Predavanje 4_a

Veliko ograničenje jezika C sastoji se u činjenici da je podržan samo jedan način prenosa parametara u funkcije, poznat pod nazivom *prenos parametara po vrijednosti* (engl. *passing by value*). Ovaj mehanizam omogućava prenošenje vrijednosti sa mjesta poziva funkcije u samu funkciju. Međutim, pri tome ne postoji nikakav način da funkcija promijeni vrijednost nekog od stvarnih parametara koji je korišten u pozivu funkcije, s obzirom da funkcija manipulira samo sa formalnim parametrima koji su *kopije stvarnih parametara*, odnosno oni predstavljaju *posve neovisne objekte od stvarnih parametara* (mada su inicijalizirani tako da im na početku izvršavanja funkcije *vrijednosti budu jednake vrijednosti stvarnih parametara*). Slijedi da se putem ovakvog prenosa parametara ne može prenijeti *nikakva informacija iz funkcije nazad na mjesto poziva funkcije*. Informacija se doduše može prenijeti iz funkcije na mjesto poziva putem *povratne vrijednosti*, ali postoje situacije gdje to nije dovoljno.

Posljedica pomenutog ograničenja je da se u jeziku C moraju prečesto koristiti pokazivači, što je čest izvor grešaka. Zamislimo, na primjer, da želimo napisati funkciju "`Udvostruci`" koja udvostručava vrijednost svog argumenta. Recimo, želimo da sljedeća sekvenca naredbi ispiše broj 10:

```
int a(5);
Udvostruci(a);
std::cout << a;
```

Jasno je da to ne možemo učiniti funkcijom poput sljedeće, s obzirom da se prema uobičajenom mehanizmu prenosa parametara u funkcije, stvarni parametar pri pozivu (u našem slučaju "`a`") samo *kopira* u formalni parametar funkcije "`x`", i svaka dalja manipulacija sa formalnim parametrom "`x`" utiče samo na njega, a ne na stvarni parametar:

```
void Udvostruci(int x) {
  x *= 2;                              // Ovo neće ispravno raditi...
}
```

Kako postići da funkcija "`Udvostruci`" radi ono što smo naumili? U jeziku C *nikako* – naime, u C-u ne postoji nikakav način da funkcija promijeni vrijednost svog stvarnog parametra putem formalnog parametra. Najbliže tome što u C-u možemo uraditi je da formalni parametar funkcije deklariramo kao *pokazivač*, koji unutar te funkcije moramo eksplicitno dereferencirati pomoću operatora "`*`" sa ciljem da pristupimo sadržaju na koji on pokazuje, a ne samom pokazivaču:

```
void Udvostruci(int *x) {
  *x *= 2;
}
```

Međutim, posljedica ovakvog rješenja je da ćemo pri pozivu funkcije "`Udvostruci`", umjesto samog stvarnog parametra "`a`" morati prenostiti njegovu *adresu*, koju možemo dobiti pomoću operatora "`&`" (s obzirom da i formalni parametar, koji je pokazivač, očekuje adresu):

```
int a(5);
Udvostruci(&a);
std::cout << a;
```

Ovim jesmo postigli traženi efekat, ali korisnika funkcije tjeramo da pri pozivu funkcije eksplicitno prenosi adresu argumenta umjesto samog argumenta, dok sama funkcija mora dereferencirati svoj formalni argument primjenom operatora "`*`". Ovo se mora raditi kod svih funkcija koje treba da promijene vrijednost svog stvarnog argumenta ili da smjeste neku vrijednost u njega (sjetite se koliko ste puta zaboravili da upotrijebite operator "`&`" u funkciji "`scanf`"). Navedimo još jedan karakterističan primjer. Neka je potrebno napisati funkciju "`Razmijeni`" koja razmjenjuje vrijednost svojih argumenata (realnog tipa). Na primjer, ukoliko (realna) promjenljiva "`a`" ima vrijednost 5.12, a (realna) promjenljiva "`b`" vrijednost 8, želimo da nakon poziva

```
Razmijeni(a, b);
```

promjenljiva "`a`" dobije vrijednost 8, a promjenljiva "`b`" vrijednost 5.12. U C-u je tačno ovo nemoguće postići. Naime, ukoliko napišemo funkciju poput sljedeće:

```
void Razmijeni(double x, double y) {
  double pomocna(x);                          // Ovo također neće raditi!
  x = y; y = pomocna;
}
```

nećemo postići željeni efekat – razmjena će se obaviti nad formalnim parametrima "x" i "y" koji predstavljaju *kopije* stvarnih parametara "a" i "b", ali su *potpuno neovisni od njih*. Djelomično rješenje ponovo možemo postići primjenom pokazivača, odnosno prepravljanjem funkcije "Razmijeni" da izgleda ovako:

```
void Razmijeni(double *x, double *y) {
  double pomocna(*x);
  *x = *y; *y = pomocna;
}
```

Međutim, nedostatak ovog rješenja je što kao stvarne parametre ponovo moramo zadavati adrese:

```
Razmijeni(&a, &b);
```

Pored ovog nedostatka, rješenje zasnovano na upotrebi pokazivača osjetljivo je na mogućnost formiranja grešaka koje mogu nastati ukoliko se na nekom mjestu zaboravi staviti zvjezdica ili se stavi zvjezdica tamo gdje ne treba. Detaljan opis grešaka koje se tom prilikom mogu napraviti zajedno sa analizom njihovih posljedica može se naći u dodatku predavanjima posvećenom pokazivačima.

Izloženi problem se u jeziku C++ elegantno rješava upotrebom tzv. *referenci* (ili *upućivača*, kako se ovaj termin ponekad prevodi). Reference su specijalni objekti koji su po internoj građi vrlo slični pokazivačima, ali se sa aspekta upotrebe veoma razlikuju. I reference i pokazivači u sebi interno sadrže adresu objekta na koji upućuju (odnosno *pokazuju* kako se to kaže u slučaju kada koristimo pokazivače). Međutim, suštinska razlika između referenci i pokazivača sastoji se u činjenici da su reference izvedene na takav način da *u potpunosti oponašaju objekat na koji upućuju* (odnosno, svaki pokušaj pristupa referenci se automatski preusmjerava na objekat na koji referenca upućuje, dok je sam pristup internoj strukturi reference nemoguć). S druge strane, kod pokazivača je napravljena striktna razlika između pristupa *samom pokazivaču* (odnosno *adresi* koja je u njemu pohranjena) i pristupa *objektu na koji pokazivač pokazuje*. Drugim riječima, za pristup objektu na koji pokazivač pokazuje koristi se *drugačija sintaksa* u odnosu na pristup internoj strukturi pokazivača ("*p" u prvom slučaju, odnosno samo "p" u drugom slučaju).

Zbog činjenice da se reference u potpunosti poistovjećuju sa objektima na koje upućuju, reference je najlakše zamisliti kao *alternativna imena* (engl. *alias names*) za druge objekte. Reference se deklariraju kao i obične promjenljive, samo se prilikom deklaracije ispred njihovog imena stavlja oznaka "&". Bitno je uočiti da ovaj znak upotrijebljen ispred imena promjenljive ima *potpuno drugačije značenje* kada se koristi za potrebe deklaracije neke promjenljive, nego upotrijebljen u drugim kontekstima. Naime, ovaj znak upotrijebljen prilikom deklaracije promjenljive ispred njenog imena označava da se radi o referenci a ne o običnoj promjenljivoj, dok isti znak upotrijebljen u proizvoljnim izrazima ispred imena neke promjenljive označava da želimo uzeti njenu *adresu*. Eventualnu zbrku još više pojačava činjenica da isti znak može označavati i *konjunkciju po bitima* ako se upotrijebi kao binarna operacija između dva cjelobrojna operanda. Sretna okolnost je da se ova operacija ne viđa tako često, pogotovo kod početnika.

