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 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. - **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 *(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 $\begin{cases} \textbf{Stanje} \\ \textbf{Ponašanje} \\ \textbf{Identitet} \end{cases}$
- 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()` ∼ `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 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 \ *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
  - 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 ∼ ispred imena
  - Nema tip rezltata 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 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 → 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
  - 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

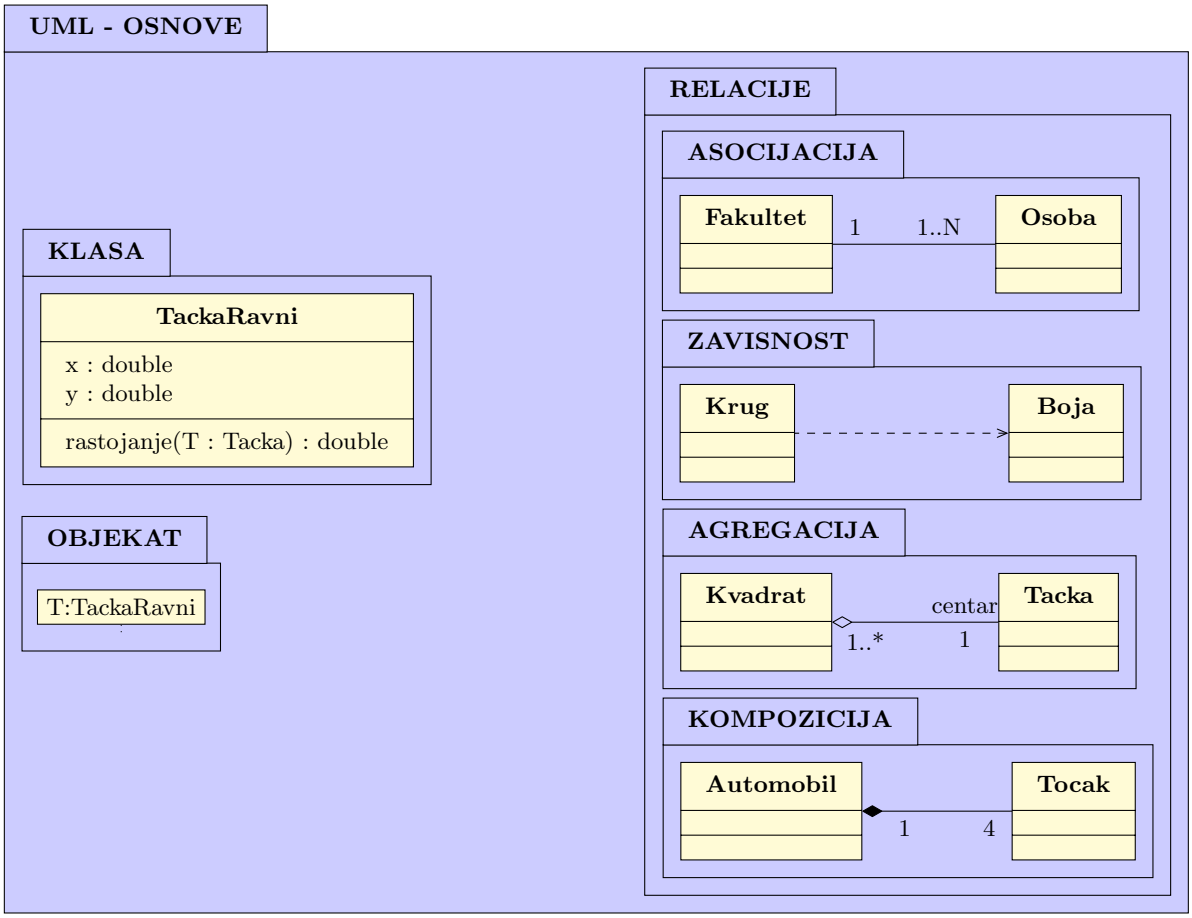
```
class Alfa {... public: int a, b; };

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;
```

- 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` $\not\Longrightarrow$ `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 definišu 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
  - Opreatorski metod ne dozvoljava konverziju levog operanda
- Unarni i binarni operatori
  - Unarni
    ```
    tip operator@() // Metod
    tip operator@(X x) // Globalna funkcija
    ```
  - Binarni
    ```
    tip operator@(X x_desno)
    tip operator@(X x_levo, X x_desno)
    ```
- Inicijalizacija i dodela vrednosti
  - Inicijalizacija podrazumeva da objekat još ne postoji, dok dodela 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 konverzioni konstruktor
  - U oba slučaja može biti pozvan konstruktor sa više paramteara, ako svi ostali parametri imaju podrazumevane vrednosti
  - Dodela → 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 odele (*deep copy*)
- Preklapanje operatora dodele
  - Mora biti nestatički metod
  - Ugrađena varijanta vriši 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 dodela 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 destruktor, konstruktor kopije ili operator dodele, sva je prilika da trea 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 { `kopiraj(x)`, `premesti(x)`, `brisi(x)`
    - Ovi metodi se koriste na mnogo mesta, pa je zgodno napisati ih samo jednom
    - Alternativa - *copy-and-swap* koja objedinjuje dva tipa operatora dodele
      Umesto po referenci, prosledi se kopija desnog operanda na stek, zatim se izvrši razmena objekata i destruktor obriše ostatke levog operanda kada skidamo objekta sa steka

```
/*...*/
int main(){
    X x = X("OOP");      // X(const char*); Konverzioni konstruktor
      y = x,             // X(const X&); Konstruktor kopije
      z = X("me cini");  // X(const char*); X(const X&&); Premestajuci konstruktor
    x = y;               // operator=(const X&); Kopirajuca dodela
    y = X("tuznim");     // X(const char*); operator=(const X&&); Premestajuca dodela
/* Paziti na optimizacije kompajlera. Nekad kompajler izvrsi nesto sto ne ocekujemo*/

}
```

