

Objektno orijentisano programiranje 1

Klase i objekti



Klase i objekti – osnovni pojmovi

- Klasa je strukturirani korisnički tip koji obuhvata:
 - podatke koji opisuju stanje objekta klase i
 - funkcije namenjene definisanju operacija nad podacima klase
- Klasa je formalni opis apstrakcije koja ima:
 - internu implementaciju i
 - javni interfejs
- Jedan primerak (instanca, pojava) klase naziva se *objektom* te klase
- Podaci koji su deo klase nazivaju se:
 - *podaci članovi* klase (engl. *data members*), *polja* ili ***atributi***
- Funkcije koje su deo klase nazivaju se:
 - *funkcije članice* (engl. *member functions*), *primitivne operacije*, ***metodi***

Komunikacija objekata

- Jedan od osnovnih principa OO programiranja:
 - objekti klasa komuniciraju (sarađuju) da ostvare složene funkcije
- Poziv metoda se naziva *upućivanjem poruke* objektu klase
- Objekat klase može da menja svoje stanje predstavljeno atributima kada se pozove njegov metod, odnosno kada mu se uputi poruka
- Objekat koji šalje poruku (poziva metod) naziva se objekat-*klijent*
- Objekat koji prima poruku (čiji se metod poziva) je objekat-*server*
- Iz svog metoda objekat može pozivati metod drugog objekta
 - pozvan metod može biti član iste ili druge klase
 - tok kontrole "prolazi kroz" razne objekte koji komuniciraju
- Unutar metoda, članovima objekta čiji je metod pozvan pristupa se navođenjem njihovog imena

Prava pristupa (1)

- Klasa ima sekcije koje određuju pravo pristupa članovima
- Svaka sekcija započinje labelom sekcije (ključna reč)
- Članovi (podaci ili funkcije) klase iza ključne reči `private`:
 - nazivaju se *privatnim* (engl. *private*) članovima:
 - zaštićeni su od pristupa spolja (kapsulirani su)
 - njima mogu pristupati samo metodi klase (i prijatelji)
- Članovi iza ključne reči `public`:
 - nazivaju se *javnim* (engl. *public*) članovima:
 - dostupni su spolja bez ograničenja
- Članovi iza ključne reči `protected`:
 - nazivaju se *zaštićenim* (engl. *protected*) članovima:
 - dostupni su metodima date klase, kao i klasa izvedenih iz te klase
 - nisu dostupni ostalim funkcijama

Prava pristupa (2)

- Redosled sekcija `public`, `protected` i `private` je proizvoljan
 - preporučuje se navedeni redosled
- Može biti više sekcija iste vrste
- Podrazumevana sekcija (do prve labele) je privatna
- Preporuka je da se klase projektuju tako da nemaju javne atribute
- Kontrola pristupa članovima nije stvar objekta, nego klase:
 - metod klase `x` pozvan za objekat `o1` može da pristupi privatnim članovima drugog objekta `o2` iste klase `x`
- Kontrola pristupa članovima je odvojena od koncepta dosega imena:
 - najpre se, na osnovu oblasti važenja imena i njihove vidljivosti, određuje entitet na koga se odnosi dato ime na mestu obraćanja u programu
 - zatim se određuje da li je tom entitetu dozvoljen pristup sa tog mesta

Definisanje klase (1)

- Atributi (podaci članovi, polja):
 - mogu da budu i inicijalizovani (tek od C++11 je to dozvoljeno)
 - ne mogu da budu tipa klase koja se definiše, ali mogu biti pokazivači ili reference na tu klasu
- Metodi (funkcije članice):
 - u definiciji klase mogu da se:
 - deklarishu (navodi se samo prototip) ili
 - definišu (navodi se i telo)
 - funkcije koje su definisane u definiciji klase:
 - mogu pristupati članovima klase direktnim navođenjem imena tih članova
 - jesu `inline` funkcije
 - funkcije koje su samo deklarisanе u definiciji klase:
 - moraju biti definisane kasnije, izvan definicije klase
 - treba proširiti doseg klase (`<ime_klase> : : <ime_funkcije>`) za pristup imenima članova
 - vrednost rezultata, kao i parametri mogu biti tipa klase čiji su članovi kao i tipa pokazivača ili reference na tu klasu

Definisanje klase (2)

- Definicija klase mora biti prisutna tamo gde se klasa koristi
 - zato se definicija klase uobičajeno piše u datoteci zaglavlju (*.h)

- Primer definicije klase `Kompleksni` :

```
class Kompleksni {  
  public:  
    Kompleksni zbir(Kompleksni); Kompleksni razlika(Kompleksni);  
    float re(); float im();  
  private:  
    float real, imag;  
};
```

- Nepotpuna definicija klase (bez tela) se naziva deklaracijom:

```
class Kompleksni;
```

- Pre definicije klase, a posle deklaracije:
 - mogu da se definišu pokazivači i reference na tu klasu
 - ne mogu da se definišu objekti te klase

Objekti (primerenci) klase

- Primerenci klase se definišu na način koji je uobičajen za definisanje objekata standardnih tipova:
 - identifikator klase se koristi kao oznaka tipa, npr. `Kompleksni c;`
- Za svaki objekat klase formira se poseban komplet svih nestatičkih atributa te klase
- Nestatički metodi klase se pozivaju za objekte klase
- Lokalne statičke promenljive metoda
 - su zajedničke za sve objekte date klase
 - žive od prvog nailaska na njihovu definiciju do kraja programa
 - imaju sva svojstva lokalnih statičkih promenljivih globalnih funkcija

