetimo se za trenutak klase complex i metoda sabcomp.

```
complex complex::sabcomp(complex y){
complex temp;
temp.re=re+y.re;
temp.im=im+y.im;
return temp;}
```

Konverzija tipova

- Posmatrajmo sada dio glavnog programa:

```
main(){
complex z1,z2,z3;
//naredbe koje postavljaju vrijednost z1 i z2
z3=z1.sabcomp(z2);}
```

- Ovo je korektan način poziva metoda za kompleksni broj z1 sa argumentom - kompleksnim brojem z2.
- Što će se dogoditi sa pozivom:

```
z3=z1.sabcomp(2.5);
```

- Namjera pozivaoca metoda je jasna. On želi da sabere kompleksni sa realnim brojem, što je u matematici dozvoljeno, ali ovdje ako ne odradimo jednu operaciju, nije.
- Naime, sabcomp očekuje argument tipa complex, a proslijeđen mu je realan broj. Ne postoji implicitna konverzija ugrađenog tipa u korisnički, već je to naš zadatak, i to uz pomoć kontruktora, koji se automatski poziva prilikom prosleđivanja podataka funkciji.

Konstruktor kod konverzije tipova

- Dodajmo klasi complex konstruktor u javnom dijelu koji prima kao argument realni broj:

```
complex (double a):re(a),im(0){}
```

- Što će se sada dogoditi prilikom prethodnog poziva?
- Program uočava da ne postoji metod sabcomp, član klase complex, koji kao argument prima realni broj, već samo kompleksan broj, ali će vidjeti da se na osnovu realnog broja može konstruisati kompleksan broj pa će to i uraditi, odnosno doći će do implicitne konverzije u kompleksni tip podataka.
- Bitno je zapamtiti da do ove implicitne konverzije dolazi samo ako smo mi kreirali odgovarjući kontruktor koji vrši tu konverziju!!!

Eksplicitna konverzija

- Naravno, moguće je kada napravimo konstruktor pozvati i eksplicitnu konverziju, na isti način kao za ugrađene tipove podataka:
  complex(a) ili preko cast operatora (complex)a, gdje je a realni broj.
- Što će se dogoditi ako dođe do poziva:

```
z3=z1.sabcomp(2);
```

- Pretpostavimo da nijesmo napravili konstruktor koji prima cijeli broj. Prilikom izvršavanja programa uočava se ta činjenica, ali se poziva konstruktor za realni broj jer dolazi do standardne konverzije iz cijelog u realni broj.

Statički podaci članovi

- Svi radnici u firmi mogu da imaju maksimalno 40 godina staža, dok se zakon ne promijeni, kada će možda biti dozvoljeno do 42 godine. Svi objekti na crtežu moraju da stanu u granice 20x20cm dok neko ne dozvoli crtanje na većem papiru.
- Očigledno, postoje podaci članovi klasa koje dijele svi objekti tog klasnog tipa.
- Zgodno bi bilo ovakve podatke članove zapisati na jednom mjestu i na jednom mjestu ih eventualno mijenjati za sve objekte toga klasnog tipa.
- Ovakvi podaci članovi se nazivaju zajedničkim podacima članovima klase ili statičkim, jer ih najavljuje ključna riječ static.

Primjer klase sa static

```
class Radnik{
  private:
     int gs;
     static int maxgs;
     //još ponešto
  public:
     int penzija(){
     if(gs>=maxgs) return 1; return 0;}
     //još metoda i slično
  };
  int Radnik::maxgs=40;//inicijalizacija
```

Uočimo da smo inicijalizaciju izvršili van klase. Ovo je obavezno jer bi se drugačije inicijalizacija asocirala samo sa klasom, a ne sa konkretnom aplikacijom (negdje je veličina papira na kojem se nanose objekti predefinisana na A4, a u USA je možda Letter i u drugoj aplikaciji će početna veličina papira biti Letter)!

Napomenimo da za inicijalizaciju ne važe pravila pristupa.