- Preklapanje operator `++` i `--`
  - Problem nastaje jer postoje i prefiksna i postfiksna verzija
  - Prefiksna
    `T& operator@@()`
    `T& operator@@(T&)`
  - Postfiksna
    `T operator@@(int)`
    `T operator@@(T&, int)`
  - 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 preklopljenog 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`
- Preklopljeni 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 → funkcijska klasa
  - Objekat funkcijske klase → funkcijski 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 odredjenom 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* opereator->() { 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` $\iff$ `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 eksplicitno tako deklarisane
  - Pozivaju se za objekat koji u datom trenutku ne postoji
  - Unutar tela ovih funkcija
    1. Ne treba eksplicitno pozivati konstruktor/destruktor
    2. Ove operacije se pozivaju implicitno pre (kod `new`) ili posle (kod `delete`)
  - Služe samo da rezervišu ili oslobode 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 prostoro koji treba osloboditi u bajtovima
  - Ako ga nema, funkciaj 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`
  - Eksplicitno → 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
  - Preklopljeni 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& oeprator<<(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 nabrajanja

– Celobrojni tip, ali su dozvoljene

1. Operacije dodele istom tipu nabrajanja
2. Operacija konverzije u celobrjnu vrednost - može i implicitno (osim za nabrajanja sa ograničenim dosegom)
3. Operacija konverzije iz celobrojne vrednosti - samo eksplicitno

– Dozvoljeno preklapanje operatora koji se ne preklapaju kao metodi

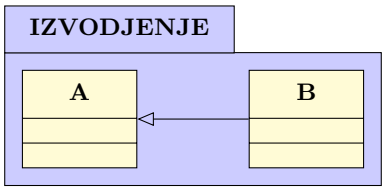
1. Ne može `=, [], (), ->, (tip), new, delete`
2. Ugrađeni `= i (tip)` zadovoljavaju realne potrebe, ostali nemaju mnogo smisla

```
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 (atribute)
    2. Karakteristike ponašanja (metode)
  - Pored nasleđenih ima i svoje čanove
  - Nasleđivanje (*inheritance*) ili generalizacija/specijalizaicija



Slika 1: Klasa B je izvedena iz klase A

– Termini

	A	B
1.	*base*	*derived*
2.	*superclass*	*subclass*
3.	*parent*	*child*

- Jezik je **objektno-orijentisan** 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

```
class Osnovna { private: int i;
public: void f();
};
class Izvedena: public Osnovna { private: int j;
public: void g();
};
int main() {
    Osnovna b; Izvedena d;
    b.f();
    b.g(); // ERROR: g je metod izvedene
    d.f(); d.g(); // OK
}
```

- 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 de klase kojem 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: privatan clan
    b.zb=1; // ERROR: zasticeni 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
  - Određu se 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**		
	javni	**zaštićeni**	**privatan**
javno	javni	zaštićen	privatan
zaštićeno	zaštićen	zaštićen	privatan
privatno	privatan	privatan	privatan

- Eksplicitna promena prava pristupa
  - Može eksplicitno da se promeni pravo pristupa nasleđenom 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::clan`
  - Može i bez `using`, ali izbegavati

```
class O {
            int pb;
protected:  int z1b, z2b;
public:     int j1b, j2b;
};
class PI: O {
public:
    using O::j1b;
    // Vracanje starog prava pristupa
    using O::z1b;
    // Povecanje prava pristupa
    PI() { z2b = 1; }
protected:
    using O::j2b;
    // Delimicno vracanje prava pristupa
};
```

```
class IPI: public PI {
public:
    IPI() {
        j1b=0;      // OK
        j2b=0;      // OK
        z1b=0;      // OK
        z2b=0;      // ERROR
    }
};
int main() {
    PI pi;
    pi.j1b=0;       // OK
    pi.j2b=0;       // ERROR: zasticen clan
    pi.z1b=0;       // OK
    pi.z2b=0;       // ERROR: privatni clan
}
```

- 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
- Sakrivanje članova
  - Redefinisanje identifikatora člana osnovne klase u izvedenoj pritom sakrivajući originalni
  - Pristup originalu:
    `osnovna_klasa::clan`
  - Ako u izvedenoj klasi napišemo neki metod koji ima ime kao neki metod u osnovnoj klasi
    1. Takav metod sakriva sve nasleđene metode istog imena
    2. Iz metoda izvedene klase se ne može pristupiti skrivenim metodama
    3. Nasleđeni sakriveni metodi se ne mogu pozivati ni za objekte izvedene klase (osim preko pokazivača na osnovni tip objekta)
- Uvoz članova
  - U izvedenoj klasi sve sakrivene metode treba
    1. Redefinisati
    2. Restaurirati
  - Nije dobro da izvedena klasa sadrži samo neke metode osnovne
    1. Nije pravo nasleđivanje
    2. Korisnik klase očekuje isti ili prošireni interfejs
  - Uvoz
    `using::osnovna_klasa::ime_metoda`
  - Nisu vidljivi samo metodi osnovne klase sa istim potpisom kao izvedena klasa
  - Ne može da se restaurira vidljivost pojedinačnog metoda osnovne klase