Podrazumevane operacije

- Podrazumevano je moguće:
 - definisanje objekata, pokazivača i referenci na objekte i nizova objekata klase
 - dodela vrednosti (operator =) jednog objekta drugom
 - uzimanje adrese objekata (operator &)
 - posredno pristupanje objektima preko pokazivača (operator *)
 - neposredno pristupanje atributima i pozivanje metoda (operator .)
 - posredno pristupanje atributima i metodima preko pokazivača (operator ->)
 - pristupanje elementima nizova objekata (operator [])
 - prenošenje objekata kao argumenata i to po vrednosti, referenci ili pokazivaču
 - vraćanje objekata iz funkcija po vrednosti, referenci ili pokazivaču
- Neke od ovih operacija korisnik može drugačije da definiše
 - preklapanjem operatora (pisanjem odgovarajućih operatorskih funkcija)

Pokazivač `this` (1)

- Unutar svakog nestatičkog metoda
 - implicitni (podrazumevani, ugrađeni) lokalni pokazivač `this`
- Ovaj pokazivač ukazuje na objekat čiji je metod pozvan (tekući objekat)
- Tip ovog pokazivača je "konstantni pokazivač na klasu čiji je metod član"
- Ako je klasa `X`, `this` je tipa `X*const`
- Primer:

```
// definicija metoda zbiri(Kompleksni) klase Kompleksni
Kompleksni Kompleksni::zbiri(Kompleksni c) {
    Kompleksni t=*this; //u t se kopira tekuci objekat
    t.real+=c.real;    t.imag+=c.imag;
    return t;
}
//...
int main() {Kompleksni c,c1,c2;... c=c1.zbiri(c2); ...}
```

Pokazivač `this` (2)

- Pristup članovima objekta čiji je metod pozvan obavlja se neposredno
- Implicitno je to pristup preko pokazivača `this` i operatora `->`
- Može se i eksplicitno pristupiti članovima preko ovog pokazivača unutar metoda

```
// nova definicija metoda zbir(Kompleksni) klase Kompleksni
Kompleksni Kompleksni::zbir(Kompleksni c) {
    Kompleksni t;
    t.real=this->real+c.real;  t.imag=this->imag+c.imag;
    return t;
}
```

- Pokazivač `this` je, u stvari, jedan skriveni argument metoda
- Poziv `objekat.f()` se prevodi u kôd koji ima semantiku kao `f(&objekat)`
- Primeri korišćenja:
 - tekući objekat treba vratiti kao rezultat metoda
 - adresa tekućeg objekta je potrebna kao stvarni argument poziva neke funkcije
 - tekući objekat treba uključiti u listu

Inspektori i mutatori

- Metod može da samo čita stanje ili da menja stanje objekata
- Metod koji ne menja stanje objekta:
 - *inspektor* ili *selektor* (engl. *inspector*, *selector*)
- Metod koji menja stanje objekta:
 - *mutator* ili *modifikator* (engl. *mutator*, *modifier*)
- Dobra praksa je da se korisnicima klase navede da li neki metod menja unutrašnje stanje objekta
- Da je metod inspektor, govori reč `const` iza liste parametara
- Reč `const` je jedan od 4 modifikatora metoda (`volatile`, `& i &&`)
- Sreće se i naziv *konstantan* metod, ali može biti dvoznačan

Deklarisanje inspektora

- Deklarisanje metoda kao inspektora je:
 - samo notaciona pogodnost i "stvar lepog ponašanja"
 - to je obećanje projektanta klase da funkcija ne menja stanje objekta
 - prevodilac nema načina da osigura da inspektor ne menja neke attribute
 - inspektor može da menja attribute uz pomoć eksplicitne konverzije
 - eksplicitna konverzija "probija" kontrolu konstantnosti
- U inspektoru tip pokazivača `this` je `const X*const`
 - pokazivač `this` pokazuje na nepromenljivi objekat
- Nije moguće menjati objekat preko pokazivača `this`
 - svaki neposredni pristup članu je implicitni pristup preko `this`
- Za nepromenljive objekte klase nije dozvoljeno pozivati metod koji nije deklarisan da je inspektor

Inspektori – primer

```
class X {
    public:
        int citaj () const    { return i; }
        int pisi (int j=0)   { int t=i; i=j; return t; }
    private:
        int i;
};

X x; const X cx;
x.citaj(); // u redu: inspektor promenljivog objekta
x.pisi(); // u redu: mutator promenljivog objekta
cx.citaj(); // u redu: inspektor nepromenljivog objekta
cx.pisi(); // ! GRESKA: mutator nepromenljivog objekta
```

Nepostojani metodi

- Suprotnost konstantnog metoda je nepostojan (*volatile*) metod

```
int f() volatile { /*...*/ }
```
- Pri izvršenju nepostojanog metoda `f` objekat (`*this`) može promeniti stanje nezavisno od toka funkcije `f`
- Modifikator `volatile` – napomena prevodiocu da ne vrši neke optimizacije

```
class X {  
public:  
    X(){kraj=false;}  
    int f() volatile { // da nije volatile, moguća optimizacija:  
        while(!kraj){ /*...*/ } // if (!kraj) while() { /*...*/ }  
    } // u telu (...) se ne menja kraj  
    void zavrшено(){kraj=true;}  
private:  
    bool kraj;  
}
```

- Metod `volatile` može da se poziva za nepostojane i promenljive objekte
- Metod `const volatile` – može se pozvati za sve vrste objekata:
 - promenljive, nepromenljive i nepostojane objekte

Modifikatori metoda & i &&

- Bez modifikatora & ili && metod se može primeniti na *lvrednost* i *dvrednost*
- Modifikator & – tekući objekat može biti samo *lvrednost*
- Modifikator && – tekući objekat može biti samo *dvrednost*
- Ovi modifikatori su, kao i `const` i `volatile`, deo potpisa metoda
 - mogu da postoje metodi čiji se potpisi razlikuju samo po ovom modifikatoru
- Primer:

```
class U {public:
    int f() & {return 1;}
    int f() const & {return 2;}
    int f() && {return 3;}
};
U u1;                const U u2=u1;
int i = u1.f();      int j = u2.f();           int k = U().f();
```