Reference se obavezno moraju *inicijalizirati* prilikom deklaracije, bilo upotrebom znaka "=", bilo pomoću konstruktorske sintakse (navođenjem inicijalizatora unutar okruglih zagrada) ili pomoću jednoobrazne sitnakse počev od verzije C++11 (navođenjem inicijalizatora unutar vitičastih zagrada). Međutim, za razliku od običnih promjenljivih, reference se ne mogu inicijalizirati proizvoljnim izrazom, već samo nekom *drugom promjenljivom* potpuno istog tipa (dakle, konverzije tipova poput konverzije iz tipa "int" u tip "double" *nisu dozvoljene*) ili, općenitije, nekom *l-vrijednošću* istog tipa. Pri tome, referenca postaje *vezana* (engl. *tied*) za promjenljivu (odnosno l-vrijednost) kojom je inicijalizirana u smislu koji ćemo uskoro razjasniti. Na primjer, ukoliko je "a" neka cjelobrojna promjenljiva, referencu "b" vezanu za promjenljivu "a" možemo deklarirati na neki od sljedećih ekvivalentnih načina:

```
int &b = a;          // Sintaksa u duhu C-a
int &b(a);           // Ova sintaksa je više u duhu C++-a
int &b{a};           // Samo od C++11
```

Kada bi "b" bila obična promjenljiva a ne referenca, efekat ovakve deklaracije bio bi da bi se promjenljiva "b" inicijalizirala trenutnom vrijednošću promjenljive "a", nakon čega bi se ponašala kao posve neovisan objekat od promjenljive "a", odnosno eventualna promjena sadržaja promjenljive "b" ni na kakav način ne bi

uticala na promjenljivu "a". Međutim, reference se *potpuno poistovjećuju* sa objektom na koji su vezane. Drugim riječima, "b" se ponaša kao *alternativno ime* za promjenljivu "a", odnosno svaka manipulacija sa objektom "b" odražava se na identičan način na objekat "a" (kaže se da je "b" referenca na promjenljivu "a"). To možemo vidjeti iz sljedećeg primjera:

```
b = 7;                                  // b je faktički a...
std::cout << a;                         // Ispisuje 7
```

Ovaj primjer će ispisati broj 7, bez obzira na eventualni prethodni sadržaj promjenljive "a", odnosno dodjela vrijednosti 7 promjenljivoj "b" zapravo je promijenila sadržaj promjenljive "a". Objekti "a" i "b" ponašaju se kao da se radi o istom objektu! Konceptualno, reference su slične *prečicama* (engl. *shortcuts*) u Windows seriji operativnih sistema. Naime, prečica na neki dokument ne predstavlja neovisnu kopiju tog dokumenta, nego samo *alternativni način* da pristupimo nekom već postojećem dokumentu, slično kao što je referenca alternativni način da pristupimo nekom već postojećem objektu.

Bitno je naglasiti da nakon što se referenca jednom veže za neki objekat, *ne postoji nikakav legalan način da se ona preusmjeri na neki drugi objekat*. Zaista, ukoliko je npr. "c" također neka cjelobrojna promjenljiva, tada naredba dodjele poput

```
b = c;                                  // Ovo je faktički a = c...
```

neće preusmjeriti referencu "b" na promjenljivu "c", nego će promjenljivoj "a" dodijeliti vrijednost promjenljive "c", s obzirom da je referenca "b" i dalje vezana za promjenljivu "a". Svaka referenca čitavo vrijeme svog života uvijek upućuje na objekat za koji je vezana prilikom inicijalizacije. Pored toga, kako referenca uvijek mora biti vezana za neki objekat, to deklaracija poput

```
int &b;                                 // NEDOZVOLJENO!
```

nema nikakvog smisla.

Referenca na neki objekat *nije* taj objekat (tehnički gledano, u većini implementacija ona je zapravo neka vrsta prerušenog pokazivača na njega), ali je bitno da se ona u svemu ponaša kao da ona *jeste* taj objekat. Drugim riječima, ne postoji apsolutno nikakav način kojim bi program mogao utvrditi da se referenca i po čemu razlikuje od objekta za koji je vezana, odnosno da ona *nije* objekat za koji je vezana. Čak i neke od najdelikatnijih operacija koje bi se mogle primijeniti na referencu obaviće se nad objektom za koji je referenca vezana. Dakle, referenca i objekat za koji je referenca vezana tehnički posmatrano *nisu isti objekat*, ali program *nema način da to sazna*. Sa aspekta izvršavanja programa, referenca i objekat za koji je ona vezana predstavljaju *potpuno isti objekat*! Ipak, potrebno je naglasiti da tip nekog objekta i tip reference na taj objekat *nisu isti*. Dok je, u prethodnom primjeru, promjenljiva "a" tipa *cijeli broj* (tj. tipa "`int`") dotle je promjenljiva "b" tipa *referenca na cijeli broj* (tipa "`int &`"). Početnik se ovom razlikom u tipu između reference i objekta na koji referenca upućuje ne treba da zamara, s obzirom da je ona uglavnom nebitna, osim u nekim vrlo specifičnim slučajevima, na koje ćemo ukazati onog trenutka kada se pojave. Ipak, nije loše znati da ova razlika *postoji*, s obzirom da je u jeziku C++ pojam tipa izuzetno važan, jer se mnoge operacije različito izvode ovisno od tipa objekta na koji su primijenjene. Ipak, bez obzira na ovu razliku u tipu, čak ni ona se ne može iskoristiti da program sazna da neka promjenljiva predstavlja referencu na neki objekat, a ne sam objekat. Reference u jeziku C++ su zaista izuzetno dobro "zamaskirane".

Reference u jeziku C++ imaju mnogobrojne primjene. Jedna od najočiglednijih primjena je tzv. *prenos parametara po referenci* (engl. *passing by reference*) koji omogućava rješenje problema postavljenog na početku ovog predavanja. Naime, da bismo postigli da funkcija promijeni vrijednost svog stvarnog parametra, *dovoljno je da odgovarajući formalni parametar bude referenca*. Na primjer, funkciju "`Udvostruci`" trebalo bi modificirati na sljedeći način:

```
void Udvostruci(int &x) {
  x *= 2;
}
```

Da bismo vidjeli šta se ovdje zapravo dešava, pretpostavimo da je ova funkcija pozvana na sljedeći način ("a" je neka cjelobrojna promjenljiva):

```
Udvostruci(a);
```

Prilikom ovog poziva, formalni parametar "x" koji je *referenca* biće inicijaliziran stvarnim parametrom "a". Međutim, prilikom inicijalizacije referenci, one se *vezuju* za objekat kojim su inicijalizirane, tako da se formalni parametar "x" vezuje za promjenljivu "a". Stoga će se svaka promjena sadržaja formalnog parametra "x" zapravo odraziti na promjenu stvarnog parametra "a", odnosno za vrijeme izvršavanja funkcije "Udvostruci", promjenljiva "x" se ponaša kao da ona u stvari *jeste* promjenljiva "a". Ovdje je iskorištena osobina referenci da se one ponašaju tako kao da su one upravo objekat za koji su vezane. Po završetku funkcije, referenca "x" se *uništava*, kao i svaka druga lokalna promjenljiva na kraju bloka kojem pripada. Međutim, za vrijeme njenog života, ona je iskorištena da promijeni sadržaj stvarnog parametra "a", koji razumljivo ostaje takav i nakon što referenca "x" prestane "živjeti"! Naravno, pri novom pozivu funkcije "Udvostruci" (eventualno sa nekim drugim argumentom) referenca "x" se ponovo *rađa* i može biti vezana za neki drugi objekat, ali to je njen novi "život" (u toku jednog "života", referenca uvijek ostaje trajno vezana za *jedan objekat*, od trenutka njenog "rođenja" pa do "smrti").