Zajednički član klase – kad i gdje

- Statički član klase postoji samo jedan za čitavu klasu.
- Ako dobro analiziramo slučaj sa inicijalizacijom, mi smo mu pristupili prije nego je kreiran ijedan objekat tog klasnog tipa.
- Stoga zaključujemo da se statički podatak član alocira prije nego ijedan objekat tog klasnog tipa (prvo kreiramo papir, pa zatim po njemu crtamo), te da mu se može pristupiti prije nego se objekti deklarišu!?
- Zajedničkom podatku članu se može pristupiti preko operatora dosega Radnik::maxgs ili se, uz poštovanje prava pristupa, može pristupati preko objekta tog klasnog tipa: Janko.maxgs (ovo je u našem primjeru nedozvoljeno, ali može preko neke funkcije članice za objekat Janko). U svakom slučaju, ono što uradi Janko sa zajedničkim članom klase ima uticaj na sve objekte.

Zajedničke funkcije članice

- Funkcije članice klase se mogu deklarisati kao static (zajedničke za klasu).
- Ova funkcija nije asocirana za objekte, već za klasu, premda se može pozivati i preko objekata, ali ne mora!
- Naime, za pozivanje ovih funkcija se može koristiti naziv klase i operator dosega!
- Ove funkcije nemaju operator this, jer se u principu i ne pozivaju za objekte već za klasu, odnosno i kada se pozivaju za objekat odnose se na klasu!
- Najčešći razlog za kreiranje statičkih funkcija članica je rad sa statičkim podacima članovima.

Statičke funkcije - dodaci

- Za statičke funkcije članice važe neka složena (ali logična) pravila kojih je mali broj i koje nećemo posebno nabrajati.
- Ako se uključi 5% logike ne može se pogriješiti u njihovoj upotrebi (za početak ih koristite samo da radite sa statičkim podacima članovima).
- Npr. ne mogu funkcije imati isto ime i listu argumenata, a jedna biti statička, a druga standardna, jer kompajler odluku o načinu pozivanja funkcije donosi na osnovu argumenata, a ne na osnovu deklaracije i pratećih modifikatora.
- Sva pravila su ukratko opisana i demonstrirana u knjizi od Milićeva, strane 248 i 249.

Pokazivači na podatke članove klase

- Sa uvođenjem pojma klase programski jezik C++ je uveo i specijalni tip podatka: pokazivač na člana klase.
- Postoje i pokazivač na podatak član, kao i pokazivač na funkciju članicu (metod), i ovo drugo se objektivno češće koristi.
- Deklarišimo klasu:

```
class Primjer{
public:
    int a; int b; int c;/*još koješta*/};
```

  Uočite da smo radi primjera deklarisali podatke članove kao javne.
- Sada deklaracija `int Primjer::*px;` deklariše promjenljivu px koja je pokazivač na cijeli broj koji je podatak član klase Primjer.

Pokazivači na podatke članove klase

- Sada je dozvoljeno uzeti adresu podatka člana klase:

```
px=&Primjer::a;
```

- Preko pokazivača možemo pristupati datom (ili nekom drugom) podatku članu klase:

```
Primjer x;
x.*px=3;
```

- Na ovaj način pristupamo zapravo podatku članu a iz objekta x (ekvivalentno x.a=3;). Naravno, ovo je moguće samo ako je a javni podatak član.
- Očigledno, pokazivač na podatak član klase sam po sebi bez objekta ne predstavlja praktično ništa, jer tek sa objektom on predstavlja cjelinu: Pokazivač na podatak član objekta x.

Pokazivači na članove klase

- Pristup podacima članovima preko pokazivača na članove klase je moguć i preko pokazivača na objekte:

```
Primjer *PX=&x;
PX->*px=5;
```

- Postavlja se pitanje kako se pokazivač na člana klase realizuje?
- Suprotno očekivanom, odgovor nije - kao i ostali pokazivači.
- Naime, pokazivač na podatak član klase pamti samo koliko je rastojanje datog podatka člana klase od početka objekta.