Pojam konstruktora

- *Konstruktor*:
 - specifična funkcija klase koja definiše početno stanje objekta
 - nosi isto ime kao i klasa
 - nema tip rezultata (čak ni `void`)
 - ima proizvoljan broj parametara proizvoljnog tipa
 - osim tipa klase čiji je konstruktor
 - dozvoljen je tip pokazivača i referenci na *lvrednost* i *dvrednost* date klase
- Konstruktor se implicitno poziva prilikom kreiranja objekta
- Pristup članovima objekta unutar tela konstruktora
 - kao i u bilo kojem drugom metodu (preko pokazivača `this`)
- Konstruktor, kao i svaki metod, može biti preklopljen
 - važe ista opšta pravila kao za preklapanje funkcija

Podrazumevani konstruktor

- *Podrazumevani konstruktor (PK):*
 - konstruktor koji se može pozvati bez stvarnih argumenata
 - nema parametre ili
 - ima sve parametre sa podrazumevanim vrednostima
- Ugrađeni (implicitno definisan) podrazumevani konstruktor je:
 - bez parametara i
 - ima pazno telo
- Ugrađeni PK postoji samo ako klasa nije definisala ni jedan konstruktor
 - definisanjem nekog konstruktora se suspenduje (briše) ugrađeni PK
 - može da se restaurira navođenjem deklaracije iza koje stoji `=default`
- Kada se kreira niz objekata neke klase, poziva se PK za svaki element
 - redosled poziva PK elemenata je po rastućem redosledu indeksa

Pozivanje konstruktora

- Konstruktor se poziva uvek kada se kreira objekat klase:
 - kada se izvršava definicija statičkog objekta
 - kada se izvršava definicija automatskog objekta
 - kada se stvara dinamički objekat operatorom `new`
 - kada se stvarni argument klasnog tipa prenosi u formalni (parametar se inicijalizuje stvarnim argumentom)
 - kada se kreira privremeni objekat, pri povratku iz funkcije (inicijalizuje se izrazom iz `return` naredbe)

Argumenti konstruktora

- Pri stvaranju objekta moguće je navesti inicijalizator iza imena
- Inicijalizator sadrži listu argumenta konstruktora u zagradama
 - zagrade mogu biti `()` ili `{ }`
 - ako se lista navodi u zagradama `{ }`, može se pisati i `= { ... }`
 - nisu dozvoljene prazne zagrade `()`
 - to bi se prevelo kao deklaracija f-je
- Moguća je i notacija `<objekat>=<vrednost>`
 - ukoliko se konstruktor može pozvati sa samo jednim argumentom
- Poziva se onaj konstruktor koji se najbolje slaže po potpisu
 - kao kod preklapanja imena funkcija
- Konstruktor može da ima podrazumevane vrednosti argumenata

Argumenti konstruktora – primer

```
class X {
    char a; int b;
public:
    X ();
    X (char, int=0);
    X (const char*);
    X(X); // ! GRESKA
    X(X*);
    X(X&);
    X(X&&);
};

X f () {
    X x1; // X()
    X x2{}; // X()
    X x3={}; // X()
    X x(); // dekl. f-je
    return x1;
}
```

```
void g () {
    char c='a';
    const char *p="Niska";
    X x1(c); // X(char,int)
    X x2=c;
    X x3(c,10);
    X x4{c,20};
    X x5={c,30};
    X x6(p); // X(char*)
    X x7(x1); // X(X&)
    X x8{x1};
    X x9={x1};
    X x10=f(); // X(X&&)
    X* p1=new X; // X()
    X* p2=new X();
    X* p3=new X(c); // X(char,int)
    X* p4=new X{c,10};
}
```

Konstrukcija članova (1)

- Pre izvršavanja tela konstruktora
 - inicijalizuju se atributi prostih tipova
 - pozivaju se konstruktori atributa klasnih tipova
- Inicijalizatori mogu da se navedu i u zaglavlju definicije (ne deklaracije) konstruktora, iza znaka :
- Ako atribut ima inicijalizator u telu klase i u definiciji konstruktora, primenjuje se inicijalizator iz definicije konstruktora

```
class YY { public: YY (int j) {...} };
```

```
class XX {  
    YY y; int i=0;  
public:  
    XX (int);  
};
```

```
XX::XX (int k) : y(k+1), i(k-1) {...} // y=k+1, i=k-1
```

- Inicijalizacija atributa – redosledom navođenja u klasi
 - bez obzira da li su primitivnog ili klasnog tipa
 - bez obzira na redosled u listi inicijalizatora

Konstrukcija članova (2)

- Navođenje inicijalizatora u zaglavlju definicije konstruktora predstavlja specifikaciju *inicijalizacije* članova
- Inicijalizacija je različita od *operacije dodele* koja se može vršiti jedino unutar tela konstruktora
- Inicijalizacija je neophodna:
 - kada ne postoji podrazumevani konstruktor klase atributa
 - kada je atribut nepromenljiv podatak
 - kada je atribut referenca
- Do C++11 nije bila dozvojena inicijalizacija atributa u def. klase
 - jedini način inicijalizacije je bio kroz inicijalizatore u def. konstruktora

Primer konstrukcije

- Primer konstrukcije dva objekta od kojih jedan sadrži drugi:
 - objekat klase `Kontejner` sadrži po vrednosti objekat klase `Deo`, pri čemu objekat `deo` treba da "zna" ko ga sadrži

```
class Kontejner {
    public:
        Kontejner () : deo(this) {...}
    private:
        Deo deo;
};

class Deo{
    public:
        Deo(Kontejner* kontejner) :mojKontejner(kontejner) {...}
    private:
        Kontejner* mojKontejner;
};
```


