Dodatak predavanjima: Sve što treba znati o pokazivačima

Bez obzira na činjenicu da je C++ uveo novu vrstu objekata nazvanu *reference*, koja umnogome rješava brojne probleme vezane za nerazumijevanje i pogrešnu upotrebu pokazivača, pokazivači se u jeziku C++ još uvijek intenzivno koriste, s obzirom da se reference u jeziku C++ nisu mogle izvesti na način koji bi u potpunosti uklonio potrebu za pokazivačima, kao što je to učinjeno u čistim objektno orijentirani, jezicima (npr. u *Javi*). Naime, činjenica da se od jezika C++ zahtijevala visoka kompatibilnost sa jezikom C, dovela je do toga da je "bijeg" od pokazivača u jeziku C++ ipak nemoguć. S obzirom da je nerazumjevanje koncepta pokazivača čest razlog za brojne probleme u razumijevanju naprednijih koncepata jezika C++, svrha ovog dodatka je da pruži pregled onih znanja o pokazivačima koje je neophodno posjedovati za ispravno razumijevanje predavanja koja kasnije slijede. Većinu tih znanja studenti su trebali steći kroz prethodne kurseve programiranja, ali pitanje je da li su ih zaista stekli, zbog dosta složene materije. To je i glavni razlog zbog koje je ovaj dodatak uopće pisan.

Pokazivači su jedna od dvije vrste objekata u jeziku C++ pomoću kojih se može vršiti *indirektni pristup* drugim objektima (druga vrsta su *reference*). Pokazivači su u C++ direktno naslijeđeni iz jezika C, u kojem predstavljaju *jedine objekte* preko kojih je moguć indirektni pristup drugim objektima. U suštini, pokazivači nisu ništa drugo nego promjenljive koje kao svoj sadržaj sadrže *adresu nekog drugog objekta* (kaže se "pokazuju na neki objekat"), podržane odgovarajućim operatorima koji omogućavaju da se *pristupi objektu na koji pokazuju*. Pokazivači, dakle, nisu ništa komplicirano. Najveći problem kod početnika je što se često pobrka sam *pokazivač* (engl. *pointer*), sa onim *na šta on u tom trenutku pokazuje* (engl. *pointee*). Dodatni problem (i to ne samo kod početnika) je to što pokazivači pružaju izuzetno veliku fleksibilnost, koju je veoma lako *zloupotrebiti*. Zbog tog razloga se programerima u jeziku C++ savjetuje da dobro prouče reference, i da sve probleme koji se mogu rješavati pomoću referenci zaista i *rješavaju pomoću referenci*, a ne pomoću pokazivača. Bez obzira na to, u jeziku C++ i dalje postoji mnogo stvari koje je moguće uraditi samo pomoću pokazivača, a koje je nemoguće realizirati niti pomoću referenci, niti na bilo koji drugi način (koji ne koristi pokazivače).

Pokazivače nije moguće objasniti i razumjeti bez nešto detaljnijeg objašnjenja o tome *kako se promjenljive čuvaju u memoriji računara* (između ostalog i zbog toga što kod pokazivača možemo *direktno pristupiti* informaciji o tome gdje se u memoriji čuva objekat na koji pokazivač pokazuje). Zbog toga ćemo prvo u osnovnim crtama razmotriti način smještanja promjenljivih u računarskoj memoriji. Poznato je da svaka promjenljiva koja se deklarira u programu tokom svog života zauzima određeni prostor u memoriji. Mjesto u memoriji koje je dodijeljeno nekoj promjenljivoj određeno je njenom *adresom*. Adresa promjenljive je zapravo neka informacija koja na *jednoznačan način* određuje mjesto u memoriji gdje se ta promjenljiva nalazi, i ona je *fiksna* sve dok ta promjenljiva postoji. Ta informacija je kod većine računarskih arhitektura *brojčane prirode* (tj. adrese su obično *neki brojevi*), mada *ne mora nužno da bude*. Recimo, kod najvećeg broja računarskih arhitektura memorija je organizovana kao *jednodimenzionalni niz* memorijskih ćelija (registara), i tada je adresa prosto *redni broj* (*indeks*) odgovarajuće memorijske ćelije u kojoj se čuva sadržaj promjenljive (ili prve od više takvih ćelija u slučaju da sadržaj promjenljive ne može stati u jednu memorijsku ćeliju). Međutim, sasvim su moguće (i postoje) drugačije memorijske arhitekture kod kojih adresa *nije broj*. Recimo, postoje memorijski modeli koji nisu organizirani kao jednodimenzionalni već kao *dvodimenzionalni niz* (matrica) memorijskih ćelija, tako da je za određivanje pozicije neke memorijske ćelije potrebno poznavati *red* i *kolonu* u kojoj se ta ćelija nalazi, tako da kod takvih memorijskih modela adresa nije broj nego *par brojeva* (red i kolona). Na primjer, takav memorijski model javlja se kod prvih generacija PC računara sa starijim modelima Intel-ovih procesora poput 8086, kod kojih su adrese zaista predstavljeni kao parovi brojeva (koji se redom nazivaju *segment* i *offset*). Štaviše, postoje i koncepti *asocijativnih memorija* kod kojih je svakoj memorijskoj ćeliji pridružen neki *jednoznačni identifikator* (*tag*), koji uopće ne mora biti numeričke prirode, i koji onda preuzima ulogu *adrese* te ćelije. Čak i ako se ograničimo isključivo na računarske arhitekture kod kojih adrese memorijskih ćelija zaista jesu brojevi, prilikom rada sa pokazivačima mogu nastati velike greške u reznonovanju ukoliko o adresama razmišljamo kao običnim brojevima. Zbog toga adrese *ne treba posmatrati kao brojeve*, nego isključivo kao *informacije koje jednoznačno određuju lokaciju nekog objekta u memoriji*, pri čemu je *tačna priroda te informacije manje-više posve nebitna*.

Pretpostavimo, na primjer, da imamo deklaraciju neke klasične (recimo cjelobrojne) promjenljive, poput deklaracije

```
int broj;
```

Prilikom nailaska na ovu deklaraciju, rezervira se odgovarajući prostor u memoriji za potrebe smještanja sadržaja promjenljive "broj", koji naravno ima svoju adresu (koju ćemo nazvati prosto *adresa promjenljive* "broj"). To ćemo predstaviti sljedećom memorijskom slikom:

Rezervirani memorijski prostor smo predstavili pravougaonikom, dok upitnici označavaju da *sadržaj* rezerviranog memorijskog prostora *nije definiran*, odnosno u njemu se nalazi isti sadržaj koji se u tom dijelu memorije nalazio *od ranije*, tj. prije izvršene rezervacije (zbog toga je i početna vrijednst promjenljive "broj" nedefinirana). Ukoliko nakon ove deklaracije izvršimo dodjelu neke vrijednosti promjenljivoj "broj", npr. dodjelom poput

```
broj = 56;
```

efekat ove dodjele biće smještanje neke kodirane reprezentacije broja 56 u rezervirani memorijski prostor (za reprezentaciju cijelih brojeva najčešće se koristi prosti binarni kôd, ali sam način reprezentacije nije bitan za izlaganja koja slijede). To ćemo predstaviti sljedećom slikom:

Da smo prilikom deklaracije odmah izvršili i *inicijalizaciju* promjenljive, npr. deklaracijom poput

```
int broj(56);                                   // ili int broj = 56; u C stilu
```

ovakvu memorijsku sliku bismo imali *odmah po stvaranju promjenljive*, odnosno istovremeno sa rezervacijom memorijskog prostora za pamćenje sadržaja promjenljive "broj" bilo bi izvršeno upisivanje odgovarajućeg sadržaja u rezervirani prostor.

Razumije se da za pristup promjenljivoj "broj" nije uopće potrebno poznavati na kojoj se adresi u memoriji ona nalazi, jer njenom sadržaju uvijek možemo pristupiti koristeći njeno simboličko ime "broj" (kao što smo uvijek i činili). Međutim, pokazivači su specijalne vrste promjenljivih koje kao svoj *sadržaj* imaju *adresu neke druge promjenljive* (ili općenito, neke druge l-vrijednosti) i u takve promjenljive se *ne može upisivati ništa drugo osim adrese*. Pokazivačke promjenljive se deklariraju tako da se ispred njihovog imena stavi znak "*" (zvjezdica). Na primjer, sljedeća deklaracija deklarira promjenljivu "pok" koja nije tipa "cijeli broj" (tj. "int"), nego "pokazivač na cijeli broj" (ovaj tip se označava kao "int *"):

```
int *pok;
```

Kao i svaka druga promjenljiva, i pokazivačka promjenljiva zauzima određeni prostor u memoriji (te i ona ima svoju adresu). Pretpostavimo li da su na prethodno opisani način deklarirane cjelobrojna promjenljiva "broj" i pokazivačka promjenljiva "pok", to možemo predstaviti sljedećom memorijskom slikom:

Upitnici unutar promjenljive "pok" govore da prethodna deklaracija nije postavila nikakav inicijalni sadržaj u prostor rezerviran za potrebe te promjenljive, nego je on onakav kakav se prosto zatekao odranije u tom prostoru. Tako će ostati sve dok ovoj promjenljivoj ne dodijelimo neku vrijednost. Međutim, pokazivačke promjenljive namijenjene su da pamte *isključivo adrese*, i njima se *ne smiju neposredno dodjeljivati brojčane vrijednosti* (razlozi za to su brojni, a jedan od njih, mada ne i najbitniji, je i to što adrese ne moraju nužno biti brojevi). Stoga kompajler *neće dozvoliti* dodjelu poput sljedeće:

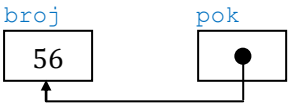
```
pok = 4325;                                     // OVO NIJE DOZVOLJENO!
```

Umjesto toga, pokazivačkim promjenljivim se mogu dodjeljivati samo *vrijednosti pokazivačke prirode* (tj. vrijednosti koje *garantirano predstavljaju adrese nekih drugih objekata*). Na primjer, pokazivaču je moguće dodjeliti adresu nekog objekta (npr. promjenljive) uz pomoć unarnog prefiksnog operatora "&",

koji možemo čitati "adresa od". Ovaj operator (koji se još naziva i *operator* referenciranja) može se primjenjivati samo na operande *koji imaju adrese* (tj. na *l-vrijednosti*), kao što su recimo *promjenljive*, ali ne i na recimo brojeve ili proizvoljne izraze (tako je, na primjer, konstrukcija poput "`&broj`" smislena, ali su konstrukcije poput "`&15`" ili "`&(broj + 2)`" besmislene). Stoga, ukoliko izvršimo dodjelu poput

```
pok = &broj;
```

u promjenljvu "`pok`" će se smjestiti *adresa promjenljive* "`broj`" (način kako je ta adresa zaista predstavljena posve je nebitan). Situacija u memoriji sada odgovara sljedećoj slici:



Na ovoj slici, da bismo izbjegli sve eventualne špekulacije o tome kako je adresa zaista predstavljena u pokazivačkoj promjenljivoj, tu adresu prosto prikazali kao kružić od kojeg vodi strelica ka objektu koji odgovara toj adresi. Kaže se da sada pokazivačka promjenljiva (engl. *pointer*) "`pok`" *pokazuje* (engl. *points to*) na promjenljivu "`broj`", za koju ćemo reći da je *pokazani objekat* (engl. *pointee*) u nedostatku boljeg prevoda za ovu englesku riječ.

Korištenje pokazivačkih promjenljivih u aritmetičkim izrazima je prilično ograničeno (npr. izraz poput "`3 * pok + 2`" nije dozvoljen), a čak i u kontekstima u kojima je dozvoljeno njihovo korištenje u aritmetičkim izrazima, pravila izvođenja računskih operacija (aritmetike) sa pokazivačima razlikuju se od klasičnih pravila aritmetike sa brojevima (još jedan razlog da nije dobro doživljavati adrese kao brojeve), o čemu će kasnije biti detaljno govora. U svakom slučaju, svaki direktni pristup promjenljivoj "`pok`" odnosi se na *adresu koja je u njoj pohranjena*, a ne na promjenljivu na koju ona pokazuje. Po tome se pokazivači bitno razlikuju od *referenci*, s obzirom da se svaki pristup referenci *automatski preusmjerava* na objekat na koji referenca ukazuje (odnosno, kod pokazivača ovog automatskog preusmjeravanja nema).

Sam sadržaj pokazivačke promjenljive nam obično nije od neposredne koristi, nego nam je važnije da *pristupimo objektu na koji ona pokazuje*. Za tu svrhu koristi se posebna operacija, koja se obično naziva *dereferenciranje*, a postiže se uz pomoć unarnog operatora "`*`". Ovaj operator primjenjuje se na pokazivačke promjenljive, ili općenitije na pokazivačke izraze (tj. izraze čiji su rezultati *adrese*, o kojima ćemo govoriti nešto kasnije), a možemo ga čitati kao "ono na šta pokazuje ...". Recimo, ako je "`pok`" neka pokazivačka promjenljiva, izraz "`*pok`" možemo čitati kao "ono na šta pokazuje `pok`". Tako će, ukoliko se prethodno izvrše naredbe iz gore napisanih primjera, naredba

```
std::cout << *pok;                  // Ispisuje sadržaj onoga na šta pok pokazuje
```

ispisaće vrijednost "56", jer promjenljiva "`pok`" trenutno pokazuje na promjenljivu "`broj`", tako da je u ovom trenutku "`*pok`" isto što i promjenljiva "`broj`".