U slučaju kada je neki formalni parametar referenca, odgovarajući stvarni parametar *mora biti l-vrijednost* (tipično neka promjenljiva), jer se reference mogu vezati samo za l-vrijednosti. Drugim riječima, odgovarajući parametar *ne smije biti broj*, ili *proizvoljan izraz* (pojedini izrazi, poput dereferenciranih pokazivača, doduše *jesu l-vrijednosti*, ali većina izraza nema status l-vrijednosti). Stoga, pozivi poput sljedećih *nisu dozvoljeni*:

```
Udvostruci(7);              // ILEGALNO: Broj 7 nije l-vrijednost!
Udvostruci(2 * a - 3);      // ILEGALNO: Izraz 2 * a - 3 nije l-vrijednost!
```

Zapravo, ako malo razmislimo, jasno je da ovakvi pozivi u suštini *nemaju smisla*. Funkcija "Udvostruci" je dizajnirana sa ciljem da *promijeni vrijednost svog stvarnog argumenta*, što je nemoguće ukoliko je on, na primjer, broj. Broj ima svoju vrijednost koju nije moguće mijenjati!

Kako su individualni elementi nizova, vektora i dekova također l-vrijednosti, kao stvarni parametar koji se prenosi po referenci može se upotrijebiti i individualni element niza, vektora ili deka. Isto vrijedi i za dereferencirane pokazivače. Tako, na primjer, ukoliko imamo deklaracije

```
int niz[10];
double *p;
```

tada se sljedeći pozivi sasvim legalno mogu iskoristiti za udvostručavanje elementa niza sa indeksom 2, odnosno za udvostručavanje sadržaja objekta na koji u tom trenutku pokazivač "p" pokazuje:

```
Udvostruci(niz[2]);             // Udvostručava niz[2]
Udvostruci(*p);                 // Udvostručava *p
```

Ovo je posljedica činjenice da se referenca može vezati za bilo koju l-vrijednost, pa tako i za individualni element niza, vektora ili deka, odnosno na dereferencirane pokazivače.. Drugim riječima, deklaracija poput

```
int &element(niz[2]);           // element faktički postaje sininim za niz[2]
double &neki_objekat(*p);       // OPREZ: neki_element nije sinonim za *p!
```

sasvim su legalne, i nakon njih ime "element" postaje alternativno ime za element niza "niz" sa indeksom 2, dok ime "neki_objekat" postaje alternativno ime za objekat na koji je pokazivač "p" pokazivao u *trenutku nailaska na ovu deklaraciju*. Tako će dodjela poput "element = 5" imati potpuno isti efekat kao i dodjela "niz[2] = 5". S druge strane, dodjela "neki_objekat = 2.5" *ne mora imati nužno isti efekat kao i dodjela* "*p = 2.15". Naime, referenca "neki_objekat" vezuje se za objekat na koji je pokazivač "p" pokazivao *u trenutku njenog kreiranja*. Ukoliko se pokazivač "p" naknadno preusmjeri da pokazuje na neki drugi objekat, "*p" tada više *neće biti isti objekat* za koji se referenca "neki_objekat" vezala. U tom smislu, pokazivači su *fleksibilniji od referenci*, s obzirom da tokom svog života mogu *mijenjati* objekte na koju pokazuju.

Pomoću referenci ćemo jednostavno riješiti i ranije pomenuti problem vezan za funkciju "Razmijeni" koja bi trebala da razmijeni vrijednosti svojih argumenata. Naime, dovoljno je funkciju prepraviti tako da joj formalni parametri budu reference:

```
void Razmijeni(double &x, double &y) {
  double pomocna(x);
  x = y; y = pomocna;
}
```

Ova funkcija će razmijeniti proizvoljne dvije realne promjenljive (ili, općenitije, l-vrijednosti) koje joj se proslijede kao stvarni parametri. Na primjer, ukoliko je vrijednost promjenljive "a" bila 2.13 a sadržaj

promjenljive "b" 3.6, nakon izvršenja poziva "Razmijeni(a, b)" vrijednost promjenljive "a" će biti 3.6, a vrijednost promjenljive "a" biće 2.13. Nije teško uvidjeti kako ova funkcija radi: njeni formalni parametri "x" i "y", koji su reference, vežu se za navedene stvarne parametre. Funkcija pokušava da razmijeni dvije reference, ali će se razmijena obaviti nad promjenljivim za koje su ove reference vezane. Jasno je da se ovakav efekat može ostvariti samo putem prenosa po referenci, jer u suprotnom funkcija "Razmijeni" ne bi mogla imati utjecaj na svoje stvarne parametre.

Kako su individualni elementi niza također l-vrijednosti, funkcija "Razmijeni" se lijepo može iskoristiti za razmjenu dva elementa niza. Na primjer, ukoliko je "niz" neki niz realnih brojeva (sa barem 6 elemenata), tada će naredba

```
Razmijeni(niz[2], niz[5]);        // Razmjenjuje niz[2] i niz[5]
```

razmijeniti elemente niza sa indeksima 2 i 5 (tj. treći i šesti element).

Reference u jeziku C++ su mnogo fleksibilnije nego u većini drugih programskih jezika. Na primjer, u C-u one uopće ne postoje, dok u jeziku Pascal samo formalni parametri mogu biti reference, odnosno referenca kao pojam ne postoji izvan konteksta formalnih parametara. Stoga se u Pascal-u pojam reference (kao neovisnog objekta) uopće ne uvodi, nego se samo govori o prenosu parametara po referenci. S druge strane, u jeziku C++ referenca može postojati kao objekat *posve neovisan o formalnim parametrima*, a prenos po referenci se prosto ostvaruje tako što se odgovarajući formalni parametar deklarira kao referenca.

