OBJEKTNO PROGRAMIRANJE 1

Oznaka predmeta: OOP

Predavanje broj: 8

Nastavna jedinica: Funkcije, reference, overloading.

Nastavne teme:

Funkcije: pozivi, neodređeni broj argumenata, podrazumevani argumenti. Reference: inicijalizacija, upotreba. Neposredno ugrađivanje u kod. Preklapanje imena funkcija. Deklaracije preklopljenih funkcija. Razrešavanje poziva. Sekvenca konverzija. Adresa preklopljene funkcije.

Predavač: prof. dr Perica S. Štrbac, dipl. ing.

Literatura:

Dragan Milićev, “Objektno orijentisano programiranje na jeziku C++”, Mikro knjiga, Beograd, 2005.

Poziv funkcije i prenos argumenata

- Poziv funkcije realizuje se operatorom poziva funkcije ()
  - Pri pozivu svi formalni argumenti inicijalizuju se stvarnim argumentima, pri čemu se implicitno primenjuju standardne i korisničke konverzije.
  - Semantika prenosa argumenata je kao i semantika inicijalizacije: formalni argument, kao lokalni automatski objekat, inicijalizuje se stvarnim argumentom, na isti način na koji se obavlja inicijalizacija u deklaraciji.
- Pri vraćanju vrednosti iz funkcije: privremeni objekat, koji na mestu poziva prihvata povratnu vrednost funkcije, inicijalizuje se izrazom navedenim iza naredbe return. Semantika je ista kao i semantika inicijalizacije u deklaraciji.
- Ako je formalni argument objekat klase, on se inicijalizuje stvarnim argumentom pomoću poziva odgovarajućeg konstruktora pre ulaska u funkciju. Isto važi i za povratnu vrednost funkcije. Na primer:

```
class X { public:   X(int);      };
X f(X x) { //...  return 1;  }  // poziva se konstruktor X(1)
void main () {
  int i=0; X iks=100;
  iks=f(i);           //poziva se f(X(i))
}
```

- Funkcija može da utiče preko formalnog argumenta na stvarni argument samo ako je formalni argument referenca ili pokazivač.

Funkcije sa neodređenim brojem argumenata

- Funkcija se može deklarisati tako da prima neodređen broj argumenata.
- Ako se *lista_argumenata* u deklaraciji funkcije završava znacima tri tačke (...), onda se za tu funkciju zna samo da je broj argumenata veći ili jednak broju deklarisanih argumenata.
  Sledeća funkcija prima dva argumenta tipa int, i još proizvoljno mnogo (nula ili više) argumenata proizvoljnog tipa:

```
void f(int,int,...);  //ili void f(int,int...);
```

- Pri pozivu ove funkcije, prva dva stvarna argumenta moraju biti tipa int; ostali stvarni argumenti su neobavezni i proizvoljnog su tipa. Prema tome, ova funkcija se može pozvati na bilo koji od sledećih načina:

```
f(1,2);
f(1,2,3+4);
f(1,2,3,4,5,6,7);
f(1,2,3.4,"Zdravo!");
```

- U telu funkcije koja ima neodređen broj argumenata, neimenovanim argumentima, koji su preneseni preko mehanizma ... , može se pristupiti na sledeći način (sledeći slajd, jedino ovaj način je garantovan standardom).
  - Standardno zaglavlje <cstdarg> sadrži deklaracije koje su potrebne u funkciji koja pristupa svojim neimenovanim argumentima.

Funkcije sa neodređenim brojem argumenata

- U zaglavlju <cstdarg> deklarisano je sledeće:
  - `va_list` je tip koji se koristi za reprezentaciju promenljive liste neimenovanih argumenata;
  - `void va_start(va_list valist, poslednji_arg); //makro`
    - gde je *poslednji_arg* poslednji imenovani formalni argument u listi deklarisanih argumenata;
    - ovaj makro mora biti pozvan pre nego što se pristupi neimenovanim argumentima;
    - poziv ovog makroa inicijalizuje *valist* da ukazuje na prvi neimenovani argument;
  - `tip va_arg(va_list valist, tip);  //makro`
    - gde je *tip* tip sledećeg neimenovanog argumenta u listi;
    - posle poziva va_start, svaki poziv makroa va_arg će vratiti vrednost tipa *tip* iz liste neimenovanih argumenata, i modifikovati *valist* tako da ukazuje na sledeći neimenovani argument iz liste;
  - `void va_end(va_list valist);  //makro`
    - ovaj makro mora biti pozvan posle pristupa neimenovanim argumentima, pre povratka iz funkcije.