Delegirajući konstruktor

- U listi inicijalizatora definicije delegirajućeg konstruktora može da se navede poziv drugog (ciljnog) konstruktora iste klase
- Tada se pre izvršenja tela delegirajućeg konstruktora izvršava ciljani
- Kad se navodi ciljani konstruktor, u listi inicijalizatora sme da postoji samo on
- Ako dolazi do neposrednog ili posrednog rekurzivnog delegiranja – greška
 - ne otkriva prevodilac, ulazi se u beskonačnu rekurziju
- Primer:

```
class T {  
    T(int i) {}  
    T():T(1) {} // delegirajuci: T(), ciljani: T(int)  
    T(char c): T(0.5) {} //! GRESKA - rekurzija  
    T(double d): T('a') {}  
}
```

Eksplicitni poziv konstruktora

- Konstruktor se može pozvati i eksplicitno u nekom izrazu
- Takav poziv kreira privremeni objekat klase pozivom odgovarajućeg konstruktora sa navedenim argumentima
- Isto se dešava ako se u inicijalizadoru objekta eksplicitno navede poziv konstruktora:

```
int main () {  
    Kompleksni c1(1,2.4),c2;  
    c2=c1+Kompleksni (3.4,-1.5);        // privremeni objekat  
    Kompleksni c3=Kompleksni (0.1,5); // priv. obj. se kopira u c3  
}
```

Konstruktor kopije (1)

- Drugi naziv – kopirajući konstruktor
- Pri inicijalizaciji objekta `x1` drugim objektom `x2` iste klase `X`
 - poziva se konstruktor kopije (KK, engl. *copy constructor*)
- Ugrađeni, implicitno definisani konstruktor kopije
 - vrši inicijalizaciju članova `x1` članovima `x2` (pravi plitku kopiju)
 - primitivni atributi (uključujući pokazivače) se prosto kopiraju
 - za klasne attribute se pozivaju njihovi KK
- Ugrađeni KK klase `X` se briše (suspenduje)
 - eksplicitno:
 - `X(const X&)=delete; // zabrana kopiranja`
 - implicitno:
 - pisanjem premeštajućeg konstruktora ili premeštajućeg operatora dodele
 - može se restaurirati pomoću: `X(const X&)=default`

Konstruktor kopije (2)

- Inicijalizacija ugrađenim KK ponekad nije zadovoljavajuća
 - ako objekti sadrže podobjekte po adresi (pokazivaču ili referenci)
 - tada programer treba da ima potpunu kontrolu nad inicijalizacijom celog objekta drugim objektom iste klase
- Za pravljenje duboke kopije potrebno je napisati KK
- Konstruktor kopije ima parametar tipa `XX&` ili `const XX&`
 - konstruktor ne sme imati parametar tipa svoje klase
- Ostali eventualni parametri kopirajućeg konstruktora moraju imati podrazumevane vrednosti

Pozivanje konstruktora kopije

- Konstruktor kopije se poziva sa jednim stvarnim argumentom
 - argument je tipa klase čiji se objekat stvara kopiranjem
- KK se poziva kada se objekat inicijalizuje objektom iste klase i to:
 - prilikom stvaranja trajnog, automatskog, dinamičkog ili privremnog obj.
 - oblik inicijalizatora (zagrada `()` ili `{}`, simbol `=`) je nebitan
 - prilikom prenosa arg. po vrednosti u funkciju (stvara se automatski obj.)
 - prilikom vraćanja vrednosti iz funkcije (stvara se privremeni objekat)
- Prevodilac sme da preskoči poziv KK (radi optimizacije)
 - ako se stvarani objekat inicijalizuje privremenim objektom iste klase
 - neprijatno, jer izostaju i bočni efekti – program nije prenosiv
 - čak i u ovom slučaju mora postojati KK ili premeštajući konstruktor

Konstruktor kopije – primer

```
class XX {
public:
    XX (int);
    XX (const XX&);    // konstruktor kopije
    //...
};
XX f(XX x1) {
    XX x2=x1;        // poziv konst. kopije XX(XX&) za x2
    return x2;      // poziv konst. kopije za privremeni
}                  // objekat u koji se smešta rezultat
void g() {
    XX xa=3, xb=1;
    xa=f(xb);       // poziv konst. kopije samo za parametar x1,
                    // a u xa se samo prepisuje
                    // privremeni objekat rezultata, ili se
}                  // poziva XX::operator= ako je definisan
```

Premeštajući konstruktor (1)

- Konstruktor koji se poziva za konstrukciju objekata istog tipa, pri čemu je izvorišni objekat na kraju životnog veka
 - izvorišni objekat je *nvrednost*
 - *nvrednost* – nestajuća vrednost (*xvalue*, od *expiring value*)
 - izvorišni objekat ne mora da se sačuva
 - mogu se samo “premestiti” njegovi dinamički delovi u odredišni objekat
 - nema kopiranja dinamičkih delova izvorišnog objekta,
 - dovoljna je plitka kopija
 - posledica je da je premeštajući konstruktor efikasniji od kopirajućeg
 - treba samo modifikovati izvorišni objekat
 - na način da njegovo uništavanje ne povuče razaranje već premeštenih delova u odredišni objekat

Premeštajući konstruktor (2)

- Postoji ugrađeni, implicitno definisan, premeštajući konstruktor
- Ugrađeni premeštajući konstruktor pravi plitku kopiju originala
 - za attribute tipa klase – zovu se njihovi premeštajući konstruktori
 - za polja tipa pokazivača – kopiraju se samo njihove vrednosti
- U polja pokazivača izvorišta treba eksplicitno postaviti `nullptr`
 - razlog: da se spreči uništavanje premeštenih delova u novi objekat
 - za objekte koji sadrže delove po adresi
 - nije dovoljan ugrađeni konstruktor
- Ugrađeni premeštajući konstruktor se suspenduje (briše), ako se eksplicitno definiše:
 - premeštajući konstruktor
 - kopirajući konstruktor
 - destruktor
 - operator dodele

Pozivanje premeštajućeg konst.