Reference također imaju veliku primjenu u rangovskim *for*-petljama (ovo se naravno odnosi samo na C++11). Naime, osnovni problem kod rangovskih *for*-petlji je što upravljačka promjenljiva koja kontrolira rad ove petlje podrazumijevano predstavlja samo *kopiju* odgovarajućeg elementa niza, vektora ili neke druge kolekcije podataka kroz koju se petlju kreće, a ne i sam odgovarajući element. Recimo, uz pretpostavku da je "a" neki niz ili vektor, sljedeća petlja *neće ispravno udvostručiti elemente tog niza odnosno vektora* (ovdje smo upotrijebili "**auto**" deklaraciju da ne moramo razmišljati o tome kojeg su tipa elementi tog niza odnosno vektora):

```
for(auto x : a) x *= 2;           // Ova petlja ne daje očekivani rezultat!
```

Zaista, u svakom prolazu kroz petlju, "x" će biti samo kopija odgovarajućeg elementa niza (vektora), tako da će konstrukcija "x *= 2" *udvostručiti tu kopiju, a ne i sam element niza* (*vektora*). Drugim riječima, uz pretpostavku da je "a" recimo vektor (tako da podržava "size" funkciju), prethodna petlja je efektivno ekvivalentna sljedećoj konstrukciji:

```
for(int i = 0; i < a.size(); i++) {
  auto x(a[i]);                                  // x je KOPIJA od a[i]!
  x *= 2;
}
```

Problem se veoma lako rješava stavljanjem da je "x" *referenca*:

```
for(auto &x : a) x *= 2;                         // Sada je sve OK!
```

Zaista, ovo je sad ekvivalentno sljedećoj konstrukciji:

```
for(int i = 0; i < a.size(); i++) {
  auto &x(a[i]);                                 // x i a[i] su sada efektivno
  x *= 2;                                        //   jedno te isto...
}
```

Prenos parametara u funkcije po referenci dolazi do punog izražaja kada je potrebno prenijeti *više od jedne vrijednosti* iz funkcije nazad na mjesto njenog poziva. Naime, poznato je da funkcija *ne može nikada vratiti više od jedne vrijednosti*. Prije nego što nastavimo dalje, nije na odmet napomenuti da zbog vrlo neobične interpretacije znaka "zarez" u jezicima C i C++, početnici mogu steći *iluziju da je iz funkcije moguće vratiti više od jedne vrijednosti*. Naime, problem je što će sintaksno proći funkcije poput sljedećih, za koje *izgleda* da vraćaju više vrijednosti (ovdje konkretno dvije):

```
int VratiZbirIRazliku(int x, int y) {
  return x + y, x - y;                                  // OVO JE ZABLUDA!!!
}
```

Međutim, svakome ko je shvatio kako zapravo djeluje zarez-operator, jasno je da se iz ove funkcije vraća samo razlika "`x - y`". Što je još gore, sintaksno će proći i pokušaji da se ova funkcija "pozove" i da se "prihvate" dva "vraćena rezultata" poput sljedeće:

```
zbir, razlika = VratiZbirIRazliku(10, 6);            // NASTAVAK ZABLUDE...
```

I ovdje bi trebalo da bude jasno da će samo promjenljivoj "`razlika`" biti dodijeljena vrijednost, dok će promjenljiva "`zbir`" ostati onakva kakva je i bila...

Nakon ove digresije, razmotrimo šta možemo učiniti da prevaziđemo problem nemogućnošću vraćanja više vrijednosti kao rezultat iz funkcije. Jedna mogućnost je upotreba tzv. *struktura*, o kojima ćemo govoriti kasnije. Međutim, često korištena strategija je i *prenos parametara po referenci*, pri čemu funkcija koristeći reference prosto *smješta* tražene rezultate u odgovarajuće stvarne parametre koji su joj proslijeđeni. Ova tehnika je ilustrirana u sljedećem programu u kojem je definirana funkcija "`RastaviSekunde`", čiji je prvi parametar broj sekundi, a koja kroz drugi, treći i četvrti parametar prenosi na mjesto poziva informaciju o broju sati, minuta i sekundi koji odgovaraju zadanom broju sekundi. Ovaj prenos se ostvaruje tako što su drugi i treći formalni parametar ove funkcije ("`sati`", "`minute`" i "`sekunde`") deklarirani kao reference, koje se za vrijeme izvršavanja funkcije vezuju za odgovarajuće stvarne argumente:

```
#include <iostream>

void RastaviSekunde(int br_sek, int &sati, int &minute, int &sekunde) {
  sati = br_sek / 3600;
  minute = (br_sek % 3600) / 60;
  sekunde = br_sek % 60;
}

int main() {
  int sek, h, m, s;
  std::cout << "Unesi broj sekundi: ";
  std::cin >> sek;
  RastaviSekunde(sek, h, m, s);
  std::cout << "h = " << h << "  m = " << m << "  s = " << s << "\n";
  return 0;
}
```

Kao što je već rečeno, stvarni parametri koji se prenose po vrijednosti mogu bez ikakvih problema biti konstante ili izrazi, dok stvarni parametri koji se prenose po referenci *moraju* biti l-vrijednosti (obično promjenljive). Tako su uz funkciju "`RastaviSekunde`" iz prethodnog primjera sasvim legalni pozivi

```
RastaviSekunde(73 * sek + 13, h, m, s);
RastaviSekunde(12322, h, m, s);
```

dok sljedeći pozivi *nisu legalni*, jer odgovarajući stvarni parametri nisu l-vrijednosti:

```
RastaviSekunde(sek, 3 * h + 2, m, s);          // 3 * h + 2 nije l-vrijednost
RastaviSekunde(sek, h, 17, s);                 // 17 nije l-vrijednost
RastaviSekunde(1, 2, 3, 4);                    // 2, 3 i 4 nisu l-vrijednosti
RastaviSekunde(sek + 2, sek - 2, m, s);        // sek - 2 nije l-vrijednost
```

Interesantan je sljedeći primjer, u kojem funkcija istovremeno daje izlazne informacije putem prenosa parametara po referenci, i kao rezultat daje *statusnu informaciju* o obavljenom odnosno neobavljenom poslu. Prikazana funkcija "`KvadratnaJednacina`" nalazi rješenja kvadratne jednačine čiji se koeficijenti zadaju preko prva tri parametra. Za slučaj da su rješenja realna, funkcija ih *prenosi nazad* na mjesto poziva funkcije kroz četvrti i peti parametar i *vraća* vrijednost "**`true`**" kao rezultat. Za slučaj kada rješenja nisu realna, funkcija vraća "**`false`**" kao rezultat, a četvrti i peti parametar *ostaju netaknuti*. Obratite pažnju na način kako je ova funkcija upotrijebljena u glavnom programu:

```
#include <iostream>
#include <cmath>

bool KvadratnaJednacina(double a, double b, double c, double &x1, double &x2) {
  double d(b * b - 4 * a * c);
  if(d < 0) return false;                      // Signal "nema realnih"...
  x1 = (-b - std::sqrt(d)) / (2 * a);
  x2 = (-b + std::sqrt(d)) / (2 * a);
  return true;                                 // Signal "sve OK"...
}
```

```
int main() {
  double a, b, c;
  std::cout << "Unesi koeficijente a, b i c: ";
  std::cin >> a >> b >> c;
  double x1, x2;
  bool rjesenja_su_realna(KvadratnaJednacina(a, b, c, x1, x2));
  if(rjesenja_su_realna)
    std::cout << "x1 = " << x1 << "\nx2 = " << x2 << std::endl;
  else std::cout << "Jednačina nema realnih rješenja!\n";
  return 0;
}
```

Napomenimo da smo, s obzirom da funkcija "KvadratnaJednacina" vraća vrijednost tipa "**bool**" kao rezultat, mogli pisati i sekvencu naredbi poput sljedeće

```
if(KvadratnaJednacina(a, b, c, x1, x2))
  std::cout << "x1 = " << x1 << "\nx2 = " << x2 << std::endl;
else std::cout << "Jednačina nema realnih rješenja!\n";
```

u kojoj se računanje rješenja odvija kao *propratni efekat* uvjeta "**if**" naredbe. Međutim, ovakve konstrukcije treba *izbjegavati*, zbog smanjene čitljivosti. Uvjete sa propratnim efektima treba koristiti samo ukoliko se njihovom primjenom zaista postiže nešto korisno što bi na drugi način rezultiralo osjetno kompliciranijim ili neefikasnijim kodom. Također, prenos parametara po referenci treba koristiti *samo kada je to zaista neophodno*, s obzirom da kod takvog prenosa iz samog poziva funkcije nije vidljivo da vrijednosti stvarnih parametara mogu biti promijenjene. Stoga, funkcijama koje koriste prenos parametara po referenci treba davati takva imena da ta činjenica bude jasno vidljiva iz naziva funkcije (npr. ime poput "SmjestiMinIMax" ili "NadjiMinIMax" znatno je sugestivnije po tom pitanju nego ime poput "MinIMax").