Funkcije sa neodređenim brojem argumenata

```
#include <cstdarg>
double zbir (char *format ...){
  va_list valist;     // pokazuje na tekuci neimenovani argument
  va_start(valist,format); //inicijalizacija valist na prvi neim. arg.
  double suma=0.0; int i; double d;
  for (char *p=format; *p; p++)    // prolaz kroz niz znakova
    switch (*p) {                  // u zavisnosti od tipa
      case 'i':
      case 'I':
      i = va_arg(valist,int);       // neim. arg. tipa int
      suma+=i; break;
      case 'd':
      case 'D':
      d = va_arg(valist,double); // neim. arg. Tipa double
      suma+=d; break;
    }
    va_end(valist); return suma;
}
#include <iostream>
void main(){
  double d =zbir("idid",int(4),double(5.6),int(-5),double(6));
  std::cout<<"d="<<d<<std::endl;
  system("pause");}
```

Podrazumevani argumenti

- Funkcija se može deklarisati tako da se neki argumenti u pozivu te funkcije mogu izostaviti.
  - Tada izostavljeni argmenti imaju podrazumevane (*default*) vrednosti.
  - Podrazumevane vrednosti se u deklaraciji funkcije navode iza deklaratora argumenta, navođenjem izraza iza znaka =.

```
void f (int,int=3,int=0);
```

To znači da se funkcija f može pozvati sa jednim, dva, ili tri stvarna argumenta.

```
f(1);          f(2,4);          f(3,7,5);
```

znače isto što i redom pozivi:

```
f(1,3,0);  // drugi i treći argument imaju podrazumevane vrednosti
f(2,4,0);  // treći argument ima podrazumevanu vrednost
f(3,7,5);
```

- Ako neki argument funkcije ima podrazumevanu vrednost, onda svi argumenti iza njega moraju imati podrazumevanu vrednost.

```
void f(int,int=3,int);          //ne može
```

- Ne može se ostaviti nenaveden stvarni argument iza koga je naveden neki drugi stvarni argument.

```
f(1,,3);   // greška!
```

Podrazumevani argumenti

- Podrazumevani argument se ne može redefinisati u nekoj deklaraciji iste funkcije, čak ne ni na istu vrednost.
  - ako formalni argument funkcije ima podrazumevanu vrednost, onda svi formalni argumenti iza njega moraju imati navedene podrazumevane vrednosti u istoj ili prethodnoj deklaraciji iste funkcije.
  - deklaracija ne može ponovo definisati formalne argumente.

```
void f(int,int,int);
void f(int,int,int=0);// OK, poslednji je argument;
void f(int,int=3,int);// OK, svi argumenti iza njega
// imaju default vrednosti,navedene u prethodnoj liniji koda
void f(int,int,int);
void f(int,int=3,int);  // greška: nije poslednji argument!
void h(int,int,int=0);
void h(int,int=3,int=0);// greška: redefinicija istog arg.
```

- Podrazumevani argument može biti rezultat izraza pri čemu se imena navedena u izrazu kao i provera tipova, vezuju se za oblast važenja u kome se nalazi deklaracija funkcije nečlanice. Vrednost ovog izraza izračunava se pri svakom pozivu.

```
int a = 1; int f(int);
int g(int = f(a));              // podrazumevani argument je f(::a)
void h() { int a = 3;  g(); } // poziv g(f(::a))
```

Podrazumevani argumenti

- Lokalne promenljive ne mogu se koristiti u izrazima koji definišu podrazumevanu vrednost argumenta.

```
void f() { int i;  extern void g(int x = i);  ... } // greška!
```

- Formalni argumenti ne mogu se koristiti u izrazima koji definišu podrazumevane vrednosti argumenata.

```
int a;
void f(int a, int b = a);  //greška:formalni argument kao default
```

- Postojanje podrazumevane vrednosti argumenta nije deo tipa funkcije.

```
int f(int=0);    // tip int(int)
int g(int);      // tip int(int)
int h();         // tip int()
int (*pf)()=&f; // greška: pf je tipa   int(*)(),
                //         a &f je tipa int(*)(int)!
```

- Deklaracija funkcije u unutrašnjoj oblasti važenja sakriva deklaraciju funkcije sa istim imenom u spoljašnjem opsegu. Zato nije moguće u unutrašnjoj oblasti važenja dodavati podrazumevane vrednosti u deklaraciju funkcije na sledeći način:

```
void f(int,int=0);
void h() { extern void f(int=0,int); ...} // greška: nije poslednji
```

- **Preklopljena operatorska funkcija ne može imati podrazumevane vrednosti argumenata.**

Reference

- Reference su posebna vrsta izvedenog tipa u jeziku C++.
- Reference se veoma često koriste u programima, ali se ne sreću često u drugim programskim jezicima.
- U jeziku C++, argumenti funkcije mogu se, osim po vrednosti, preneti i po referenci. To znači da će se svaka promena formalnog argumenta reflektovati na original, stvarni argument.

```
void f(int i, int &j)
{                          // i se prenosi po vrednosti,
                           // j po referenci
  i++;                     // stvarni argument se neće promeniti
  j++;                     // stvarni argument će se promeniti
}
void main ()
{
  int si=0, sj=0;
  f(si,sj);
  // ovde će si biti jednako 0, a sj jednako 1
}
```

- Referenca predstavlja izvedeni tip tako da se"referenca na tip T" označava se sa T&.

