

# OOP1

## Objektno-orientisano programiranje 1

Veljko Selaković

prof. dr Igor Tartalja  
prof. dr Dragan Milićev

Ako nadjete greske recite mi odmah da ispravim Nebitno da li su slovne ili sam nesto pogresno lupio. Mozete i sami izmeniti na <https://github.com/veljkoselakovic/OOP1> i napraviti merge request

- UML - *Unified Modeling Language* (Kasnije predmet *Projektovanje softvera*, 5. semestar)

## 1 Osnovni ciljevi OOP

- Problem korišćenja postojećeg koda
  - **Biblioteka funkcija** - skupo održavanje, otklanjanje grešaka i proširivanje sistema
- Evolucija programskih jezika
  1. Apstrakcija izraza ~1950. - **FORTTRAN**
    - Registri skriveni
  2. Apstrakcija kontrole ~1960. - **Algol60**
    - Tok kontrole programa - *petlje*
  3. Apstrakcija podataka ~1970. - **Pascal**
    - Razdvajanje detalja prezentacije podataka od apstraktnih operacija koje se definišu nad podacima
      - npr. *tipovi nabranja*
- Dodatni koncepti
  1. Zasebno prevodenje modula - **FORTTRAN, C, Ada**
  2. Razdvajanje interfejsa od implementacije - **Ada**
  3. Koncept klase - **Simula67**
- 4 Osnovna principa OOP
  - Apstrakcija
  - (En)kapsulacija
  - Nasleđivanje
  - Polimorfizam

## 2 C++

- Razvoj C++
  - C → C sa klasama → C++
  - Svake 3 godine novi standard
  - ISO 98 → ISO 03 → ISO 11 → ISO 14 → ISO 17 → ISO 20 (*još nije standardizovano*)
  - Spontani razvoj, za razliku od **Ada**
- Aspekti C++
  1. Da bude dovoljno blizak mašini
    - C++ je *nadskup* u velikoj većini slučajeva
  2. Da bude dovoljno blizak problemu
    - Klase iz **Simula67**
    - Preklapanje operatora iz Algola

### 3 Pregled gradiva koje će se raditi

- Klase i objekti
  - Klase su apstrakcije zajedničkih atributa i zajedničkog ponašanja jednog skupa srodnih objekata
  - Klasa sadrži
    1. Podatke članove (*atributi ili polja*)
    2. Funkcije članove (*metodi*)
  - Pristupačnost određenim članovima deklariše programer
  - **Implementaciju** klase čine ←— *Kako radi?*
    1. Privatni podaci članovi
    2. Definicije funkcija
  - **Interfejs** klase čine ←— *Šta radi?*
    1. Javni podaci članovi
    2. Deklaracije javnih funkcija
  - Instanca klase → objekat 
$$\left\{ \begin{array}{l} \text{Stanje} \\ \text{Ponašanje} \\ \text{Identitet} \end{array} \right.$$
- Konstruktori i destruktori
  - Prilikom kreiranja i uništavanja objekta
  - Nemaju **return** tip
  - Automatsko izvršavanje prilikom kreiranja/uništavanja objekta
- Izvođenje i nasleđivanje
  - Iz opštije klase izvodimo specifične klase
  - Izvedene klase nasleđuju atribute i metode osnovne klase, i dodaju nove
  - Objekti izvedene klase su i indirektne instance osnovne klase
  - U izrazima izvedeni objekti mogu zameniti osnovnu klasu - *Liskov substitution principle*
  - Nasleđeni metodi se mogu redefinisati
- Polimorfizam
  - Ako se funkcija proglaši virtual na nju se primeni **dinamičko vezivanje**
  - **Dinamičko vezivanje** - Adresa se ne određuje u vreme povezivanja, poziv se vezuje za funkciju u vreme izvršenja
  - Ponašanje objekta ne zavisi samo od tipa pokazivača, već i od tipa pokazanog objekta
- Klasifikacija objektnih jezika
  - 1. Objektno-bazirani
    - Apstrakcija, (en)kapsulacija, modularnost  
**Ada83, Visual Basic 6**
  - 2. Objektno-orientisani
    - Princip nasleđivanja  
**Simula, Smalltalk, Ada95, C++, Java, VB.Net, C#**
- Obrada izuzetnih situacija
  - Nepostojeće datoteke, prekoračenje opsega indeksa,...
  - Tradicionalni jezici u funkciji vraćaju vrednost koja signalizira grešku, koja se naknadno analizira
  - Kod postane nepregledan
  - **try/catch/throw** ključne reči
- Šabloni (*Templates*)
  - Odredene obrade ne zavise od tipa podataka
  - *Generičko programiranje*
  - Statički mehanizam - *u prevodu se zameni*
- Preklapanje operatora
  - Sam koncept nije OO, ali se dobro uklapa
  - Redefinicija standardnih jezičkih operatora  
**operator<simbol>**
  - Ne mogu se preklopiti svi operatori, a za neke važe posebna pravila

## 4 Proširenja jezika C

- Deklaracija vs definicija

- Deklaracija je iskaz koji
  1. Uvodi ime u program
  2. Govori prevodiocu kojoj jezičkoj kategoriji pripada ime
- Definicija
  1. Kreira objekat ILI
  2. Navodi telo funkcije ILI
  3. U potpunosti navodi strukturu korisničkog tipa

---

```
void f(int x, float y); // Deklaracija
void f(int x, float y) {...} // Definicija
extern int x; // Deklaracija
int x; //Definicija
class X; //Deklaracija
class X { ... }; //Definicija
```

---

- Samo jedna definicija, a proizvoljno mnogo deklaracija
- Objekat može biti definisan i deklarisan u istom redu

- Objekti

- Objekat u širem smislu - **podatak**
- Objekat u užem smislu - **instanca klase**
- Objekat ima
  1. Stanje
  2. Ponašanje
  3. Identitet → primitivni podaci nemaju identitet
- **Promenljiva** je lokacija u kojoj se čuva podatak
- Podela promenljivih
  1. Statička
  2. Automatska
  3. Dinamička
  4. Privremena (*tranzijentna*)

- Lvrednosti (*LVALUE*)

- Izraz koji upućuje na objekat ili funkciju
- Operatori čiji operandi moraju biti LVAL  
**unarni** `&`, `++`, `--`, **levi operandi** svih operatora dodele
- Operatori čiji su rezltati LVAL  
**unarni** `*`, `[]`, **prefiksni** `++` i `--`, operatori dodele
- DVrednost je sve što nije LVrednost (*RVALUE*)

---

```
int *q[10];
q[10]=&i; //q[10] je lvrednost
*q[10]=1; // *q[10] je lvrednost
q = &i; // ERROR, ime niza nije vrednost
int a=1, b=2, c=3;
(a=b)=c; // OK
(a+b)=c; // ERROR
++ ++i; // OK
i++ ++; // ERROR
```

---

- Oblast važenja (*Scope*)

- Onaj deo teksta programa u kome se deklarisano ime može koristiti
- **Dinamičko vezivanje** imena → od mesta deklaracije do kraja datoteke (globalna imena)
- **Lokalna** imena → od mesta deklarisanja do kraja odgovarajućeg bloka
- Sakrivanje imena
  1. Ako se definiše u nekom bloku, globalno je skriveno
  2. Ako se redefiniše u unutrašnjem bloku, ime iz spoljašnjeg bloka je sakriveno do izlaska iz bloka
- Pristup globalnom imenu koristeći operator ::

---

```
int x = 0; // Globalno x
void f() {
    int y=x, x; // y dobija vrednost globalnog x
    x=1; // Lokalno x
    ::x=5; // Globalno x
{
    int x; // Novo lokalno x, sakriva prethodno
    x=2;
}
x=3; // Pristup prvom lokalnom
}
int *p = &x; // Globalno x
```

---

BITNO

- Specifični dosezi

- Oblast važenja funkcije imaju samo labele
- **for** petlja
- Ranije verzije kompjerala su imale doseg promenljivih koji je bio 1 blok **van for** (MS VC++ 6.0)
- U **if** doseg do kraja **else**

- Klasni/Strukturni doseg

- Oblast važenja imaju svi njeni članovi
  1. . → levi operand objekat
  2. .→ → levi operand pokazivač na objekat
  3. :: → levi operand ime klase

- Životni vek objekata

- Vreme u toku izvršavanja programa u kojem objekat postoji i za koje mu se može pristupati
- Vek atributa klase = vek objekta
- Vek parametra = vek automatskog objekta
  1. Statički objekti
  2. Automatski objekti
  3. Dinamički objekti
  4. Privremeni (*transientni*) objekat

- Statički i automatski objekti

- Automatski objekat je lokalni objekat koji nije definisan kao *static*
  1. **Životni vek** - od definicije do kraja oblasti važenja
  2. Svaki put se kreira iznova prilikom poziva bloka u kom je definisan
  3. Prostor se alocira na *stack*
- Statički objekat je globalni objekat ili lokalni deklarisani kao *static*

Globalni {  
1. **Životni vek** - od definicije do kraja izvršenja *main*  
2. Kreiraju se jednom, na početku izvršavanja, pre funkcija objekta  
Lokalni {  
3. Počinju da žive pri prvom nailasku na njih

- Dinamički i privremeni objekti
  - Dinamički objekti se kreiraju i uništavaju posebnim operacijama
    1. **Životni vek** - kontroliše programer
    2. **new/delete**
    3. Prostor alocira na *heap*
  - Privremeni objekti se kreiraju pri izračunavanju izraza
    1. **Životni vek** - kratak i nedefinisani
    2. Odlaganje medurezultata i privremeno smeštanja vraćenih vrednosti funkcije
    3. Najčešće se uništavaju čim nisu potrebni
- Leksički elementi
  - Komentari
    1. *//...*
    2. */ \* ... \*/*
  - 73 ključne reči + alternative
  - C++11 **bool**, C11: **\_Bool**
  - Specijalne (ali ne i rezervisane) reči: **final** i **override** (*Backwards compatibility*)
  - Ne treba započinjati imena donjom crtom
- Tipizacija
  - **Stroga tipizacija** - objekti različitih tipova se ne mogu proizvoljno zamenjivati
  - C++ je hibridan
    1. Za osnovne primitivne tipove ✓
    2. Za sve ostalo X
- Konverzija
  - Operatori zahtevaju određene tipove operanada
  - Naredbe zahtevaju određene tipove operanada  
**for**, **if**, **while**...
- Vrste konverzija tipova
  - **Standarna konverzija** - ugrađeno u jezik  
npr. **int** → **float**, **char** → **int**
  - **Korisnička konverzija** - definiše programer
  - Pored toga, konverzija može biti
    1. **Implicitna** - prevodilac je automatski vrši
    2. **Eksplicitna** - zahteva programer
  - C koristi *cast* operator  
**(tip)izraz**
  - C++ uvodi 4 specifčna *cast* operatorka
  - Postoji i konverzionalni konstruktor
- Pridruživanje imena tipu
  - C-stil
 

```
typedef opis_tipa = ime_tipa
```
  - C++ stil, čitljivije
 

```
using ime_tipa = opis_tipa
```