- Parametar premeštajućeg konst. je referenca na *dvrednost*: `XX&&`
 - kao kod KK, ostali parametri moraju imati podrazumevane vrednosti
- Prevodilac će pozvati premeštajući konstruktor
 - ako izvorišni objekat nestaje
 - ako u klasi postoji (ugrađeni ili napisani) premeštajući konstruktor
- Ako u klasi ne postoji premeštajući konstruktor
 - poziva se kopirajući konstruktor
 - semantika nije promenjena
 - samo može biti promene u brzini izvršavanja

Premeštajući konstruktor – primer

- Primer:

```
class Niz {
    double* a; int n;
public:
    ...
    Niz(Niz&& niz) {a=niz.a; niz.a=nullptr; n=niz.n;}
} ...
Niz f(Niz niz) {return niz;}
```

- Kopiranje `a` i `n` bi uradio i ugrađeni premeštajući konstruktor
- Ugrađeni konstruktor ne bi postavio `niz.a=nullptr`;
- Funkcija `f` vraća objekat tipa `Niz` po vrednosti,
pa se poziva premeštajući konstruktor
 - razlog: `niz` kao lokalni automatski objekat (na steku) nestaje
 - nestajućim objektom `niz` se inicijalizuje privremeni objekat rezultata `f`

Konverzioni konstruktor (1)

- Konverzija između tipova od kojih je barem jedan klasa
 - definiše je korisnik
- Jedna mogućnost konverzije tipova
 - pomoću *konstruktor konverzije*
 - određišni tip mora biti klasa
- Ako u klasi U postoji $U::U(T\&)$, ili $U::U(T)$ gde je T klasa ili standardni tip:
 - vrednost izraza $U(t)$, gde je t tipa T , je privremeni objekat tipa U
 - to predstavlja konverziju iz tipa T u tip U
- Korisničke konverzije će se primenjivati automatski (implicitno)
 - ako je jednoznačan izbor konverzije
 - izuzev u slučaju *explicit* konstruktora
- Automatska konverzija mora biti neposredna:
 - za $U::U(T\&)$ i $V::V(U\&)$ moguće je samo eksplicitno $V(U(t))$

Konverzioni konstruktor (2)

- Pomoću konv. konst. nije moguća konverzija u standardni tip:
 - ovaj nije klasa za koju korisnik može da definiše konverzioni konst.
- Druga mogućnost definisanja konverzija
 - preklapanje *kast operatora*
- Konverzija argumenata i rezultata funkcije
 - pri pozivu funkcije:
 - inicijalizuju se parametri stvarnim argumentima, uz eventualnu konverziju tipa
 - parametri se ponašaju kao automatski lokalni objekti pozvane funkcije
 - ovi objekti (ako su klasnih tipova) se konstruišu pozivom odgovarajućih konstruktora
 - pri povratku iz funkcije:
 - konstruiše se privremeni objekat koji prihvata vrednost `return` izraza na mestu poziva

Primer konverzije arg. i rez. f-je

```
class T {  
public:  
    T(int i);           // konstruktor  
};  
T f (T k) {  
    //...  
    return 2;         // poziva se konstruktor T(2)  
}  
int main () {  
    T k(0);  
    k=f(1);           // poziva se konstruktor T(1)  
    //...  
}
```

Konstante klasnog tipa

- Kao i globalne f-je i metodi – i konstruktor može biti “konstantan”
 - modifikator deklaracije i definicije `constexpr`
 - prazno telo (C++14 dozvoljava naredbe koje su konstantni izrazi)
 - za sve attribute inicijalizatori moraju biti konstantni izrazi
 - za attribute klasnog tipa mora postojati konstantni konstruktor
- Ugrađeni podrazumevani konstruktor
 - konstantan – ako za attribute klase postoji konstantna inicijalizacija
- Za restaurirani podrazumevani konstantni konstruktor
 - nije potreban `constexpr`
- Konstantan objekat se stvara
 - konstantnim konstruktorom
 - sa konstantnim argumentima
- Konstantan objekat se koristi efikasno kao i konstanta

Konstantni objekti – primer (1)

```
class A{
    int a=1, b=2; // konstantni inicijalizatori
public:
    constexpr A(int x, int y): a(x+y), b(x-y){}
    A()=default; // restauracija podrazumevanog k.
};

constexpr A a1(3,4); // konstantan objekat
int k=1;
constexpr A a2(k,k+1); // ! GRESKA
        A a3(k,k+1); // promenljiv objekat
constexpr A a4(a1); // postoji ugrađeni kopirajući k.
constexpr A a5; // postoji restauriran podrazumevani k.
```

Konstantni objekti – primer (2)

- Nastavak:

```
class Beta {  
    Alfa a; int b;  
public:  
    constexpr Beta(int x) :b(2*x) {}  
    Beta()=default;  
}  
...  
constexpr Beta b1(3);  
constexpr Beta b2(b1);  
constexpr Beta b3; // !GRESKA - PK nije konstantan
```


Destruktor

- *Destruktor:*
 - specifična funkcija članica klase koja "razgrađuje" objekat
 - nosi isto ime kao klasa, uz znak ~ ispred imena
 - nema tip rezultata i ne može imati parametre
 - može da postoji najviše jedan u klasi
- Destruktor se piše kada treba osloboditi (dealocirati)
 - memoriju koju je konstruktor zauzeo (alocirao)
 - druge (nememorijske) resurse
- Čest slučaj potrebe za destruktorom
 - kada klasa sadrži članove koji su pokazivači ili reference na sadržane podobjekte (sadržanje delova po adresi)
 - dobra praksa: metod za uništavanje delova koji se poziva iz destruk.
- U destrukturu se članovima pristupa kao i u nekom metodu
 - preko pokazivača `this` (implicitno ili eksplicitno)

Pozivanje destruktora (1)

- Destruktor se implicitno poziva na kraju životnog veka objekta

```
class X {  
public:  
    ~X () { cout<<"Poziv destruktora klase X!\n"; }  
}  
int main () { X x; //...  
} // ovde se poziva destruktor objekta x
```