Reference

- Pri definiciji reference tipa T&, ona se mora i inicijalizovati objektom tipa T.
- Od trenutka svoje definicije, referenca upućuje (referencira) na objekat kojim je inicijalizovana, referenca postaje alternativno ime (sinonim) za objekat kojim je inicijalizovana.

```
int i=1;              // celobrojni objekat i;
int &j=i;             // j je referenca tipa int&, upućuje na i;
i=3;                  // menja se i;
j=5;                  // opet se menja i
int *p=&j;            // isto što i &i
j+=1;                 // isto što i: i+=1;
int k=j;              // posredan pristup do i preko reference;
int m=*p;             // posredan pristup do i preko pokazivača;
```

- Referenca se prilikom svoje inicijalizacije neraskidivo vezuje za objekat kojim se inicijalizuje.
- Pristup preko pokazivača i reference do objekta je posredni pristup: referenca upućuje na neki objekat, pokazivač ukazuje na neki objekat.
- Posredni pristup preko pokazivača zahteva navođenje operacije dereferenciranja pokazivača (*), dok se kod posrednog pristupa preko reference ne navodi nikakva operacija.
- Referenca trajno upućuje na isti objekat.

Reference

- Reference se najčešće realizuju u prevodiocima kao konstantni pokazivači.
- Reference treba shvatati samo kao imena izvedenog tipa koja referenciraju neki objekat, u smislu da svako obraćanje referenci u stvari predstavlja obraćanje referenciranom objektu.
  Nad referencama se, kao i nad ostalim tipovima, u jeziku C++ definišu operacije koje su dozvoljene, konverzije nad referencama itd.
- Reference nisu objekti, u smislu nečega što zauzima prostor u memoriji podataka: prevodilac na nekim mestima može realizovati referencu i kao konstantu, koja predstavlja adresu referenciranog objekta, i čiju vrednost zamenjuje direktno u kôd na mestima korišćenja.
- Znak & za reference se vezuje samo za ime uz koje se nalazi u deklaracijama.

```
int &ri=i,      // ri je referenca na objekat tipa int;
    (&rf)()=f,  // rf je referenca na funkciju f bez argumenata
                // koja vraća int;
    &g(int),    // g je funkcija koja ima argument tipa int
                // a vraća rezultat tipa referenca na int;
    *&rp = pi;  // rp je referenca na pokazivač na int
```

Reference

- Referenca može biti deklarisana kao const i volatile, ali to nema nikakvo posebno značenje.

```
int &const ri = i,    // referenca je const
    &volatile rj = j; // referenca je volatile
```

ovo je sasvim drugačije od referenci na const ili volatile objekat, što ima odgovarajuće značenje za referencirani objekat:

```
const int &ri = i;    // ri je referenca na const int
volatile int &rj = j; // rj je referenca na volatile int
```

- Ne mogu postojati:
  - nizovi referenci,
  - pokazivači na reference,
  - reference na void (void&),
  - reference na bit-polja,
  - reference na reference.
- Pošto referenca nije objekat onda se ne može ni kreirati kao dinamički objekat operatorom new.
- Razlikovati operaciju uzimanja adrese (&) sa znakom tipa reference (&). Znak & upotrebljen u izrazu, sa odgovarajućim operandom, predstavlja operaciju uzimanja adrese, a znak & upotrebljen u imenu tipa (u deklaraciji) predstavlja tip reference.

Inicijalizacija referenci

- Deklaracija reference ne mora sadržati inicijalizaciju reference u slučajevima:
  - Kada deklaracija sadrži eksplicitno naveden specifikator extern;
  - Kada deklaracija predstavlja deklaraciju formalnog argumenta funkcije ili povratne vrednosti funkcije.
  - Kada je deklaracija reference, kao člana klase, unutar deklaracije klase;
- Referenca na tip T mora se inicijalizovati objektom tipa T, ili objektom koji se može konvertovati u tip T (inicijalizator može biti i druga referenca na isti tip T:

```
int i = 0;
int &ri1 = i;   // ri1 je referenca na objekat i
int &ri2 = ri1; // ri2 je referenca na isti objekat i
```

- Ako inicijalizator nije lvrednost, i ako referenca nije referenca na konstantu, deklaracija nije ispravna. Na primer:

```
int i = 0;    // objekat i je lvrednost,
int &ri = i;  // pa ri zato upućuje na i;
const int &r1 = 1;  // inicijalizator nije lvrednost, pa r1 mora biti
                    // referenca na konstantu; kreira se privremeni
                    // objekat od izraza 1, na koga r1 upućuje;
int &r2 = 2;  // greška: r2 nije referenca na konstantu!
```

- Referenca na volatile T može biti inicijalizovana objektom tipa T ili volatile T, ali ne tipa const T.