Dereferencirani pokazivač može se koristiti u bilo kojem kontekstu u kojem bi se mogao koristiti bilo koji izraz čiji tip odgovara tipu na koji pokazivač pokazuje. Stoga je sljedeća naredba potpuno legalna, i ispisuje brojeve 57 i 112:

```
std::cout << *pok + 1 << std::endl << 2 * *pok;
```

U ovoj naredbi trebamo obratiti pažnju na dva detalja. Prvo, operator dereferenciranja ima *veći prioritet* od aritmetičkih operatora (sabiranja, množenja, itd.) tako da se izraz poput "`*pok + 1`" ne interpretira kao "`*(pok + 1)`" već kao "`(*pok) + 1`" (kasnije ćemo vidjeti kakvo bi tačno značenje imala prva interpretacija, koja je također smislena). Drugo, u izrazu "`2 * *pok`" prva zvjezdica predstavlja operator množenja, dok druga zvjezdica predstavlja operator dereferenciranja (odnosno, postoje dva različita operatora istog imena "`*`", pri čemu je jedan binarni a drugi unarni). Ova činjenica može dovesti do zbrke ukoliko prilikom pisanja izraza u kojima se javljaju dereferencirani pokazivači *izostavimo razmake*, odnosno ukoliko prethodnu naredbu napišemo na sljedeći način:

```
std::cout<<*pok+1<<std::endl<<2**pok;    // LEGALNO, ALI NEMOJTE PISATI OVAKO!!!
```

Međutim, na ovaj način je veoma nejasno šta koja zvjezdica predstavlja, tako da takvo pisanje treba izbjegavati. Vjerovatno najjasnije značenje dobijamo ukoliko upotrijebimo i dodatne zagrade:

```
std::cout << *pok + 1 << std::endl << 2 * (*pok);
```

Veoma važna činjenica je da dereferencirani pokazivač *ima status promjenljive*, tačnije *status l-vrijednosti*, tako da se može naći i sa *lijeve strane znaka jednakosti*. Štaviše, na dereferencirane pokazivače mogu se primjenjivati i sve druge operacije koje se normalno mogu primjenjivati samo nad promjenljivim (i drugim l-vrijednostima) kao što su recimo "++" i "--", tako da su sasvim dozvoljene naredbe poput sljedećih:

```
*pok = 100;                 // Smješta 100 u ono na šta pokazuje pok
(*pok)++;                   // Uvećava sadržaj onoga na šta pokazuje pok za 1
```

Prva naredba postavlja sadržaj memorijske lokacije na koju ukazuje pokazivačka promjenljiva "pok" na vrijednost 100, dok druga naredba uvećava sadržaj memorijske lokacije na koju ukazuje pokazivačka promjenljiva "pok" za jedinicu. U konkretnom primjeru, kada pokazivačka promjenljiva "pok" pokazuje na promjenljivu "broj", efekat ovih naredbi je isti kao da smo neposredno pisali

```
broj = 100;                 // *pok je ovdje faktički promjenljiva "broj"
broj++;
```

Drugim riječima, *dereferencirani pokazivač koji pokazuje na neku promjenljivu ponaša se identično kao i promjenljiva na koju pokazuje*. Treba još napomenuti da su u izrazu "(*pok)++" zagrade *obavezne*. Naime, operator "++" ima *veći prioritet* u odnosu na operator dereferenciranja. Stoga bi se izraz "*pok++" interpretirao kao "*(pok++)", koji je također smislen. Uskoro ćemo razjasniti i smisao ove druge interpretacije.

Operatori referenciranja "&" i dereferenciranja "*" su *međusobno inverzni*. Ako je "x" objekat tipa "NekiTip", tada izraz "&x" ima tip "pokazivač na tip NekiTip", a izraz "*&x" svodi se prosto na "x". S druge strane, ukoliko je "pok" pokazivač tipa "pokazivač na tip NekiTip", tada je tip izraza "*pok" prosto "NekiTip", a izraz "&*pok" svodi se na "pok".

Veoma je bitno istaći da znak "*" upotrijebljen ispred imena promjenljive ima *potpuno drugačije značenje prlikom deklaracije promjenljive* i *prilikom njene upotrebe*. Nerazumijevanje ove činjenice uzrok je velikih zbrki kod početnika. Zaista, prilikom deklaracije promjenljive, zvjezdica ispred njenog imena (npr. u deklaraciji poput "**int** *pok;") označava da se radi o pokazivačkoj a ne o običnoj promjenljivoj, dok prilikom upotrebe (pokazivačke) promjenljive zvjezdica ispred njenog imena (npr. u izrazu poput "std::cout << *pok") označava da ne želimo pristupiti toj (pokazivačkoj) promjenljivoj nego onome na šta ona pokazuje. Ovo su očito dva posve različita značenja (da zbrka bude još veća, znamo da u nekim kontekstima zvjezdica može označavati i *množenje*). Ipak, postoji jedan način kako se deklaracije pokazivačkih promjenljivih mogu čitati tako da se ukloni razlika između gore pomenuta dva značenja znaka "*". Naime, umjesto da deklaraciju oblika

```
int *pok;
```

čitamo kao "pok je tipa pokazivač na cijeli broj", možemo je čitati kao "*pok je tipa cijeli broj", odnosno "ono na šta pokazuje pok je tipa cijeli broj". Uz ovakvu interpretaciju, tumačenje znaka "*" upotrijebljenog unutar deklaracija i unutar proizvoljnih izraza postaje konzistentnije, odnosno fraza poput "*pok" može se uvijek čitati kao "ono na šta pokazuje pok", bez obzira što se u deklaracijama pokazivačka promjenljiva ("pok" u ovom primjeru) tek uvodi. Nije na odmet napomenuti ni da znak "&" nema isto značenje u svim kontekstima. Upotrijebljen u proizvoljnim izrazima ispred imena promjenljive, on označava adresu te promjenljive. S druge strane, prilkom deklaracije promjenljive, isti znak upotrijebljen njenog imena označava da se *ne radi o običnoj promjenljivoj*, nego o tzv. *referenci* (što je posve drugo značenje). Konačno, u nekim kontekstima isti znak može označavati i *operaciju konjunkcije po bitima* (ukoliko se upotrijebi kao binarna operacija između dva cjelobrojna operanda).

Kao što svaka promjenljiva (osim ako je označena kvalifikatorom "**const**") može tokom svog života mijenjati svoj sadržaj, tako i pokazivačke promjenljive mogu mijenjati svoj sadržaj, što zapravo znači da tokom života mogu *pokazivati na različite objekte*. Ovo ilustrira sljedeći primjer:

```
int broj1(5), broj2(8);
int *pok;
pok = &broj1;                           // *pok je sada isto što i broj1
std::cout << *pok << std::endl;         // Ispisuje "5"
pok = &broj2;                           // *pok je sada isto što i broj2
std::cout << *pok << std::endl;         // Ispisuje "8"
```

Jednom pokazivaču se može dodijeliti drugi pokazivač samo ukoliko *oba pokazuju na posve isti tip* (uz neke iznimke vezane za tzv. nasljeđivanje, na koje ćemo ukazati kada se za to ukaže potreba). Na primjer, ukoliko imamo deklaraciju

```
int *pok1, *pok2;
```

tada je dodjela "`pok1 = pok2`" posve legalna, i nakon nje "`pok1`" i "`pok2`" pokazuju na *istu lokaciju u memoriji*, i to onu na koju je prije dodjele pokazivao pokazivač "`pok2`" (da bi ova dodjela imala smisla, prethodno je potrebno pokazivaču "`pok2`" dodijeliti adresu nekog drugog objekta). Drugim riječima, nakon ove dodjele, "`*pok1`" i "`*pok2`" ne samo da imaju iste vrijednosti, nego su oni *jedan te isti objekat* (inače, pojava da se istom objektu može pristupiti na više različitih načina u programiranju se naziva *aliasing*). Slično vrijedi za dodjelu "`pok2 = pok1`". Sljedeći primjer ilustrira pojavu aliasinga:

```
int broj;
int *pok1, *pok2;
pok1 = &broj1;                          // *pok1 je sada isto što i broj
pok2 = pok1;                            // *pok2 je sada također isto što i broj
*pok1 = 8;                              // U promjenljivu broj se smješta 8
std::cout << *pok1 << *pok2;            // Ispisuje dvije osmice...
*pok1 = 5;                              // U promjenljivu broj se smješta 5
std::cout << *pok1 << *pok2;            // Ispisuje dvije petice...
```

Kao što je već rečeno gore, pokazivači koji ne pokazuju na objekte istih tipova *ne mogu se međusobno dodjeljivati*. Recimo, ukoliko imamo deklaracije

```
int *pok1;
double *pok2;
```

tada *nije dozvoljena* niti dodjela "`pok1 = pok2`" niti dodjela "`pok2 = pok1`". Međutim, bitno je razlikovati ove dodjele od dodjela "`*pok1 = *pok2`" i "`*pok2 = *pok1`" koje su *potpuno legalne*, s obzirom da se dereferencirani pokazivači "`pok1`" i "`pok2`" ponašaju kao cjelobrojna odnosno realna promjenljiva, a međusobne dodjele između cjelobrojnih i realnih promjenljivih su dozvoljene (uz automatsku konverziju tipa).

S obzirom da se u jeziku C++ razmaci u tekstu programu mogu po volji ubacivati i izbacivati sve dok to ne dovede do promjene smisla programa (npr. ne smijemo izbaciti razmak između "`int`" i "`a`" u konstrukciji "`int a`", jer bismo u suprotnom dobili riječ "`inta`" koja nema nikakvog smisla u C++-u), pokazivačke promjenljive se mogu *stilski* deklarirati na nekoliko različitih načina sa aspekta *gdje pisati razmake*, pro čemu do danas *ne postoji općeprihvaćeni konsenzus* koji je stil najbolji. Recimo, neki stilovi deklaracije promjenljive "`pok`" koja je tipa "pokazivač na cijeli broj" su sljedeći:

```
int*pok;                // Nepregledan stil...
int* pok;               // Mnogi danas preporučuju ovaj stil...
int *pok;               // Međutim, izgleda da je ovaj klasični stil bolji...
int * pok;              // Ovaj stil nesretno asocira na množenje...
```

Od ova četiri stila, rijetko ko će zagovarati prvi ili četvrti stil. S druge strane, intenzivne su prepirke oko toga da li je bolji drugi ili treći stil. U literaturi (pogotovo onoj posvećenoj C++-u a ne C-u) se često zagovara drugi stil, uz objašnjenje da u deklaracijama oznaka "`*`" bolje pristaje *uz oznaku tipa*, odnosno treba je tumačiti kao *sastavni dio oznake tipa* (u skladu sa tumačenjem "`pok` je tipa pokazivač na cijeli broj"). Mada ovakvo tumačenje nije bez osnova, ono ima jedan ozbiljan nedostatak. Naime, pogledajmo sljedeću deklaraciju:

```
int* a, b;                              // Kojeg su tipa "a" i "b"?
```

Uz prethodno tumačenje, moglo bi se pomisliti da su u ovoj deklaraciji i "`a`" i "`b`" pokazivačke promjenljive (tj. da su "`a`" i "`b`" tipa pokazivač na cijeli broj). Međutim, istina je da je *samo* "`a`" *pokazivačka promjenljiva*, dok je "`b`" *obična cjelobrojna promjenljiva* (tipa "`int`"). Autor ovih materijala smatra da je manje zbunjujuće ukoliko se ista deklaracija napiše kao

```
int *a, b;              // Ovdje se jasnije vidi da je samo "a" pokazivač...
```

i ista interpretira kao "`*a` i `b` su tipa cijeli broj". Zbog toga će se u ovim materijalima koristiti ovaj (stariji) stil pisanja deklaracija pokazivačkih promjenljivih, bez obzira što se u posljednje vrijeme intenzivno propagira drugi stil.