---

```
typedef unsigned long long int Ceo1;
using Ceo2 = unsigned long long int;
```

---
- Određivanje tipa izrazom
 

```
decltype (izraz) promenljiva [= vrednost]
```

  - Izraz se **ne** izračunava
  - Primena kod *template-ova*

---

```
int x=1; double y=2.3;
decltype(x) a = x; // a je int
decltype(y) b = y; // b je float
decltype(a++) c = a; // c == a == 1
```

---

- Automatsko određivanje tipa
  - `auto` ključna reč određuje tip na osnovu inicijalne vrednosti u C, `auto` označava automatsku lokalnu promenljivu, i ne piše se

---

```
auto int a = 10; // Samo u C
auto a = 10; // C++, a je int
```

---
- Odloženo navođenje tipa funkcije
 

```
auto ime_funkcije(parametri) -> tip
```

  - Koristi se kod *template-ova*
  - C++14 tip može da se izostavi
    1. U tom slučaju tip se odredi preko `return` tipa ili definicije
    2. Funkcija ne sme da se poziva pre navođenja definicija na mestima gde je tip bitan

---

```
auto func(int x) -> double { ... } // return tip je double
auto f() { return 1; }
auto g();
auto a = g(); // ERROR
auto g() { return 0.5}
auto b = g() // OK
```

---
- Konstante
  - Izvedeni tip
 

```
const tip ime = vrednost;
```
  - Mora da se inicijalizuje pri definisanju
  - Izraz ne morada bude konstantan
 

Konstante inicijalizovane **konstantnim** izrazom (*simboličke* ili *kompilacione* konstante) mogu da se koriste u izrazima koji moraju biti konstantni (računaju se u **toku** prevođenja)

npr. Dimenzija statickog niza

BITNO
  - **Simboličke** konstante NE alociraju memoriju
 

```
const char* pk = niz; ← pokazivač na konstantu
char* const kp = niz; ← konstantni pokazivač
```
  - Ubacivanje `const` u parametre funkcije obezbeđuje da se dati objekat ne menja
  - Ubacivanje `const` u `return` tipu funkcije obezbeđuje da se privremeni objekat rezultata ne može menjati
  - POGLEDATI `constexpr`

---

```
char niz[] = { 'i', 'd', 'e', ' ', 'g', 'a', 's', '\0' };
const char* pk = niz; // Pokazivac na konstantu
pk[3] = '-'; // ERROR
pk = "OOP:("; // OK
char* const pk = niz // Konstantni pokazivac
pk[3] = '-'; // OK
pk = "OOP:("; // ERROR
```

---
- Znakovne konstante
  - U C → int (65 i 'A' su ista stvar)
  - U C++ → char
  - U izrazima `false` → 0, a u dodeli vrednosti logičkim promenljivama 0 → `false`

- Prostori imena (**namespace**)
  - Mehanizam za izbegavanje konflikata imena  
`namespace ID { sadržaj }`
  - Jednoznačno ime
    1. Celo ime `A::i`
    2. Uvoz imena `using A::i`
    3. Uvoz svih imena `using namespace A`
- Stringovi
  - C stil - niz znakova koji se završava sa `\0`
  - Literal C++ stringa je `const char*`
  - Literal C stringa je `char*`
  - C++ - podrazumevani string je
- Tipovi **enum**, **struct** i **union**
  - Identifikatori ova 3 tipa mogu da se koriste kao oznaka tipa, bez ključne reči
  - Ako u dosegu postoji objekat sa istim identifikatorom, sam ID označava objekat, a ne tip

---

```
enum RadniDan {Pon, Uto, Sre, Cet, Pet};
RadniDan r_dan = Uto;
int RadniDan;
enum RadniDan r1 = Sre; // OK
RadniDan r1 = Pon;     // ERROR
```

---
- Tip nabranja (**enum**)
  - Svaki **enum** je poseban celobrojni tip
  - Definisana je samo operacija dodelje vrednosti
    1. Eksplisitna konverzija celobrojne vrednosti u tip nabranja je obavezna
    2. Ne otkriva se greška ako konvertovana vrednost nije u opsegu
  - U aritmetičkim i relacijskim izrazima, kao i pri dodelji promenljivoj tipa int, konverzija je automatska

---

```
enum Dani {PO=1, UT, SR, CE, PE, SU, NE, POSLEDNJI=7}; // NE i POSLEDNJI su 7
Dani dan=SR; // OK
Dani d=4; // ERROR - nije eksplisitna konverzija
dan++; // ERROR - nije definisana operacija ++
dan=(Dani)(dan+1); // OK
if (dan<NE) { ... } // OK
dan=(Dani)8; // Ne prijavljuje se logicka greska
```

---
- Pripadajući tip nabranja
  - Numerička reprezentacija nabranja
  - Kompaktnije, podrazumeva se int
 

```
enum ime: pripadajući_tip {imenovane_konstante}
```
  - Paziti opsege
- Nabranja sa ograničenim dosegom
  - Isti doseg kao i tip nabranja
  - Rešenje - **struct** ili **class** iza **enum**
  - Pristup konstanti sa `::`
  - Obavezna eksplisitna konverzija u ceo broj
 

```
int i = (int)tip::ime
```

---

```
enum SemaforPesaci {CRVENO, ZELENO};
enum SemaforVozila {ZELENO, ZUTO, CRVENO}; // ERROR
enum struct SemaforPesaci {CRVENO, ZELENO};
enum struct SemaforVozila {ZELENO, ZUTO, CRVENO};
SemaforPesaci sp = SemaforPesaci::CRVENO;
SemaforVozila sv = ZUTO; // ERROR
int i = (int) SemaforVozila::ZELENO; // Obavezna konverzija
```

---

- Inicijalizatorske liste  
`{vrednost, vrednost, ..., vrednost}`
  - Inicijalizacija **svih** vrsta podataka, čak i prostih
  - Paziti na nebezbedne konverzije
  - Vrednosti se dodeljuju redom (čak i strukturama, uniji se popuni prvo polje)
  - Manjak vrednosti → popunjava se nulama
  - Višak vrednosti → Greška
  - Argumenti funkcija i izrazi u `return` mogu biti ove liste
  - Bezimeni podatak
  - Niz ne može da dobije vrednost liste nakon inicijalizacije, sem ako je deo neke strukture

ČUDNO

---

```
int i1={1}, i2{1}, i3={i1+i2};
i1={2};
int i4={0.5}; // ERROR - nije bezbedno
int *pi={&i1};
int n1[5]={1,2,3}, n2[5]{1,2,3}, n3[] {1,2,3};
int m[] [3]{{1,2},{}, {1,2,3}};
n1={4,5,6}; // ERROR
struct S1{int a,b;};
S1 s11={1,2}, s12{1,2}; s11={3,4};
struct S2{int a; S1 b; int c[3];};
S2 s21={1,{2,3}, {4,5,6}}, s22{1,2,3,4,5,6};
s21 = {6, {5,4}, {3,2,1}};
```

---

- Bezimena unija
  - Predstavlja objekat koji sadrži u raznim trenucima razne tipove podataka
  - Datotečki ili blokovski doseg
  - Unija za koju je definisan barem 1 objekat ili pokazivač → nije bezimena unija, iako nema ime

---

```
union{ int i; double d; char *pc; };
i=55; d=123.456; pc="ABC";
```

---

- Uvek promenljiva polja (**mutable**)
  - Polje označeno sa **mutable** može da se menja čak i za **const** parametre

---

```
struct X{
    int a;
    mutable int b;
};
int main(){
    X x1;
    const X x2;
    x1.a = 4;
    x1.b = 2;
    x2.a = 3; // ERROR
    x2/b = 4; // OK
}
```

---

- Dinamički objekti

- `new/delete`
- Operand operatora `new` je identifikator tipa T sa eventualnim inicijalizatorima
  1. Alocira potreban prostor za objekat datog tipa
  2. Poziva konstruktor tipa
- Ako nema mesta `bad_alloc` exception  
U nestandardizovanom C++ vraća se `nullptr`
- Vraća pokazivač na kreirani objekat  
`T *t = new (nothrow)T;` ← Ignorisanje exceptiona, vraća `nullptr` ako ne uspe
- Stavlja na `heap`

- Uništavanje dinamičkih objekata

- `delete` ima 1 operand (pokazivač nekog tipa)
- Mora biti objekat kreiran pomoću `new`, inače će ponašanje biti nepredviđeno
- `delete nullptr` ne radi ništa
  1. Poziva destruktorn za pokazani objekat
  2. Oslobađa zauzeti prostor
- `delete` vraća `void`

- Dinamički nizovi

- Sve dimenzije niza osim prve moraju biti konstanti izrazi
- Inicijalizacija
  1. Podrazumevani konstruktor ILI
  2. Generisain konstruktor
- `delete [] pt;`
- Redosled konstrukcije po rastućem indeksu
- Redosled destrukcije obrnut od redosleda konstrukcije
- Može inicijalizatorska lista

- Reference

- C isključivo po vrednosti prenosi argumente
- C++ prenosi argumente i po referenci

---

```
void f(int i, int &j){ // i po vrednosti, j po referenci
    i++; // stvarni argument se neće promeniti
    j++; // stvarni argument ce se promeniti
}
int main () {
    int si=0,sj=0;
    f(si,sj);
    cout<<"si="<

- Definisanje referenci

```

- Reference na LVrednosti (`lvalue`)
- Znak `&` ispred imena
- Sinonim za objekat, ne može se promeniti
- U definiciji mora da se inicijalizuje objektom
- Svaka naredba nad referencom je operacija nad pokazanim objektom

---

```
int i=1; // celobrojni objekat i
int &j=i; // j upucuje na i
i=3; // menja se i
j=5; // opet se menja i
int *p=&j; // isto sto i &i
j+=1; // isto sto i i+=1
int k=j; // posredan pristup do i preko reference
int m=*p; // posredan pristup do i preko pokazivaca
```