S obzirom na *smjer prenosa informacija* koji se ostvaruje putem prenosa parametara između funkcije i onoga ko poziva funkciju, sve parametre možemo podijeliti u tri klase: *ulazni*, *izlazni* i *ulazno-izlazni* parametri. Ulazni parametri su oni preko kojih se informacije prenose samo u smjeru od pozivaoca funkcije ka funkciji, i to su svi oni koji se prenose po vrijednosti. Za realizaciju izlaznih i ulazno-izlaznih parametara neophodno je koristiti reference. Izlazni su oni parametri kod kojih se informacija prenosi samo u smjeru iz funkcije ka pozivaocu funkcije, odnosno vrijednosti stvarnih parametara kakvi su bili prije poziva funkcije potpuno je nebitna (bitna je samo njihova vrijednost koju će dobiti *nakon* poziva funkcije). Takvi su recimo parametri "sati", "minute" i "sekunde" u funkciji "RastaviSekunde", ili parametri "x1" i "x2" u funkciji "KvadratnaJednacina". Konačno, kod ulazno-izlaznih parametara imamo prenos informacije u oba smjera između pozivaoca funkcije i same funkcije, odnosno bitna je vrijednost stvarnih parametara i prije poziva funkcije i nakon njenog završetka (pri tome se očekuje da će funkcija *promijeniti* njihovu vrijednost). Takav je recimo bio parametar "x" u funkciji "Udvostruci", kao i parametri "x" i "y" u funkciji "Razmijeni".

Bitno je naglasiti da se kod prenosa parametara po referenci formalni i stvarni parametri *moraju u potpunosti slagati po tipu*, jer se reference mogu vezati samo za promjenljive odgovarajućeg tipa (osim u nekim izuzecima vezanim za tzv. nasljeđivanje, sa kojima ćemo se susresti kasnije). Na primjer, ukoliko funkcija ima formalni parametar koji je referenca na tip "**double**", odgovarajući stvarni parametar ne može biti npr. tipa "**int**". Razlog za ovo ograničenje nije teško uvidjeti. Razmotrimo, na primjer, sljedeću funkciju:

```
void Problem(double &x) {
  x = 3.25;
}
```

Pretpostavimo da se funkcija "Problem" može pozvati navodeći neku cjelobrojnu promjenljivu kao stvarni argument. Formalni parametar "x" trebao bi se nakon toga vezati i poistovjetiti sa tom promjenljivom. Međutim, funkcija smješta u "x" vrijednost koja nije cjelobrojna. U skladu sa djelovanjem referenci, ova vrijednost trebala bi da se zapravo smjesti u promjenljivu za koju je ova referenca vezana, što je očigledno nemoguće s obzirom da se radi o cjelobrojnoj promjenljivoj. Dakle, smisao načina na koji se reference ponašaju ne može biti ostvaren, što je ujedno i razlog zbog kojeg se formalni i stvarni parametar u slučaju prenosa po referenci moraju u potpunosti slagati po tipu.

Posljedica činjenice da se parametri koji se prenose po referenci moraju u potpunosti slagati po tipu je da se maloprije napisana funkcija "Razmijeni" ne može primijeniti za razmjenu dvije promjenljive koje nisu tipa "**double**", npr. dvije promjenljive tipa "**int**" (pa čak ni promjenljive tipa "**float**"), bez obzira što sama funkcija ne radi ništa što bi suštinski trebalo ovisiti od tipa promjenljive. Način kako se može riješiti ovaj problem razmotrićemo nešto kasnije.

U programiranju je poznata pojava nazvana *aliasing*, koja nastaje kada je jednom te istom objektu moguće pristupiti na više različitih načina. Aliasing može biti posebno izražen kad se u igru uključe i reference, jer tada jedan te isti objekat *može imati više različitih imena*. Ova pojava može uzrokovati ozbiljne previde ukoliko do nje dođe *nesvjesno* (tj. bez namjere). Posmatrajmo recimo sljedeću funkciju:

```
void Funkcija(int &x, int &y) {
  x += 2; y += 5;
  std::cout << x << " " << y << std::endl;
}
```

Pretpostavimo sada da smo ovu funkciju pozvali ovako:

```
int a(1);
Funkcija(a, a);                          // Šta će se ispisati na ekranu???
```

Šta će ova funkcija ispisati na ekran ukoliko se izvrši ovakav poziv? Ukoliko je Vaš odgovor "`3 6`", nesvjesno ste previdili pojavu aliasinga koji je ovdje nastao, s obzirom da se nakon ovakvog poziva reference "`x`" i "`y`" vežu na *istu promjenljivu* "`a`" (odnosno, one zapravo predstavljaju *dva različita imena* za istu promjenljivu "`a`"). Stoga je tačan odgovor "`8 8`".

Do sada smo vidjeli da formalni parametri funkcija mogu biti reference. Međutim, interesantno je da rezultat koji vraća funkcija *također može biti referenca* (nekada se ovo naziva *vraćanje vrijednosti po referenci*). Ova mogućnost se ne koristi prečesto, ali kasnije ćemo vidjeti da se neki problemi u objektno-orijentiranom pristupu programiranju ne bi mogli riješiti da ne postoji ovakva mogućnost. U slučaju funkcija čiji je rezultat referenca, nakon završetka izvršavanja funkcije ne vrši se *prosta zamjena* poziva funkcije sa vraćenom *vrijednošću* (kao u slučaju kada rezultat nije referenca), nego se poziv funkcije *potpuno poistovjećuje* sa vraćenim *objektom* (koji u ovom slučaju mora biti promjenljiva, ili općenitije l-vrijednost). Pri tome, to poistovjećivanje ide dotle da i *sam poziv funkcije postaje l-vrijednost*, tako da se poziv funkcije čak može upotrijebiti sa lijeve strane operatora dodjele, ili prenijeti u funkciju čiji je formalni parametar referenca (obje ove radnje zahtijevaju l-vrijednosti).