Prije nego što se upoznamo sa primjenama pokazivača, bitno je napomenuti da bez obzira što pokazivači i dalje imaju veliku primjenu u jeziku C++, oni se mnogo manje koriste nego što se koriste u jeziku C. Naime, u jeziku C su pokazivači *jedini način da se ostvari indirektni pristup nekoj promjenljivoj*, dok se ta indirekcija u jeziku C++ često može ostvariti znatno jednostavnije putem tzv. *referenci*. Recimo, u jeziku C, ne postoji nikakav način da funkcija *promijeni vrijednosti svojih stvarnih parametara*. Tako, ukoliko na primjer želimo napisati funkciju "`Razmijeni`" koja *razmjenjuje sadržaje* promjenljivih koje joj se prenose kao argumenti, u C-u *ne postoji nikakav način* da to izvedemo tako da poziv funkcije izgleda poput sljedećeg:

```
Razmijeni(a, b);            // Ovo je u C-u nemoguće postići
```

Naime, kako god bila napisana funkcija "`Razmijeni`", ona uvijek radi sa *kopijom* stvarnih argumenata "`a`" i "`b`", tako da ih ona ne može izmijeniti (napomenimo da je, zahvaljujući referencama, u jeziku C++ moguće postići da se koristi *upravo ovakav poziv*). U jeziku C jedino što možemo uraditi je da funkciji "`Razmijeni`" *eksplicitno prenesemo adrese* gdje se nalaze odgovarajući stvarni parametri, odnosno u pozivu funkcije moramo eksplicitno uzeti adrese stvarnih parametara pomoću adresnog operatora "`&`". To zahtijeva da poziv funkcije izgleda poput sljedećeg:

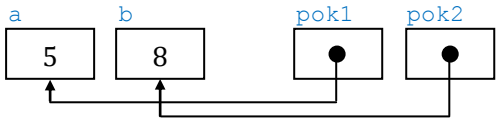
```
Razmijeni(&a, &b);
```

Da bi ovakav poziv bio sintaksno ispravan, potrebno je funkciju "`Razmijeni`" napisati tako da umjesto cijelih brojeva kao parametre prihvata *pokazivače na cijele brojeve*. Tada je moguće upotrijebiti operator dereferenciranja da se pomoću njega *indirektno* pristupi sadržaju memorijskih lokacija gdje se nalaze stvarni parametri i da se na taj načia izvrši razmjena. Na taj način funkcija "`Razmijeni`", napisana u duhu jezika C, mogla bi izgledati ovako:

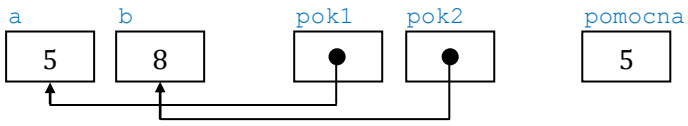
```
void Razmijeni(int *pok1, int *pok2) {
  int pomocna(*pok1);
  *pok1 = *pok2;
  *pok2 = pomocna;
}
```

Mada napisana funkcija izgleda dosta jednostavno, početniku nije posve lako shvatiti kako ona radi. Neka su, na primjer, vrijednosti promjenljivih "`a`" i "`b`" respektivno "5" i "8", i neka je funkcija "`Razmijeni`" pozvana na opisani način (tj. uz eksplicitno navođenje operatora referenciranja "`&`" pri pozivu funkcije). Sljedeća slika ilustrira situaciju nakon izvršavanja svake od naredbi u funkciji "`Razmijeni`":

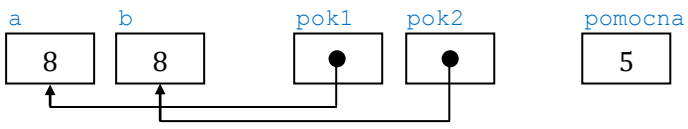
```
Razmijeni(&a, &b);
```



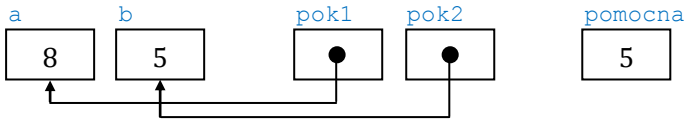

```
int pomocna(*pok1);
```




```
*pok1 = *pok2;
```




```
*pok2 = pomocna;
```



Vidimo da su promjenljive "a" i "b" zaista razmijenile svoje vrijednosti. S druge strane, u jeziku C++ isti problem može se riješiti na mnogo jednostavniji i jasniji način, korištenjem prenosa parametara po referenci umjesto pokazivača. Bez obzira na to, ovaj primjer veoma ilustrativno demonstrira na koji način djeluju pokazivači, a ujedno i demonstrira kako se pokazivači koriste kao parametri funkcija. Interesantno je napomenuti da se potpuno ista situacija u memoriji poput situacije prikazane na prethodnim slikama dešava i u rješenju u kojem se umjesto pokazivača koristi prenos parametara po referenci, samo što toga programer nije svjestan. U suštini, reference uvedene u jezik C++ se, na izvjestan način, ponašaju kao "veoma dobro prerušeni pokazivači".

Prethodni primjer je također veoma poučan, jer ćemo kod njega razmotriti i neke od tipičnih grešaka koje mogu nastupiti pri radu sa pokazivačima. Naime, početnici koji često brkaju razliku između pokazivača i pokazanog objekta često nisu sigurni *kad treba staviti zvjezdicu a kada ne*. Razmotrićemo sada neke od tipičnih grešaka koje u prethodnom primjeru mogu nastati ukoliko se *ne stavi zvjezdica tamo gdje bi trebala*, ili se *stavi tamo gdje ne bi trebala*. Idealna situacija je ukoliko nam kompajler pri tome prijavi grešku, jer onda smo *barem sigurni da nešto nije u redu*. Mnogo gora situacija je ukoliko pri tome *ne dođe do sintaksne greške*, ali funkcija ne radi ispravno. Pri tome je još i dobro ukoliko funkcija daje očito pogrešan rezultati, jer tada ponovo znamo da nešto nije u redu. Najgora stvar koja se može desiti je da sve izgleda kao da funkcijaradi ispravno, a zapravo pravi neku štetu sa strane koja nije odmah vidljiva.

Krenimo prvo sa najmanje problematičnim greškama, tj. onima koje će dovesti do prijave sintaksne greške. Slijedi spisak šta bismo sve mogli pogriješiti u prethodnoj funkciji po pitanju upotrebe zvjezdice, a da to dovede do prijave greške:

```
int *pomocna(*pok1);                    // Greška (1)
int pomocna(pok1);                      // Greška (2)
pok1 = *pok2;                           // Greška (3)
*pok1 = pok2;                           // Greška (4)
pok2 = pomocna;                         // Greška (5)
pok2 = *pomocna;                        // Greška (6)
*pok2 = *pomocna;                       // Greška (7)
```

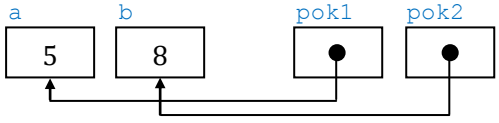
Konstrukcija (1) je sintaksno neispravna jer se u njoj promjenljiva "pomocna" deklarira kao pokazivač, a pokušava se inicijalizirati onim na šta pokazuje pokazivač "pok", što je cijeli broj (a pokazivač ne možemo inicijalizirati cijelim brojem). U konstrukciji (2) pokušavamo inicijalizicati cjelobrojnu promjenljivu pokazivačem, što također nije dozvoljeno. U konstrukcijama (3) i (5) pokušavamo dodijeliti pokazivaču cijeli broj, dok u konstrukciji (4) pokušavamo cijelom broju dodijeliti pokazivač. Konačno, u konstrukcijama (6) i (7) pokušavamo dereferencirati promjenljivu "pomocna" koja nije pokazivačkog tipa, što je naravno nedozvoljeno, osim ako nismo istovremeno napravili još jednu grešku i ovu promjenljivu također deklarirali kao pokazivačku (o posljedicama takve greške govorićemo nešto niže u nastavku teksta).

Sada ćemo preći na malo podmuklije greške, s obzirom na činjenicu da one ostaju nedetektirane od strane kompajlera. Pretpostavimo da smo greškom umjesto dodjele "*pok1 = *pok2" napisali dodjelu "pok1 = pok2" (koja je u načelu sasvim legalna), odnosno da smo napisali sljedeću funkciju:

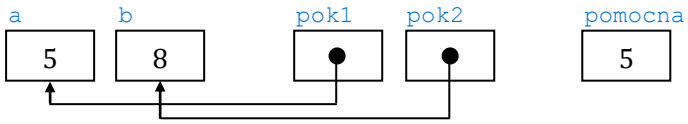
```
void Razmijeni(int *pok1, int *pok2) {
  int pomocna(*pok1);
  pok1 = pok2;                          // OVA FUNKCIJA NE RADI ISPRAVNO...
  *pok2 = pomocna;
}
```

Ova funkcija neće raditi ispravno, jer umjesto da kopiramo ono na šta pokazuje pokazivač "pok2" tamo gdje pokazuje pokazivač "pok1", mi zapravo kopiramo *same pokazivače*, odnosno adrese koje su u njima sadržane, nakon čega će doći do *aliasinga* (oba pokazivača će pokazivati na istu promjenljivu). Sljedeća analiza pokazuje šta će se dogoditi uz iste početne pretpostavke kao u prethodnoj analizi:

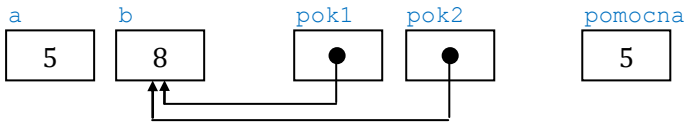
```
Razmijeni(&a, &b);
```



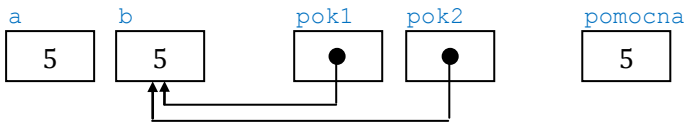
```
int pomocna(*pok1);
```




```
pok1 = pok2;
```




```
*pok2 = pomocna;
```



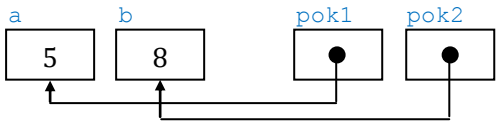
U svakom slučaju, vidimo da sadržaji promjenljivih "a" i "b" *nisu razmijenjeni* (nego je zapravo sadržaj promjenljive "a" samo iskopiran u promjenljivu "b"). Ova greška i nije toliko opasna, jer ćemo u fazi testiranja *barem vidjeti* da funkcija ne radi (gore su greške koje se čak ni testiranjem ne mogu uvijek uočiti, a uskoro ćemo vidjeti da su nažalost i takve greške moguće).

Druga greška koja bi se mogla napraviti je sljedeća, u kojoj se promjenljiva "pomocna" greškom deklarira kao pokazivačka promjenljiva, a ona to ne treba da bude. Naravno, da bi takva funkcija prošla sintaksnu provjeru, treba napraviti još "poneku greškicu", recimo staviti zvjezdicu ispred promjenljive "pomocna" u dodjeli "*pok2 = pomocna". Sve u svemu, pretpostavimo da smo greškom napisali ovakvu, sintaksno ispravnu ali logički nekorektnu funkciju:

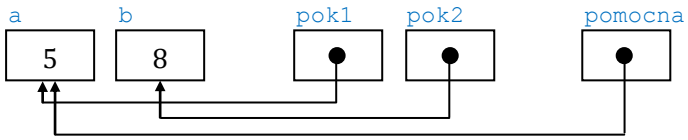
```
void Razmijeni(int *pok1, int *pok2) {
  int *pomocna(pok1);
  *pok1 = *pok2;                          // NI OVA FUNKCIJA NE RADI ISPRAVNO...
  *pok2 = *pomocna;
}
```

Kod ove funkcije već na početku imamo aliasing uzrokovan činjenicom da pokazivači "pomocna" i "pok2" pokazuju na isti objekat. Sljedeća analiza pokazuje šta se dešava prilikom izvršavanja ove funkcije:

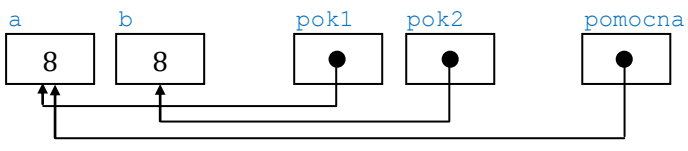
```
Razmijeni(&a, &b);
```



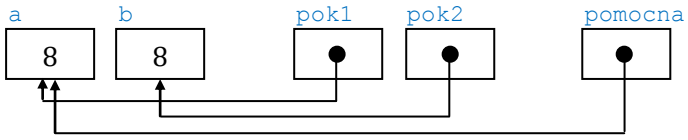

```
int *pomocna(pok1);
```




```
*pok1 = *pok2;
```



```
*pok2 = *pomocna;
```



Kao što vidimo, sadržaji promjenljivih "a" i "b" ni ovdje nisu ispravno razmijenjeni. Varijacijom ove neispravne funkcije može se dobiti još nekoliko neispravnih (mada sintaksno korektnih) varijanti. Recimo, umjesto dodjele "*pok1 = *pok2" neko bi mogao napisati pogrešnu dodjelu "pok1 = pok2". Isto tako, neko bi umjesto dodjele "*pok2 = *pomocna" mogao napisati dodjelu "pok2 = pomocna" (koja bi sad sintaksno prošla, jer je promjenljiva "pomocna" deklarirana kao pokazivač). I naravno, mogle bi se istovremeno napraviti obje spomenute greške. Kao korisnu vježbu, analizirajte šta bi se desilo u svakom od gore opisanih scenarija. Uglavnom, ni u jednom od gore opisanih scenarija funkcija "Razmijeni" ne bi radila ono što treba da radi, ali bi njeno testiranje jasno pokazalo da traženi cilj (razmjena sadržaja promjenljivih "a" i "b") nije postignut.