---

- Implementacija referenci

- Slično konstanom pokazivaču

```
int &j = *new int(2);
delete &j;
```
- Ako je referenca na `const` objekat, ne sme se menjati
- Ne postoje nizovi referenci, pokazivači na referencu, kao ni reference na reference
- Referenca na pokazivač je dozvoljena

- Funkcije koje vraćaju reference

- Funkcija mora da vrati referencu na objekat koji je *živ* i posle funkcije
- Rezultat poziva funkcija je LVrednost (`lvalue`) samo kao funkcija vraća referencu

---

```
int& f(int &i) { int &r = *new int(i); return r; } // OK
int& f(int &i) { return *new int(i); } // OK
int& f(int &i) { return i; } // OK
int& f(int &i) { int r = i; return r; } // NIJE OK
int& f(int i) { return i; } // NIJE OK
int& f(int &i) { int r = *new int(i); return r; } // NIJE OK
int& f(int &i) { int j = i; &r = j; return r; } // NIJE OK
```

---

- Obilazak elemenata niza u petlji  
`for(tip ime: niz) naredba`

- `foreach`, range petlja
- Može se staviti referenca na objekat, čime omogućavamo menjanje svakog elementa kom pristupamo - bez njega su *read-only*

---

```
for(auto& it: arr){ // it od iterator
    cout<<it++<<endl;
}
```

---

- Reference na DVrednosti (`rvalue`)

- Tip reference na DVrednost  
`osnovni_tip && ime = vrednost;`
- Referenca na DVrednost je LVrednost
- Posledica - privremeni podaci dobijaju imena, pa možemo da ih menjamo
- Podatak može biti nepromenljiv

---

```
int i=1; // i je promenljiv podatak
const int ci=i; // ci je nepromenljiv podatak
int && rd1=i; // ERROR - i je promenljiva lvrednost
int && rd2=ci; // ERROR - ci je nepromenljiva lvrednost
int && rd3=i+1; // (i+1) je promenljiva dvrednost
int && rd4=10; // 10 je nepromenljiva dvrednost
rd3++; rd4++; // rd3==3, rd4==11
```

---

- Reference na DVrednosti kao parametri

- Ne postoji *bočni efekat*
- `const` nema smisla

- Podrazumevane vrednosti argumenata

- Može biti samo nekoliko poslednjih argumenata
- Proizvoljni izrazi

- Neposredno ugrađivanje funkcije u kod

- Jednostavne, kratke funkcije  
`inline tip ime( parametri ) { definicija }`
- Izbegavanje prenosa argumenata i poziva funkcija
- Funkcija članica klase je **inline** ako se definiše u definiciji klase
- Ako se definiše van definicije klase, mora se staviti ključna reč
- Prevodilac ne mora da poštuje **inline**
- Ako je u više datoteka, mora se definisati u svakoj
- Često rešenje je sprovođenje sa dodatnom datotekom-zaglavljem, ali se tad funkcija može direktno videti od strane drugih korisnika
- Eliminiše potrebu za makroima

---

```
#define max(i,j)((i)>(j))?(i):(j)
max(i++, k++);
((i++)>(j++))?(i++):(j++); // 2x se inkrementira
```

---

- Preklapanje imena funkcija

- *Function name overloading*
- Funkcije koje realizuju logički istu operaciju, sa različitim tipovima argumenata
- U C nema preklapanja - funkcije moraju imati različita imena
- Mora da se razlikuje broj i/ili tip argumenata
- Tip rezultata **ne mora** da se razlikuje
- Takođe, **nije dovoljno** da se razlikuje samo **return tip**
- Statički koncept, sve se odvija u prevodenju
- Prevodilac prioritira slaganje tipova
  1. Potpuno slaganje - uključuje niz → pokazivač, ili referenca → objekat
  2. Slaganje standardnim konverzijama  
npr. `char` → `int`
  3. Slaganje korisničkim konverzijama

---

```
double max (double i, double j)
{ return (i>j) ? i : j; }
const char* max (const char *p, const char *q)
{ return (strcmp(p,q)>=0)?p:q; }
double r=max(1.5,2.5); // max(double,double)
double p=max(1,2.5); // (double)1; max(double,double)
const char *q=max("Pera","Mika");// max(const char*,const char*)
```

---

- Pristup elementima

- Složeni podaci
- Problem 2 definicije koje imaju identično telo sa različitim parametrima
- Druga funkcija poziva prvu ← Rešenje
- Slično za pokazivače i reference
- Čudan slajd, izgleda kao dodatno objašnjavanje overloadinga

---

```
int& elem( int *a, int i) { return a[i]; }
const int& elem(const int *a, int i) { return a[i]; }
int a[20],i=10;
const int b[20]={0};
elem(a,i)=1;
elem(b,i)=1; // ERROR
int x=elem(b,i);
```

---

- Napomene
  - Uputstva za prevodioca, **anotacije**  
[[napomena]]
  - Služe prevodiocu za provere i optimizacije
  - Prevodilac može da ih zanemari
- Funkcije koje se ne vraćaju
  - Postoje funkcije koje se ne vraćaju na mesto poziva
  - Nasilno prekida rad programa sa `exit(kod)`
    1. Kod = 0 ✓
    2. Kod ≠ 0 X
  - Anotacija [[noreturn]]
  - Ako ima negde `return`, kod postaje nepredvidiv
- Operatori i izrazi
  - Novi operatori (12)
    - unarni ::, ::, new, delete, .\*, ->, typeid, throw, alignof, 4 cast operatora
  - Postfiksni ++, -- imaju viši prioritet od prefiksnih
  - Prefiksni ++, -- → lvalue
  - Dodela vrednosti → lvalue
  - Ternarni operator je lvalue ako su drugi i treći operator lvalue
- Operatori konverzije tipa
  - C cast  
(tip) izraz ← Ne preporučuje se
  - Novi cast operatori
    1. `static_cast <oznaka.tipa> (izraz)`
    2. `reinterpret_cast <oznaka.tipa> (izraz)`
    3. `const_cast <oznaka.tipa> (izraz)`
    4. `dynamic_cast <tip_pokazivača_ili_reference> (izraz)`
  - Bezbedne i nebezbedne konverzije
    - int → float ✓
    - float → int X
  - Notacija je nezgrapna i kabasta iz 2 razloga
    1. Lako se uoči u tekstu
    2. Da programeri ne bi koristili
  - Ako postoji potreba za eksplisitim konverzijama → Preispitati projektne odluke
- Statička konverzija
  - Prenosive konverzije
    1. Između numeričkih tipova
    2. Između pokazivača i void\*
    3. Nestandardne konverzije - definiše programer
  - Primjenjuju se automatski kad su bezbedne
  - Eksplisitan poziv kad je nebezbedno
    - npr. void\* → drugi pokazivač, numerički tip → char
  - nullptr može da se dodeli bilo kom tipu pokazivača
  - Ne preporučuje se korišćenje NULL ili 0
- Reinterpretirajuća konverzija
  - Konverzija tipova bez logičke veze
    - npr. int → pokazivač
  - Nema pretvaranja vrednosti, istu vrednost različito interpretiramo
  - Jako nebezbedno
- Konstanta konverzija
  - Dodavanje ili uklanjanje const
  - Dodavanje je bezbedno, uklanjanje nije

## 5 Klase i objekti

- Osnovni pojmovi
  - Klasa je strukturirani korisnički tip koji obuhvata
    1. Podatke koji opisuju stanje objekta klase
    2. Funkcije namenjene definisanju operacija nad podacima
  - Klasa je formalni opis apstrakcije koja ima
    1. Internu implementaciju
    2. Javni interfejs
  - Instanca klase → objekat
  - Podaci klase → **atributi**, polja, podaci članovi
  - Funkcije klase → **metodi**, primitivne operacije, funkcije članice
- Komunikacija objekata
  - Objekti klase komuniciraju da ostvare složene funkcije
  - Poziv metoda → **upućivanje poruke**
  - Objekat može da menja stanje kad se pozove metod
  - **Objekat-klijent** - poziva metod
  - **Objekat-server** - metod mu je pozvan
  - Iz svog metoda se može pozvati metod drugog objekta iste ili druge klase
  - Unutar metode, članovima objekta-servera pristupa se navođenjem imena
- Pravo pristupa
  - Sekcije
- 1. **private**
  - Zaštićeni od spolja (**kapsulirani**)
  - Pristupaju im samo metodi klase
- 2. **protected**
  - Dostupni metodima iste klase + sve klase izvedene iz nje
- 3. **public**
  - Dostupni spolja bez ograničenja
  - Preporučuje se redosled **private** → **protected** → **public**
  - Može da postoji više sekcija iste vrste
  - Podrazumevana labela je **private**
    - Kontrola pristupa je stvar klase, a ne objekta
    - Metod jednog objekta može da pristupa privatnim članovima drugog objekta iste klase
  - Kontrola pristupa je odvojena od koncepta dosega
    1. Odredi se postojanje
    2. Proveravanje prava pristupa

BITNO

- Definisanje klase

- Atributi

- 1. Mogu da budu i inicijalizovane - od C++11
    - 2. Ne mogu da budu tipa klase koja se definiše, ali su dozvoljeni pokazivači i reference na tu klasu

- Metodi

- 1. U definiciji mogu da se
      - Deklarišu - samo prototip
      - Definišu - kompletno telo
    - 2. Funkcije definisane u definiciji klase su **inline** i mogu pristupati članovima imenom
    - 3. Funkcije koje su samo deklarisane u definiciji klase moraju biti definisane kasnije, van definicije, sa proširenim dosegom za pristup članovima  
`<ime_klase>::<ime_funkcije>`
    - 4. Vrednost rezultata metoda može biti tipa klase koja se definiše, kao i pokazivač ili referenca na nju

- Definicija se piše tamo gde se klasa koristi, obično u *header* fajlu (**.h**)

- Nepotpuna definicija klase je **deklaracija**

- Pre definicije, a posle deklaracije

- 1. Mogu da se definišu pokazivači i reference
    - 2. Ne mogu da se definišu objekti te klase

- Objekti klase

- Uobilajeno definisanje, kao kod standardnih tipova
  - Za svaki objekat formira se poseban komplet svih nestatičkih atributa
  - Nestatički metodi se pozivaju za objekte, a statički za klase
  - Lokalne **static** promenljive metoda
    - 1. Zajedničke za sve objekte
    - 2. Žive od nailaska na njih do kraja programa
    - 3. Imaju svojstva lokalnih promenljivih globalnih funkcija