Inicijalizacija referenci

- Referenca na const T može biti inicijalizovana objektom tipa T ili const T, ali ne tipa volatile T. Referenca na čisti tip T može biti inicijalizovana samo objektom čistog tipa T. Referenca na osnovnu klasu B može se inicijalizovati objektom izvedene klase D, pod uslovom da je B dostupna osnovna klasa klase D.
- Ako je formalni argument tipa reference, onda će on upućivati na stvarni argument.
- Ako je povratna vrednost funkcije tipa reference, onda privremeni objekat, koji na mestu poziva prihvata rezultat funkcije, jeste referenca na objekat koji je vraćen naredbom return. Na primer:

```
int& f(int &i) {  // f je funkcija koja prima argument tipa int&
   return i;      // a vraća rezultat tipa int&;
}
void main () {
  int x = 0;
  f(x)=1;         // x postaje 1!
}
```

Na mestu poziva funkcije f, formalni argument ***i*** se inicijalizuje tako da upućuje na stvarni argument x. Sve što se unutar funkcije f bude radilo sa formalnim argumentom ***i***, odnosiće se na referencirani objekat, a to je ovde stvarni argument x. Nareda return vraća ***i*** a to je ovde objekat x.

Upotreba referenci

- Ne postoji operator koji operiše sa referencama. Svi operatori, kada se primene na referencu, operišu sa referenciranim objektom.

```
int i = 0, j = 1;
int &ri = i;
int *pi = &ri;  // &ri je isto što je &i, pa pi ukazuje na i
*pi=2;          // pristup do i preko pokazivača
ri=j;           // pristup do i preko reference; i postaje 1
if (sizeof(int) == sizeof(int&)) // uvek zadovoljeno
```

- Rezultat operacije kreiranja dinamičkog objekta (new) je pokazivač. Taj pokazivač se ne mora direktno zapamtiti u objekat tipa pokazivača, kako bi se omogućio pristup do dinamičkog objekta. Za dinamički objekat se može vezati i referenca:

```
int &ri = *new int(2);// new vraća pokazivač na dinamički objekat;
                      // operacija *new vraća taj objekat;
                      // ri upućuje na dinamički objekat -
                      // ri je ime za dinamički objekat;
ri++;                 // dinamički objekat postaje 3;
int *pi = &ri;        // pi ukazuje na dinamički objekat;
delete &ri;           // ukida se dinamički objekat;
```

Upotreba referenci

- Postoji samo jedna standardna konverzija tipa reference na izvedenu klasu u tip reference na osnovnu klasu.
- Objekat nekog tipa se može eksplicitno konvertovati u referencu tipa X&, samo ako se pokazivač na taj objekat može eksplicitno konvertovati u tip X*;
  - Pri toj konverziji se ne pozivaju konstruktori ili korisnički definisane konverzije.
  - Rezultat eksplicitne konverzije u tip reference jeste lvrednost (odnosi se na referencirani objekat), dok rezultati ostalih eksplicitnih konverzija nisu.
- Kada se referenca vezuje za neki objekat, treba voditi računa o životnom veku tog objekta: on treba da bude jednak ili duži od životnog veka reference.

```
int& f(int  i){int &r=*new int(1);return r;}//ako nije bilo delete &r
int& f(int &i){return i;}
```

- Funkcija vraća referencu na objekat koji prestaje da živi posle povratka iz funkcije:

```
int& f(int &i) { int r=1; return r; }     //automatski lokalni objekt
int& f(int i) {  return i; }//referenca na formalni argument, koji je
                            //lokalni automatski objekat
int& f(int &i) {int r=*new int(i); return r;}
//int r predstavlja zasebnu kopiju dinamičkog objekta, a nepovratno
//se gubi dinamički objekat
```

Neposredno ugrađivanje u kôd, *inline*

- Kratke funkcije prosleđuju svoje parametre pozivom neke druge funkcije, ili samo vraćaju rezultat nekog jednostavnog izraza. Smisao ovakvih funkcija je najčešće da obezbede enkapsulaciju i proveru tipova argumenata.
- Vreme koje se troši u toku izvršavanja programa na njihov poziv i prenos argumenata, može da bude veće od vremena izvršavanja samog tela funkcije.
- Zato u jeziku C++ postoji mogućnost da se od prevodioca zahteva da kôd tela funkcije ugrađuje neposredno u kôd programa na mestima poziva te funkcije. Ovakve funkcije nazivaju se *inline* funkcijama i deklarišu se navođenjem specifikatora inline ispred deklaracije.

```
inline int max (int a, int b) {return (a>=b)?a:b;}
```