Pređimo sada na opis vjerovatno najpodmuklije greške koju bismo mogli napraviti prilikom pisanja ove funkcije. Pretpostavimo da neko nije navikao da vrši inicijalizaciju promjenljivih odmah prilikom njihove deklaracije, nego da je više sklon da umjesto konstrukcije

```
int pomocna(*pok1);
```

obavi prvo deklaraciju pa dodjelu vrijednosti promjenljivoj "pok", odnosno da napiše

```
int pomocna;
pomocna = *pok1;
```

Na taj način dobijamo sljedeću izvedbu funkcije "Razmijeni":

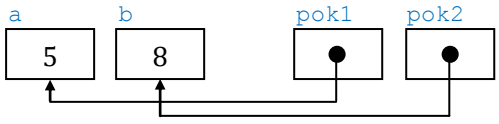
```
void Razmijeni(int *pok1, int *pok2) {
  int pomocna;
  pomocna = *pok1;                              // Ova funkcija je korektna...
  *pok1 = *pok2;
  *pok2 = pomocna;
}
```

Jasno je da je ova funkcija korektna, jer deklaracija promjenljive uz inicijalizaciju ima u konačnici isti efekat kao deklaracija promjenljive bez inicijalizacije iza koje odmah slijedi dodjela željene početne vrijednosti. Međutim, pretpostavimo da je neko napisao verziju gornje funkcije pri čemu je stavio zvjezdice "i gdje treba i gdje ne treba", odnosno da je napisao sljedeću (sintaksno ispravnu) funkciju:

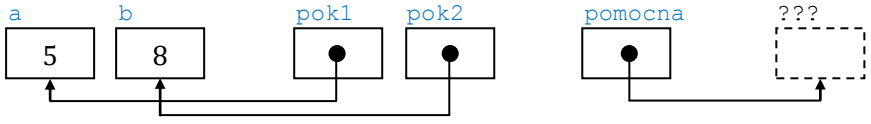
```
void Razmijeni(int *pok1, int *pok2) {
  int *pomocna;
  *pomocna = *pok1;                       // OVA FUNKCIJA IMA FATALNU GREŠKU
  *pok1 = *pok2;                          //    IAKO PRILIKOM TESTIRANJA MOŽE
  *pok2 = *pomocna;                       //    IZGLEDATI KAO DA RADI ISPRAVNO!!!
}
```

Ova funkcija je *neispravna*, ali što je najgore, to *vrlo vjerovatno neće biti otkriveno prilikom njenog testiranja*. Analizirajmo izvršavanje ove funkcije (obratite pažnju da na početku pokazivač "pomocna" *nije inicijaliziran* tako da pokazuje na *slučajnu lokaciju u memoriji*, koja je označena crticama):

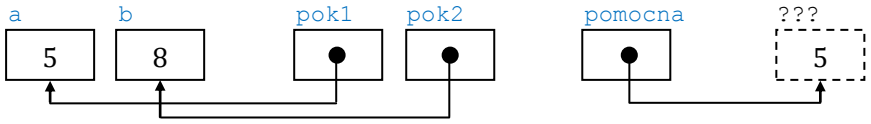
```
Razmijeni(&a, &b);
```



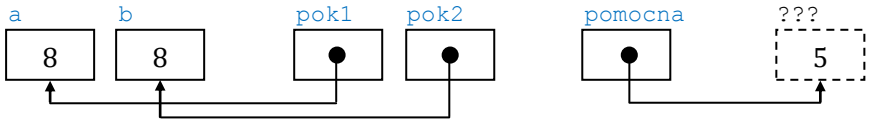

```
int *pomocna;
```



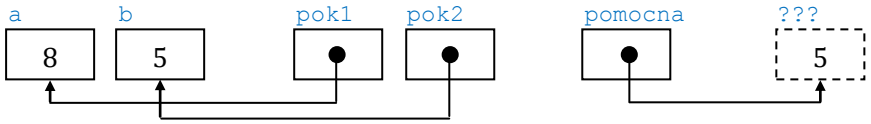
```
*pomocna = *pok1;
```




```
*pok1 = *pok2;
```




```
*pok2 = pomocna;
```



Spolja gledano, testiranje funkcije bi moglo pokazati da funkcija *radi ispravno*, odnoso ona *zaista razmjenjuje* sadržaj promjenljivih "a" i "b"! U čemu je onda problem? Glavni problem je upravo u pristupanju nepoznatom mjestu u memoriji koje nastaje dereferenciranjem *neinicijaliziranog* pokazivača, za koji *ne znamo gdje pokazuje*. Dodjelom "*pomocna = *pok1" mi sadržaj onoga na šta pokazuje pokazivač "pok1" (a to je promjenljiva "a") kopiramo tamo gdje pokazuje pokazivač "pomocna", ali problem u tome je što mi *nemamo nikakve ideje o tome koje je to mjesto u memoriji*. Desi li se slučajno da se na tom mjestu u memoriji ne nalazi ništa bitno, neće biti nikakvih štetnih posljedica, i izgledaće nam da je sve ispravno. Međutim, može se dogoditi da se u tom dijelu memorije na koji pokazivač "pomocna" nalaze recimo neki podaci bitni za funkcioniranje operativnog sistema, ili općenito neki podaci *zabranjeni za pristup*. U tom slučaju će vjerovatno operativni sistem *prekinuti rad programa*, uz obavještenje da program radi nedozvoljene stvari. Ni to nije najgori scenario, jer tad *bar znamo da nešto nije u redu*. Najgore je ako se na tom mjestu u memoriji na koje slučajno pokazivač "pomocna" u tom trenutku pokazuje nalazi recimo *neka druga promjenljiva istog programa*. Ova funkcija će tada *poremetiti* vrijednost te promjenljive, a da *niko toga neće biti svjestan*. Posljedice tog "napada" biće vidljive tek možda *mnogo kasnije*, kada programu zatreba vrijednost te promjenljve koju je ova funkcija uništila! Ovo je najgora vrsta grešaka, jer se teško otkrivaju testiranjem, s obzirom da se njihove moguće posljedice tipično manifestiraju mnogo kasnije u odnosu na trenutak kada je do greške zaista došlo.

Važno je napomenuti da do ove greške *ne bi moglo doći da smo inicijalizaciju vršili odmah prilikom deklaracije*. Naime, treba primijetiti da u prethodnoj konstrukciji

```
int *pomocna;                          // Problematična konstrukcija...
*pomocna = *pok1;
```

smisao zvjezdice ispred promjenljive "pomocna" u prvom i drugom redu *nije isti*. Stoga ova konstrukcija *nije ekvivalentna* konstrukciji

```
int *pomocna(*pok1);                   // A ova nije ni sintaksno ispravna...
```

koja usput *nije ni sintaksno ispravna*, jer se pokazivačka promjenljiva "pomocna" ne može inicijalizirati cjelobrojnom promjenljivom, a "*pok1" se upravo tretira kao cjelobrojna promjenljiva. Sve prethodne (ispravne i neispravne) primjere *treba dobro proučiti*, jer je njihovo proučavanje najbolji način da se ispravno shvati smisao pokazivačkih promjenljivih i izbjegnu greške pri njihovoj upotrebi.

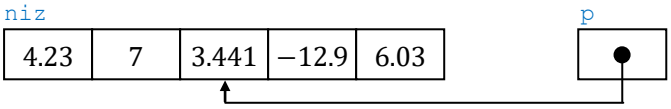
Sada ćemo preći na opis *pokazivačke aritmetike*. Jezici C i C++ pokazivačke promjenljive tretiraju pokazivačke promjenljive *posve drugačije nego obične brojčane promjenljive*, čak i u situacijama kada se adrese koje su u njima pohranjene zaista mogu zamisliti kao obični brojevi (što je zapravo slučaj kod većine računarskih arhitektura). Jedan od razloga za to je i činjenica da nije dobro pretpostavljati ništa o tome kako je memorija zaista organizirana, s obzirom da bi smisao jezika trebala da što manje ovisi od svojstava hardvera na kojem se program izvršava. Posljedica ovoga je da mnoge aritmetičke operacije nad pokazivačkim promjenljivim *nisu dozvoljene*, a i one koje jesu dozvoljene *interpretiraju se bitno drugačije* nego sa običnim brojčanim promjenljivim (na način koji *ne ovisi* od same arhitekture računara).

Stoga je neophodno pokazivačke tipove treba shvatiti kao *sasvim posebne tipove*, bitno različite od cjelobrojnih tipova (bez obzira što su većini slučajeva oni interno realizirani upravo kao cijeli brojevi). Na primjer, *nije dozvoljeno* sabrati, pomnožiti ili podijeliti dva pokazivača. Razlog za ovu zabranu je činjenica da se ne vidi *na šta bi smisleno mogao pokazivati* pokazivač koji bi se dobio recimo množenjem adresa dva druga objekta, čak i uz pretpostavku da su te adrese obični brojevi. Slično, nije dozvoljeno pomnožiti ili podijeliti pokazivač sa brojem, ili obrnuto. Također, mada se interna vrijednost pokazivača *može ispisati na izlazni tok* (pri čemu se kao rezultat ispisa dobija interna reprezentacija adrese pohranjene u pokazivaču u vidu heksadekadnog broja, što za većinu korisnika nije osobito interesantna informacija), *nije dozvoljeno njihovo unošenje* preko ulaznog toka (npr. tastature). Međutim, dozvoljeno je *sabrati pokazivač i cijeli broj* (i obratno), zatim *oduzeti cijeli broj od pokazivača*, kao i *oduzeti jedan pokazivač od drugog*. U nastavku ćemo razmotriti smisao ovih operacija.

Smisao pokazivačke aritmetike očitava se u slučaju kada pokazivač pokazuje na neki element niza (kako operator referenciranja "`&`" kao svoj argument prihvata bilo kakvu *l-vrijednost*, njegov argument može biti i *indeksirani element niza*, za razliku od recimo izraza poput "`&2`" ili "`&(a + 2)`" koji su besmisleni). Da bismo to ilustrirali, pretpostavimo da imamo sljedeće deklaracije:

```
double niz[5] = {4.23, 7, 3.441, -12.9, 6.03};          // Ili bez "=" u C++11
double *p(&niz[2]);
```

Ovdje smo inicijalizirali pokazivač "`p`" tako da pokazuje na element niza "`niz[2]`". Nakon ovih deklaracija, stanje u memoriji možemo predstaviti ovako:



Pokazivač "`p`" pokazuje na element "`niz[2]`", tako da je "`*p`" upravo "`niz[2]`". Razmotrimo sada izraz "`p + 1`". Ovaj izraz *ponovo predstavlja pokazivač* (tako da se i on *može dereferencirati* poput svih drugih pokazivača, odnosno izraz poput "`*(p + 1)`" je *legalan*) ali koji pokazuje na element "`niz[3]`". Dakle, dodavanje jedinice na neki pokazivač koji pokazje na neki element niza daje kao rezultat novi pokazivač koji pokazuje na *sljedeći element niza* u odnosu na element na koji pokazuje izvorni pokazivač. Međutim, treba naglasiti da čak i u slučajevima kada adrese možemo interpretirati kao brojeve, *brojčana vrijednost adrese koju predstavlja pokazivač* "`p + 1`" *gotovo sigurno nije jednaka brojčanoj vrijednosti adrese koju predstavlja pokazivač* "`p`" *uvećanoj za* 1! Naime, kod većine računarskih arhitektura memorija se adresira po *bajtima*, odnosno uzima se da jedna memorijska ćelija zauzima tačno jedan bajt. S druge strane, promjenljive tipa "**`double`**" *sasvim sigurno ne mogu stati u jednu jednobajtnu memorijsku lokaciju* (najčešći način kodiranja promjenljivih tipa "**`double`**" koji se danas koristi propisan je IEEE 754 standardom, prema kojem se jedna promjenljiva tipa "**`double`**" kodira u 64 bita, odnosno u 8 bajta, što znači da promjenljive tipa "**`double`**" prema tom standardu zauzimaju 8 jednobajtnih memorijskih lokacija). Stoga, ukoliko je brojčana vrijednost adrese pohranjene u pokazivaču "`p`" recimo 1000, tada pokazivaču "`p + 1`" ne odgovara adresa 1001 nego recimo 1008, s obzirom da element niza "`niz[2]`" ne zauzima samo memorijsku ćeliju sa adresom 1000, nego 8 memorijskih ćelija sa adresama od 1000 do 1007 uključivo!