WTF

- Podrazumevane operacije

- Definisanje objekata, pokazivača i referenci na objekte i nizove objekata
  - Dodela vrednosti jednog objekta drugo
  - Uzimanje adrese &
  - Pristupanje objektu preko pokazivača \*
  - Pristupanje atributima i pozivanje metode neposredno pomoću .
  - Pristupanje atributima i pozivanje metoda posredno pomoću pokazivača ->
  - Pristupanje elementima niza []
  - Prenošenje objekta kao argumenata po vrednosti, referenci ili pokazivaču
  - Vraćanje objekta iz funkcije po vrednosti, referenci ili pokazivaču
  - Preklapanje operatora može redefinisati dosta gore navedenog

- Pokazivač **this**

- Pokazivač na tekući objekat
  - Unutar svakog nestatičkog objekta je implicitno, **this** je skriveni argument svakog metoda  
`objekat.f() ~ f(&objekat)`
  - Konstanti pokazivač na klasu čiji je metod član  
Klasa X, **this** → X\* const
  - Pristup se obavlja neposredno
  - Primeri korišćenja
    - 1. Tekući objekat vratiti kao rezultat metoda
    - 2. Adresa objekta je potrebna kao argument
    - 3. Tekući objekat ubaciti u listu

---

```

// Definicija metoda zbir(Kompleksni) klse Kompleksni
Kompleksni Kompleksni::zbir(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.zbir(c2);
}

```

---

- Inspektori i mutatori

- **Inspektor** ili selektor → ne menja stanje objekta
- **Mutator** ili modifikator → menja stanje objekta
- Dobra praksa da se kaže koji tip od ova dve je metod
- `const` iza liste parametara → inspektor
- Postoji konstantan metod, ali je to druga stvar

- Definisanje inspektora

```
<tip> ime(parametri) const {definicija}
```

- Notaciona pogodnost
- Prevodilac nema načina da osigura da inspektor ne menja attribute  
Eksplicitna konverzija može da probiju kontrolu konstantnosti
- U inspektoru `this` je `const`
- Nije moguće menjati objekat pomoću `this`
- Za nepromenljive objekte nije dozvoljeno pozivati metod koji nije inspektor

---

```

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(); // OK - inspektor promenljivog objekta
x.pisi(); // OK - mutator promenljivog objekta
cx.citaj(); // OK - inspektor nepromenljivog objekta
cx.pisi(); // ERROR - mutator nepromenljivog objekta

```

---

- Nepostojani metodi (**volatile**)
  - Suprotnost konstantnog metoda
  - Veza sa konkurentnim programiranjem
  - Neki drugi *thread* može u svakom trenutku da promeni stanje objekta
  - Prevodilac ne izvršava optimizaciju
  - **volatile** može da se poziva za nepostojane i promenljive objekte
  - **const volatile** - za sve vrste objekata

---

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

---

- Modifikatori metoda **&** i **&&**
  - Bez modifikatora **&** i **&&** metod se može primeniti na **lvalue** i **rvalue**
  - Modifikator **&** - tekući objekat može biti samo **lvalue**
  - Modifikator **&&** - tekući objekat može biti samo **rvalue**
  - Mogu da postoje metodi čiji se potpisi razlikuju samo po ovom modifikatoru

BITNO

---

```
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
  - Specifična funkcija klase koju definiše početno stanje objekta
  - Isto ime kao klasa
  - Nema **return** tip, čak ni **void**
  - Proizvoljan broj proizvoljnih tipova parametara
    1. Ne sme biti tip klase koju definiše ako je jedini parametar ili ako svi ostali imaju podrazumevanu vrednost
    2. Dozvoljen tip pokazivača na **lvalue** i **rvalue** date klase
  - Implicitno se poziva prilikom kreiranja
  - Pristup članovima objekta kao i bilo koji drugi metod
  - Može biti preklopljen \ *overloaded*
- Podrazumevani konstruktor
  - Može se pozvati bez stvarnih argumenata - nema parametre ili su svi podrazumevani
  - Ugrađeni podrazumevani konstruktor je bez parametara i ima prazno telo
  - Ugrađeni konstruktor postoji smao ako klasa nije definisala nijedan drugi konstruktor
  - Definisanje nekog konstruktora se suspenduje ugrađeni  
Restauracija ugrađenog konstruktora - deklaracija iza koje sledi = **default**
  - Kad se kreira niz objekata poziva se podrazumevani konstruktor po rastućem redosledu indeksa

- Pozivanje konstruktora
  - Stvaranje bilo kakvog objekta
  - 1. Definicija statičkog objekta
  - 2. Definicija automatskog objekta
  - 3. Dinamički objekat kreiran operatorom `new`
  - 4. Kad se stvarni argument klasnog tipa prenosi u formalni
  - 5. Kada se kreira privremeni objekat pri povratku iz funkcije
- Argumenti konstruktora
  - Pri stvaranju objekta moguće je navesti inicijalizator iza imena
  - Inicijalizator sadrži listu argumenata konstruktora u zagradama
    1. () ili {}
    2. Ako {} → može se pisati `i = {...}`
    3. Nisu dozboljene prazne zagrade ()  
Deklaracija funkcije
  - Moguća notacija `<objekat> = <vrednost>`
  - Poziva se onaj konstruktor koji se najbolje salže po potpisu
  - Može da ima podrazumevane vrednosti

---

```

class X {
    char a; int b;
public:
    X ();
    X (char, int=0);
    X (const char*);
    X(X); // ERROR
    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="Ne volim OOP";
    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(char,int)
    X* p4=new X{c,10};
}

```

---

- Konstrukcija članova

- Pre izvršavanja tela konstruktora
    1. Inicijalizuju se prosti tipovi
    2. Pozivaju se konstruktori za klasne tipove
  - Inicijalizatori mogu da se navedu u zaglavlju definicije (NE deklaracije) konstruktora, iza znaka :
    - Ako atributi ima inicijalizatoru telu klase i u definiciji konstruktora → primenjuje se inicijalizator iz definicije konstruktora
  - Inicijalizacija atributa - **redosled navođenja u definiciji klase**
    - Bez obzira da li su primitivni ili klasni tipovi
    - Bez obzira na redosled u listi inicijalizatora
- BITNO
- 
- ```
class X {
private:
    int i = 0;
}

```
- Do C++11 nije bila dozvoljena inicijalizacija atributa u definiciji klase
  - Inicijalizacija je različita od operacije dodele koja se može vršiti jedino unutar tela konstruktora
  - Inicijalizacija je neophodna
    1. Kada ne postoji podrazumevani konstruktor klase atributa
    2. Kada je atribut nepromenljiv
    3. Kada je atribut referenca

---

```
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

```

---



---

```
// Primer konstrukcija dva objekta od kojih jedan sadrzi drugi
class Kontejner {
public:
    Kontejner () : deo(this) {...}
private:
    Deo deo;
};
class Deo{
public:
    Deo(Kontejner* kontejner):mojKontejner(kontejner) {...}
private:
    Kontejner* mojKontejner;
};
```

---

- Delegirajući konstruktor

- 
- ```
class T {
    T(int i){}
    T():T(1){}
    T(char c): T(0.5){}
    T(double d): T('a'){}
}
```
- 
- U listi inicijalizatora definicije delegirajućeg konstruktora može da se navede poziv drugog konstruktora
  - Pre izvršenja tela delegirajućeg konstruktora, izvršava se ciljni konstruktor
  - Kad se navodi ciljni konstruktor, navodi se samo on
  - Ako dolazi do neposrednog ili posrednog delegiranja → greška  
Prevodilac ne otkriva ovakav tip greške

- Eksplisitni poziv konstruktora

- Ovakav poziv kreira primvremeni objekat klase pozivom odgovarajućeg konstruktora
- Isto se dešava ako se u inicijalizatoru objekta eksplisitno navede poziv konstruktora  
`Kompleksni c = Kompleksni(0.1, 5);`  
 Privremeni objekat se kopira u `c` - zavisi od prevodioca

- Konstruktor kopije

- Kopirajući konstruktor
- Pri inicijalizaciji objekta O1 drugim objektom O2 iste klase poziva se konstruktor kopije
- Ugrađeni, implicitno definisani, konstruktor kopije
  1. Vrši inicijalizaciju članova O1 članovima O2 (pravi **plitku kopiju** - *shallow copy*)
  2. Primitivni atributi se prosto kopiraju - uključujući i pokazivače
  3. Za klasne atribute se pozivaju njihovi konstruktori kopije
- Ugrađeni konstruktor kopije se briše ili suspenduje
  1. Eksplisitno  
`X(const X&) = delete`
  2. Implicitno - pisanjem premeštajućeg konstruktora ili premeštajućeg operatora deodele  
`Restauriranje konstruktora kopije X(const X&) = default`
- Problem pokazivača → pravimo **duboku** kopiju - *deep copy*
- Parametri konstruktora kopije su `X&` ili `const X&`
- Ostali eventualni parametri moraju biti podrazumevane vrednosti

- Pozivanje konstruktora kopije

- Poziva se jednim stvarnim argumentom
- Konstruktor kopije se poziva kada se objekat inicializuje objektom iste klase i to
  1. Prilikom stvaranja trajnog, automatskog, dinamičkog ili privremenog objekta
  2. Prilikom prenosa argumenata po vrednosti u funkciju (stvara se automatski objekat)
  3. Prilikom vraćanja vrednosti iz funkcije (stvara se privremeni objekat)
- Prevodilac sme da preskoči poziv konstruktora kopije zbog optimizacije
  - Ako se stvarani objekat inicializuje privremenim objektom iste klase
  - Izostaju bočno efekti koje programer očekuje
  - Čak i tada mora postojati konstruktor kopije ili premeštajući konstruktor

---

```

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 smesta 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

- Konstruktor koji se poziva za konstrukciju objekta istog tipa, pri čemu je izvorišni objekat na kraju životnog veka
- Izvorišni objekat je **nvrednost** (nestajuća vrednost) - *xvalue (expiring value)*
- Izvorišni objekat ne mora da se sačuva
- Samo prenestimo njegove dinamičke delove u odredišni objekat
- Nema kopiranja dinamičkih delova
- Posledica → premeštajući konstruktor je efikasniji od kopirajućeg
- Modifikovati izvorišni objekat da njegova destruktacija ne povuče razaranje premeštenih delova
- Postoji ugrađeni, implicitno definisani, premeštajući konstruktor, ali ona ima problem - ne briše originalne pokazivače u izvorišnom objektu
- Ugrađeni premeštajući konstruktor se briše ako se eksplisitno definiše bar jedan od navedenih:
  1. Premeštajući konstruktor
  2. Kopirajući konstruktor
  3. Destruktor
  4. Operator dodele