```
class O {
public:
    int a=1;
    void m1();
    void m1(int);
    void m2();
    void m2(int);
};
class I: public O {
public:
    using O::m2;
    int a=2;
    void m1(int);
    void m2(int);
    void m(){
    int x=a;       // x=I::a
    int y=O::a;    // x=O::a
    m1();          // ERROR
    m1(x);         // I::m1(int)
    O::m1();       // O::m1();
    O::m1(y);      // O::m1(int);
    m2();          // O::m2();
    m2(x);         // I::m2(int)
    O::m2(y);      // O::m2(int);
} };
void f(){
    I i;
    int p=i.a;     // I::a
    int q=i.O::a;  // O::a
    i.m1();        // ERROR
    i.O::m1();     // O::m1()
    i.m1(p);       // I::m1(int)
    i.O::m1(q);    // O::m1(int)
    i.m2();        // O::m2()
    i.m2(p);       // I::m2(int)
    i.O::m2(q);    // O::m2(int)
```

- Uvoz konstruktora
  - Konstruktori osnovne klase se ne nasleđuju
  - Može se izvršiti uvoz svih konstruktora osnovne klase
    `using ime_klase::ime_klase`
  - Na taj način se generišu konstruktori izvedene sa istim potpisima
  - Generisani konstruktor izvedene klase imaju prazno telo, samo pozivaju konstruktore osnovne sa istim potpisom

```
class A { public: A(int i){...} };
class B: public A {public: using A::A; }
B b(1); // poziva se A::A(1)
```

- Izvođenje struktura
  - Strukture su ravnopravne sa klasama
    1. Izvođenje strukture iz klase/strukture - javno
    2. Izvpđenje klase iz strukture - privatno
  - Unije se ne mogu izvoditi, niti se može izvoditi iz njih

```
struct OS { };
class OK { };
class IKS: OS{ };     // privatno izvodjenje
class IKK: OK{ };     // privatno izvodjenje
struct ISK: OK{ };    // javno izvodjenje
struct ISS: OS{ };    // javno izvodjenje
```

- Konstruktori i destruktori
  - Prilikom kreiranja objekata izvedene klase poziva se konstruktor te klase, ali i konstruktor osnovne klase
  - Analogno za uništavanje objekata (ovo se ne odnosi na redosled, već samo da se obe operacije izvršavaju)
  - Prenos parametara u konstruktor osnovne klase
    1. U listu inicijalizatora konstruktora izvedene klase moguće je upisati i inicjalizator osnovne klase
    2. Inicijalizator osnovne klase se sastoji od imena klase i argumenata poziva konstruktora
  - Nije moguće vršiti inicijalizaciju pojedinih nasleđenih atributa
- Redosled konstrukcije
  - Pri kreiranju objekta izvedene klase
    1. Inicijalizuje se podobjekat osnovne klase
       - Pozivom odgovarajućeg konstruktora osnovne klase
       - U slučaju višestrukog izvođenja, inicijalizacija po redosledu u definiciji, a ne u listi inicijalizatora
    2. Inicijalizuju se specifični atributi
       - Po redosledu navođenja u definiciji a ne u listi inicijalizatora
       - Klasni tipovi pozivom odgovarajućeg konstruktora
       - Ugrađeni tipovi na osnovu inicijalizatora
       - Nizovi objekata po redosledu rastućeg indeksa podrazumevanim konstruktorom
    3. Telo konstruktora
  - Pri uništavanju objekta izveden klase, redosled poziva destruktora je obratan
- Konverzija
  - Objekat javno izvedene klase je i objekat osnovne (*Liskov substitution principle*)

1. Pokazivač na objekat izvedene klase može se implicitno konvertovati u pokazivač na objekat osnovne klase
2. Pokazivač na objekat osnovne klase mora se eksplicitno konvertovati u pokazivač na objekat izvedene klase
3. Isto važi za reference
4. Objekat osnovne klase se može inicijalizovati objektom izvedene klase
5. Objektu osnovne klase se može dodeliti objekat izvedene klase

- Objekat privatno/zaštićeno izvedene klase nije i objekat osnovne klase
- Pokazivač na objekat takve klase se može implicitno konvertovati u pokazivač na objekat osnovne klase samo unutar izvedene klase

- Pojam polimorfizma
  - Izvedena klasa može da definiše određene metode na svoj način
- Decentralizacija odgovornosti
  - Svaki objekat prepozna kojoj izvedenoj klasi pripada iako možemo da mu se obraćamo kao objektu osnovne klase
  - **Polimorfizam** - svaki objekat izvedene klase izvršava metod onako kako je to definisano u njegovoj izvedenoj klasi, iako mu se pristupa kao objektu osnovne klase
  - *Polymorphism* = *poly* (više) + *morph* (oblik)
  - Isti metod ima više oblika
  - Mehanizam je potpuno dinamički - klasa objekta se određuje pri **izvršavanju** programa
- Virtuelni (polimorfni) metodi
  - Virtuelni metodi su metodi osnovne klase koji se u ivzedenim klasama mogu redefinisati
  - Metod u izvedenoj klasi nadjačava (*override*) metod osnovne klase
  - Polimorfna klasa je ona koja sadrži barem jedan virtuelni metod
    `virtual tip ime(...)`
  - Prilikom deklarisanja virtuelnih metoda u izvedenim klasama → ne mora `virtual`
  - Pozivom preko pokazivača/referece na osnovnu klasu - izvršava se onaj metod koji pripada klasi pokazanog objekta
- Redefinisanje virtuelnih metoda
  - Deklaracija virtuelnog metoda u izvedenoj klasi mora **potpuno** da se slaže sa deklaracijom u osnovnoj klasi (broj + tipovi argumenata + tip rezultata)
  - U specijalnom slučaju, ako je tip rezultata referenca/pokazivač na osnovnu kasu, tada, redefinisani metod može da vrati pokazivač/referencu na javno izvedenu klasu iz date osnovne
  - Neslaganje potpisa → sakrivanje metoda
  - Samo razlika u tipu rezultata → greška
  - Virtuelni metod ne mora da se redefiniše u svakoj izvedenoj klasi - tamo gde nije, naseldi osnovni metod
- Pozivanje virtuelnih metoda
  - Mehanizam se aktivira samo ako se objektu pristupa indirektno - preko reference ili pokazivača