Ukoliko pretpostavimo da je memorijski model *linearan* (jednodimenzionalan) tako da su adrese interno reprezentirane kao obični brojevi, tada će stvarna adresa koju predstavlja pokazivač "`p + 1`" biti veća od adrese koju predstavlja pokazivač "`p`" za *broj memorijskih lokacija* (tipično *broj bajta*) *koje zauzima objekat na koji pokazivač* "`p`" *pokazuje*. Ova konvencija je izuzetno praktična, jer programer *ne mora da vodi brigu* o tome koliko zaista neki element niza zauzima prostora: ako "`p`" pokazuje na neki element niza, "`p + 1`" pokazuje na *sljedeći element istog niza*, ma gdje da se on nalazio. Što je još značajnije, to vrijedi bez obzira kako je memorija organizirana (tj. čak i u slučaju da ona nije organizirana na linearan način). Izraz "`p + 1`" uvijek pokazuje na element niza koji neposredno slijedi elementu na koji pokazuje pokazivač "`p`", ma kako da je memorija organizirana, i ma kakav da je interni sadržaj pokazivača "`p`". To i jeste osnovni smisao pokazivačke aritmetike. Analogno tome, izraz "`p - 1`" predstavlja pokazivač koji pokazuje na *prethodni element niza* u odnosu na element na koji pokazuje pokazivač "`p`" (u navedenom primjeru na element "`niz[1]`"). Generalno, izraz oblika "`p + n`" odnosno "`p - n`" gdje je "`p`" pokazivač a "`n`" cijeli broj (odnosno ma kakav cjelobrojni izraz) predstavlja novi pokazivač koji pokazuje na element niza čiji je indeks veći odnosno manji za "`n`" u odnosu na indeks

elementa niza na koji pokazuje pokazivač "p". Stoga će sljedeća naredba, za niz iz razmatranog primjera, ispisati vrijednosti "3.441", "–12.9" i "6.03":

```
for(int i = 0; i <= 2; i++) std::cout << *(p + i) << " ";
```

Uočite razliku između izraza "*(p + i)" i izraza "*p + i" (odnosno "(*p) + i") koji, kao što smo ranije vidjeli, imaju posve drugačije značenje. Također, kako je svaki dereferencirani pokazivač *l-vrijednost*, to su izrazi poput izraza

```
*(p - 1) = 8.5
```

sasvim smisleni (ovaj izraz će postaviti sadržaj elementa niza "niz[1]" na vrijednost "8.5").

Bitno je napomenuti da je smisao pokazivačke aritmetike preciziran standardima jezika C i C++ *samo za slučaj kada pokazivač pokazuje na neki element nekog niza*. Za slučaj kada pokazivač pokazuje na neki drugi objekat (npr. na običnu promjenljivu), smisao pokazivačke aritmetike *nije definiran*. Na primjer, za ranije definiran pokazivač "pok" koji je pokazivao na cjelobrojnu promjenljivu "broj", smisao izraza "pok + 1" i "pok - 1" nije preciziran standardom (tj. oni su *sintaksno ispravni*, ali nije precizirano šta im je vrijednost, stoga se ne smiju koristiti). Za slučaj linearnih memorijskih modela, ovi izrazi bi se gotovo sigurno interpretirali kao pokazivači koji pokazuju na mjesta u memoriji čije su adrese veće odnosno manje od adrese zapisane u pokazivaču "pok" za onoliko memorijskih lokacija koliko zauzima promjenljiva "broj". Međutim, ako memorijski model nije linearan, interpretacije zaista mogu biti svakakve. Stoga se trebamo pridržavati standarda koji strogo naglašava da pokazivačku aritmetiku treba koristiti samo ukoliko pokazivač pokazuje na neki element niza. Ovog pravila se zaista moramo pridržavati strogo: u nekim slučajevima operativni sistem ima pravo da prekine izvršavanje programa u kojem je pokazivač "odlutao" zbog pogrešne primjene pokazivačke aritmetike, čak i ako se "zalutali" pokazivač uopće ne dereferencira (ovo se dešava iznimno rijetko, ali se dešava)!

Pokazivačka aritmetika nije uvijek definirana čak ni za slučaj pokazivača koji pokazuju na elemente niza. Postoji još pravilo koje kaže da su izrazi oblika "p + n" odnosno "p - n" gdje je "p" pokazivač koji pokazuje na neki element niza a "n" cjelobrojni izraz definirani samo ukoliko novodobijeni pokazivač ponovo pokazuje na neki element niza koji *zaista postoji* u skladu sa specificiranom dimenzijom niza, ili eventualno na mjesto koje se nalazi neposredno *iza posljednjeg elementa niza*. Drugim riječima, vrijednost izraza "p - n" postaje nedefinirana ukoliko novodobijeni pokazivač "odluta" ispred prvog elementa niza, a vrijednost izraza "p + n" postaje nedefinirana ukoliko novodobijeni pokazivač "odluta" daleko iza posljednjeg elementa niza. Za konkretan primjer niza "niz" iz maloprije navedenog primjera koji sadrži 5 elemenata i pokazivača "p" koji je postavljen da pokazuje na element "niz[2]", izrazi "p + 1", "p + 2", "p + 3", "p - 1" i "p - 2" su precizno definirani, za razliku od izraza "p + 4" ili "p - 3" koji nisu, jer su "odlutali" u "nepoznate vode" (njihov sadržaj je *nepredvidljiv*, posebno u slučajevima kada memorijski model nije linearan).

Nad pokazivačkim promjenljivim se mogu primjenjivati i operatori "+=", "-=", "++" i "--" čije je značenje analogno kao za obične promjenljive, samo što se koristi pokazivačka aritmetika. Na primjer, izraz "p += 2" će promijeniti sadržaj pokazivačke promjenljive "p" tako da pokazuje na element niza koji se nalazi dvije pozicije naviše u odnosu na element niza na koji "p" trenutno pokazuje, dok će izraz "--p" promijeniti sadržaj pokazivačke promjenljve "p" tako da pokazuje na prethodni element niza u odnosu na element na koji "p" trenutno pokazuje (u oba slučaja, "p" postaje nedefiniran u slučaju da odlutamo izvan prostora koji niz zauzima; eventualno se smijemo pozicionirati tik iza kraja niza). Na taj način moguće je koristiti pokazivače čija vrijednost tokom izvršavanja programa pokazuje na razne elemente niza, kao da se pokazivač "kreće" odnosno "klizi" kroz elemente niza. Ovakve pokazivače obično nazivamo *klizeći pokazivači* (engl. *sliding pointers*), kao što je ilustrirano u sljedećem primjeru (u kojem pretpostavljamo da su "niz" i "p" deklarirani kao u ranijim primjerima):

```
p = &niz[0];
for(int i = 0; i < 5; i++) {
  std::cout << *p  << " ";
  p++;
}
```

Dereferenciranje i inkrementiranje (odnosno dekrementiranje) pokazivača se često obavlja u istom izrazu, što je ilustrirano u sljedećem primjeru (koji je ekvivalentan prethodnom):

```
p = &niz[0];
for(int i = 0; i < 5; i++) std::cout << *p++ << " ";
```

U posljednjem primjeru, izraz "`*p++`" interpretira se kao izraz "`*(p++)`" a ne kao izraz "`(*p)++`" koji ima drugačije, ranije objašnjeno značenje.

Oba navedena primjera su posve legalna i korektna. Međutim, sljedeći primjer, koji bi trebao da ispiše elemente niza u obrnutom poretku, *nije legalan* (iako je sintaksno ispravan), bez obzira što će u većini slučajeva *raditi ispravno*:

```
p = &niz[4];
for(int i = 0; i < 5; i++) {                    // NELEGALNO: Na kraju petlje,
  std::cout << *p  << " ";                      //   pokazivač p će "odlutati"
  p--;                                          //   u nedozvoljeno područje!!!
}
```

Isto vrijedi i za sažetu varijantu ovog primjera:

```
p = &niz[4];
for(int i = 0; i < 5; i++) std::cout << *p-- << " ";        // NELEGALNO!
```

U čemu je problem? Poteškoća leži u činjenici da nakon završetka ovih sekvenci, pokazivač "`p`" pokazuje *ispred prvog elementa niza*, što po standardu nije legalno. Operativni sistem ima pravo da prekine program u kojem se generira takav pokazivač ukoliko mu njegovo postojanje ugrožava rad (to se vrlo vjerovatno neće desiti, ali se *može desiti*). Rješenje ovog problema je posve jednostavno — nemojmo generirati takve pokazivače. Na primjer, možemo "`p`" na početku postaviti da pokazuje tačno *iza* posljednjeg elementa niza, tj. na nepostojeći element niza "`niz[5]`" (što je, začudo, legalno — razloge za to ćemo shvatiti kasnije) i umanjivati pokazivač *prije nego što ga dereferenciramo*, kao u sljedećem (legalnom) primjeru:

```
p = &niz[5];
for(int i = 0; i < 5; i++) {
  p--;
  std::cout << *p << " ";
}
```

ili u njegovoj kompaktnijoj varijanti:

```
p = &niz[5];
for(int i = 0; i < 5; i++) std::cout << *--p << " ";
```

Pokazivači se mogu *porediti* pomoću operatora "`<`", "`>`", "`<=`", "`>=`", "`==`" i "`!=`" pod uvjetom da *pokazuju na isti tip*. Dva pokazivača na različite tipove (npr. pokazivači na tipove "**`int`**" i "**`double`**") ne mogu se porediti, čak i ukoliko su tipovi na koje pokazivači pokazuju uporedivi. Pokušaj poređenja pokazivača koji pokazuju na različite tipove *prijavljuje se kao sintaksna greška od strane kompajlera*. Zbog toga, pretpostavimo da imamo dva pokazivača koji pokazuju na objekte istog tipa. Rezultat poređenja primjenom operatora "`==`" i "`!=`" je uvijek jasno definiran: dva pokazivača su jednaka ako i samo ako pokazuju na *isti objekat*. Međutim, rezultat poređenja dva pokazivača primjenom ostalih relacionih operatora je precizno definiran standardnom samo za slučaj *kada oba pokazivača pokazuju na elemente jednog te istog niza*. Na primjer, neka su "`p1`" i "`p2`" dva pokazivača koji pokazuju na neke elemente istog niza (eventualno, oni smiju pokazivati i na fiktivni prostor iza posljednjeg elementa niza). Pokazivač "`p1`" je *manji* od pokazivača "`p2`" ukoliko "`p1`" pokazuje na element niza *sa manjim indeksom* u odnosu na element na koji pokazuje "`p2`", a *veći* ukoliko pokazuje na element niza *sa većim indeksom* u odnosu na element na koji pokazuje "`p2`". S druge strane, ukoliko pokazivači "`p1`" i "`p2`" ne pokazuju na dva elementa istog niza, pojmovi "veći" i "manji" nisu definirani, tako da rezultat poređenja *zavisi od implementacije*. Stoga se takva poređenja *ne smiju koristiti*. Ukoliko je memorijski model linearan, sva poređenja pokazivača se obično interno realiziraju prostim poređenjem *brojčanih vrijednosti adresa* pohranjenih u pokazivaču, tako da su rezultati poređenja praktično uvijek u skladu sa intuicijom, čak i ukoliko kršimo pravila kada je poređenje pokazivača legalno (npr. kod linearnih modela uvijek je manji onaj pokazivač koji pokazuje na nižu adresu). Međutim, kod memorijskih modela koji nisu linearni, svakakva iznenađenja su moguća. Da biste spriječili moguća iznenađenja, samo poštujte pravilo: *nemojte porediti po veličini pokazivače koji ne pokazuju na elemente istog niza*. Također, bitno je uočiti razliku između poređenja poput "`p1 == p2`" koje poredi *dva pokazivača* (koji mogu biti različiti čak i ukoliko su sadržaji lokacija na koje oni pokazuju isti) i poređenja poput "`*p1 == *p2`" koje poredi *sadržaje lokacija na koje pokazuju dva pokazivača* (koji mogu biti isti čak i ukoliko su pokazivači različiti).

Mada nije moguće sabrati, pomnožiti ili podijeliti dva pokazivača, moguće je *oduzeti* jedan pokazivač od drugog, pod uvjetom da su oba pokazivača *istog tipa*. Pri tome, rezultat oduzimanja je, slično kao za slučaj poređenja, precizno definiran *samo za slučaj kada oba pokazivača pokazuju na elemente istog niza*. Po definiciji, rezultat oduzimanja takva dva pokazivača je *cijeli broj* (a ne pokazivač) koji je jednak *razlici indeksa elemenata niza na koje pokazivači pokazuju*. Na primjer, ukoliko pokazivač "`p1`" pokazuje na element niza "`niz[4]`" a pokazivač "`p2`" na element niza "`niz[1]`", tada je vrijednost izraza "`p1 - p2`" cijeli broj 3 (tj. 4 – 1). Ovakva definicija obezbjeđuje da vrijedi pravilo da je vrijednost izraza "`p1 + (p2 - p1)`" uvijek jednaka vrijednosti "`p2`", što je u skladu sa intuicijom.

Zahvaljujući mogućnosti poređenja pokazivača, prethodni primjeri za ispis elemenata niza uz korištenje klizećih pokazivača mogu se napisati tako da se uopće ne koristi pomoćna promjenljiva "`i`", već samo klizeći pokazivač. Na primjer, elemente niza u standardnom poretku možemo ispisati ovako:

```
for(double *p = &niz[0]; p < &niz[5]; p++)
  std::cout << *p << " ";
```

Pri ispisu unazad ponovo treba biti oprezniji, zbog mogućnosti da pokazivač "odluta" ispred prvog elementa niza. Na primjer, sljedeći primjer *nije korektan* (u skladu sa standardom):

```
for(double *p = &niz[4]; p >= &niz[0]; p--)                    // NELEGALNO!!
  std::cout << *p << " ";
```

Naime, nakon što "`p`" dostigne prvi element niza, vrijednost izraza "`p--`" postaje nedefinirana, pa je pogotovo nedefinirano šta je rezultat poređenja "`p >= &niz[0]`". Stoga je veoma upitno kada će napisana petlja uopće završiti (i da li će uopće završiti). Ovo nije samo prazno filozofiranje: napisani primjer *provjereno* neće raditi sa C i C++ kompajlerima za MS DOS operativni sistem (kod kojeg memorijski model *nije linearan*) u slučajevima kada je niz "`niz`" deklariran kao statički ili globalni. Sigurno je moguće navesti još mnoštvo situacija u kojima ovaj primjer *ne radi kako treba*. Rješenje ovog problema je isto kao u ranijim primjerima: ne smijemo dozvoliti da pokazivačka aritmetika dovede do izlaska pokazivača izvan dozvoljenog opsega. Sljedeća varijanta je, na primjer, korektna (sjetimo se da se u "**`for`**" petlji bilo koji od njena tri argumenta mogu izostaviti – u ovom primjeru izostavljen je treći argument):

```
for(double *p = &niz[5]; p > &niz[0];)
  std::cout << *--p << " ";
```

Treba naglasiti da su veoma rijetke knjige o programskim jezicima C i C++ u kojima se upozorava na opisanu opasnost pokazivačke aritmetike, koja posebno dolazi do izražaja kada klizeće pokazivače koristimo za kretanje kroz niz *naniže*.