Nisam 100%  
siguran za ovo  
BAR

- Pozivanje premeštajućeg konstruktora

- Paramater je **X&&**, ostali su podrazumevani parametri
- Prevodilac poziva premeštajući konstruktor
  1. Ako izvorišni objekat nestaje
  2. Ako u klasi postoji premeštajući konstruktor
- Ako u klasi ne postoji premeštajući
  1. Poziva se kopirajući konstruktor
  2. Semantika je ista
  3. Promena je samo u efikasnosti

BEZ CONST

---

```
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; }
```

---

- Konverzionalni konstruktor

- Konverzija između tipova od kojih je bar jedan klasa
- Odredišni tip mora biti klasa  
**X::X(T&)** **X:: X(T)** → konverzija tipa T u X
- Korisničke konverzije se primenjuju automatski ako je jednoznačan izbor konverzije, izuzev u slučaju **explicit** konstruktora
- Konverzija mora biti posredna  
**U::U(T&), V::V(U&)** → **V(U(t))** eksplisitno
- Nije moguće konvertovati u primitivni tip
- Konverzija argumenata i rezultat funkcije
  1. Pri pozivu funkcije
    - Inicijalizuju se parametri stvarnim argumentima uz eventualnu konverziju tipa
    - Parametri se ponašaju kao automatski lokalni objekti pozvane funkcije
    - Ovi objekti se konstruišu pozivom odgovarajućih konstruktora
  2. Pri povratku iz funkcije
    - Konstruiše se privremeni objekat koji prihvata vrednost **return** izraza na mestu poziva

---

```
//Konverzionalni konstruktor - PRIMER
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)
    //...
}
```

---

- Destruktor

- Specifična funkcija članica koja uništava objekat
- Nosi isto ime kao klasa, uz ~ ispred imena
- Nema tip rezultata i ne može imati parametre → najviše 1 po klasi
- Destruktor se piše kada treba osloboditi memoriju i ostale resurse
- Česta potreba → klasa sadrži članove koji su pokazivači ili reference na druge objekte  
Dobra praksa tad → metod za uništavanje delova, pozvan iz konstruktora
- Ponašanje kao i drugim metodima

- Pozivanje destruktora

- Implicitno se poziva na kraju životnog veka objekta
- Pri uništavanju dinamičkog objekta koristeći **delete**
- Pri uništavanju dinamičkog niza - u smeru opadajućih indeksa
- Redosled je uvek obrnut od konstruktora
- Eksplisitno pozivanje  
`X::~X(), px->~X(), this->~X()`
  - Ne preporučuje se, objekat nastavi da živi i posle ovoga
- Posle izvršenja automatskog destruktora se oslobada zauzeta memorija

- Statički (zajednički) atributi

- Pri stvaranju objekta klase → poseban komplet nestatičkih atributa
- Ključna reč - **static**
- Jedan primerak za celu klasu, svi objekti ga dele  
`static <tip> ime;`

- Definisanje statičkog atributa

- U klasi se samo deklariše
- Mora da se definiša na globalnom nivou
- Svi oblici inicijalizatora ✓
- Inicijalizacija
  1. Pre prvog pristupa njemu
  2. Pre stvaranja objekta date klase
- Obraćanje `int <klasa>::X=5; // bez static`
- Ako se navede inicijalizator → 0
- Imenovana **celobrojna** konstanta može se definisati i u definiciji klase

ČUDNO  
Ima veze sa  
constexpr?

- Statički i globalni podaci
  - Sličnosti
    1. Trajni podaci → sličan životni vek
    2. Definicija na globalnom nivou
  - Razlike
    1. Statički atributi pripadaju klasi
    2. Doseg imena statičkog atributa je klasa
    3. Statičkim atributima je moguće ograničiti pristup
  - Statički atribut ima sva svojstva globalnog statičkog podatka osim dosega imena i kontrole pristupa
  - Smanjuje se potreba za globalnim objektima
- Statički (zajednički) metodi
  - Funkcija klase, a ne svakog posebnog objekta
  - Zajednički za sve objekte
  - Primena
    1. Opšte usluge
    2. Obrada statičkih atributa
  - Deklaraju se dodavanjem **static** ispred deklaracije
  - Svojstva globalnih funkcija osim dosega i kontrole pristupa
  - Nemaju **this**
    1. Ne mogu pristupati nestatičkim članovima direktnim imenovanjem
    2. Modifikatori **const** i ostali nemaju smisla
  - Mogu pristupati nestatičkim članovima konkretnih objekata
    1. Pristup preko parametra
    2. Pristup lokalnom objekta
    3. Pristup globalnom objektu
  - Direktni pristup statičkim članovima  
`<klasa>::<ime_funkcije>(argumenti);`
  - Može se pozvati za konkretni objekat, ali izbegavati  
Levi operand tada samo nade tip bez ikakvog izračunavanja
  - Mogu se pozivati i pri stvaranju objekta klase
  - Uslužna klasa → sve statički metodi, obrisanog ugrađenog konstruktora - kao biblioteka

---

```

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; // ERROR - X::y nije staticki
    int k=x1.y; // ovo moze;
    (x1++.x); // x1++ (ako je definisan post inkrement operator) x.x izracunava pozBartalji
    return x1.x; // i ovo moze, ali nije preporucljivo
} // izraz "x1" se ne izracunava

```

---

```

int X::g () {
    int i=x; // nestaticki metod moze da koristi
    int j=y; // i staticke i nestaticke atribute
    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();
    // ERROR - za X::g mora konkretan objekat
    p=xx.f(xx); // i ovako moze, ali nije preporucljivo
}

```

---

---

```
// Zadatak koji se pojavio na kolokvijumu
class X {
public: static X* kreiraj () { return new X; }
private: X(); // Konstruktor je privatан
};
int main() {
    X x; // ERROR
    X* px=X::kreiraj(); // OK
}
```

---

- Prijatelji klase

- Kad je potrebno da klasa ima povlašćene korisnike koji mogu da pristupaju njenim privatnim članovima
- Povlašćene mogu biti
  1. Funkcije
  2. Cele klase
- Nazivamo ih **prijateljima** - *friends*
- Prijateljstvo, kao relacija između klasa
  1. Ne nasleđuje se
  2. Nije simetrično  $\Theta$
  3. Nije tranzitivno
- Reguliše isključivo pravo pristupa, a ne i oblast važenja i vidljivost identifikatora

- Prijateljski funkcije

- Nisu članice klase ali imaju pristup privatnim članovima
- Mogu biti metode druge klase ili globalne funkcije
- Funkcija je prijateljska ako se u definiciji klase navede njena deklaracija ili definicija sa modifikatorom **friend**
- Klasa mora eksplicitno da naglasi prijateljstvo
- Ako u definiciji klase pišemo prijateljsku funkciju
  1. I dalje nije članica klase iako je definišemo unutar nje
  2. Podrazumeva se da je **inline**
  3. Funkcija nema klasni doseg već doseg identifikatora klase
- Nevažno je pravo pristupa za **friend** funkciju
- Nema **this**
- Funkcija može biti prijatelj većem broju klasa istovremeno

---

```
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;} // ERROR - 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); // P preko metoda
    g(6,x); // Postavljanje preko prijateljske funkcije
    j=o(x); // Citanje preko prijateljske funkcije
}
```

---

- Prijateljske funkcije i metodi
  - Nekad je bolja prijateljska funkcija od metoda
  - Metod mora da se pozove za objekat date klase, dok globalnoj funkciji možemo dostaviti i oblik drugog tipa  
Nemoguća konverzija skrivenog argumenta u metodu
  - Pristup privatnim članovima više klasa - simetrično rešenje
  - Nekad je jenotacija pogodnija  
`max(a,b)` ili `a.max(b)`
  - Kad se preklapaju operatori, često je jednostavnije definisati globalne operatorske funkcije nego metode

- Prijateljske klase

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

```
class X {
    friend Y; // Ako je klasa Y definisana ili deklarisana
    friend class Z; // Ako Z nije ni definisana ni deklarisana
};
```

- Svi metodi klase Y pristupaju privatnim članovima klase X
- Prijateljske klase se često koriste kad neke dve klase imaju tesnu vezu

- Ugnježdene klase

- Klase mogu da se deklarišu ili definišu unutar definicije druge klase
- Koristi se kada neki tip semantički pripada samo datoj klasi
- Povećava čitljivost i smanjuje potrebu za globalnim tipovima
- Unutar definicije klase se mogu navesti i definicije nabranjanja `enum` i tipova `typedef`
- Ugnježdena klasa se nalazi u dosegu imena okružujuće klase (izvan nje pristup samo sa `::`)
- Iz okružujuće klase u ugnježdenu `., ->, ::`
- Doseg imena okružujuće klase O se proteže na ugnježdenu klasu U  
Pristup iz U do članova O samo sa `., ->`
- U ugnježdenoj klasi mogu direktno da se koriste identifikatori

1. Tipova iz okružujuće klase  $\leftarrow$  Samo od konkretnog objekta
2. Konstanti tipa nabranjanja okružujuće klase
3. Statički članovi okružujuće klase

- Ovo važi ako ime nije sakriveno imenom člana **ugnježdene** klase

Zar nije obrnuto?

```
<id_okružujuće>::<id_ugnježdene>::<id_statičkog>
```

- Ugnježdena klasa je implicitno prijatelj okružujuće
- Okružujuća klasa nije prijatelj ugnježdene  $\otimes \leftarrow$  ugnježdena klasa

---

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

Unutrasnja u; // ERROR
Spoljna::Unutrasnja u; // OK
```

---

- Strukture

- Struktura je klasa kod koje su svi članovi podrazumevano javni  
Može se menjati eksplisitnim korišćenjem `public:` i `private:`
- C++ struktura može imati i metode
- Strukture se koriste za definisanje strukturiranih podataka koje ne predstavljaju apstrakciju i generalno nemaju značajnijih operacija
- Tipično imaju samo konstruktor, uz eventualni destruktor

- Lokalne klase

- Definišu se unutar funkcija
- Identifikator ima doseg od deklaracije do kraja bloka u kom je deklarisan
- Unutar klase dozvoljeno je korišćenje iz okružujućeg dosega
  1. Identifikatora tipova
  2. Konstanti tipa nabranja
  3. Trajnih podataka (statičkih atributa, statičkih lokalnih i globalnih)
  4. Spoljašnjih (`extern`) podataka i funkcija
- Metodi lokalne klase moraju da se definišu unutar definicije klase
- Lokalna klasa ne može da ima statičke atrubute, dok može imati statičke metode

---