- Pri uništavanju dinamičkog objekta pomoću operatora `delete`
 - destruktor se implicitno poziva pre oslobađanja memorije
- Pri uništavanju dinamičkog niza
 - destruktor se poziva za svaki element niza, po opadajućem indeksu
- Redosled poziva destruktora
 - uvek obrnut od redosleda poziva konstruktora

Pozivanje destruktora (2)

- Destruktor može i eksplicitno da se pozove
 - `x.~X()` ili
 - `px->~X()` ili
 - unutar metoda klase `X`: `this->~X()`
- Eksplicitan poziv se ne preporučuje
 - poziv ima efekat kao i poziv metoda datog objekta
 - objekat nastavlja da živi
- Posle izvršavanja tela automatski pozvanog destruktora
 - automatski se oslobađa memorija koju je sam objekat zauzimao

Primer konstruktora i destruktora

```
class Tekst {
public:
    Tekst() { niska=nullptr; }           // podrazumevani konstruktor
    Tekst(const char*);                 // konverzioni konstruktor
    Tekst(const Tekst&);                // kopirajuci konstruktor
    Tekst(Tekst&&);                     // premestajuci konstruktor
    ~Tekst();                           // destruktork
private:
    char *niska;
};
#include <cstring>
using namespace std;
Tekst::Tekst(const char* t) {
    niska=new char [strlen(t)+1]; strcpy(niska,t);
}
Tekst::Tekst(const Tekst& t) {
    niska=new char [strlen(t.niska)+1]; strcpy(niska,t.niska);
}
Tekst::Tekst(Tekst&& t) { niska=t.niska; t.niska=nullptr; }
Tekst::~~Tekst() { delete [] niska; niska=nullptr; }
int main () {
    Tekst a("Pozdrav!"), b=a; // Tekst(const char*); Tekst(const Tekst&);
}
```

Statički (zajednički) atributi

- Pri stvaranju objekata klase
 - za svaki objekat - poseban komplet nestatičkih atributa
- Atribut može biti deklarisan kao *statički*
 - pomoću modifikatora `static`
- Postoji samo jedan primerak statičkog atributa za celu klasu
 - zajednički podatak za sve objekte klase – objekti ga dele

```
class X { private:  
    static int i; // samo jedan za sve objekte  
    int j; // svaki objekat ima svoj j  
};
```

- Primeri:
 - klasa definiše elemente unikatne liste objekata
 - glava liste je zajednički član klase
 - brojač stvorenih objekata klase (inkrementira se u konstruktorima)

Definisanje statičkog atributa

- U klasi se statički atribut samo deklariše
- Mora da se definiše na globalnom nivou
 - izvan definicije klase i svih funkcija
- Čak se i privatni statički atributi moraju definisati na taj način
- Dozvoljeni su svi oblici inicijalizatora
- Inicijalizacija statičkog atributa se obavi
 - pre prvog pristupa njemu i pre stvaranja objekta date klase
- Obraćanje statičkom atributu van klase vrši se preko operatora `::`
`int X::i=5; // obavezno bez static`
- Ako se ne navede inicijalizator u definiciji, podrazumeva se nula
- Imenovana celobrojna konst. se može definisati i u definiciji klase

Statički atributi - primer

```
class X {  
    static int psa=10;        // ! GRESKA - promenljivi stat. atr.  
    static const int ick=10;    // imenovana celobrojna konst.  
    static const double irk=10.0; // ! GRESKA - mora celobrojno  
    static const int nsa;        // nepromenljivi stat. atr.  
};  
int X::nsa = 10;
```

Statički atributi i globalni podaci

- Sličnosti sa globalnim podacima:
 - trajni podaci (sličan životni vek)
 - moraju se definisati na globalnom nivou
- Razlike od globalnih podataka:
 - statički atributi logički pripadaju klasi
 - doseg imena statičkih atributa je klasa
 - statičkim atributima je moguće ograničiti pristup
- Statički atribut ima sva svojstva glob. statičkog podatka osim dosega imena i kontrole pristupa
- Statički članovi smanjuju potrebu za globalnim objektima

Statički atributi – primeri upotrebe

- Primer: glava jedinstvene liste

```
class Element {  
public:  
    static Element *glava;  
    //..  
private:  
    int vrednost;  
    Element *sledeci;  
};
```

- Primer: vođenje evidencije o broju kreiranih primeraka

```
class Objekat {  
public:  
    Objekat () {brojac++;}  
    //..  
private:  
    static int brojac;  
};
```

Statički (zajednički) metodi (1)

- Statički metodi su funkcije klase, a ne svakog posebnog objekta
- Metodi su “zajednički” za sve objekte klase
- Primena:
 - za opšte usluge klase
 - prvenstveno za obradu statičkih atributa
- Deklarišu se dodavanjem modifikatora `static` ispred deklaracije
- Imaju sva svojstva globalnih funkcija osim doseg a i kontrole pristupa
- Ne poseduju pokazivač `this`
 - ne mogu neposrednim imenovanjem pristupati nestatičkim članovima
 - modifikator `const` i drugi iza liste parametara nemaju smisla

Statički (zajednički) metodi (2)

- Mogu pristupati nestatičkim članovima konkretnog objekta
 - parametra
 - lokalnog
 - globalnog
- Mogu neposredno pristupati samo statičkim članovima klase
- Pozivaju se za klasu:
 - `<klasa> :: <poziv>`
- Mogu se pozivati i za konkretan objekat (što treba izbegavati)
 - levi operand izraza se ne izračunava
- Statički metodi se mogu pozivati i pre stvaranja objekata klase