```
class Osnovna { public: virtual void f(); };
class Izvedena : public Osnovna { public: void f(); };
void g1(Osnovna b) { b.f(); }
void g2(Osnovna *pb) { pb->f(); }
void g3(Osnovna &rb) { rb.f(); }
int main () {
Izvedena d;
g1(d);                              // poziva se Osnovna::f
g2(&d);                             // poziva se Izvedena::f
g3(d);                              // poziva se Izvedena::f
Osnovna *pb=new Izvedena; pb->f();  // poziva se Izvedena::f
Osnovna &rb=d; rb.f();              // poziva se Izvedena::f
Osnovna b=d; b.f();                 // poziva se Osnovna::f
delete pb; pb=&b; pb->f();          // poziva se Osnovna::f
}
```

- Modifikatori `override` i `final`
  - Implicitna vrednost metoda u izvedenoj klasi nije rObUsNa
  - Kao modifikator (iza liste parametara) može da se navede modifikator `override`
  - Eksplicitno se iskazuje da metod nadjačava odgovarajući virtuelni metod
  - Prevodilac prijavljuje grešku ako nema odgovarajućeg virtuelnog metoda

- Modifikator `final` sprečava nadjačavanje virtuelnih metoda u izvedenim klasama
- Ovi modifiaktori **nisu** rezervisane reči, niti deo potpisa metoda
- Moguća kombinacija i `override` i `final`

```
class A { public:
virtual void vm1();
virtual void vm1(int);
void m1();
};
class B: public A {public:
void vm1() override;
void vm1(int) override final;
void vm1(float) override;        // ERROR - nema A::vm1()
void vm2() override;             // ERROR - nema A::vm2()
void m1() override;              // ERROR - A::m1() nije virtuelna
void m2() final;                 // ERROR - B::m2() nije virtuelna
};
class C: public B { public:
void vm1(int);                   // ERROR - B::vm1(int) je konacna
};
int override=1; int final=2;     // U redu, nije preporucljivo
```

- Dinamičko vezivanje
  - Dinamičko vezivanje - *Dynamic binding* je mehanizam koji obezbeđuje da se metod koji se poziva određuje
    1. Po tipu objekta
    2. Ne po tipu pokazivača ili reference na taj objekat
  - Odlučivanje koji će se virtuelni metod pozvati se obavlja **dinamički** - u toku programa
  - Po tome se razlikuje u odnosu na preklapanje
- Implementacija virtuelnih poziva
  - Za jednostruko nasleđivanje
    1. Za virtuelne metode nove klase → jedinstven kod metoda
    2. Za svaku polimorfnu klasu postoji tabela pokazivača na virtuelne metode te klase - **TVF** - Tabela virtuelnih funkcija
- Osobine polimorfnih metoda
  - Virtuelni metodi ne mogu biti statički
  - Ako nam treba statički metod sa polimorfnim ponašanjem
    1. Obaj metod treba da poziva nestatički virtuelni metod za neki objekat
    2. Pokazivač/referenca na taj objekat se prenosi kao argument poziva statičkog metoda
  - Globalne prijatljeske funkcije **ne mogu** biti polimorfne (ako je potrebno, mora biti implementirano kao i kod statičkih metoda)
  - Virtuelni metodi mogu biti prijatelji drugih klasa
- Virtuelni destruktor
  - Konstruktor ne može biti virtuelna funkcija jer se poziva pre nego što se objekat kreira
  - Destruktor može biti virtuelna funkcija
  - Tek u vreme izvršenja se zna koji destruktor pozivamo
  - Kada se uništava dinamički objekat, destruktor osnovne klase se uvek izvršava, ili kao jedini, ili posle destruktora izvedene klase
  - Kada neka klasa ima virtuelnu funkciju, verovatno i njen destruktor treba da bude virtuelan
  - Unutar destruktora izvedene klase ne treba pozivati destruktor osnovne

```
class OVD { public: virtual ~OVD(); };
class IVD: public OVD { public: ~IVD(); };
class OnVD{ public: ~OnVD(); };
class InVD: public OnVD { public: ~InVD(); };
void oslobodi (OVD *pb) { delete pb; }
void oslobodi (OnVD *pb) { delete pb; }
int main () {
OVD *pb=new OVD; oslobodi(pb);          // ~OVD()
IVD *pd=new IVD; oslobodi(pd);          // ~IVD(), ~OVD()
OnVD *pbn=new OnVD; oslobodi(pbn);      // ~OnVD()
InVD *pdn=new InVD; oslobodi(pdn);      // ~OnVD()
}
```

- Nizovi i izvođenje
  - U C++, niz objekata nije objekat
  - Niz objekata izvedene klase nije jedna vrsta niza objekata osnovne klase
  - Ako se niz objekata izvedene klase prenese funkciji koja očekuje niz osnovne klase, može doći do greške
  - Objekti osnovne klase su manj od izvedeni objekata, a funkcija smatra da je dobila niz objekata osnovne klase

```
class Osnovna { public: int bi; };
class Izvedena : public Osnovna { public: int di; };
void f(Osnovna *b, int i) { cout<<b[i].bi; }
int main () {
Izvedena d[5];
d[2].bi=77;
f(d,2); // nece se ispisati 77
}
```

- Prenos zbirki objekata funkcijama
  - Ako je u igri nasleđivanje pravilo je da se koriste nizovi pokazivača na objekte umesto nizova objekata
  - Nije dozvoljena konverzija `Izvedena**` u `Osnovna**`
- Apstraktni metodi i klase
  - Apstraktni metod je virtuelni metod koji nije definisan za osnovnu klasu
  - Umesto tela stoji `= 0`
  - Klasa koja sadrži bar 1 apstraktan metod naziva se apstraktnom klasom - *abstract class*
  - Apstraktna kalsa ne može imati objekte, već stiže samo za izvođenje
  - Mogu se formirati pokazivači i reference na apstraktnu klasu
  - Pokazivači i reference na asptraktnu klasu mogu samo da pokazuju na objekte izvedenih klasa