Sve ovo može na prvi pogled djelovati dosta nejasno. Stoga ćemo vraćanje reference kao rezultata ilustrirati konkretnim primjerom. Posmatrajmo sljedeću funkciju, koja obavlja posve jednostavan zadatak (vraća kao rezultat veći od svoja dva parametra):

```
int VeciOd(int x, int y) {
  if(x > y) return x;
  else return y;
}
```

Pretpostavimo sada da imamo sljedeći poziv:

```
int a(5), b(8);
std::cout << VeciOd(a, b);
```

Ova sekvenca naredbi će, naravno, ispisati broj 8, s obzirom da će poziv funkcije "`VeciOd(a, b)`" po povratku iz funkcije biti zamijenjen izračunatom vrijednošću (koja očigledno iznosi 8, kao veća od dvije vrijednosti 5 i 8). Modificirajmo sada funkciju "`VeciOd`" tako da joj formalni parametri postanu reference:

```
int VeciOd(int &x, int &y) {
  if(x > y) return x;
  else return y;
}
```

Prethodna sekvenca naredbi koja poziva funkciju "`VeciOd`" i dalje radi ispravno, samo što se sada vrijednosti stvarnih parametara "`a`" i "`b`" ne *kopiraju* u formalne parametre "`x`" i "`y`", nego se formalni parametri "`x`" i "`y`" *poistovjećuju* sa stvarnim parametrima "`a`" i "`b`". U ovom konkretnom slučaju, krajnji efekat je isti. Ipak, ovom izmjenom smo ograničili upotrebu funkcije, jer pozivi poput

```
std::cout << VeciOd(5, 7);        // NEISPRAVNO: 5 i 7 nisu l-vrijednosti!
```

više nisu mogući, s obzirom da se reference ne mogu vezati za brojeve. Međutim, ova izmjena predstavlja korak do posljednje izmjene koju ćemo učiniti: modificiraćemo funkciju tako da kao rezultat *vraća referencu*:

```
int &VeciOd(int &x, int &y) {
  if(x > y) return x;
  else return y;
}
```

Ovako modificirana funkcija vraća kao rezultat *referencu na veći od svoja dva parametra*, odnosno sam poziv funkcije ponaša se kao da je on *upravo vraćeni objekat*, a ne samo njegova *vrijednost*. Na primjer, pri pozivu "`VeciOd(a, b)`" formalni parametar "`x`" se poistovjećuje sa promjenljivom "`a`", a formalni parametar "`y`" sa promjenljivom "`b`". Uz pretpostavku da je vrijednost promjenljive "`b`" veća od promjenljive "`a`" (kao u prethodnim primjerima poziva), funkcija će vratiti referencu na formalni parametar "`y`". Kako reference na reference ne postoje, biće zapravo vraćena referenca na onu promjenljivu koju referenca "`y`" predstavlja, odnosno promjenljivu "`b`". Kad kažemo da reference na reference ne postoje, mislimo na sljedeće: ukoliko imamo deklaracije poput

```
int &q(p);          // q je referenca na p
int &r(q);          // r je također referenca na p (a ne na referencu q)
```

tada "`r`" ne predstavlja referencu na referencu "`q`", već referencu na promjenljivu za koju je referenca "`q`" vezana, odnosno na promjenljivu "`p`" (odnosno "`r`" je također referenca na "`p`"). Dakle, funkcija je vratila referencu na promjenljivu "`b`", što znači da će se poziv "`VeciOd(a, b)`" ponašati upravo kao promjenljiva "`b`". To znači da postaje moguća ne samo upotreba ove funkcije kao obične vrijednosti, već je moguća njena upotreba u bilo kojem kontekstu u kojem se očekuje neka promjenljva (ili l-vrijednost). Tako su sada, na primjer, sasvim legalne konstrukcije poput sljedećih (ovdje su "`Udvostruci`" i "`Razmijeni`" funkcije iz ranijih primjera, samo što je funkcija "`Razmijeni`" modificirana da razmjenjuje *cjelobrojne vrijednosti*, a "`c`" je neka cjelobrojna promjenljiva):

```
VeciOd(a, b) = 10;
VeciOd(a, b)++;
VeciOd(a, b) += 3;
Udvostruci(VeciOd(a, b));
Razmijeni(VeciOd(a, b), c);
```

Posljednji primjer razmjenjuje onu od promjenljivih "`a`" i "`b`" čija je vrijednost veća sa promjenljivom "`c`". Drugim riječima, posljednja napisana verzija funkcije "`VeciOd`" ima tu osobinu da se poziv poput "`VeciOd(a, b)`" ponaša ne samo kao *vrijednost* veće od dvije promjenljive "`a`" i "`b`", nego se ponaša kao da je ovaj poziv *upravo ona promjenljiva* od ove dvije čija je vrijednost veća! Kako je poziv funkcije koja vraća referencu l-vrijednost, za njega se može vezati i referenca (da nije tako, ne bi bili dozvoljeni gore navedeni pozivi u kojima je poziv funkcije "`VeciOd`" iskorišten kao stvarni argument u funkcijama kod kojih je formalni parametar referenca). Stoga je sljedeća deklaracija posve legalna:

```
int &v(VeciOd(a, b));
```

Nakon ove deklaracije, referenca "`v`" ponaša se kao ona od promjenljivih "`a`" i "`b`" čija je vrijednost *bila veća u trenutku deklariranja ove reference*.

Vraćanje referenci kao rezultata funkcije treba koristiti samo u izuzetnim prilikama, i to sa dosta opreza, jer ova tehnika podliježe brojnim ograničenjima. Na prvom mjestu, jasno je da se prilikom vraćanja referenci kao rezultata iza naredbe "**`return`**" mora naći isključivo neka promjenljiva ili općenitije l-vrijednost, s obzirom da se reference mogu vezati samo za l-vrijednosti. Kompajler će prijaviti grešku ukoliko ne ispoštujemo ovo ograničenje. Mnogo opasniji problem je ukoliko vratimo kao rezultat referencu na neki objekat koji *prestaje živjeti nakon prestanka funkcije* (npr. na neku lokalnu promjenljivu, uključujući i formalne parametre funkcije koji nisu reference). Na primjer, zamislimo da smo funkciju "`VeciOd`" napisali ovako:

```
int &VeciOd(int x, int y) {
  if(x > y) return x;              // FATALNA GREŠKA: Vraćamo referencu na
  else return y;                   //   objekte koji prestaju postojati!!!
}
```

Ovdje funkcija i dalje vraća referencu na veći od parametara "`x`" i "`y`", međutim ovaj put ovi parametri nisu reference (tj. ne predstavljaju neke objekte koji postoje izvan ove funkcije) nego *samostalni objekti* koji imaju smisao samo unutar funkcije i koji se *uništavaju* nakon završetka funkcije. Drugim riječima, funkcija će vratiti referencu na *objekat koji je prestao postojati* (u smislu da je prostor u memoriji koji je objekat zauzimao sada raspoloživ za druge potrebe). Ovakva referenca naziva se *viseća referenca* (engl. *dangling reference*). Ukoliko za rezultat ove funkcije vežemo neku referencu, ona će se vezati za objekat koji zapravo

ne postoji (odnosno, za mjesto u memoriji kojem se ne bi smjelo pristupati)! Posljedice ovakvih akcija su potpuno nepredvidljive i često se završavaju potpunim krahom programa ili operativnog sistema. Viseće reference su na neki način analogne tzv. *divljim pokazivačima* (engl. *wild pointers*), odnosno pokazivačima koji su "odlutali" u nepredviđene dijelove memorije (jedino je sretna okolnost što divlje reference znatno rjeđe nastaju od divljih pokazivača). Dobri kompajleri mogu uočiti većinu situacija u kojima ste eventualno napravili viseću referencu i prijaviti upozorenje prilikom prevođenja, ali postoje i situacije koje ostaju nedetektirane od strane kompajlera, tako da dodatni oprez nije na odmet.