```

int x;
void f() {
    static int s;
    int x;
    extern int g();
    class Lokalna {
        public:
            int h () { return x; } // ERROR - 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; // ERROR - nije u dosegu

```

---

- Pokazivači na članove klase

- Dodelom vrednosti pokazivaču na članove klase označi se nei član klase  
Kao pokazivačka aritmetika indeksa u nizu
- Deklaracija  
`<tip_člana><klasa>::*<identifikator>`
- Formiranje adrese  
`<identifikator> = &<klasa>::<član>`
- Pristup  
`<objekat>.*<identifikator>`  
`<pokazivač_na_objekat>-*>*<identifikator>`
- `.* i ->*` imaju prioritet 14 i asocijativnost sleva na desno

---

```

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;

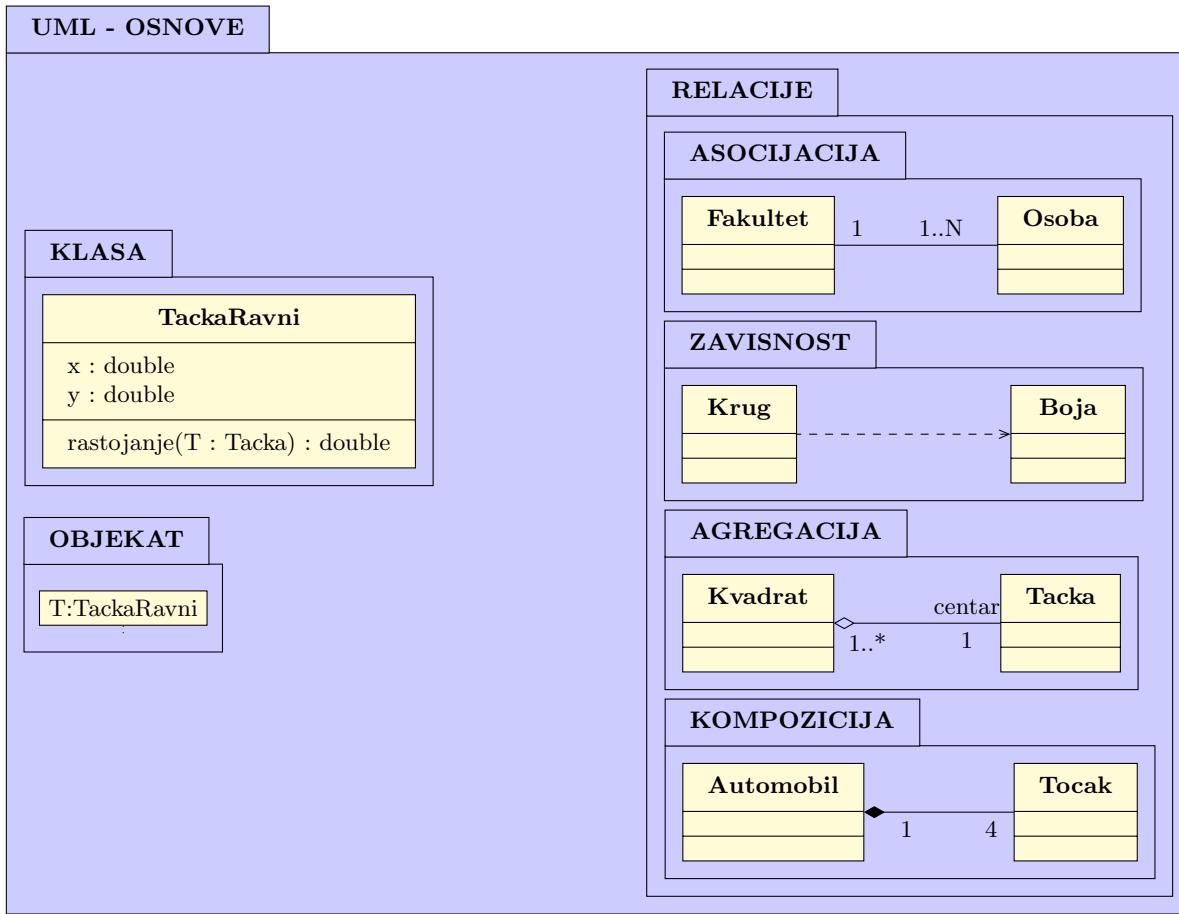
pc=&Alfa::b; // pc pokazuje na clanove b objekata klase Alfa
alfa.*pc = 2; // alfa.b=2;
beta->*pc = 2; // beta->b=2;

```

---

- UML

- Grafička notacija za modeliranje



SI KOLOKVIJUM 1

## 6 Preklapanje operatora

- Pojam preklapanja operatora

- Drugi naziv - **preopterećivanje operatora** - *operator overloading*
- Nova značenja operatora za korisničke tipove
- Sličan princip kao kod preklapanja funkcija
- Operatorske funkcije nose ime `operator@`, gde je @ neki operator ugrađen u jezik
- Izraz `T1 @ T2` može biti tumačen kao
  1. `operator@(T1, T2);` - friend funkcija
  2. `T1.operator@(T2);` - operatorski metod

---

```
class Kompleksni {
private:
    double real, imag;
public:
    Kompleksni (double, double); // Konstruktor
```

```

// Operator + u obliku globalne prijateljske funkcije
friend Kompleksni operator+(Kompleksni, Kompleksni);
Kompleksni operator-(Kompleksni K){ // Klasni metod
    Kompleksni novi_broj;
    novi_broj.real = this->real - K.real;
    novi_broj.imag = this->imag - K.imag;
    return novi_broj;
}
};

// Definicija konstruktora
Kompleksni::Kompleksni(double r=0.0, double i = 0.0) : real(r), imag(i) {}
Kompleksni operator+ (Kompleksni K1, Kompleksni K2){
    Kompleksni novi_broj;
    novi_broj.real = this->real + K.real;
    novi_broj.imag = this->imag + K.imag;
    return novi_broj;
}
/*...*/
Kompleksni K1(1.0, 1.0), K2(2.0, 2.0), K3, K4;
K3 = K1+K2; // operator+(K1, K2);
K4 = K3-K2; // K3.operator-(K2);

```

---

- Ograničenja preklapanja operatora

- Sledeći operatori se **ne** mogu preklopiti:  
., .\*, ::, ?:, sizeof, alignof, typeid, throw
- Ne može se menjati operator nekog primitivnog tipa

- Ne može mo praviti nove operatore, kao ni menjati osobine postojećih
 

Prioritet
n-arnost
Asocijativnost
- Neki operatori imaju specijalna ograničenja  
=, postfiksni ++ i --, [], -, (tip), new, delete

- Pravila preklapanja operatora

- Operatori koji su definisani implicitno  
=, &, ., ,(zapeta), \*, -, [], ← Poslednja 3 rade normalno i kad su preklopljeni, samo se sad mogu primeniti i na obektu
- Vraćeni rezultat je proizvoljan
- Ako se simbol piše slovima (**new**, **delete**), onda stoji odvojeno od **operator**
- Operatorske funkcije ne mogu imati podrazumevane vrednosti
- Operatorski metodi ne mogu biti statički, osim **new** i **delete**
- Deo potpisa operatorskog metoda čine i modifikatori tekućeg objekta  
**const**, **volatile**, &, &&

- Bočni efekti i veze između operatora

- Ne podrazumeva se bočni efekat kod operatora koji ga inače imaju  
++, --, =, +=, -=, ...
- Loša praksa
 

Može se preklopiti da nema bočnog efekta tamo gde obično postoji
Može se napraviti bočni efekat tamo gde ga inače nema
- Veze koje postoje između primitivnih operatora nisu garantovane  
**a+=b**  $\not\Rightarrow$  **a = a + b**
- Ovi operatori se moraju eksplicitno preklopiti

- Preporuke kod preklapanja operatora

- Preklopljeni operatori treba da imaju očekivana značenja
  1. Ako standardni operator ima bočni efekat, i preklopljeni bi trebao da ga ima
  2. Ako standardni operator vraća **lvalue**, i preklopljeni bi trebao da ga vraća
- Kada se definiju operatori za klasu, treba težiti da njihov skup bude kompletan  
Ako imam definisane **+**, **=** trebalo bi da definišem i **+=**

- Operatorski metodi/globalne prijateljske funkcije
  - Operatorske funkcije mogu biti
    1. Metodi klase
    2. Globalne prijateljske funkcije
  - Kod globalne funkcije, bar 1 parametar mora biti klasnog tipa  
`a.operator@(b) ← Metod klase`  
`operator@(a,b) ← Globalna prijateljska funkcija`
  - Ne možemo imati oba
- Ograničenja operatorskog metoda
  - Ako levi operand binarne operacije treba da bude standardnog tipa → moramo definisati globalnu funkciju
  - Kod operatorskog metoda levi operand je uvek skriveni parametar, i uvek je tipa klase kojoj pripada  
 Primer je operacija oduzimanja
  - Operatorski metod ne dozvoljava konverziju levog operanda
- Unarni i binarni operatori
 

<ul style="list-style-type: none"> <li>– Unarni</li> <li><code>tip operator@() // Metod</code></li> <li><code>tip operator@(X x) // Globalna funkcija</code></li> </ul>	<ul style="list-style-type: none"> <li>– Binarni</li> <li><code>tip operator@(X x_desno)</code></li> <li><code>tip operator@(X x_levo, X x_desno)</code></li> </ul>
---	---
- Inicijalizacija i dodela vrednosti
  - Inicijalizacija podrazumeva da objekat još ne postoji, dok dodata podrazumeva suprotno
  - Inicijalizacija se vrši uvek kada se kreira objekat
  - Inicijalizacija poziva konstruktor, a ne operator dodele
  - Konstruktor se poziva čak iako je notacija za inicijalizaciju operator =
  - Ako je izraz sa desne strane simbola =
    1. Istog tipa kao objekat koji se kreira, poziva se ili konstruktor kopije, ili premeštajući konstruktor
    2. Različitog tipa u odnosu na objekat koji se kreira, poziva se konverzionalni konstruktor
  - U oba slučaja može biti pozvan konstruktor sa više parametara, ako svi ostali parametri imaju podrazumevane vrednosti
  - Dodata → Izvršavanje izraza sa operatorom dodele =
  - Podrazumevano značenje je kopiranje objekta član po član
    1. Primitivni tipovi → kopiranje vrednosti
    2. Klasni tipovi → poziva se operator dodele odgovarajuće klase
    3. Pokazivački tipovi → kopiraće se samo pokazivač, a ne i pokazivana vrednost/objekat. Ako treba kopirati i pokazani objekat mora se preklopiti operator odelje (*deep copy*)
- Preklapanje operatora dodele
  - Mora biti nestatički metod
  - Ugrađena varijanta vrši plitko kopiranje (*shallow copy*)
  - Ugrađena varijanta vraća `lvalue`, pa je preporuka da se rezultat vraća po referenci
  - Najčešća optimizacija
    1. Prvo se ispita da nije slučaj poziva `a = a`, jer tada ništa ne radimo (bitno je da se ovo uradi, jer može da dođe do slučajnog brisanja objekta)
    2. Uništavaju se delovi levog operanda (nije neophodno, pogotovo ako su iste veličine kao i delovi desnog operanda)
    3. Kopiraju se (ili premeštaju) delovi desnog operanda u levi

---