```
class Osnovna {
public:
virtual void cvf () = 0;            // apstraktni metod
virtual void vf ();                 // virtuelni metod
};
class Izvedena : public Osnovna {
public:
void cvf();
};
int main () {
    Izvedena izv, *pi=&izv;
    Osnovna osn;                    // ERROR - Osnovna je apstraktna klasa
    Osnovna *po=&izv;
    po->cvf();                      // Poziva se Izvedena::cvf()
    po->vf();                       // Poziva se Osnovna::vf()
}
```

- Apstraktni destruktor
  - Kada treba sprečiti stvaranje objekta čiji su svi metodi konkretni, a ne apstraktni deklarišemo apstraktni destruktor
  - Apstraktni destruktor mora biti definisan i za osnovnu klasu, a definicija mora biti van tela klase ZAŠTO?
  - Postojanje definicije za destruktor omogućava kreaciju podklasnih objekata

```
class O {
public:
virtual ~O() = 0;
};
O::~O(){}
class I:public O {};
O o; // ERROR
I i; // U redu
```

- Generalizacija i konkretizacija
  - Apstraktna klasa predstavlja **generalizaciju** izvedenih klasa
  - Klasa koja se izvodi iz apstraktne je **konkretizacija** apstraktne klase
  - Ako se u izvedenoj klasi ne navede definicija nekog apstraktnog metoda iz osnovne klase, i izvedena klasa je apstraktna
  - Apstraktna klasa može imati konstruktor - pozivaće se kao konstruktor osnovne klase pri konstrukciji objekata izvedenih konkretnih klasa
- Problem konverzije naniže
  - Preko pokazivača/reference na osnovnu klasu nije moguće pristupiti članovima izvedene klase
  - Samo ako znamo tačan tip objekta na koji ukazuje pokazivač/referenca možemo eksplicitno konvertovati pokazivač na osnovnu klasu u pokazivač na izvedenu - konverzija naniže - *downcast*
  - Ovakva konverzija uopšte nije bezbedna
  - Rešenje je da se nakon konverzije naniže proveri njena ispravnost - operator za dinamičku konverziju tipa
- Operator dinamičke konverzije
  `dynamic_cast<izvedena_klasa*>(pokazivac)`
  `dynamic_cast<izvedena_klasa&>(referenca)`
  - Konvertuju pokazivač/referencu na polimorfnu osnovnu klasu u pokazivač/referencu na izvedenu klasu
  - Ako pokazivač ne pokazuje na objekat izvedene klase u koju se vrši konverzija, ili njene podklase rezultat je `nullptr`
  - Ako referenca ne upućuje na na objekat izvedene klase u koju se vrši konverzija, ili njene podklase rezultat je `bad_cast` iz `<typeinfo>`

```
class A {public: virtual void vm(){}};  // Polimorfna klasa
class B: public A{/*...*/};
class C: public A{/*...*/};
int main(){
    A *pa=new B();
    B *pb=dynamic_cast<B*>(pa);
    C *pc=dynamic_cast<C*>(pa);          // pc==nullptr
    B &rb=dynamic_cast<B&>(*pa);
    C &rc=dynamic_cast<C&>(*pa);         // bad_cast
}
```

- Dinamičko određivanje tipa
  - C++ omogućava da se u vreme izvršavanja (*runtime*) odredi tip izraza
    `typeid(izraz)`
    `typeid(tip)`
  - `izraz`
    - Proizvoljnog tipa

    - Vrednost se **ne** izračunava
    - Ako ukazuje na polimorfnu klasu, rezultat se odnosi na dinamički tip operanda, u suprotnom na statički
    - `typeid(*p) p == nullptr` → `bad_typeid`
  - `tip` - proizvoljan tip, rezultat je `const type_info&`
  - Introspekcija ili refleksija je način da program sazna o sebi
- Klasa `type_info`
  - `type_info` je iz `<typeinfo>`
  - Metodi klase `type_info`
    - `bool operator==(const type_info&) const;`
    - `bool operator!=(const type_info&) const;`
    - `bool before (const type_info&) const;` - uređuje dva objekta tipa `type_info`
    - `const char* name() const;` - vraća neko ime - zavisi od kompajlera
  - Nema javnih konstruktora, ne mogu se kreirati podaci ovog tipa, i zato se i ne mogu prenositi podaci ovog tipa po vrednosti
  - Operator dodele je takođe privatan

```
class O {};
// Nepolimorfna klasa
class I: public O{public: virtual void f(){}};
class II: public I{};
int main(){
O *po=new I;
I *pi=new II;
// Izlaz (Borland):
cout<<(typeid(*po)==typeid(I))<<endl;   // 0
cout<<(typeid(*pi)==typeid(II))<<endl;  // 1
cout<<typeid(O).name()<<endl;           // O
cout<<typeid(*po).name()<<endl;         // O
cout<<typeid(*pi).name()<<endl;         // II
cout<<typeid(po).name()<<endl;          // O *
cout<<typeid(pi).name()<<endl;          // I *
int m[100][20];
cout<<typeid(m).name()<<endl;           // int[100][20]
```

- Zloupotreba određivanja tipa
  - Poenta objektno-orijantisanih jezika je u decentralizaciji odgovornosti
  - Kada se odredi tip objekta on se može porediti sa klasama
  - Polimorfizam se zamenjuje selekcijom i program ponovo postaje sa centralnom odgovornošću