Realizacija pokazivača na podatke članove klase

- Mi pokazivač na podatak član klase nazivamo ofsetom, pošto on ukazuje samo na to koliko je posmatrani podatak član klase u memoriju udaljen od početka objekta.
- Stoga se ovom podatku može pristupiti samo kada se zna na koji se objekat odnosi.
- Da napomenemo još jednom, podatku članu klase se preko pokazivača na podatak član klase može pristupiti samo ako je taj podatak član klase javan, a u suprotnom će doći do greške u kompajliranju ako je pristup van klase.
- Pored navedenih, jedina važnija operacija koja se provodi kod pokazivača na podatak član klase je poređenje.

Poređenje pokazivača na podatke članove klase

- Zapamtite da se mogu porediti samo pokazivači na podatke članove koji pripadaju istoj grupi (public ili private sekciji).
- Ako u klasi imate više, recimo, public sekcija **(ovo se dešava kad naknadno shvatite da nešto treba da bude javno, ne morate to prebacivati u postojeću public sekciju, već ga najaviti na datom mjestu)** onda se poređenje može smisleno obaviti samo za podatke u okviru iste sekcije.
- Razlog je u činjenici što ANSI C++ standard ne determiniše na koji način se zauzima memorija za podgrupe podataka članova, pa kompajleri u tom pogledu imaju slobodu **(na jednom može na jedan način, recimo, prvo kasnije navedeni podaci članovi, a na drugom obrnuto).**

Pokazivač na funkcije članice

- Pokazivači na funkcije članice se relativno često koriste **(za razliku od pokazivača na podatke članove).**
- Ako imamo klasu:

```
class Primjer{
public:
int f(int); int g(int); int h(int);};
```

deklaracija: `int (Primjer::*pokfun)(int);`
deklariše pokfun koji je pokazivač na funkciju članicu klase Primjer. Sada: `pokfun=&Primjer::f; Primjer x;` `int j=(x.*pokfun)(3);` je ekvivalentno sa `j=x.f(3);`

- Odgovor na pitanje zbog čega koristiti pokazivače na funkcije članice je, ja se nadam, očigledan, a ako nije pogledajmo naredni slajd.

Upotreba pokazivača na funkcije članice

- Pretpostavimo da nam klasa realizuje neki matematički koncept.
- Neka u zavisnosti od situacije neka operacija sa tim konceptom može da se provodi na različite načine **(npr. računanje integrala različitim približnim, tačnim ili simboličkim formama).**
- Korisnika obično ne interesuje način na koji se to postiže, a i kada ga i interesuje on ne želi da za te semantički iste poslove **(računanje integrala u našem primjeru)** koristi i pamti više funkcija.
- Jedan način da se to uradi je da se kreira funkcija **(članica ili nečlanica? članica je neophodna ako su funkcije koje se žele pozivati privatne)** i da jedan od njenih argumenata bude pokazivač na funkciju koja je član klase. Na ovaj način jedna funkcija izvršava više semantički **(smisleno)** istih uslova, ali koji se realizuju na različite načine.

Strukture

- Ključna riječ struct se zadržala u C++ zbog tradicije, odnosno iz razloga da bi se programi napisani u C-u mogli prekompajlirati u C++ (ponekad se kaže zbog kompatibilnosti unazad).
- Za divno čudo, C++ dozvoljava da strukture u svom sastavu imaju funkcije članice!
- Postavlja se pitanje da li onda postoji ikakva razlika između ključnih riječi struct i class u C++-u?
- Suštinske ne, osim što se kod strukture svi članovi klase podrazumijevaju kao javni (zbog kompatibilnosti unazad), dok su kod klase podrazumijevano privatni.

Strukture i Unije

- Nadamo se da ste shvatili: strukture mogu da imaju privatnu i javnu sekciju, baš kao i klase, ali ako dođete do tih sekcija ipak je preporučljivo da koristite klase **(sad zbog kompatibilnosti unaprijed jer će strukture prije ili kasnije biti izbačene).**
- Još veće iznenađenje je činjenica da unija koja je, što se tiče podataka članova, ista kao ona u C-u, može imati funkcije članice koje su podrazumijevano javne.
- Ostali detalji vezani za unije su isti kao u C-u.