- Standard C99 sugeriše da se uz globalne funkcije koje su *inline* stavlja i static tako da imaju interno povezivanje (definicija *inline* funkcije, koja se koristi u više programskih fajlova, navodi u nekom fajlu-zaglavlju, koje se zatim uključuje u sve potrebne programske fajlove.
- Funkcije članice su podrazumevano *inline* ako se njihova definicija navodi unutar deklaracije klase:

```
class counter { int i;
  public:     counter(int x) {i=x;} // ovo je inline konstruktor
              int count() {return ++i;} // ovo je inline funkcija članica
};
```

Neposredno ugrađivanje u kôd, *inline*

- Funkcija članica može, ali ne mora eksplicitno biti naznačena kao inline u deklaraciji klase, da bi bila *inline*.
- Ona se može specificirati kao inline u deklaraciji koja predstavlja njenu definiciju, van deklaracije klase:

```
class counter
{
      int i;
  public:
          counter (int x) {i=x;}      // ovo je inline konstruktor
      int count();
};

inline int counter::count() {return ++i;} // inline funkcija članica
```

- Specificiranje neke funkcije kao *inline* predstavlja samo zahtev prevodiocu da optimizuje pozive funkcije direktnim ugrađivanjem njenog tela u kôd na mestu poziva. Taj zahtev prevodilac ne mora da ispuni.
- Program nimalo ne menja značenje ako je neka funkcija *inline* ili ne.
  - O mehanizmu *inline* treba razmišljati samo kao o zahtevu za optimizaciju kôda.

```
#define max(a,b) ((a>=b)?(a):(b)) //treba koristiti inline funkciju
```

Preklapanje imena funkcija

- U jeziku C++ postoji mogućnost da više različitih funkcija nosi isto ime. Ovaj mehanizam naziva se *preklapanje imena funkcija* (*overloading*).
- U jeziku C++ može se za jedno ime funkcije navesti više različitih deklaracija.
- Tada se za to ime kaže da je *preklopljeno* (*overloaded*).
- Moguće je deklarisati i definisati više različitih funkcija sa istim identifikatorom.
- Ove funkcije treba da se razlikuju po broju i/ili tipu svojih argumenata.
- Kada se preklopljeno ime funkcije koristi (poziv funkcije) bira se odgovarajuća funkcija poređenjem tipova stvarnih argumenata sa tipovima formalnih argumenata.

```
char* max (const char *p, const char *q)
{ return (strcmp(p,q)>=0)?p:q; } // vraća veći od dva niza znakova

double max (double i, double j){
 return (i>j) ? i : j; }// vraća veći od dva broja

void main () {
   double r=max(1.5,2.5);          // poziva se max(double,double)
   char *q=max("Pera","Mika");     // poziva se max(char*,char*)
}
```

Preklapanje imena funkcija

- Odluka o tome koja se od preklopljenih funkcija poziva, donosi se u vreme prevođenja, upoređivanjem tipova stvarnih argumenata sa tipovima formalnih argumenata.
- Prevodilac mora jednoznačno da odredi koja se funkcija poziva.
- Funkcije koje se razlikuju samo po tipu rezultata ne mogu imati isto ime.
- Funkcija se eksplicitno poziva navođenjem stvarnih argumenata, čiji se tipovi znaju u vreme prevođenja.
  - nije uvek jednoznačno određeno kog tipa treba da bude rezultat poziva funkcije, upotrebljen u nekom izrazu.

```
int zbir (int,int) {/*...*/}
double zbir (int,int) {/*...*/}  // greška!
```

- Tipovi argumenata preklopljenih funkcija treba da se dovoljno razlikuju, dok tip rezultata nije od značaja: može se, ali i ne mora razlikovati.
- Kako za bilo koji tip T, imena tipova T i T& prihvataju isti skup vrednosti inicijalizatora, funkcije koje se razlikuju samo u ovom pogledu ne mogu imati isto ime. Na primer:

```
int f(int) {/*...*/}
int f(int&) {/*...*/}  // greška!
```

Preklapanje imena funkcija

- Za bilo koji tip T, imena tipa T, const T i volatile T prihvataju isti skup vrednosti inicijalizatora, pa funkcije ne mogu imati isto ime prema ovom pogledu.
  - mogu se razlikovati T&, const T& i volatile T&, T*, const T* i volatile T*, pa se preklopljene funkcije mogu razlikovati
- Ime tipa deklarisano pomoću typedef deklaracije nije ime posebnog tipa:

```
typedef int CEO;
void f(int i) {/*...*/}
void f(CEO i) {/*...*/}  // greška: redefinisana ista funkcija!
```