```
// Ohrabruje ovakvu vrstu pisanja, preuzeto sa StackOverflow
PolymorphicType *pType = ...;
if(typeid(*pType) == typeid(Derived1))
  pType->Func1();
else if(typeid(*pType) == typeid(Derived2))
  pType->Func2();
else if(typeid(*pType) == typeid(Derived3))
  pType->Func3();
```

- Višestruko nasleđivanje
  - Klasa direktno nasleđuje osobine više osnovnih klasa, a roditeljske nisu jedna drugoj vrsta - *multiple inheritance*
  - Klasa se deklariše kao naslednik više klasa tako što se u zaglavlju navode osnovne klase - ispred svake osnovne treba da stoji `public`
- Konstrukcija i destrukcija
  - Pravila o nasleđenim članovima važe
  - Konstruktori svih osnovnih klasa se izvršavaju pre

1. Konstruktora članova izvedene klase
2. Konstruktora izvedene klase

– Konstruktori osnovnih klasa se pozivaju po redesledu deklarisanja

– Destruktori osnovnih klasa se izvršavaju posle

1. Destruktora izvedene klase
2. Destruktora članova izvedene klase

– Destruktori osnovnih klasa se pozivaju obrnutim redom

- Problem dijamant strukture
  - Kada su osnovne klase, pri višestrukom izvođenju, izvedene iz iste roditeljske klase nastaje problem dijamant strukture

```
class B {int i; /*...*/};
class X : public B {/*...*/};
class Y : public B {/*...*/};
class Z : public X, public Y {/*...*/};
```

  - Objekat nove klase će imati dva skupa članova klase B - razlikujemo ih pomoću `z.X::i` ili `z.Y::i`
  - Konstruktor osnovne kase B se izvršava dva puta
- Virtuelne osnovne klase
  - Ako nisu potrebna dva skupa članova, pri izvođenju treba deklarisati klasu B kao virtuelnu osnovnu klasu

```
class B {int i; /*...*/};
class X : virtual public B {/*...*/};
class Y : virtual public B {/*...*/};
class Z : public X, public Y {/*...*/};
```

  - Sada klasa Z nasleđuje samo jedan skup klase B i nema dvoznačnosti
  - Konstruktor klase B se poziva samo jednom
  - Klasa B mora biti virtuelna osnovna i za X i za Y, ako je samo jedna virtuelna ostaje dva skupa članova sa dva konstruktora
  - Redefinicija metod `m()` definisanog u klasi B
    - U klasi X ili Y - za `z.m()` se poziva redifinicija ✓
    - I u klasi X i u klasi Y - dvoznačnost X
- Redosled konstrukcije
  - Konstruktori virtuelnih osnovnih klasa se pozivaju pre konstruktora nevirtuelnih osnovnih klasa
  - Precizan redosled
    1. Konstruktori virtuelnih osnovnih klasa prvo po dubini grafa do korena, a zatim sleva-udesno na istom nivou
    2. Konstruktori nevirtuelnih osnovnih klasa
    3. Konstruktori atributa
    4. Konstriktor izvedene klase
  - Konstruktori virtulenih osnovnih klasa se pozivaju samo jednom

```
#include <iostream.h>
class B {
public: B(){cout<<" B";}
};
class X : virtual public B {
public: X(){cout<<" X";}
};
class Y : virtual public B {
public: Y(){cout<<" Y";}
};
class Z : public X, public Y {
public: Z(){cout<<" Z"<<endl;}
};
int main(){Z z;}
```

```
// Originalni redosled:        B X Y Z
// Da je stajalo (jedna po jedna zamena):
class X: public B {...    //  B B X Y Z
class Y: public B {...    //  B X B Y Z
class Z: public X,
virtual public Y {...     //    B Y X Z
class Z: virtual public X,
virtual public Y {...     //    B X Y Z
```

8 Izuzeci (*Exceptions*)

- Pojam izuzetaka
  - Izuzeci (*exceptions*) su događaji koje treba posebno obraditi van toka programa
  - Kod jezika koje ne podržavaju obradu izuzetaka, dolazi do sledećih problema
    - Problem migracije udesno - Posle izvršenja dela programa u kojem može doći do greške se testira status. OK status ide u jednu granu `if`, a obrada greške u drugu
    - Problem propagacije unazad - U lancu poziva funkcija, ako greška treba da se propagira nazad ka prethoodnom nivou, svaki nivo treba da testira da li je došlo do greške, i ako greška ne može da se obradi, funkcija u `return` mora da vrati kod greške
- Izuzeci u jeziku C++
  - Na mestu otkrivanja izuzetne situacije se baca izuzetak (operator `throw`) i dalja regularna obrada tekućeg bloka se trajno prekida
  - Izuzetak može biti objekat klase ili nekog drugog tipa
  - Obrada iuzetaka se nastavlja u posebnom bloku (`catch`) - rukovaocu obradom izuzetka (*handler*) se dostavlja izuzetak kao argument
  - Za svaki tip izuzetka postoji zaseban *handler*
  - Posle uspešne obrade izuzetka, obrada se nastavlja regularno
  - Ako ne postoji odgovarajući rukovalac izuzetak se automatski propagira unazad
- Otkrivanje i obrada izuzetaka
  - Obrada izuzetaka je vezana za blok naredbe `try`
  - Dopuniti sintaksnim dijagramima
- Definicija rukovaoca i tok obrade
  - Definicija rukovaoca liči na definiciju funkcije sa tačno jednim parametrom
  - Završiti sa sintaksnim dijagramom i njegovim objašnjenjem
  - Ovde samo objašnjava kako se koristi try/catch