```
X& X::operator=(const X& x){
    if(&x != this) { // Proveravamo da li su isti objekat
        // Formiramo nove delove
        // Kopiramo sadrzaje iz x
    }
    return *this;
}
```

---

- Varijante operatora dodele

- Kopirajuća i premeštajuća verzija
- Razlika samo u efikasnosti, semantički su isti
- Razlika u tipu parametra
  1. `X& X::operator=(const X& x)` ← Kopirajući konstruktor
  2. `X& X::operator=(X&& x)` ← Premeštajući konstruktor
- Obe verzije imaju ugrađene definicije koje rade plitku kopiju
- Ugrađena kopirajuća se briše ako se definiše
  1. Premeštajući konstruktor
  2. Premeštajuća dodela

ČUDNO

- Ugrađena premeštajuća dodata se briše ako se definiše

1. Kopirajući konstruktor
2. Premeštajući konstruktor
3. Konstruktor
4. Destruktor
5. Kopirajuća dodela

- Restauracija kopirajućeg sa `=default`

PREM.?

- Ako u klasi ne postoji premeštajuća dodela, koristi se kopirajuća

- Preporuke za operator dodele

- Ako se za klasu pišu destruktore, konstruktor kopije ili operator dodele, sva je prilika da treba ispisati sve te funkcije
- Ako se za klasu piše operator dodele sva je prilika da treba napisati i kopirajuću i premeštajuću dodelu
- Implementacija

- Napraviti privatne metode  $\begin{cases} \text{kopiraj}(x) \\ \text{premesti}(x) \\ \text{brisni}(x) \end{cases}$ 
  - Ovi metodi se koriste na mnogo mesta, pa je zgodno napisati ih samo jednom
  - Alternativa - *copy-and-swap* koja objedinjuje dva tipa operatora dodele  
Uместо po referenci, prosledi se kopija desnog operanda na stek, zatim se izvrši razmena objekata i destruktore obriše ostatke levog operanda kada skidamo objekta sa steka

---

```
/*...*/
int main(){
    X x = X("OOP");           // X(const char*); Konverzionalni konstruktor
    y = x;                     // X(const X&); Konstruktor kopije
    z = X("me cini");         // X(const char*); X(const X&&); Premestajući konstruktor
    x = y;                     // operator=(const X&); Kopirajuća dodela
    y = X("tuznim");          // X(const char*); operator=(const X&&); Premestajuća dodela
    /* Paziti na optimizacije kompjajlera. Nekad kompjajler izvrši nesto što ne očekujemo*/
}
```

---

- Preklapanje operatora `++` i `--`

- Problem nastaje jer postoje i prefiksna i postfiksna verzija
- Prefiksna
 

<code>T&amp; operator@@()</code>	– Postfiksna
<code>T&amp; operator@@(T&amp;)</code>	<code>T operator@@(int)</code>
	<code>T operator@@(T&amp;, int)</code>
- Za poziv postfiksne, parametar tipa `int` ima vrednost 0
- Može i `t.operator@@(k)` ili `operator@@(t, k)`, gde je  $k \neq 0$ , ali je besmisleno

- Preklapanje operatora `[]`

- Mora biti nestatički metod, i ne može biti globalna funkcija
- Kod standardnog indeksiranja indeksni izraz mora biti celobrojnog tipa

- Kod prekloppljenog indeksiranja indeksni izraz može biti proizvoljnog tipa  
`X[ind] = x.operator[](ind)`
  - Preklapanje omogućava `o[i]`, gde je o objekat date klase
  - Primer primene → kontejner za niz koji prati maksimalni i minimalni indeks i prati prekoračenja
  - Dve varijante
    1. `T& operator[](ind) // Mozemo da menjamo objekat na ind`
    2. `const T& operator[](ind) const // Mozemo samo da citamo objekat na ind`
  - Prekloppljeni operator `[]` nije indeksiranje, već koristi notaciju indeksiranja
  - Deluje na objekat klase, a ne na niz objekata
  - Takođe, ne radi pokazivačka aritmetika, što je i logično
  - Objekat je kolekcija elemenata, i operatorom `[]` pristupamo nekoj komponenti
- Preklapanje operatora `()`
    - Mora biti nestatički metod, i ne sme biti globalna funkcija
    - Proizvoljan broj parametara proizvoljnog tipa  
`f(a1, a2, ..., an) = f.operator()(a1, a2, ..., an)`  
`o(a1, a2, ..., an) // o je objekat neke klase`
    - Klasa sa `()` operatorom → funkcionalna klasa
    - Objekat funkcionalne klase → funkcionalni objekat
  - Preklapanje operatora `->`
    - Mora biti nestatički metod, i ne sme biti globalna funkcija
    - Bez parametara iako je binarni operator  
`o->clan = (o.operator->())->clan`
    - Ili vraća neki pokazivač na objekat klase koja sadrži `clan`, ili objekat (ili referencu) klase za koju je takođe definisan operator `->`
    - Jedna od primena su pametni pokazivači
- 
- ```
// Pratimo koliko puta smo pristupili određenom objektu
struct X { int val; };
class Xptr {
    X* p; int bp;
public:
    Xptr(X *px): p(px), bp(0) { }
    X& operator*() { bp++; return *p; }
    X* operator->() { bp++; return p; }
};
int main() {
    X x; Xptr pp=&x;
    (*pp).val = 1;
    int i = pp->val;
}
```
- 

- Preklapanje operatora (tip)
  - Mora biti nestatički metod, i ne sme biti globalna prijateljska funkcija
  - Drugi način za konverziju klasnih tipova  
 (Prvi je pomoću konstruktora konverzije)
  - Unarni operator `operatorT()`
    1. Konverzija tipa klase koje je objekat u tip `T`
    2. `T` može biti standardni, pokazivački ili klasni tip
  - Nema parametara
  - Tip rezultata ne sme da bude naveden u deklaraciji/definiciji jer se podrazumeva na osnovu imena
  - Pozivanje
    1. `T(x), (T)x <=> X.operatorT()`
    2. `static_cast<T>(x)`

- Razlike između konstruktora konverzije i preklopljenog operatora
  1. Može da se koristi za T koje je standardni tip
  2. Operand x mora biti objekat klase X, odnosno ne može biti primitivan tip
- T(x) ne može da se koristi ako je tip sa većim brojem reči  
`unsigned long (x) ≠ (unsigned long) x`
- Konverzija može biti i implicitna
- Ako ne želimo implicitnu konverziju dodajemo `explicit`
- Konverzija se primenjuje automatski, ako je izbor jednoznačan
  1. Ako imamo konstruktor konverzije i `operatorT, t=x X`
  2. Ako su definisane konverzije `T(x), X(t)` i `operator+, x+t X`
- Pri prenosu paramtera po referenci, rezultat konverzije stvarnog argumenta je privremeni objekat, pa se u funkciji prenosi njegova adresa, i izmene u funkciji se odnose samo na taj objekat

---

```
class X {
public:
    operator int() { return 1; }
    explicit operator double() { return 2; }
};

/*...
int a=X();           // a == 1
int b = (int) X();   // b == 1
double c = (double) X(); // c == 2.0
double d = X();       // d == 1.0
```

---

- Preklapanje `new` i `delete`

- Preuzimanje kontrole nad alokacijom memorije
- Oba metoda su statička, čak i ako nisu eksplisitno tako deklarisane
- Pozivaju se za objekat koji u datom trenutku ne postoji
- Unutar tela ovih funkcija
  1. Ne treba eksplisitno pozivati konstruktor/destruktur
  2. Ove operacije se pozivaju implicitno pre (kod `new`) ili posle (kod `delete`)
- Služe samo da rezervišu ili oslobođe prostor za objekat

- Preklapanje `new`

- `void* operator new (size_t sz, T1 t, ..., TN tn)`
- `void* operator new[] (size_t sz, T1 t, ..., TN tn)`
- `size_t` je celobrojni tip definisan u `<stddef.h>` (C), odnosno u `<cstddef>` (C++)
- Parametar `sz` daje veličinu prostora koji treba alocirati u bajtima
- Argument za `sz` je `sizeof(T)`, i formira se na osnovu tipa T
- Opciona lista izraza je inicijalizator
- `operator new` vrati pokazivač na alocirani prostor
- Klasa može imati više preklopljenih `new`

---

```
new (arg2, ..., argN) T(izraz, ..., izraz)
new (arg2, ..., argN) T[duzina]
```

---

- Preklapanje `delete`

- `velicina` određuje prostoru koji treba oslobođiti u bajtovima
- Ako ga nema, funkcija mora sama da odredi veličinu na osnovu ranije alokacije
- Ne vraća rezultat
- Samo po 1 `delete` po klasi za podatak i niz

---

```
void operator delete (void* pokazivac);
void operator delete (void* pokazivac, size_t velicina);
void operator delete[] (void* pokazivac);
void operator delete[] (void* pokazivac, size_t velicina);
```

---

- Dohvatanje ugrađenih `new` i `delete`

- Eksplisitno → korišćenjem ::
- Implicitno → samo za druge tipove
- `new` i `delete` ne mogu biti virtuelni, ali se nasleđuju normalno

---