- Nabrajanja su sasvim drugi tip od ostalih ugrađenih celobrojnih tipova, pa se preklopljene funkcije mogu razlikovati u tom pogledu.
- Takođe, tipovi unsigned i signed su različiti tipovi od svojih osnovnih tipova, pa se preklopljene funkcije mogu razlikovati u tom pogledu.
- Tipovi argumenata koji se razlikuju samo u tipu "pokazivač" (*) ili "niz" ([]) su identični u ovom kontekstu. Prema tome, funkcije se ne mogu razlikovati samo po tipu nekog argumenta T* ili T[].
  - ne mogu se razlikovati argumenti samo po prvoj dimenziji. Na primer:

```
void g(int *);
void g(int[7]);  // deklaracija iste funkcije g
void g(int[9]);  // deklaracija iste funkcije g
```

Deklaracije preklopljenih funkcija

- Dve deklaracije sa istim imenom funkcije odnose se na istu funkciju, samo ako se nalaze u istom opsegu važenja i imaju iste tipove argumenata.
- Ako su dve deklaracije funkcije sa istim imenom u istom opsegu važenja, ali se tipovi argumenata dovoljno razlikuju, onda se deklaracije odnose na različite funkcije sa preklopljenim imenom.
- Ako se deklaracije dve funkcije sa istim imenom nalaze u različitom opsegu važenja, bez obzira da li su tipovi argumenata isti ili različiti, deklaracija u unutrašnjoj oblasti važenja (npr. ugnežđeni blok) sakriva deklaraciju u spoljašnjoj oblasti važenja.
  - Pri tom, čak i ako se tipovi argumenata dovoljno razlikuju, funkcija čija je deklaracija u unutrašnjoj oblasti važenja ne preklapa, već sakriva funkciju iz spoljašnje oblasti važenja. Na primer:

```
extern void f(char*);
void g() {
  extern void f(double);// s a k r i v a globalnu deklaraciju;
  f("Zdravo!");         // greška: funkcija f(char*) je skrivena!
}
```

- Funkcija članica izvedene klase nije u istom opsegu važenja kao i funkcija članica osnovne klase. Deklaracija funkcije članice u izvedenoj klasi sakriva, a ne preklapa isto ime funkcije članice osnovne klase.

Deklaracije preklopljenih funkcija

```
class B {              // osnovna klasa B;
public:
  int f(int);
};

class D : public B {// izvedena klasa D;
public:
  int f(char*);      // sakriva deklaraciju f iz osnovne klase;
};

void g(D* pd) {      // *pd je objekat izvedene klase D;
  pd->f(1);          // greška: f(int) ne postoji u klasi D!
  pd->B::f(1);       // ovako može: pristup funkciji osnovne klase;
  pd->f("Zdravo!"); // i ovako može;
}
```

- Ovo pravilo treba imati na umu svaki put kada se u izvedenoj klasi želi preklopiti neko ime funkcije osnovne klase.
- Funkciji članici osnovne klase može se pristupiti, za neki objekat izvedene klase, eksplicitnim navođenjem imena osnovne klase ispred operatora razrešavanja opsega važenja (::). Ako se želi direktan pristup onda se mora svaka funkcija članica osnovne klase, koja se preklapa sa nekom funkcijom izvedene klase, ponovo definisati u izvedenoj klasi.

Deklaracije preklopljenih funkcija

```
class B {           // osnovna klasa B;
  public:
    int f(int);
};

class D : public B {// izvedena klasa D;
  public:
    int f(int i) {    // definisana nova funkcija, uz eksplicitni
      return B::f(i); // poziv B::f; a i dobar primer za inline;
    }
    int f(char*);
};

void g(D* pd) {     // *pd je objekat izvedene klase D;
  pd->f(1);         // sada je u redu: poziva se D::f(int);
  pd->B::f(1);      // ovako opet može;
  pd->f("Zdravo!"); // ovako i dalje može;
}
```

- Nova funkcija f(int) je definisana u klasi D, pa je sada isti oblik funkcije f prenesen iz osnovne u izvedenu klasu gde se poziva u njoj poziva B::f(int).
- Razne preklopljene funkcije članice iste klase mogu, čak, imati različita prava pristupa: neke mogu biti privatne, neke zaštićene, a neke javne članice te klase.