```
try {
...
radi();                  // Funkcija mozda baca izuzetak
...
} catch(const char *pz){ // Obrada izuzetka tipa znakovnog niza
} catch(const int i){    // Obrada izuzetka celobrojnog tipa
} catch(...){            // Obrada izuzetka proizvoljnog tipa
                         // koji nije jedan od gornjih
}
```

- Izazivanje izuzetaka
  - Prijavljivanje ili bacanje izuzetaka se vrši naredbom `throw izraz;` gde izraz svojim tipom određuje koji handler će biti aktivan, dok se vrednost izraza izračunava i prosleđuje handleru kao argument

- Izuzetak se može izazvati iz bloka ili bilo koje funkcije direktno ili indirektno pozvane iz bloka naredbe `try`
- Funkcije iz kojih se izaziva izuzetak mogu biti članice klasa, operatori, konstruktori a i destruktori
- Za (dinamički) ugnežđene naredbe `try` - ugnežđeni rukovalac može da izazove izuzetak, koji se prosleđuje rukovaocu spoljašnjeg `try`-a
- Takođe, izuzetak može da se izazove i pomoću naredbe `trow;` bez izraza, i takav izuzetak ima tip rukovaoca u kojem je izazvan

- Specifikacija izuzetaka funkcije
  - U deklaraciji ili definiciji funkcija može da se navede spisak tipova izuzetaka koje funkcija izaziva `throw(niz_identifikatora)` - iza liste argumenata
  - Ako se stavi `throw()` klauzula kad funnkcija izazove izuzetak tipa koji nije nabrojan → greška
  - Ako se izostavi `throw()` klauzula funkcija sme da prijavi izuzetak proizvoljnog tipa
  - Virtuelna metoda u izvedenoj klasi ne sme da proširi listu izuzetaka iz `throw()` klauzule, ali sme da ih suzi
  - Standard C++11 ne preporučuje korišćenje `throw(niz_identifikatora)`, već samo vraćanje informacije da li funkcija baca ili ne baca izuzetke

```
void radi(...)throw(const char *, int){
if(...) throw "Izuzetak!";
if(...) throw 100;
if(...) throw Tacka(0,0); // ERROR nije naveden tip izuzetka Tacka
}
```

- Modifikator i operator `noexcept`
  - Modifikator
    - `throw(niz_identifikatora)`iza liste parametara, izvršava se u toku prevođenja i vrednost `true` označava da funkcije ne baca izuzetke
  - Operator
    - `throw(niz_identifikatora)`, gde je rezultat izraza logičkog tipa i izraz može biti proizvoljnog tipa (čak i `void`)
    - Izraz se ne izračunava, samo proverava u toku izvođenja, proverava se da li bi u slučaju izvršenja izraza moglo doći do izuzetka
    - Može da se koristi i u gorenavedenom modifikatoru

```
void f() noexcept {}                // ne baca
int g() noexcept(false){return 0;}  // moze da baca
double h() {return 0.0;}            // moze da baca
int i() throw(){return 0;}          // ne baca
void j() throw (int, double){}
                                    // moze da baca int i double
bool p=noexcept(f());               // true
bool q=noexcept(g());               // false
bool r=noexcept(h());               // false
bool s=noexcept(i());               // true
bool t=noexcept(j());               // false
bool v=noexcept(new int);           // false
```

- Prihvatanje izuzetaka - pravila
  - Rukovalac tipa R može da primi izuzetak tipa I ako
    1. R i I su isti tip
    2. R je javna osnovna klasa za izvedenu klasu I
    3. R i I su pokazivački/refrentni tipovi i I standardno može da se konvertuje u R

    ,
  - Prilikom navođenja rukovaoca treba se držati sledećih pravila
    1. Rukovaoce tipa izvedenog iz neke osnovne klase treba stavljati ispred rukovaoca tipa te osnovne klase
    2. Univerzalni rukovalac na poslednje mesto

- Prosleđivanje objekta izuzetka
  - Na mestu izazivanja izuzetka formira se objekat koji se prosleđuje rukovaocu
  - Objekat koji se prosleđuje je po pravilu kopija objekta rezultata izraza operatora `throw`
  - Prevodilac može da odluči da ne napravi kopiju - tada se za objekaz uzima rezultat izraza
  - To može biti i bezimeni privremeni objekat ili lokalni automatski objekat, ali svakako mora da ima kopirajući i/ili premeštajući konstruktor i destruktor
  - Objekat se prosleđuje i obrađuje u najbliže, prvom odgovarajućem rukovaocu
  - Ako se ne pronađe odgovarajući rukovalac objekat se prosleđuje prethodnom nivou `try`
- Obrada izuzetka u ugnežđenim `try`
  - Ako su `try` naredbe ugnežđene i ako se pronađe rukovalac tekuće naredbe - on obrađuje izuzetak, i ako se ne baci dalje, nastavlja se naredbom koja se nalazi posle poslednjeg `catch` bloka
  - Ako se ne pronađe odgovarajući rukovalac - izuzetak se prosleđuje rukovaocu prethodnog nivoa naredbe `try`, a ni jedna naredba posle `catch` se **neće** izvršiti
  - U slučaju da se izuzetak baci iz `catch` bloka naredbom `throw;`, i on se prosleđuje rukovaocu prethodnog nivoa, a objekat izuzetka ostaje živ i u `catch` bloku tog `try`
- Uništavanje lokalnih objekata
  - Predaja kontrole rukovaocu podrazumeva napuštanje bloak gde se dogodio izuzetak
  - Uništavaju se svi lokalni objekti tog i svih ugnežđenih blokova
  - Uništava se redosledom obrnutim od stvaranja
  - Kod izuzetaka bačenih iz konstruktora uništavaju se stvoreni klasni atributi i nasleđeni podobjekti
  - Nije dobro da rezultat izraza `throw` pokazuje/upućuje na lokalni objekat, jer će on biti uništen pre nego što dođe do rukovaoca
- Funkcijska naredba `try`
  - Telo `try` bloka se poklapa sa telom funkcije
  - Modifikatori metode (npr `const`), i `throw` klauzula idu ispred try
  - U handlerima mogu da se koriste parametri funkcije a ne mogu da se koriste lokalne promenljive
  - Ako funkcije nije `void`, mora da izvrši return ako ne baci izuzetak