Pokazivači su tijesno povezani sa nizovima. Naime, kad god se ime niza upotrijebi *samo za sebe*, bez navođenja indeksa u uglastim zagradama, ono se *automatski konvertira u pokazivač* (*odgovarajućeg tipa*) *koji pokazuje na prvi element niza* (preciznije, u *konstantni pokazivač*, što ćemo precizirati malo kasnije). Drugim riječima, ukoliko je "`niz`" neki niz, tada je izraz "`niz`" potpuno ekvivalentan izrazu "`&niz[0]`". Zbog automatske konverzije nizova u pokazivače, slijedi da se pokazivačka aritmetika *može primijeniti i na nizove*. Razmotrimo, na primjer, izraz "`niz + 2`". Ovaj izraz je, na prvi pogled, besmislen. Međutim, on je ekvivalentan izrazu "`&niz[0] + 2`" koji je, zahvaljujući pokazivačkoj aritmetici (ne zaboravimo da je izraz "`&niz[0]`" *pokazivač*), ekvivalentan izrazu "`&niz[2]`", što je lako provjeriti. Dakle, izrazi "`niz + 2`" i "`&niz[2]`" su *ekvivalentni*. Slijedi da elemente niza "`niz`" možemo ispisati i ovako:

```
for(double *p = niz; p < niz + 5; p++) std::cout << *p << " ";
```

Naravno, iz istog razloga, sasvim je legalna i sljedeća konstrukcija:

```
for(int i = 0; i < 5; i++) std::cout << *(niz + i) << " ";
```

Iz ovoga vidimo da su, zahvaljujući pokazivačkoj aritmetici i automatskoj konverziji nizova u pokazivače, izrazi "`niz[i]`" i "`*(niz + i)`", gdje je "`niz`" ma kakva nizovna promjenljiva, zapravo sinonimi. Ovim je u jezicima C i C++ uspostavljena veoma tijesna veza između pokazivača i nizova. Ovi jezici su pomenutu vezu učinili još tijesnijom (na način kakav ne postoji niti u jednom drugom programskom jeziku) uvođenjem konvencije da izrazi oblika "`a[n]`" i "`*(a + n)`" *uvijek budu potpuni sinonimi*, bez obzira da li je "`a`" ime niza ili pokazivač. Drugim riječima, indeksiranje se može primijeniti i na pokazivače! Jednostavno, izraz oblika "`a[n]`" u slučaju kada "`a`" nije ime niza, *po definiciji se*

interpretira kao "`*(a + n)`", bez obzira kojeg je tipa "`a`". Stoga je izraz "`a[n]`" ispravan kad god se zbir "`a + n`" može interpretirati kao pokazivač (jedna pomalo čudna posljedica ove činjenice je da su sinonimi i izrazi poput "`niz[3]`" i "`3[niz]`" mada se ovo svojstvo teško može iskoristiti za nešto pametno). Indeksiranje primjenjeno na pokazivače ilustriraćemo sljedećim primjerom koji ispisuje elemente 4.23, 7, 3.441 i −12.9 (uz pretpostavku da je niz "`niz`" deklariran kao u prethodnim primjerima) i koji pokazuje kako se u jeziku C++ mogu simulirati nizovi sa *negativnim indeksima* kakvi postoje u nekim drugim programskim jezicima (npr. u Pascalu i FORTRAN-u):

```
double *p(niz + 2);
for(int i = -2; i <= 1; i++) std::cout << p[i] << " ";
```

Podsjetimo se da izraz "`niz + 2`" ima isto značenje kao i izraz "`&niz[2]`". Kasnije ćemo vidjeti i mnoge druge korisne primjene činjenice da se indeksiranje može primijeniti na pokazivače. Međutim, treba napomenuti da sljedeći primjer, koji se čak može naći u nekim knjigama o jezicima C i C++, *nije legalan*. Programer je htio da iskoristi indeksiranje pokazivača sa ciljem simulacije nizova čiji indeksi počinju od *jedinice* umjesto od nule:

```
double *p(niz - 1);                                          // NELEGALNO!
for(int i = 1; i <= 5; i++) std::cout << p[i] << " ";
```

Problem sa ovim primjerom je u tome što izraz "`niz - 1`" nije legalan, u skladu sa već opisanim pravilima vezanim za pokazivačku aritmetiku (isto vrijedi i za njemu ekvivalentan izraz "`&niz[-1]`"). Stoga ovaj primjer može ali i ne mora raditi, ovisno od konkretnog kompajlera i operativnog sistema. Postoji velika vjerovatnoća da će on raditi ispravno (inače ga ne bi citirali u mnogim knjigama), ali postoji vjerovatnoća i da neće. To je, bez sumnje, sasvim dovoljan razlog da *ne koristimo ovakva rješenja*.

Bez obzira na tijesnu vezu između pokazivača i nizova, na imena nizova *ne mogu se primijeniti* operatori "`+=`", "`-=`", "`++`" i "`--`" koji *mijenjaju* sadržaj pokazivačke promjenljive, odnosno lokaciju na koju oni pokazuju. Imena nizova (sama za sebe, bez indeksiranja) uvijek se tretiraju kao pokazivači na *prvi element niza*, i to se *ne može promijeniti*. Oni su uvijek *vezani za prvi element niza*. Preciznije, ime niza upotrijebljeno samo za sebe konvertira se u *konstantni* (*nepromjenljivi*) pokazivač na prvi element niza. Stoga se imena nizova *ne mogu koristiti kao klizeći pokazivači*.

Treba napomenuti da su *jedini izuzeci* u kojima se ime niza upotrijebljeno samo za sebe ne konvertira u odgovarajući pokazivač slučajevi kada se ime niza upotrijebi kao argument operatora "`sizeof`" ili "`&`". Tako, npr. "`sizeof niz`" daje *veličinu čitavog niza* a ne veličinu pokazivača. Također, izraz "`&niz`" ne predstavlja adresu pokazivača, već adresu prvog elementa niza. Naravno, izrazi "`niz`" ili "`&niz[0]`" također predstavljaju adresu prvog elementa niza, ali oni *nisu ekvivalentni* izrazu "`&niz`". Naime, tip izraza "`niz`" ili "`&niz[0]`" je "pokazivač na element niza" (npr. pokazivač na cijeli broj). S druge strane, tip izraza "`&niz`" (koji se inače vrlo rijetko koristi) je "pokazivač na čitav niz" (odnosno "pokazivač na niz kao cjelinu"). Drugim riječima, *tipovi izraza* "`niz`" (ili "`&niz[0]`") i "`&niz`" se *razlikuju* (iako im je vrijednost ista). Pokazivači na "nizove kao cjeline" koriste se jedino kod *dinamičke alokacije višedimenzionalnih nizova*, i o njima ćemo govoriti kada budemo obrađivali tu tematiku.

Kako se nizovi po potrebi automatski konvertiraju u pokazivače kada se upotrijebe bez indeksa, to će svaka funkcija koja kao svoj formalni parametar očekuje pokazivač na neki tip kao stvarni parametar prihvatiti niz elemenata tog tipa (naravno, formalni parametar će pri tome pokazivati na prvi element niza). Međutim, interesantno je da vrijedi i obrnuta situacija: funkciji koja kao formalni parametar očekuje niz možemo kao stvarni parametar proslijediti pokazivač odgovarajućeg tipa. Zapravo, jezici C i C++ *uopće ne prave razliku* između formalnog parametra tipa "pokazivač na tip `Tip`" i formalnog parametra tipa "Niz elemenata tipa `Tip`" (u oba slučaja funkcija dobija samo adresu prvog elementa niza). Drugim riječima, formalni parametri nizovnog tipa, ne samo da se mogu koristiti kao pokazivači, već oni zaista i *jesu pokazivači*, bez obzira što u općem slučaju, nizovi i pokazivači nisu u potpunosti ekvivalentni, kao što smo već naglasili. Zbog tog razloga, operator "`sizeof`" *neće dati ispravnu veličinu niza* ukoliko se primijeni na formalni argument nizovnog tipa (naime, kao rezultat ćemo dobiti *veličinu pokazivača*).

Činjenica da su formalni argumenti nizovnog tipa zapravo pokazivači, omogućava nam da funkcije koje obrađuju nizove možemo realizirati i preko klizećih pokazivača. Na primjer, funkciju koja će ispisati na ekranu elemente niza realnih brojeva možemo umjesto uobičajenog načina (sa *for*-petljom i indeksiranjem elemenata niza) realizirati i na sljedeći način:

```
void IspisiNiz(double *pok, int n) {
  for(int i = 0; i < n; i++) std::cout << *pok++ << std::endl;
}
```

Ovu funkciju možemo pozvati na posve uobičajeni način:

```
IspisiNiz(niz, 5);
```

Nikakve razlike ne bi bilo ni da smo formalni parametar deklarirali kao "`double pokazivac[]`". U oba slučaja on bi se mogao koristiti kao klizeći pokazivač. Dakle, za formalne parametre nizovnog tipa *ne vrijedi pravilo da su uvijek vezani isključivo za prvi element niza*. Oni se mogu koristiti u potpunosti kao i pravi pokazivači (iz prostog razloga što *oni to i jesu*). Zapravo, mogućnost da se formalni parametri deklariraju kao da su nizovnog tipa podržana je samo sa ciljem da se *jasnije izrazi namjera* da funkcija kao stvarni parametar očekuje *niz*. Deklaracija formalnog parametra kao pokazivača početniku bi svakako djelovala mnogo nejasnije.

Moguće je deklarirati i *pokazivače na konstantne objekte*. Na primjer, deklaracijom

```
const double *p;                    // Nekonstantni pokazivač na konstantni objekat
```

deklariramo pokazivač "`p`" na *konstantnu realnu vrijednost* (tip ovog pokazivača je "`const double *`", za razliku od pokazivača na nekonstantnu realnu vrijednost, čiji je tip prosto "`double *`"). Ovakvi pokazivači omogućavaju da se sadržaj lokacije na koju oni pokazuju *čita*, ali ne i da se *mijenja*. Na primjer, sa ovakvim pokazivačem izrazi "`std::cout << *p`" i "`broj = p[2]`" su dozvoljeni (pod uvjetom da je promjenljiva "`broj`" deklarirana kao realna promjenljiva), ali izrazi "`*p = 2.5`" i "`p[2] = 0`" nisu. Odavde odmah vidimo da se formalni parametar "`pok`" u prethodnoj funkciji mogao deklarirati kao "`const double *pok`", što bi čak bilo veoma preporučljivo. Naime, na taj način bismo spriječili eventualne nehotične promjene objekta na koji pokazivač "`pok`" pokazuje.

Na ovom mjestu potrebno je razjasniti neke česte zablude vezane za pokazivače na konstantne objekte. Prvo, pokazivač na konstantni objekat *neće učiniti objekat na koji on pokazuje konstantom*. Na primjer, ako je "`p`" pokazivač na konstantnu realnu vrijednost, a "`broj`" neka realna (nekonstantna) promjenljiva, dodjela "`p = &broj`" neće učiniti promjenljivu "`broj`" konstantom. Promjenljiva "`broj`" će se i dalje moći mijenjati, *ali ne preko pokazivača* "`p`"! Ipak, adresu *konstantnog objekta* (npr. adresu neke konstante) moguće je dodijeliti samo pokazivaču na konstantni objekat. Također, pokazivaču na nekonstantni objekat nije moguće (bez eksplicitne primjene operatora za konverziju tipa) dodijeliti pokazivač na konstantni objekat (čak i ukoliko su tipovi na koje oba pokazivača pokazuju isti), jer bi nakon takve dodjele preko pokazivača na nekonstantni objekat mogli promijeniti sadržaj konstantnog objekta, što je svakako nekonzistentno. Obrnuta dodjela je *legalna*, tj. pokazivaču na konstantni objekat moguće je dodijeliti pokazivač na nekonstantni objekat istog tipa, s obzirom da takva dodjela ne može imati nikakve štetne posljedice.

