OOP1

Objektno-orijentisano 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 pull 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. - **FORTRAN**
     - 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 nabrajanja*
- Dodatni koncepti
  1. Zasebno prevođenje modula - **FORTRAN, 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 **Ade**
- 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 *(atribut ili polje)*
    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 $\begin{cases} \textbf{Stanje} \\ \textbf{Ponašanje} \\ \textbf{Identitet} \end{cases}$
- 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 proglasi 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-orijentisani
     - 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)*
  - Određene 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[100];
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 BITNO
  - Pristup globalnom imeu 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
```

- Specifični dosezi
  - Oblast važenja funkcije imaju samo labele
  - `for` petlja
  - Ranije verzije kompajlera su imale doseg promenljivih koji je bio 1 blok **van** (`for` (MS VC++.6)
  - 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 *(tranzijentni)* 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 deklarisan 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 nedefinisan
    2. Odlaganje međurezultata 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* operatora
  - Postoji i konverzioni konstruktor
- Priduživanje 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 vrednost `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 statičkog 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 nabrajanja (`enum`)
  - Svaki `enum` je poseban celobrojni tip
  - Definisana je samo operacija dodele vrednosti
    1. Eksplicitna konverzija celobrojne vrednosti u tip nabrajanja je obavezna BITNO
    2. Ne otkriva se greška ako konvertovana vrednost nije u opsegu
  - U aritmetičkim i relacijskim izrazima, kao i pri dodeli 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 eksplicitna 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 nabrajanja
  - Numerička reprezentacija nabrajanja
  - Kompaktnije, podrazumeva se int
    `enum ime: pripadajući_tip {imenovane_konstante}`
  - Paziti opsege

- Nabrajanja sa ograničenim dosegom
  - Isti doseg kao i tip nabrajanja
  - Rešenje - `struct` ili `class` iza `enum`
  - Pristup konstanti sa `::`
  - Obavezna eksplicitna 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 destruktor 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 nece promeniti
    j++; // stvarni argument ce se promeniti
}
int main () {
    int si=0,sj=0;
    f(si,sj);
    cout<<"si="<<si<<", sj="<<sj<<endl;
}
Izlaz: si=0, sj=1
```

- 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 prevođenju
  - 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. Lakše se uoči u tekstu
    2. Da programeri ne bi koristili
  - Ako postoji potreba za eksplicitnim konverzijama → Preispitati projektne odluke
- Statička konverzija
  - Prenosive konverzije
    1. Između numeričkih tipova
    2. Između pokazivača i `void*`
    3. Nestandardne konverzije - definiše programer
  - Primenjuju se automatski kad su bezbedne
  - Eksplicitan 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 struktuirani korisnički tip koji obuhvata
    1. Podatke koji opisuju stanje objekta klase
    2. Funkcije namenjene definisanju operacija nad podacima
  - Klasa je formalni opis aptrakcije 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 klasa 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	BITNO
  - Kontrola pristupa je odvojena od koncepta dosega
    1. Odredi se postojanje
    2. Proveravanje prava pristupa

- Definisanje klase
  - Atributi
    1. Mogu da budu i inicijalizovane - od C++11
    2. Ne moguda 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* fajl (`.h`)
  - Nepotpuna definicija klase je **deklaracija**
  - Pre defincije, 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 `[]`
  - Prenošenenje objekta kao argumenata po vrednosti, referenci ili pokazivaču
  - Vraćanje objekta iz funkcije po vrednosti, referenci ili pokazivaču
  - Preklapanje operatora može redefinisati dosta gorenavedenog
- Pokazivač `this`
  - Pokazivač na tekući objekat
  - Unutar svako nestatičkog objekta je implicitno, `this` je skriveni argument svakog metoda
    `objekat.f()` $\sim$ `f(&objekat)`
  - Konstanti pokazivač na klasu čiji je metod član
    Klasa `X`, `this` $\rightarrow$ `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 atribute
    Eksplicitna konverzija može da probiju konrolu konstantnosti
  - U inspektoru `this` je `const X* 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. Dozboljen 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 \ *overloadoded*
- 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
  - U listi inicijalizatora definicije delegiruajućeg 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

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

- Eksplicitni poziv konstruktora
  - Ovakav poziv kreira primvremeni objekat klase pozivom odgovarajućeg konstruktora
  - Isto se dešava ako se u inicijalizatoru objekta eksplicitno 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 inicijalizaicju 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. Eksplicitno
       `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 kopij 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 inicijalizuje 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 premestimo njegove dinamičke delove u odredišni objekat
  - Nema kopiranja dinamičkih delova
  - Posledica → premeštajući konstruktor je efikasniji od kopirajućeg
  - Modifikovati izvorišni objekat da njegova destrukcija 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 eksplicitno 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; }
```

- Konverzioni 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))` eksplicitno
  - 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

```
//Konverzioni 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 $\sim$ ispred imena
  - Nema tip rezltata i ne može imati parametre $\rightarrow$ najviše 1 po klasi
  - Destruktor se piše kada treba osloboditi memoriju i ostale resurse
  - Česta potreba $\rightarrow$ klasa sadrži članove koji su pokazivači ili reference na druge objekte
    Dobra praksa tad $\rightarrow$ 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 objekte koristeći `delete`
  - Pri uništavanju dinamičkog niza - u smeru opadajućih indeksa
  - Redosled je uvek obrnut od konstruktora
  - Eksplicitno pozivanje
    `X.~X(), px->~X(), this->~X()`
    - Ne preporučuje se, objekat nastavi da živi i posle ovoga
  - Posle izvršenja automatskog destruktora se oslobađa zauzeta memorija
- Statički (zajednički) atributi
  - Pri stvaranju objekta klase $\rightarrow$ 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 $\rightarrow$ 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
  - Deklarišu 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
  - Direktan pristup statičkim članovima
    `<klasa>::<ime_funkcije>(argumenti);`
  - Može se pozvati za konkretan objekat, ali izbegavati
    Levi operand tada samo nađe tip bez ikakvog izračunavanja
  - Mogu se pozivati i pre stvaranja 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) xx.g(); // ovako moze
    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
    xx.g(); // ovako moze, ali nije preporucljivo
    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 privatan
};
int main() {
    X x; // ERROR
    X* px=X::kreiraj(); // OK
}
```

- Prijatelji klasa
  - 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 ☹
    3. Nije tranzitivno
  - Reguliše isključivo pravo pristupa, a ne i oblast važenja i vidljivost identifikatora
- Prijateljski funkcije
  - Nisu članice klasa 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 operator, 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 nabrajanja `enum` i tipova `typedef`
  - Ugnježdena klasa se nalaziu 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 ⟵ Samo od konkretnog objekta
    2. Konstanti tipa nabrajanja 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 ☹⟵ 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 eksplicitnim korišćenjem `public:` i `private:`
  - C++ struktura može imati i metode
  - Strukture se koriste za definisanje struktuiranih 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 nabrajanja
    3. Trajnih podataka (statičkih atributa, statičkih lokalnih i globalnih)
    4. Spoljašnjih (`extern`) podataka i funkcija
  - Metodi lokalne klase moraju da se definišu unutar definicije klasa
  - Lokalna klasa ne može da ima statičke atribute, 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;
```