```
tip fun(parametri) try { /* telo funkcije */ }
catch (parametar1) { /* telo rukovaoca 1 */ }
catch (parametar2) { /* telo rukovaoca 2 */ }
//-------------------------------------------------------------
int f(int x) throw(double) try {
int y=0;
if (...) throw 1;          // baca se i obradjuje
if (...) throw 2.0;        // baca se i propagira dalje
...
return x+y;               // regularan rezultat funkcije
} catch (int g) {
int a=x;                  // u redu
int b=y;                  // ERROR
return -1;                // rezultat u izuzetnoj situaciji
```

- Funkcijski `try` u konstruktoru
  - Omogućava hvatanje izuzetaka koji se bacaju iz
    1. Inicijalizatora atributa primitivnog tipa
    2. Konstruktora atributa klasnog tipa
    3. Konstruktora osnovnih klasa
  - Lista inicijalizatora u definiciji konstruktora se piše iza `try`
  - Ako se iz rukovaoca pristupa atributima ili nasleđenom podobjektu posledice su nepredvidive jer neki još nisu inicijalizovani, a klasni koji su već bili konstruisani su uništeni prilikom bacanja izuzetka

```
class A {
public:
A(int x) {... if (...) throw 'x';...}
A(char x) {... if (...) throw 2;...}
}
class B {
    A a1=A(3);
    A a2;
public:
    B() try : a2('a'){              // Izuzetak tipa int
    ...
    if(...) throw 4.0;              // Izuzetak tipa double
    } catch (double g) {            // Rukovalac za tip double
    A a3(a1);                       // ERROR - a1 unisten
    } catch (char g) {              // Rukovalac za tip char
    ...
    }                               // Nije obradjen tip int
}
```

- Neprihvaćeni izuzeci
  - Ako se za neki izuzetak ne pronađe rukovalac - izvršava se sistemska funkcija `void terminate();`
  - Podrazumeva se da ova funkcija poziva funkciju `abort()` - ona vraća kontrolu operativnom sistemu
  - Ovo se može promeniti pomoću funkcije `set_terminate()`
    - Dostavlja se pokazivač na funkciju koju `terminate()` poziva umesto `abort()`
    - Pokazana funkcija mora biti bez argumenata i rezultata `void`
    - Vrednost funkcije `set_terminate()` je pokazivač na staru funkciju koja je bila pozvana iz `terminate()`
- Zamena za `abort()`
  - Tip korisničke funkcije koja zamenjuje `abort()` `typedef void (*PF) ();`
  - Prototip funkcije `set_terminate()` `PF set_terminate(PF pf);`
  - Iz korisničke funkcije `(*PF)` treba pozvati `exit(int), terminate()` ili `abort()`
  - Pokušaj povratka sa `return` iz korisničke funkcije dovešće do nasilnog prekida programa sa `abort()` - funkcija ne sme išta da vraća
- Neočekivani izuzeci
  - Ako se u nekoj funkciji izazove neočekivani izuzetak (nije na spisku izuzetaka) izvršava se funkcija `void unexpected();`
  - Podrazumeva se da ova funkcija pozove funkciju `terminate()`
  - Ovo ponašanje se može promeniti pomoću `set_unexpected()`
    - Dstavlja joj se pokazivač na funkciju koju treba da pozove umesto `terminate()`
    - Pokazana funkcija mora biti bez argumenata i rezultata (`void`)
    - Vrednost funkcije `set_unexpected()` je pokazivač na staru funkciju koja je bila pozvana iz `unexpected()`
- Zamena za `terminate()`
  - Tip korisničke funkcije koja zamenjuje `terminate()` `typedef void (*PF) ();`
  - Prototip funkcije `set_terminate()` `PF set_terminate(PF pf);`
  - Iz korisničke funkcije `(*PF)` treba pozvati `exit(int), terminate()` ili `abort()` ili baciti izuzetak sa liste dozvoljenih ili baciti `bad_exception`
  - Pokušaj povratka sa `return` iz korisničke funkcije dovešće do nasilnog prekida programa sa `abort()` - funkcija ne sme išta da vraća
- Standardni izuzeci
  - Klasa `exception` u zaglavlju `<exception>` je predak svih standardnih izuzetaka

- Metode standardnih klasa i neki operatori prijavljuju izuzetke klasa izvedenih iz `exception`, a preporučuje se i da se korisnički izuzeci izvode odatle
- Klasa omogućava da se pojedini tipovi izuzetaka mogu obrađivati
  1. Pojedinačno
  2. U srodnim grupama
  3. Svi zajedno (ali ne kao ...)

- Definicija klase `exception`
  - `what()` vraća pokazivač na tekstualni opis izuzetka (standard ne propisuje tekst poruka)
  - Ni jedna metoda ne sme da prijavi izuzetak, to se obezbeđuje sa `noexcept`

```
class exception {
public:
    exception() noexcept;
    exception(const exception &) noexcept;
    exception& operator=(const exception &) noexcept;
    virtual ~exception() noexcept;
    virtual const char* what() const noexcept;
};
```