Pored običnih referenci, postoje i *reference na konstante objekte*. One se deklariraju isto kao i obične reference, uz dodatak ključne riječi "`const`" na početku. Reference na konstantne objekte se također vezuju za objekat kojim su inicijalizirane, ali se nakon toga ponašaju kao konstantni objekti, odnosno pomoću njih nije moguće *mijenjati* vrijednost vezanog objekta. Na primjer, neka su date sljedeće deklaracije:

```
int a(5);
const int &b(a);           // a se ne može mijenjati preko reference b...
```

Referenca "`b`" će se vezati za promjenljivu "`a`", tako da će pokušaj ispisa promjenljive "`b`" ispisati vrijednost 5, ali pokušaj promjene vezanog objekta pomoću naredbe "`b = 6`" dovešće do prijave greške. Naravno, promjenljiva "`a`" nije time postala konstanta: ona se i dalje može direktno mijenjati (recimo naredbom poput "`a = 6`"), ali ne i indirektno preko reference "`b`". Treba uočiti da se prethodne deklaracije bitno razlikuju od sljedećih deklaracija, u kojoj je "`b`" obična konstanta, a ne referenca na konstantni objekat:

```
int a(5);
const int b(a);            // b je konstantna KOPIJA od a...
```

Naime, u posljednjoj deklaraciji, ukoliko promjenljiva "`a`" promijeni vrijednost recimo na vrijednost 6, vrijednost konstante "`b`" i dalje ostaje 5. Sasvim drugačiju situaciju imamo ukoliko je "`b`" referenca na konstantni objekat: promjena vrijednosti promjenljive "`a`" ostavlja identičan efekat na referencu "`b`", s obzirom da je ona vezana na objekat "`a`" (ona je faktički *drugo ime* za promjenljivu "`a`"). Dakle, svaka promjena promjenljive "`a`" odražava se na referencu "`b`", mada se ona, tehnički gledano, ponaša kao konstantan objekat!

Postoji još jedna bitna razlika između prethodnih deklaracija. U slučaju kada "`b`" nije referenca, u nju se prilikom inicijalizacije kopira *čitav sadržaj* promjenljive "`a`", dok se u slučaju kada je "`b`" referenca, u nju kopira *samo adresa* mjesta u memoriji gdje se vrijednost promjenljive "`a`" čuva, tako da se pristup vrijednosti promjenljive "`a`" putem reference "`b`" obavlja *indirektno*. U slučaju da promjenljiva "`a`" nije nekog jednostavnog tipa poput "`int`", nego nekog masivnog tipa koji zauzima veliku količinu memorije (takvi tipovi su, recimo, tipovi "`vector`", "`deque`" i "`string`"), kopiranje čitavog sadržaja može biti zahtjevno, tako da upotreba referenci može znatno povećati efikasnost.

Kako se reference na konstantne objekte ne mogu iskoristiti za promjenu objekta za koji su vezane, omogućeno je da se reference na konstante mogu vezati i za konstante, brojeve, pa i proizvoljne izraze. Tako su, na primjer, sljedeća deklaracije sasvim legalne:

```
int a(5);
const int &b(3 * a + 1);       // Legalno, iako 3 * a + 1 nije l-vrijednost!
const int &c(10);              // Legalno, iako 10 nije l-vrijednost!
```

Tehnički, ovakve konstrukcije se interno izvode tako što se kreiraju pomoćne *bezimene promjenljive* koje se inicijaliziraju na navedenu vrijednost, a zatim se reference vezuju na te bezimene promjenljive. Drugim riječima, efekat prethodnih deklaracija je isti kao da smo imali nešto poput

```
int a(5);
int bezimena_1(3 * a + 1), bezimena_2(10);
const int &b(bezimena_1), &c(bezimena_2);
```

Također, reference na konstantne objekte mogu se vezati za objekat koji nije istog tipa, ali za koji postoji automatska pretvorba u tip koji odgovara referenci (i ovdje se automatski kreira bezimena promjenljiva na koju se referenca zaista veže). Na primjer:

```
int p(5);
const double &b(p);        // Legalno, iako p nije tipa int!
```

Činjenica da kad god se neka referenca na konstantni objekat inicijalizira nečim što nije l-vrijednost odgovarajućeg tipa dolazi do kreiranja pomoćne bezimene promjenljive na koju se referenca zaista veže, može ponekat imati prilično neočekivane efekte. Pogledajmo na primjer sljedeći programski isječak:

```
int a(5);
const int &b(a), &c(a + 0);
a++;
std::cout << b << " " << c << std::endl;       // Ispisuje 6 6 a ne 6 5!
```

Mada bi neko mogao pomisliti da je "a" i "a + 0" jedno te isto, samo je referenca "b" zaista vezana za promjenljivu "a", dok je referenca "c" vezana za bezimenu promjenljivu koja sadrži rezultat izračunavanja izraza "a + 0" (s obzirom da "a + 0" nije l-vrijednost). Stoga ovaj isječak neće ispisati "6 6", nego "6 5", suprotno očekivanjima.

Interesantna stvar nastaje ukoliko referencu na konstantni objekat upotrijebimo kao *formalni parametar funkcije*. Na primjer, pogledajmo sljedeću funkciju:

```
double Kvadrat(const double &x) {
  return x * x;
}
```

Kako se referenca na konstantni objekat može vezati za proizvoljan izraz (a ne samo za l-vrijednosti), sa ovako napisanom funkcijom pozivi poput

```
std::cout << Kvadrat(3 * a + 2);
```

postaju potpuno legalni. Praktički, funkcija "Kvadrat" može se koristiti na posve isti način kao da se koristi prenos parametara po vrijednosti (jedina formalna razlika je u činjenici da bi eventualni pokušaj promjene vrijednosti parametra "x" unutar funkcije "Kvadrat" doveo do prijave greške, zbog činjenice da je "x" referenca na konstantni objekat, što također znači i da ovakva funkcija ne može promijeniti vrijednost svog stvarnog parametra). Ipak, postoji suštinska tehnička razlika u tome šta se zaista interno dešava u slučaju kada se ova funkcija pozove. U slučaju da kao stvarni parametar upotrijebimo neku l-vrijednost (npr. promjenljivu) kao u pozivu

```
std::cout << Kvadrat(a);
```

dešavaju se iste stvari kao pri klasičnom prenosu po referenci: formalni parametar "x" se poistovjećuje sa promjenljivom "a", što se ostvaruje prenosom adrese. Dakle, ne dolazi ni do kakvog kopiranja vrijednosti. U slučaju kada stvarni parametar nije l-vrijednost (što nije dozvoljeno kod običnog prenosa po referenci), automatski se kreira privremena bezimena promjenljiva koja se inicijalizira stvarnim parametrom, koja se zatim klasično prenosi po referenci, i uništava čim se poziv funkcije završi (činjenica da će ona biti uništena ne smeta, jer njena vrijednost svakako nije mogla biti promijenjena putem reference na konstantu). Drugim riječima, poziv

```
std::cout << Kvadrat(3 * a + 2);
```

načelno je ekvivalentan pozivu

```
{
  const double privremena_bezimena(3 * a + 2);
  std::cout << Kvadrat(privremena_bezimena);
}
```