```
#include <cstddef>
using namespace std;
class XX {
public:
    void* operator new (size_t sz) { return new char [sz]; } // ugradjeni new
    void operator delete (void* p) { delete [] p; } // koristi se ugradjeni delete
};
```

---

- Standardni U/I tokovi

- Kao ni C, ni C++ nema ugrađene U/I naredbe i realizuju se pomoću standardne biblioteke
- Standardna biblioteka C++ je napisana u duhu OOP-a
- Mogu se koristiti i C funkcije (`scanf`, `printf`), ali nije preporučljivo
- C++ biblioteka je `<iostream>`
- Dve osnovne klase
  1. `istream` - *input stream*
  2. `ostream` - *output stream*
- Iz navedenih klasa izvedene su i klase `ifstream` i `ofstream` za rad sa datotekama
- Pristup datotekama isključivo preko klasa

- Standardni objekti i operacije za U/I

- Dva globalna objekta
  1. `cin` iz `istream` - obično povezan sa tastaturom
  2. `cout` iz `ostream` - obično povezan sa monitorom
- Preklopjeni su operatori `<< i >>` za sve standardne tipove
 

```
istream& operator>>(istream& is, T& t);
ostream& operator<<(ostream& os, const T& t);
```
- Rezultati su reference pa samim tim dozvoljavaju **ulančavanje**

- Korišćenje operatora `<< i >>`

- Vraćaju referencu na levi operand → višestruki U/I u istoj naredbi
- Sleva-udesno asocijativni → podaci se ispisuju/učitavaju u prirodnom redosledu
- Za jednostavne U/I operacije

- Preklapanje operatora `<< i >>`

- Mora uvek da se koristi globalna prijateljska funkcija zbog toga što je levi operand tipa `istream&` ili `ostream&`, a ne trenutni objekat klase

---

```
#include <iostream>
using namespace std;
class Kompleksni {
    double real, imag;
    friend ostream& operator<<(ostream&, const Kompleksni&);
public: /* ... */
};
ostream& operator<<ostream& os, const Kompleksni& c{
    return os<<"("<<c.real<<", "<<c.imag<<")";
}
void main() {
    Kompleksni c(0.5, 0.1);
    cout<<"c="<<c<<endl; // ISPIS: c=(0.5,0.1)
}
```

---

- Operatori za nabranjanja

- Celobrojni tip, ali su dozvoljene
  1. Operacije dodele istom tipu nabranja
  2. Operacija konverzije u celobrojnu vrednost - može i implicitno (osim za nabranja sa ograničenim dosegom)
  3. Operacija konverzije iz celobrojne vrednosti - samo eksplisitno
- Dozvoljeno preklapanje operatora koji se ne preklapaju kao metodi
  1. Ne može `=`, `[]`, `()`, `->`, `(tip)`, `new`, `delete`
  2. Ugrađeni `= i (tip)` zadovoljavaju realne potrebe, ostali nemaju mnogo smisla

---

```

enum Dani { PO, UT, SR, CE, PE, SU, NE};
inline Dani operator+(Dani d, int k) {
    k=(int(d)+k)%7;
    if(k<0) k+=7;
    return Dani(k);
}
inline Dani& operator+=(Dani& d, int k) {
    return d=d+k;
}
inline Dani& operator++(Dani& d) {
    return d=Dani(d<NE?int(d)+1:PO);
}
inline Dani operator++(Dani& d, int) {
    Dani e(d);
    ++d;
    return e;
}

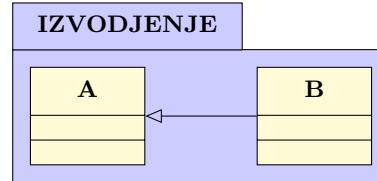
```

---

## ER KOLOKVIJUM

## 7 Izvođenje klasa

- Pojam specijalizacije opšteg
  - Jedna klasa može biti podvrsta neke druge klase
  - Klasa B je specijalni slučaj (*a-kind-of*) klase A
  - Objekat klase B je (*is-a*) i objekat klase A
- Izvođenje i nasleđivanje
  - Objekti klase B imaju sve osobine klase A, sa još nekim, specifičnim, osobinama
  - Specijalnija klasa B se izvode iz generalnije (opštije) klase A
  - Klasa B nasleđuje
    1. Strukturne karakteristike (attribute)
    2. Karakteristike ponašanja (metode)
  - Pored nasleđenih ima i svoje čanove
  - Nasleđivanje (*inheritance*) ili generalizacija/specijalizacija



Slika 1: Klasa B je izvedena iz klase A

- Termini

| A                    | B               |
|----------------------|-----------------|
| 1. <i>base</i>       | <i>derived</i>  |
| 2. <i>superclass</i> | <i>subclass</i> |
| 3. <i>parent</i>     | <i>child</i>    |

- Jezik je **objektno-orientisan** ako podržava nasleđivanje, a **objektno-baziran** u suprotnom
  - Pri izvođenju nije potrebno menjati postojeće klase
  - Postojeće klase se **ne** prevode ponovo
- Definisanje izvedene klase
 

```
class ime: ime_osnovne { ... };
```

    - Ako postoji samo jedna osnovna klasa → **jednostruko izvođenje**
    - Ako postoji više osnovnih klasa → **višestruko izvođenje**
    - Izvođenje može biti i u više koraka, gde je izvedena klasa osnovna klasa za sledeće izvođenje
- NIJE  
VIŠESTRUKO
- Konačna klasa
    - Modifikator **final** → dalje se ne može izvoditi
    - **final** nije rezervisana reč, može biti i identifikator ali se ne preporučuje
  - Primer izvođenja
    - Objekat izvedene klase sadrži
      1. Bezimeni podobjekat osnovne klase koji sadrži nasleđene članove
      2. Specifične čanove navedene u definiciji izvedene klase
  - Pristup nasleđenim članovima
    - Javnim članovima osnovne klase se pristupa isto kao i članovima izvedene klase
  - Konstruktor, destruktor, **operator=**
    - Izvedena klasa ih **ne** nasleđuje
    - Kao i za osnovne klase, postoje ugrađeni
      1. Podrazumevani konstruktor praznog tipa
      2. Kopirajući konstruktor koji kopira član po član (plitka kopija)
      3. Premeštajući konstruktor koji kopira član po član
      4. Destruktor praznog tela
      5. Kopirajući operator dodele koji vrši dodelu kopiranjem, član po član
      6. Premeštajući operator dodele koji vrši dodelu premeštanjem, član po član
  - Prava pristupa
    - Članovi izvedene klase imaju prava pristupa javnim članovima osnovne klase, a nemaju pristup privatnim
    - Posebno pravo pristupa - **zaštićeno**  
**protected:**
      - Označava da klase kojima imaju pravo pristupa metode te i izvedenih klasa - *zašto te?*
      - Zaštićenim članovima može da se pristupi iz metoda izvedene klase kao nasleđenim članovima
- 
- ```
class Osnovna {
    int pb;
protected: int zb;
public:   int jb;
};

class Izvedena : public Osnovna {
public:
    void m(int x) {
        jb=zb=x;    // OK
        pb=x;       // ERROR: privatni clan
        Osnovna c;
        o.zb=x;     // ERROR: ne moze preko osnovne
```

```

    }
};

void f(){
    Osnovna b;
    b.pb=1; // ERROR: privatni clan
    b.zb=1; // ERROR: zaštićeni clan
    b.jb=1; // OK
}

```

---

- Načini izvođenja

- Određuje ga modifikator ispred imena osnovne klase
- Može biti
  1. Javno **public**
  2. Zaštićeno **protected**
  3. Privatno **private**
- Kaže se i za osnovnu klasu da je javna, zaštićena ili privatna
- Podrazumevano izvođenje je privatno
- Način izvođenja određuje nasleđivanje prava pristupa
- Određuje se stepen kontrole pristupa članovima osnovne klase preko objekta izvedene klase
- Objekat izvedene klase može biti osnovna klasa u dalje izvođenju
- Ne utiče na pravo pristupa iz metoda dotične izvedene klase

Izvođenje	član osnovne klase		
	<b>javni</b>	<b>zaštićeni</b>	<b>privatan</b>
<b>javno</b>	javni	zaštićen	privatan
<b>zaštićeno</b>	zaštićen	zaštićen	privatan
<b>privatno</b>	privatan	privatan	privatan

- Eksplisitna promena prava pristupa

- Može eksplisitno da se promeni pravo pristupa nasledenom članu
  1. Da se uveća u odnosu na originalno (zaštićeni postaje javni)
  2. Može da se restaurira na originalnu koje je umanjeno načinom izvođenja
  3. Delimično da se restaurira
  4. Da se umanji u odnosu na originalno
- Ne može da se promeni pravo pristupa privatnom članu osnovne klase
- Postiže se uvozom člana u odgovarajuću sekciju izvedene klase **using klasa::član**
- Može i bez **using**, ali izbegavati

---

<pre> <b>class</b> O {     int pb; <b>protected:</b> int z1b, z2b; <b>public:</b>   int j1b, j2b; };  <b>class</b> PI: O { <b>public:</b>     <b>using</b> O::j1b;           // Vracanje starog prava pristupa     <b>using</b> O::z1b;           // Povecanje prava pristupa     PI() { z2b = 1; }  <b>protected:</b>     <b>using</b> O::j2b;           // Delimicno vracanje prava pristupa };</pre>	<pre> <b>class</b> IPI: <b>public</b> PI { <b>public:</b>     IPI() {         j1b=0;      // OK         j2b=0;      // OK         z1b=0;      // OK         z2b=0;      // ERROR     } };  <b>int</b> main() {     PI pi;     pi.j1b=0;      // OK     pi.j2b=0;      // ERROR: zaštićeni clan     pi.z1b=0;      // OK     pi.z2b=0;      // ERROR: privatni clan }</pre>
---	--

---

- Razlika između privatnog i javnog izvođenja
  - Javno izvođenje realizuje koncept nasleđivanja
    1. **B je vrsta A** (*a-kind-of*)
    2. Izvedena klasa zadržava interfejs roditelja
    3. Objekat izvedene klase može da zameni objekat osnovne
    4. Jedino pravo nasleđivanje
  - Privatno izvođenje realizuje koncept sadržanja
    1. **A je deo B** (*a-part-of*)
    2. Semantički slično kao kad B sadrži atribut tipa A
    3. Nije nasleđen interfejs jer sve nasleđeno postaje privatno
    4. Objekat izvedene klase ne može da zameni objekat osnovne
  - Zaštićeno izvođenje
    1. Unutar izvedene klase → koncept nasleđivanja
    2. Van izvedene klase → koncept sadržanja