Statički metodi – primer (1)

```
class X {
    static int x;    // staticki atribut
    int y;

public:
    static int f(X); // staticki metod (deklaracija)
    int g();
};

int X::x=5;        // definicija statickog atributa

int X::f(X x1){    // definicija statickog metoda
    int i=x;       // pristup statickom atributu X::x
    int j=y;       // ! GRESKA: X::y nije staticki
    int k=x1.y;    // ovo moze;
    return x1.x;   // i ovo moze, ali nije preporucljivo
}                  //      izraz "x1" se ne izracunava
```

Statički metodi – primer (2)

```
int X::g () {
    int i=x;           // nestaticki metod moze da koristi
    int j=y;           // i staticke i nestaticke attribute
    return j;         //      y je ovde this->y;
}

int main () {
    X xx;
    int p=X::f(xx);   // X::f moze neposredno, bez objekta;
    int q=X::g();     // ! GRESKA: za X::g mora konkretan objekat
    xx.g();           // ovako moze;
    p=xx.f(xx);       // i ovako moze, ali nije preporucljivo
}
```

Statički metodi – primeri upotrebe

- Primeri:
 - dohvaćanje glave ili broja članova jedinstvene liste
 - dohvaćanje broja stvorenih objekata klase koja ima njihov brojač
 - uslužna klasa (svi statički metodi, obrisani ugrađeni konstruktori)
- Ograničenje na kreiranje isključivo dinamičkih objekata neke klase:

```
class X {  
public:    static X* kreiraj () { return new X; }  
private: X();           // konstruktor je privat  
};  
int main() {  
    X x;                // ! GRESKA  
    X* px=X::kreiraj(); // moze  
}
```

- Primena:
 - funkcija `kreiraj` može ograničiti broj kreiranih objekata klase

Prijatelji klasa

- Ponekad je potrebno da klasa ima i "povlašćene" korisnike koji mogu da pristupaju njenim privatnim članovima
- Povlašćeni korisnici mogu biti:
 - funkcije (globalne ili metodi drugih klasa) ili
 - cele klase
- Takve funkcije i klase nazivaju se *prijateljima* (eng. *friends*).
- Prijateljstvo, kao relacija između klasa:
 - se ne nasleđuje
 - nije simetrična relacija
 - nije tranzitivna relacija
- Prijateljstvo je relacija koja reguliše isključivo pravo pristupa, a ne oblast važenja i vidljivost identifikatora

Prijateljske funkcije (1)

- Prijateljske funkcije su funkcije koje
 - nisu članice klase, ali
 - imaju dozvoljen pristup do privatnih članova klase
- Prijateljske funkcije mogu da budu:
 - globalne funkcije ili
 - metodi drugih klasa
- Funkcija je prijateljska:
 - ako se u definiciji klase navede njena
 - deklaracija ili
 - definicija
 - sa modifikatorom `friend`
- Potrebno je da klasa eksplicitno proglasi funkciju prijateljskom

Prijateljske funkcije (2)

- Ako se u definiciji klase navodi definicija prijateljske funkcije
 - podrazumeva se `inline`
 - ime funkcije nema klasni doseg, već doseg identifikatora klase
- Nevažno je pravo pristupa sekciji klase (privatno, zaštićeno, javno) u kojoj se navodi deklaracija prijateljske funkcije
- Prijateljska funkcija nema pokazivač `this` na objekat klase kojoj je prijatelj
 - modifikator (npr. `const`) nema smisla za globalnu prijateljsku funkciju
- Funkcija može da bude prijatelj većem broju klasa istovremeno

Prijateljske funkcije – primer

```
class X {
    friend void g(int, X&);    // prijateljska globalna funkcija
    friend void Y::h();       // prijateljski metod druge klase
    friend int o(X x){return x.i;} // definicija globalne f-je
    friend int p(){return i;} // ! GRESKA - nema this
    int i;
public:
    void f(int ip) {i=ip;}
};
void g (int k, X &x) { x.i=k; }
int main () {
    X x; int j;
    x.f(5);    // postavljanje preko metoda
    g(6,x);    // postavljanje preko prijateljske funkcije
    j=o(x);    // citanje preko prijateljske funkcije
}
```

Prijateljske funkcije i metodi

- Ponekad su globalne prijateljske funkcije pogodnije od metoda:
 - metod mora da se pozove za objekat date klase, dok globalnoj funkciji može da se dostavi i objekat drugog tipa
 - nije moguća konverzija skrivenog argumenta u drugi tip
 - kada funkcija treba da pristupa privatnim članovima više klasa, prijateljska globalna funkcija je “simetrično” rešenje
 - nisu potrebni javni metodi u “ostalim” klasama
 - ponekad je notaciono pogodnije da se koriste globalne funkcije ($f(x)$) nego metodi ($x.f()$);
 - na primer: $\max(a, b)$ je čitljivije od $a.\max(b)$;
 - kada se preklapaju operatori, često je jednostavnije definisati globalne (operatorske) funkcije nego metode

Prijateljske klase

- Ako su svi metodi klase Y prijateljske funkcije klasi X, onda je Y prijateljska klasa (*friend class*) klasi X

```
class X {  
    friend Y; //ako je klasa Y definisana ili deklarirana  
    friend class Z; //ako Z jos nije ni def. ni dek.  
};
```

- Svi metodi klase Y mogu da pristupaju privatnim članovima klase X
- Prijateljske klase se tipično koriste kada neke dve klase imaju tešnje međusobne veze

- Primer:

– obezbeđuje da samo objekat `Fabrika` može da stvara objekte `Proizvod`

```
class Proizvod {  
    private:  
        friend class Fabrika;  
    Proizvod(); // konstruktor je dostupan samo klasi Fabrika  
};
```

Strukture

- Struktura je klasa kod koje su svi članovi podrazumevano javni
 - to se može promeniti eksplicitnim umetanjem `public:` i `private:`

```
struct a {          isto što i:          class a {
    //...
private:
    //...
};

                                public:
                                //...
                                private:
                                //...
};
```