Formalni parametri koji su reference na konstantu koriste se uglavnom kod rada sa parametrima masivnih tipova, što ćemo obilato koristiti u kasnijim izlaganjima. Naime, prilikom prenosa po vrijednosti uvijek dolazi do *kopiranja stvarnog parametra u formalni*, što je neefikasno za slučaj masivnih objekata. Kod prenosa po referenci do ovog kopiranja ne dolazi, a upotreba reference na konstantu dozvoljava da kao stvarni argument upotrijebimo proizvoljan izraz (slično kao pri prenosa po vrijednosti). Stoga, u slučaju masivnih objekata, prenos po vrijednosti treba koristiti samo ukoliko zaista želimo da formalni parametar bude kopija stvarnog parametra (npr. ukoliko je unutar funkcije neophodno mijenjati vrijednost formalnog parametra, a ne želimo da se to odrazi na vrijednost stvarnog parametra). Kao konkretan primjer, razmotrimo funkcije "Prosjek", "ZbirVektora" i "Sastavi" koje su rađene na prethodnim predavanjima. Ove funkcije mogu se učiniti *mnogo efikasnijim* ukoliko se formalni parametri ovih funkcija deklariraju kao *reference na konstantne objekte*. Na taj način *neće dolaziti do kopiranja masivnih objekata* prilikom prenosa u funkcije, što je naročito

bitno u slučaju da se kao stvarni parametri upotrijebe vektori ili stringovi sa velikim brojem elemenata. Strogo rečeno, kopiranje parametara koji nisu reference odvija se posredstvom tzv. *konstruktora kopije*, tako da stepen poboljšanja u efikasnosti bitno ovisi od toga kako su izvedeni konstruktori kopije tipova "`vector`" i "`string`". Uglavnom, kopiranje svakako treba izbjeći (ako nije zaista neophodno), tako da ove funkcije treba obavezno izvesti na sljedeći način:

```
double Prosjek(const std::vector<int> &v) {
  double suma(0);
  for(double x : v) suma += x;
  return suma / v.size();
}

std::vector<int> ZbirVektora(const std::vector<int> &a,
  const std::vector<int> &b) {
    std::vector<int> c;
    for(int i = 0; i < a.size(); i++) c.push_back(a[i] + b[i]);
    return c;
}

std::string Sastavi(const std::string &s1, const std::string &s2) {
  std::string s3;
  for(int i = 0; i < s1.length(); i++) s3.push_back(s1[i]);
  for(int i = 0; i < s2.length(); i++) s3.push_back(s2[i]);
  return s3;
}
```

Slične modifikacije bi trebalo uraditi u funkcijama za rad sa matricama koje su prezentirane na prethodnom poglavlju, što možete uraditi kao korisnu vježbu. Poboljšanje efikasnosti u tom primjeru može biti drastično, pogotovo pri radu sa velikim matricama. U svim daljim izlaganjima *uvijek* ćemo koristiti reference na konstantne objekte kao formalne parametre kad god su parametri masivnih tipova, osim u slučaju kada postoji jak razlog da to ne činimo (što će biti posebno naglašeno).

Reference treba koristiti i u rangovskim *for*-petljama kad god elementi kolekcije kroz koju se krećemo predstavljaju masivne tipove podataka (kao što su stringovi, ili ponovo vektori, što je situacija koju imamo kada matrice simuliramo preko vektora čiji su elementi vektori), sa ciljem da se izbjegnu stalna kopiranja elemenata kolekcije u upravljačku promjenljivu petlje, što može drastično poboljšati efikasnost. Pri tome, ukoliko nije potrebno mijenjati elemente kolekcije, poželjno je referencu označiti kvalifikatorom "**`const`**", što će spriječiti da neko nehotično proba izmijeniti elemente kolekcije. Tako bi se, recimo funkcija za ispis elemenata matrice organizirane kao vektor vektôrā realnih brojeva uz pomoć rangovske *for*-petlje trebala napisati recimo ovako (naravno, uvijek možemo "**`typedef`**" deklaracijom uvesti alternativno ime za glomaznu konstrukciju "`std::vector<std::vector<double>>`"):

```
void IspisiMatricu(const std::vector<std::vector<double>> &m) {
  for(const std::vector<double> &red : m) {
    for(double x : red) std::cout << std::setw(10) << x;
    std::cout << std::endl;
  }
}
```

Uz pomoć "**`auto`**" deklaracija, sve se može uraditi jednostavnije:

```
void IspisiMatricu(const std::vector<std::vector<double>> &m) {
  for(auto &red : m) {
    for(auto x : red) std::cout << std::setw(10) << x;
    std::cout << std::endl;
  }
}
```

Izlaganje o referencama završićemo navođenjem još jednog detalja. Rečeno je da reference na reference *ne postoje*. Kada bi postojale, sintaksa za njihovo deklariranje trebala bi da izgleda sa dva znaka "`&`" ispred imena promjenljive (npr. "**`int`** `&&ref`"), po analogiji sa činjenicom da se pokazivači na pokazivače (koji *postoje*) deklariraju konstrukcijom koja koristi dvije zvjezdice ispred imena promjenljive (npr. deklaracijom poput "**`int`** `**pok_na_pok`"). Međutim, interesantno je da *upravo ovakve deklaracije sa dva znaka* "`&`" *prolaze u varijanti* C++11, iako ne prolaze u starijim verzijama C++-a. One ne samo da prolaze, nego se intenzivno viđaju u mnogobrojnim člancima koji opisuju specifičnosti varijante C++11, te u novijim programima koji se oslanjaju na specifičnosti ove varijante. Stoga je potrebno reći barem u najosnovnijim crtama o čemu je riječ.

Činjenica da su u C++11 postale legalne deklaracije poput "`int &&ref`" ne znači da je C++11 uveo reference na reference. Naprotiv, oznaka "`&&`" ispred imena promjenljive označava jednu *posebnu novu vrstu referenci* koje su uvedene u C++11, a zovu se *r-vrijednosne reference*. Te reference se, upravno suprotno običnim referencama, mogu vezati *isključivo za nešto što nije l-vrijednost* (pri čemu je smisao takvog vezivanja da se tada kreira bezimena promjenljiva koja se inicijalizira zadanim sadržajem, a onda se referenca vezuje za tu bezimenu promjenljivu). Na prvi pogled je nejasno zbog čega su uopće takve reference uvedene i čemu bi mogle poslužiti. Ovdje nećemo ulaziti u detalje, samo ćemo reći da se zahvaljujući takvim referencama mogu napraviti funkcije koje mogu prepoznati kada im je kao parametar poslana l-vrijednost, a kada nije, i koje na osnovu toga mogu odabrati različite načine obrade parametra u ovisnosti od toga da li je parametar l-vrijednost ili nije (s obzirom da su neki načini obrade povoljniji za l-vrijednosti nego za ono što nije l-vrijednost, dok za neke druge načine obrade vrijedi suprotno). Primjer kako se to može ostvariti demonstriraćemo u kasnijim predavanjima. Između ostalog, r-vrijednosne reference su se pokazale kao neophodne da se ostvari podrška za *move-semantiku*, što je jedan od glavnih razloga za njihovo uvođenje. Kada budemo govorili o kreiranju vlastitih tipova podataka, pokazaćemo u najkraćim crtama kako se r-vrijednosne reference mogu iskoristiti da se podrži move-semantika za tip koji kreiramo. Osim na tom mjestu, r-vrijednosnim referencama se nećemo detaljnije baviti, jer one spadaju u vrlo naprednu tematiku.