Treba primijetiti da se sadržaj pokazivača na konstantne objekte može mijenjati, tj. sam pokazivač *nije konstantan*. To treba razlikovati od *konstantnih pokazivača*, čija se vrijednost ne može mijenjati, ali se sadržaj objekata na koji oni pokazuju može mijenjati (npr. imena nizova upotrijebljena sama za sebe konvertiraju se upravo u konstantne pokazivače). Konstantni pokazivači mogu se deklarirati na sljedeći način (primijetimo da se kvalifikator "`const`" ovdje piše *iza zvjezdice*):

```
double *const p(&niz[2]);           // Konstantni pokazivač na nekonstantni objekat
```

Primijetimo da se konstantni pokazivač *mora odmah inicijalizirati*, jer mu se naknadno vrijednost više ne može mijenjati. Naravno, sada operacije poput "`p++`" ili "`p += 2`" koje bi *promijenile* sadržaj pokazivača "`p`" nisu dozvoljene (s obzirom da je "`p`" konstantan pokazivač), ali je izraz "`p + 2`" posve legalan (i predstavlja konstantni pokazivač koji pokazuje na element niza "`niz[4]`"). Konačno, moguće je deklarirati i *konstantni pokazivač na konstantni objekat*, npr. deklaracijom poput sljedeće:

```
const double *const p(&Niz[2]);   // Konstantni pokazivač na konstantni objekat
```

Može se postaviti pitanje zašto se uopće patiti sa pokazivačima i koristiti pokazivačku aritmetiku ukoliko se ista stvar može obaviti i uz pomoć indeksiranja. Ako zanemarimo dinamičku alokaciju memorije i dinamičke strukture podataka, o kojima će kasnije biti govora i koji se ne mogu izvesti bez upotrebe pokazivača, osnovni razlog je *efikasnost* i *fleksibilnost*. Naime, mnoge radnje se mogu

neposrednije obaviti upotrebom klizećih pokazivača nego pomoću indeksiranja, naročito u slučajevima kada se elementi niza obrađuju sekvencijalno, jedan za drugim. Pored toga, pokazivač koji pokazuje na neki element niza sadrži u sebi *cjelokupnu informaciju koja je potrebna da se tom elementu pristupi*. Kod upotrebe indeksiranja ta informacija je razbijena u *dvije neovisne cjeline*: ime niza (odnosno adresa prvog elementa niza) i indeks posmatranog elementa.

Navedimo jedan klasični školski primjer koji se navodi u gotovo svim udžbenicima za programski jezik C (nešto rjeđe za C++, jer se C++ ne hvali toliko pokazivačima koliko C), koji ilustrira efikasnost upotrebe klizećih pokazivača. Naime, razmotrimo kako bi se mogla napisati funkcija nazvana "`KopirajString`", sa dva parametra od kojih su oba nizovi znakova, koja kopira sve znakove drugog niza u prvi niz, sve do "`NUL`" graničnika (odnosno znaka sa ASCII šifrom 0, koja po konvenciji jezika C predstavlja oznaku kraja niza znakova, odnosno stringa), uključujući i njega. Zanemarimo ovdje činjenicu da praktično identična funkcija, pod imenom "`strcpy`", već postoji kao standardna funkcija u biblioteci "`cstring`". Vjerovatno najočiglednija verzija funkcije "`KopirajString`" mogla bi izgledati ovako (kvalifikator "**`const`**" kod imena nizovnog parametra "`izvorni`" samo ukazuje da funkcija ne mijenja sadržaj ovog niza):

```
void KopirajString(char odredisni[], const char izvorni[]) {
  int i(0);
  while(izvorni[i] != 0) {
    odredisni[i] = izvorni[i];
    i++;
  }
  odredisni[i] = 0;                    // Smještanje graničnika u odredište
}
```

Zamijenimo sada parametre "`odredisni`" i "`izvorni`" klizećim pokazivačima (strogo uzevši, oni to već i sada jesu, ali ćemo ipak izmijeniti i njihovu deklaraciju, da jasnije istaknemo namjeru da ih želimo koristiti kao klizeće pokazivače). Odmah vidimo da nam indeks "`i`" više nije potreban:

```
void KopirajString(char *odredisni, const char *izvorni) {
  while(*izvorni != 0) {
    *odredisni = *izvorni;
    odredisni++; izvorni++;
  }
  *odredisni = 0;
}
```

Dalje, oba inkrementiranja možemo obaviti u istom izrazu u kojem vršimo dodjelu. Također, test različitosti od 0 možemo izostaviti (zbog automatske konverzije svake nenulte vrijednosti u logičku vrijednost "**`true`**") čime dobijamo sljedeću verziju funkcije:

```
void KopirajString(char *odredisni, const char *izvorni) {
  while(*izvorni) *odredisni++ = *izvorni++;
  *odredisni = 0;
}
```

Konačno, kako je u jezicima C i C++ dodjela također izraz, rezultat izvršene dodjele je zapravo rezultat izraza "`*izvorni`" (inkrementiranje se obavlja naknadno) koji je identičan sa izrazom u uvjetu "**`while`**" petlje. Stoga se funkcija može svesti na sljedeći oblik:

```
void KopirajString(char *odredisni, const char *izvorni) {
  while(*odredisni++ = *izvorni++);
}
```

Djeluje nevjerovatno, zar ne!? Primijetimo da je nestala potreba čak i za dodjelom "`*odredisni = 0`", jer je lako uočiti da će ovako napisana petlja iskopirati i "`NUL`" graničnik. Ovaj primjer veoma ilustrativno pokazuje u kojoj mjeri se klizeći pokazivači mogu iskoristiti za optimizaciju kôda (doduše, po cijenu čitljivosti i razumljivosti, mada se na pokazivačku aritmetiku programeri brzo naviknu kada je jednom počnu koristiti). Funkcija "`strcpy`" iz biblioteke "`cstring`" implementirana je upravo ovako. Odnosno, ne baš u potpunosti: prava funkcija "`strcpy`" iz biblikoteke "`cstring`" implementirana je tačno ovako (ako zanemarimo da su imena promjenljivih u stvarnoj implementaciji vjerovatno na engleskom jeziku, što je naravno posve nebitno):

```
char *strcpy(char *odredisni, const char *izvorni) {
  char *pocetak(odredisni);
  while(*odredisni++ = *izvorni++);
  return pocetak;
}
```

Praktično jedina razlika između maločas napisane funkcije "KopirajString" i (prave) funkcije "strcpy" je u tome što funkcija "strcpy", pored toga što obavlja kopiranje, *vraća kao rezultat* pokazivač na prvi element niza u koji se vrši kopiranje (isto svojstvo posjeduje i funkcija "strcat", također prisutna u biblioteci "cstring", koja *nadovezuje* niz znakova koji je zadan drugim parametrom na kraj niza znakova koji je predstavljen prvim parametrom, odnosno vrši dodavanje znakova iza nul-graničnika prvog niza znakova, uz odgovarajuće pomjeranje graničnika). Ovaj primjer ujedno demonstrira da funkcija može kao rezultat vratiti pokazivač. Povratna vrijednost koju vraćaju funkcije "strcpy" i "strcat" najčešće se *ignorira*, kao što smo i mi radili u svim dosadašnjim primjerima. Međutim, ova povratna vrijednost može se nekada korisno upotrijebiti za ulančavanje funkcija "strcpy" i/ili "strcat". Na primjer, posmatrajmo sljedeću sekvencu naredbi (uz pretpostavku da su "recenica", "rijec1", "rijec2" i "rijec3" propisno deklarirani nizovi znakova):

```
std::strcpy(recenica, rijec1);
std::strcat(recenica, rijec2);
std::strcat(recenica, rijec3);
std::cout << recenica;
```

Zahvaljujući činjenici da funkcije "strcpy" i "strcat" vraćaju pokazivač na odredišni niz, ove četiri naredbe možemo kompaktnije zapisati u vidu sljedeće naredbe:

```
std::cout << std::strcat(std::strcat(std::strcpy(recenica rijec1),
  rijec2), rijec3);
```

Treba napomenuti da pomenuto svojstvo funkcija "strcat" i "strcpy" može početnika da dovede u zabludu. Naime, mnogi početnici često isprobaju naredbu poput sljedeće:

```
std::cout << std::strcat("Dobar ", "dan!");                    // NELEGALNO!!!
```

Postoji velika vjerovatnoća da će ova naredba zaista ispisati pozdrav "Dobar dan!" na ekran (uskoro ćemo vidjeti zašto), iz čega početnik može zaključiti da je ona ispravna. Međutim, ovakva naredba je *strogo zabranjena*, jer dovodi do *prepisivanja sadržaja stringovne konstante*. O ovoj temi treba malo detaljnije prodiskutovati, s obzirom da ona može biti izvor brojnih frustracija. Da bismo bolje objasnili šta se ovdje dešava, razmotrimo sljedeću naredbu (u kojoj se, ukoliko će to olakšati razumijevanje, umjesto bibliotečke funkcije "strcpy" može koristiti i maločas napisana funkcija "KopirajString"):

```
std::strcpy("abc", "def");                                  // VEOMA PROBLEMATIČNO!!!
```

Mada će ovu konstrukciju kompajler prihvatiti (u boljem slučaju uz prijavu upozorenja), ona zapravo prepisuje sadržaj memorije u kojem se čuva stringovna konstanta `"abc"` sa sadržajem stringovne konstante `"def"`. Posljedice ovog su posve nepredvidljive. Jedna mogućnost je da će svaka upotreba stringovne konstante `"abc"` u programu dovesti do efekta kao da je upotrijebljena stringovna konstanta `"def"`. Druga mogućnost je da dođe do prekida rada programa od strane operativnog sistema (ovo se može desiti ukoliko kompajler sadržaje stringovnih konstanti čuva u dijelu memorije u kojem operativni sistem ne dozvoljava nikakav upis). Mada nekome efekat zamjene jedne stringovne konstante drugom može djelovati kao "zgodan trik", njegova upotreba je, čak i u slučaju da je konkretan kompajler i operativni sistem dozvoljavaju, *izuzetno loša taktika* koju ne bi trebalo koristiti *ni po koju cijenu* (činjenica da je po standardu jezika C ovakav poziv načelno *dozvoljen* ne znači da ga treba ikada koristiti). Još gore posljedice može imati naredba

```
std::strcpy("abc", "defghijk");                             // FATALNA GREŠKA!!!
```

U ovom slučaju se duža stringovna konstanta `"defghijk"` pokušava prepisati u dio memorije u kojem se čuva kraća stringovna konstanta `"abc"`, što ne samo da će prebrisati ovu stringovnu konstantu, nego će i prebrisati sadržaj nekog dijela memorije za koji ne znamo kome pripada (za stringovnu konstantu `"abc"` rezervirano je tačno onoliko prostora koliko ona zauzima, tako da već prostor koji slijedi može pripadati nekom drugom). Naravno, posljedice su potpuno nepredvidljive i nikada nisu bezazlene. Kao zaključak, nikada ne smijemo stringovnu konstantu proslijediti kao stvarni parametar nekoj funkciji

koja *mijenja sadržaj znakovnog niza* koji joj je deklariran kao formalni parametar (ovo je tako opasna stvar da bolji kompajleri imaju opciju kojom se može uključiti da se takav pokušaj tretira kao sintaksna greška). Kao konkretan primjer, kao prvi argument funkcije "`strcpy`" (odnosno "`KopirajString`") smije se zadavati isključivo niz za koji eksplicitno znamo da mu se sadržaj smije mijenjati (recimo, neka promjenljiva nizovnog tipa), i koji je dovoljno velik da primi sadržaj niza koji se u njega kopira.

Sada postaje jasno šta nije u redu sa pozivom "`std::strcat("Dobar ", "dan!")`". Naime, ovaj poziv bi pokušao da nadoveže stringovnu konstantu `"dan!"` na kraj onog dijela memorije gdje se čuva stringovna konstanta `"Dobar "`, čime ne samo da bi promijenio sadržaj ove stringovne konstante, nego bi i *prebrisao sadržaj memorijskih lokacija koje joj ne pripadaju*. Posljedice ovakvog "napada" mogu se odraziti znatno kasnije u programu, što može biti veoma frustrirajuće. Ako nam operativni sistem odmah prekine izvođenje takvog programa zbog zabranjenih akcija, još možemo smatrati da smo imali sreće. Stoga, možemo zaključiti da funkciju "`strcat`" ima smisla koristiti samo ukoliko je njen prvi argument niz koji već sadrži neki string (eventualno prazan) i koji je dovoljno veliki da može prihvatiti string koji će nastati nadovezivanjem. Možemo istu stvar reći i ovako: korisnici koji poznaju neke druge programske jezike mogu pomisliti da funkcija "`strcat`" *vraća kao rezultat* string koji je nastao nadovezivanjem prvog i drugog argumenta (ne mijenjajući svoje argumente), s obzirom da takve funkcije postoje u mnogim programskim jezicima. Međutim, to nije slučaj sa funkcijom "`strcat`" iz jezika C++: ona *nadovezuje drugi argument na prvi*, pri tome mijenjajući prvi argument. U neku ruku, razlika između funkcije "`strcat`" i srodnih funkcija u drugim programskim jezicima najlakše se može uporediti sa razlikom koja postoji između izraza "`x += y`" i "`x + y`".