- Na C++ struktura može imati i metode
- Struktura se tipično koristi:
 - za definisanje strukturiranih podataka koji ne predstavljaju apstrakciju, odnosno nemaju bitno ponašanje (nemaju značajnije operacije)
- Strukture tipično poseduju samo konstruktore i eventualno destruktore

Ugneždene klase (1)

- Klase mogu da se deklariraju ili definišu unutar definicije druge klase
- Ugnežđivanje se koristi kada neki tip (npr. klasa) semantički pripada samo datoj klasi
 - povećava se čitljivost programa
 - smanjuje se potreba za globalnim tipovima
- Unutar definicije klase se mogu navesti i
 - definicije nabiranja (`enum`) i tipova (`typedef`)
- Ugneždjena klasa (tip) se nalazi u dosegu imena okružujuće klase
 - izvan okružujuće klase imenu ugneždjene klase se može pristupiti samo preko operatora proširenja doseg imena `::`
- Iz okružujuće klase do članova ugneždjene moguć je samo pristup:
 - pomoću operatora `..`, `-> i ::`
- Doseg imena okružujuće \cup se proteže na ugneždenu klasu \cup
 - ali i iz \cup do članova objekta \circ je moguć samo pristup preko `. i ->`

Ugneždene klase (2)

- U ugneždenoj klasi mogu direktno da se koriste samo identifikatori:
 - tipova iz okružujuće klase
 - konstanti tipa nabiranja iz okružujuće klase
 - statičkih članova iz okružujuće klase
- To važi ako ime nije sakriveno imenom člana ugneždene klase
- Pristup statičkom članu ugneždene klase izvan okružujuće:
<id okružujuće> : : <id ugneždene> : : <id statičkog člana ugneždene>
- Ugneždjena klasa je implicitno prijatelj okružujuće klase
 - ima pravo pristupa privatnoj sekciji okružujuće
- Okružujuća klasa nije prijatelj ugneždene
 - mora se eksplicitno proglasiti prijateljem da bi mogla da pristupi privatnoj sekciji

Ugneždene klase – primer

```
int x,y;
class Spoljna {
public:
    int x; static int z;
    class Unutrasnja {
        void f(int i, Spoljna *ps) {
            x=i; // ! GRESKA: nepoznat objekat klase Spoljna
            Spoljna::x=i; // ! GRESKA: isti uzrok
            z=i; // pristup statičkom članu Spoljna
            ::x=i; // pristup globalnom x;
            y=i; // pristup globalnom y;
            ps->x=i; // pristup Spoljna::x objekta *ps;
        }
    };
};

Unutrasnja u; // ! GRESKA: nije u dosegu
Spoljna::Unutrasnja u; // u redu;
```


Lokalne klase

- Lokalne klase se definišu unutar funkcija
- Identifikator lokalne klase ima doseg (oblast važenja) od deklaracije do kraja bloka u kojem je deklarisan
- Unutar lokalne klase iz okružujućeg dosega je dozvoljeno samo korišćenje:
 - identifikatora tipova
 - konstanti tipa nabiranja
 - trajnih podataka (statičkih lokalnih i globalnih)
 - spoljašnjih (eksternih) podataka i funkcija
- Metodi lokalne klase moraju da se definišu unutar definicije klase
- Lokalna klasa ne može da ima statičke (zajedničke) atribute

Lokalne klase – primer

```
int x;
void f() {
    static int s;
    int x;
    extern int g();
    class Lokalna {
    public:
        int h () { return x; } // ! GRESKA: x je automatska prom.
        int j () { return s; } // OK: s je staticka promenljiva
        int k () { return ::x; } // OK: x je globalna promenljiva
        int l () { return g(); } // OK: g() je spoljasnja funkcija
    };
}

Lokalna *p = 0; // ! GRESKA: nije u doseg
```

Pokazivači na članove klase

- Dodelom vrednosti pokazivaču na članove klase označi se neki član klase
- Analogija: kao što se indeksom označi komponenta niza
- Pokazivač na članove klase se definiše:
`<tip člana klase> <klasa> : : * <identifikator pokazivača člana>`
- Formiranje adrese članova klase:
`<identifikator pokazivača člana> = &<klasa> : : <član klase>`
- Pristup članovima pomoću pokazivača na članove klase:
`<objekat klase> . * <identifikator pokazivača člana>` ili
`<pokazivač na objekat klase> -> * <identifikator pokazivača člana>`
- Operatori `. *` i `-> *` su prioriteta 14 i asocijativnost je sleva-udesno

Pokazivači na članove klase – primer

```
class Alfa {... public: int a, b; };
```

```
int Alfa::*pc;// pc je pokazivac na int clanove klase Alfa  
Alfa alfa,*beta;  
beta=&alfa;
```

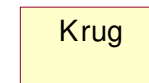
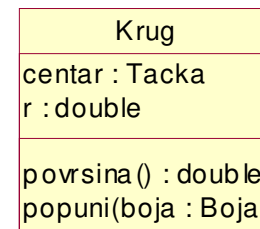
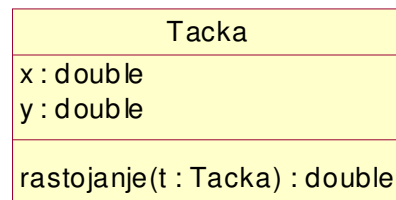
```
pc=&Alfa::a;// pc pokazuje na clanove a objekata klase Alfa  
alfa.*pc = 1;      // alfa.a=1;  
beta->*pc = 1;     // beta->a=1;
```

```
pc=&Alfa::b;// pc pokazuje na clanove b objekata klase Alfa  
alfa.*pc = 2;     // alfa.b=2;  
beta->*pc = 2;     // beta->b=2;
```

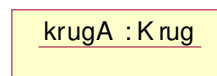
Grafička notacija (UML)

- UML – Unified Modeling Language
 - grafička notacija za modeliranje softvera

- Klasa:



- Objekat:



Relacije

- Asocijacija



- Zavisnost



- Agregacija



- Kompozicija