Razrešavanje poziva

- Za poziv se bira ona funkcija, iz skupa funkcija sa istim imenom, vidljivim u tekućoj oblasti važenja, koja "najbolje odgovara" po tipovima formalnih argumenata.
- Prevodilac razrešava poziv na sledeći način.
  - Za svaki stvarni argument, formira se skup preklopljenih funkcija koje najbolje odgovaraju po tipu odgovarajućeg formalnog argumenta.
  - Taj skup može imati jednu ili više funkcija:
    - ako jedna funkcija bolje odgovara nego sve ostale za taj argument, skup ima jednu funkciju;
    - ako više funkcija podjednako dobro odgovaraju za taj argument, skup ima više funkcija.
    - Kada se formiraju ovakvi skupovi za svaki stvarni argument, traži se presek tih skupova.
    - Ako taj presek sadrži tačno jednu funkciju, poziva se ta funkcija.
    - Inače, poziv je neregularan .
- Potpuno poklapanje tipa stvarnog argumenta sa tipom formalnog argumenta bolje je od bilo koje konverzije. Neka su deklarisane dve preklopljene funkcije:

```
void f(int,int);
void f(int,double);      i neka je poziv oblika:          f(1,2);
```

Razrešavanje poziva

- Treba primetiti da su stvarni argumenti celobrojni literali, koji su tipa int.
  - Prevodilac formira skupove funkcija koje najbolje odgovaraju za svaki stvarni argument.
  - Za prvi argument, to je skup {f(int,int), f(int,double)}.
  - Za drugi argument, to je skup {f(int,int)}, jer potpuno poklapanje tipova stvarnog i formalnog argumenta bolje odgovara od konverzije iz int u double.
  - U preseku ova dva skupa je jedino funkcija f(int,int), koja se upravo i poziva, dakle poziv je jednoznačan i regularan.
- Primer dvosmislenosti

```
void f(double,int);
void f(int,double);
f(1,2);
```

  - Skup za prvi argument je {f(int,double)}, a za drugi argument {f(double,int)}. Kako je presek ova dva skupa prazan, poziv nije regularan.
- Za potrebe slaganja argumenata, funkcija sa *n* argumenata sa podrazumevanim vrednostima, posmatra se kao *n+1* funkcija sa različitim brojem argumenata.

```
void f(int a,int b=0);   //u pogledu rezrešavanja poziva ekvivalentna je
                         //sa sledecim funkcijama
void f(int a, int b);

inline void f(int a) {f(a,0)};
```

Razrešavanje poziva

- Podrazumevani argumenti mogu biti izvor dvosmislenosti. Na primer:

```
void f();
void f(int=0);
void main () {
  f(1);  // u redu: poziva se f(int);
  f();   // greška: dvosmislenost, da li f() ili f(0)!
}
```

- Za potrebe slaganja argumenata, za nestatičku funkciju članicu se smatra da ima jedan dodatni argument koji specificira objekat čija je to funkcija članica.
  - Ovaj dodatni argument zahteva odgovaranje objekta ili pokazivača preko koga je funkcija pozvana.
  - Ovde se ne primenjuju nikakve korisničke konverzije da bi se došlo do slaganja ovog dodatnog argumenta.
- Ako je funkcija članica klase X pozvana preko pokazivača i operatora `->`, za ovaj dodatni argument se smatra da je tipa const X* za konstantne funkcije članice, volatile X* za volatile funkcije članice, odnosno X* za ostale funkcije članice.
- Ako je funkcija članica klase X eksplicitno pozvana korišćenjem operatora `.`, ili kao operatorska funkcija za prvi argument, za ovaj dodatni argument se smatra da je tipa const X& za konstantne funkcije članice, volatile X& za volatile funkcije članice, odnosno X& za ostale funkcije članice.
- Tri tačke (...) u deklaraciji funkcije odgovaraju svakom tipu stvarnog argumenta.

Sekvence konverzija

- Na osnovu sekvence konverzija kojima se tip stvarnog argumenta može prevesti u tip formalnog argumenta odlučuje se koji tip formalnog argumenta bolje odgovara datom stvarnom argumentu (int->float->double i int->double).
- Za dati stvarni argument, ne razmatra se nijedna sekvenca konverzija, koja sadrži više od jedne korisničke konverzije, niti sekvenca koja se može skratiti izostavljanjem jedne ili više konverzija, pri čemu novodobijena sekvenca konverzija takođe dovodi tip stvarnog argumenta u tip formalnog argumenta.
- Najkraća sekvenca konverzija, koja dovodi tip stvarnog argumenta u tip formalnog argumenta, i koja ne sadrži više od jedne korisničke konverzije, naziva se sekvencom konverzija koja najbolje odgovara.
- Sledeće *trivijalne* konverzije, koje uključuju proizvoljan tip T, ne utiču na to koja je od dve sevence konverzija bolja:

Iz	U (respektivno)
`T  T&  T[]`	`T&  T  T*`
`T(arg)`	`T(*)(arg)`
`T`	`const T`
`T`	`volatile T`
`T*`	`const T*`
`T*`	`volatile T*`

Sekvence konverzija

- Sekvence konverzija se, pri proceni odgovaranja tipa formalnog argumenta tipu stvarnog argumenta, razmatraju na sledeći način:
  1. Potpuno slaganje. Sekvence koje se sastoje samo od nula ili više trivijalnih konverzija, bolje su nego sve druge sekvence.
  2. Slaganje sa celebrojnim promocijama. Od sekvenci koje nisu navedene pod 1, one sekvence koje sadrže samo celobrojnu promociju, konverziju iz float u double i trivijalne konverzije, bolje su nego sve druge sekvence.
  3. Slaganje sa standardnim konverzijama. Od sekvenci koje nisu navedene pod 2, one sekvence koje sadrže samo standardne konverzije i trivijalne konverzije, bolje su nego sve druge sekvence.
     -- ako je klasa B javno izvedena (public) iz klase A,
     konverzija iz B* u A* je bolja nego konverzija iz B* u void* i const void*;
     -- ako je klasa C javno izvedena (public) iz klase B,
     konverzija iz C* u B* je bolja nego konverzija iz C* u A*,
     konverzija iz C& u B& je bolja nego konverzija iz C& u A&.
  4. Slaganje sa korisnički definisanim konverzijama. Od sekvenci koje nisu navedene pod 3, sekvence koje sadrže samo korisnički definisane konverzije, standardne konverzije i trivijalne konverzije, bolje su nego sve druge sekvence.
  5. Slaganje sa tri tačke (...). Sekvence koje sadrže slaganje sa tri tačke (...) su najgore sekvence za slaganje.

Adresa preklopljene funkcije

```
void f(int);
void f(char);
void (*pfi)(int)=&f;       // adresa funkcije f tipa void(int);
void (*pfc)(char)=&f;      // adresa funkcije f tipa void(char);
void (*pf)(char*,...)=&f; // greška: nema funkcije f tipa void(char*,...)
```

- Ako je preklopljeno ime funkcije stvarni argument u pozivu neke funkcije, bira se ona funkcija koja po tipu u potpunosti odgovara deklarisanom formalnom argumentu.
- Slično važi i za argumente operatorskih funkcija.
  - Ako je preklopljeno ime upotrebljeno u naredbi return, onda se bira funkcija koja po tipu u potpunosti odgovara deklarisanom tipu povratne vrednosti.
  - Ako je preklopljeno ime upotrebljeno na levoj strani znaka dodele (=), bira se ona funkcija koja po tipu u potpunosti odgovara tipu leve strane znaka dodele.
  - Sve druge upotrebe, koje dovode do višeznačnosti, nisu dozvoljene.
- Ne postoji standardna konverzija iz jednog tipa funkcije u drugi. Zato nije dozvoljeno čak ni ovo:

```
int f(char*,...);
int (*pf)(char*,int)=&f;  // greška: neslaganje tipova!
```

Zadatak

```
//_____________________________fajl PrintData.h _____________________________
#pragma once
#include <iostream>
using namespace std;
class PrintData {
   public:
      void Print(int i);
      void Print(double  f);
      void Print(char* c);
};
//_____________________________fajl PrintData.cpp _____________________________
#include "PrintData.h"
void PrintData::Print(int i){
    cout << "BAZNA   : int  : " << i << endl;
}
void PrintData::Print(double  f) {
    cout << "BAZNA   : float: " << f << endl;
}
void PrintData::Print(char* c){
    cout << "BAZNA   : chars: " << c << endl;
}
```

Zadatak

```
//_____________________________fajl SuperPrintData.h _____________________________
#pragma once
#include "PrintData.h"

class SuperPrintData: public PrintData {
  public:
    void Print();
    void Print(char*);
};
//_____________________fajl SuperPrintData.cpp _____________________________

#include "SuperPrintData.h"

void SuperPrintData::Print(){
  cout<<"IZVEDENA: funkcija Print() u IZVEDENOJ klasi"<<endl;
  cout<<"          sakriva funkcije Print(int) i Print(double) iz bazne
klase"<<endl;
}

void SuperPrintData::Print(char* str){
   cout<<"IZVEDENA: sada cu pozvati u BAZNOJ klasi print(char*)"<<endl;
   PrintData::Print(str);
}
```

Zadatak

```
//_____________________________fajl gde je main
#include <iostream>
#include "PrintData.h"
#include "SuperPrintData.h"
int main(void){
   PrintData pd;
   pd.Print(5);     pd.Print(500.263);       pd.Print("Hello C++");
   SuperPrintData spd;
   spd.Print();
   //spd.Print(1); //ne bi Vam dozvolio jer je skrivena
   spd.Print("moguce jer je ponovo definisana metoda Print(char*) u
izvedenoj klasi");
   system("pause");return 0;
}
//_____________________________________IZLAZ
BAZNA   : int  : 5
BAZNA   : float: 500.263
BAZNA   : chars: Hello C++
IZVEDENA: funkcija Print() u IZVEDENOJ klasi
          sakriva funkcije Print(int) i Print(double) iz bazne klase
IZVEDENA: sada cu pozvati u BAZNOJ klasi print(char*)
BAZNA   : chars: moguce jer je ponovo definisana metoda Print(char*) u
izvedenoj klasi
```