Pokazivačka aritmetika može se veoma lijepo iskoristiti zajedno sa funkcijama koje barataju sa klasičnim nul-terminiranim stringovima u stilu jezika C, kao što je maločas napisana funkcija "`KopirajString`", odnosno bibliotečkih funkcija iz biblioteke "`cstring`" poput "`strcpy`", "`strcat`", "`strcmp`" itd. Na primjer, ukoliko je potrebno iz niza znakova "`recenica`" izdvojiti sve znakove od desetog nadalje u drugi niz znakova "`recenica2`", to najlakše možemo uraditi na jedan od sljedeća dva načina (oba su ravnopravna):

```
std::strcpy(recenica2, &recenica[9]);
std::strcpy(recenica2, recenica + 9);
```

Ista tehnika može se iskoristiti i sa funkcijama koje prihvataju proizvoljne vrste nizova kao parametre, odnosno koje prihvataju pokazivače na proizvoljne tipove. Na primjer, pretpostavimo da želimo iz ranije deklariranog niza realnih brojeva "`niz`" ispisati samo drugi, treći i četvrti element (tj. elemente sa indeksima 1, 2 i 3). Nema potrebe da prepravljamo već napisanu funkciju "`IspisiNiz`", s obzirom da ovaj cilj možemo veoma jednostavno ostvariti jednim od sljedeća dva poziva, bez obzira da li je funkcija napisana da prihvata pokazivače ili nizove kao svoj prvi argument, i bez obzira da li je napisana korištenjem indeksiranja ili klizećih pokazivača:

```
IspisiNiz(&niz[1], 3);
IspisiNiz(niz + 1, 3);
```

Ova univerzalnost ostvarena je automatskom konverzijom nizova u pokazivače, kao i mogućnošću da se indeksiranje primjenjuje na pokazivače kao da se radi o nizovima.

Tijesna veza između nizova i pokazivača i činjenica da se nizovi znakova tretiraju donekle različito od ostalih nizova ima kao posljedicu da se i pokazivači na znakove tretiraju neznatno drugačije nego pokazivači na druge objekte. Tako, ukoliko je "`p`" pokazivač na znak (tj. na tip "**`char`**"), izraz "`std::cout << p`" neće ispisati internu reprezentaciju adrese pohranjene u pokazivaču "`p`" (kao što bi vrijedilo za sve druge tipove pokazivača), nego *redom sve znakove iz memorije počev od adrese smještene u pokazivač* "`p`", *pa sve do* "`NUL`" *graničnika* (jasno je da se ova konvencija odnosi samo na jezik C++ a ne na C, s obzirom da jezik C ne posjeduje objekat toka "`cout`"). Na ovaj način je, kao prvo, ostvarena konzistencija sa tretmanom znakovnih nizova od strane objekta izlaznog toka "`cout`", a kao drugo, omogućeni su neki interesantni efekti. Na primjer, sljedeća skupina naredbi

```
char recenica[] = "Tehnike programiranja";
char *p(&recenica[8]);
std::cout << p;
```

ispisaće tekst "programiranja" na ekran. Naime, nakon izvršene dodjele, pokazivač "`p`" pokazuje na deveti znak rečenice smještene u niz "`recenica`" (indeksiranje počinje od nule), a to je upravo prvi

znak druge riječi u nizu "recenica", odnosno početak dijela teksta koji glasi "programiranja". Naravno, umjesto izraza "&recenica[8]" mogli smo pisati i "recenica + 8" (s obzirom da bi se u ovom primjeru upotreba imena "recenica" bez indeksa automatski konvertirala u pokazivač na prvi element niza, nakon čega bi pokazivačka aritmetika odradila ostatak posla). Također, uopće nismo ni morali uvoditi pokazivačku promjenljivu "p": mogli smo prosto pisati

```
char recenica[] = "Principi programiranja";
std::cout << &recenica[8];
```

odnosno

```
char recenica[] = "Principi programiranja";
std::cout << recenica + 8;
```

Isti efekat proizvela bi čak i sljedeća naredba (razmislite zašto):

```
std::cout << "Principi programiranja" + 8;
```

Pokazivači na znakove su jedini pokazivači koji se mogu koristiti sa ulaznim tokom "cin" (jasno je da se i ova napomena odnosi samo na jezik C++). Tako, ako je "p" pokazivač na znak, izvršavanjem izraza "std::cin >> p" znakovi pročitani iz ulaznog toka (do prvog razmaka, u skladu sa ustaljenim ponašanjem operatora ">>") smjestiće se redom u memoriju počev od adrese smještene u pokazivaču "p". Slično vrijedi i za druge konstrukcije za unos, poput "std::cin.getline(p, 100)". Zajedno sa pokazivačkom aritmetikom, ovo se može iskoristiti za razne interesantne efekte. Na primjer, pomoću sekvence naredbi

```
char recenica[100] = "Početak: ", *p(&recenica[9]);
std::cin >> p;
```

sa tastature će se unijeti *jedna riječ*, i smjestiti u niz recenica počev od desetog znaka, odnosno tačno iza teksta "Početak: ". I u ovom primjeru smo mogli izbjeći uvođenje promjenljive "p" i pisati direktno izraz poput "std::cin >> &recenica[9]" ili "std::cin >> recenica + 9". Ukoliko bismo umjesto jedne riječi željeli unijeti *čitavu liniju teksta*, koristili bismo konstrukciju poput "std::cin.getline(p, 91)" odnosno "std::cin.getline(&recenica[9], 91)" ili "std::cin.getline(recenica + 9, 91)" ako pored toga želimo i da izbjegnemo uvoenje promjenljive "p". Maksimalna dužina 91 određena je tako što je od maksimalnog kapaciteta od 100 znakova odbijen prostor koji je već rezerviran za tekst "Početak: ".

Pokazivači i pokazivačka aritmetika omogućavaju prilično moćne trikove, koje bi inače bilo znatno teže realizirati. S druge strane, pokazivači mogu biti i *veoma opasni*, jer lako mogu "odlutati" u neko nepredviđeno područje memorije, što omogućava da pomoću njih indirektno promijenimo praktički *bilo koji dio memorije u kojem su dopuštene izmjene* (posljedice ovakvih intervencija su uglavnom nepredvidljive i često fatalne). Na primjer, već smo rekli da je smisao pokazivačke aritmetike precizno definiran *samo za pokazivače koji pokazuju na neki element niza*, i to samo pod uvjetom da novodobijeni pokazivač *ponovo pokazuje na neki element niza*, ili eventualno *tačno iza posljednjeg elementa niza*. Ukoliko se ne pridržavamo ovih pravila, pokazivači mogu "odlutati" u nedozvoljena područja memorije. Razmotrimo, na primjer, sljedeću sekvencu naredbi:

```
int a(5), *p(&a);
p++;                          // NELEGALNO: p ne pokazuje na element nekog niza!
*p = 3;
```

Ovdje smo prvo deklarirali cjelobrojnu promjenljivu "a" koju smo inicijalizirali na vrijednost "5", a zatim pokazivačku promjenljivu "p" koju smo inicijalizirali tako da pokazuje na promjenljivu "a" (odnosno, ona sada sadrži adresu promjenljive "a"). Nakon toga je na promjenljivu "p" primijenjen operator "++". Na šta sada pokazuje promjenljiva "p"? Strogo uzevši, odgovor *uopće nije definiran u skladu sa pravilima koja po standardu vrijede za pokazivačku aritmetiku*, ali u skladu sa načinom kako se pokazivačka aritmetika implementira u većini kompajlera, "p" će gotovo sigurno pokazivati na prostor u memoriji koji se nalazi *neposredno iza prostora* koji zauzima promjenljiva "a". Nakon toga će dodjela "*p = 3" *na to mjesto u memoriji* upisati vrijednost "3". Međutim, šta se nalazi "na tom mjestu u memoriji"? Odgovor je nepredvidljiv: možda neka druga promjenljiva (ovo je najvjerovatnija mogućnost), možda ništa pametno, a možda neki podatak od vitalne važnosti za rad operativnog sistema! Samim tim, *posljedica ove dodjele je nepredvidljiva*. U svakom slučaju, dodjelom "*p = 3", mi smo zapravo "napali" memorijski prostor čiju svrhu ne znamo, što je strogo zabranjeno. Za pokazivač koji je

"odlutao" tako da više ne znamo na šta pokazuje kažemo da je *divlji pokazivač* (engl. *wild pointer*). To ne mora značiti da ne znamo koja je adresa pohranjena u njemu, nego samo da ne znamo *kome pripada adresa koju on sadrži*. Divlji pokazivači *nikada se ne bi trebali pojaviti u programu*. Nažalost, greške koje nastaju usljed divljih pokazivača veoma se teško otkrivaju, a prilično ih je lako napraviti. Zbog toga se u literaturi često citira da su "pokazivači zločesti" (engl. *pointers are evil*), i oni predstavljaju drugi tipični "zločesti" koji postoji u jezicima C i C++ (prvi "zločesti" koncept su *nizovi*, koji postaju "zločesti" ukoliko indeks niza "odluta" izvan dozvoljenog opsega). Smatra se da oko 90% svih fatalnih grešaka u današnjim profesionalnim programima nastaje upravo zbog ilegalnog pristupa memoriji preko divljih pokazivača!

Treba primijetiti da je i pokazivač koji pokazuje tačno iza posljednjeg elementa niza *također divlji pokazivač*, jer i on pokazuje na lokaciju koja ne pripada nizu (niti se zna kome ona zapravo pripada). Međutim, standard jezika C++ kaže da je takav pokazivač *legalan*. Nije li ovo kontradikcija? Stvar je u tome što je dozvoljeno *generirati* odnosno *kreirati* pokazivač koji pokazuje iza posljednjeg elementa niza. Pored toga, garantira se da je on *veći* od svih pokazivača koji pokazuju na elemente tog niza. Ipak, to još uvijek ne znači da takav pokazivač smijemo *dereferencirati*. Pokušaj njegovog dereferenciranja također može imati nepredvidljive posljedice. Međutim, njega *bar smijemo kreirati bez opasnosti*. Druge vrste divljih pokazivača *ponekad ne smijemo ni kreirati* – može se desiti da i sâmo njihovo kreiranje dovede do kraha, čak i ukoliko se oni uopće ne dereferenciraju (to se, u nekim rijetkim slučajevima, može desiti npr. ukoliko primjenom pokazivačke aritmetike pokazivač odluta ispred prvog elementa niza). Dalje, sve operacije sa njima (npr. poređenje, itd.) *potpuno su nedefinirane*. U tom smislu, pokazivač koji pokazuje neposredno iza elementa niza legalan je u smislu da ga *smijemo kreirati* (tj. njegovo kreiranje garantirano neće izazvati krah) i *smijemo ga poredити* sa drugim pokazivačima, jedino ga *ne smijemo dereferencirati*. Sa drugim divljim pokazivačima *ne smijemo raditi apsolutno ništa*. Čak i sâm pokušaj njihovog *kreiranja* (koji tipično nastaje neispravnom upotrebom pokazivačke aritmetike) može biti fatalan po program!

U vezi sa divljim pokazivačima neophodno je naglasiti jednu nezgodnu osobinu: svi pokazivači su neposredno nakon deklaracije divlji (osim u slučajevima kada se uporedo sa deklaracijom vrši i inicijalizacija), sve dok im se eksplicitno ne dodijeli vrijednost! Naime, kao i sve ostale promjenljive, njihova vrijednost je *nedefinirana* neposredno nakon deklaracije (osim ukoliko se istovremeno vrši i inicijalizacija), sve do prve dodjele. Tako je i sa pokazivačima: njihova početna vrijednost je u odsustvu inicijalizacije *nedefinirana*, tako da se sve do prve dodjele ne zna ni na šta pokazuju. Na ovo smo se već ukratko osvrnuli prilikom razmatranja raznih *neispravnih* realizacija funkcije "`Razmijeni`". Recimo, sljedeća sekvenca naredbi ima nepredvidljive posljedice jer "napada" nepoznati dio memorije, s obzirom da se ne zna na šta pokazivač "`p`" pokazuje:

```
int *p;
*p = 10;                    // NELEGALNO: Gdje se smješta ovaj broj 10???
```

Sličnu situaciju imamo u sljedećem katastrofalno opasnom neispravnom primjeru, koji se veoma često može sresti kod početnika:

```
char *p;
std::cin >> p;              // NELEGALNO: Gdje se smješta uneseni tekst???
std::cout << p;
```

Mada onome ko isproba navedeni primjer može izgledati da on radi korektno (odnosno da ispisuje na ekran riječ prethodno unesenu sa tastature), on sadrži katastrofalnu grešku. Naime, kako je "`p`" neinicijaliziran pokazivač, njegov sadržaj je nepredvidljiv, pa je i potpuno nepredvidljivo gdje će se uopće u memoriji smjestiti znakovi pročitani sa tastature! Može se desiti da se program koji sadrži ovakve naredbe "sruši" tek nakon više sati, ukoliko se ispostavi da su smješteni znakovi "upropastili" neke bitne podatke u memoriji. Može se desiti da u startu "`p`" pokazuje na neki dio memorije zabranjen za upis od strane operativnog sistema. U tom slučaju program će se srušiti odmah (zapravo, operativni sistem će prekinuti njegovo izvršavanje). U tom slučaju čak možemo reći da smo imali sreće!

Opisani problem nije "specijalnost" samo jezika C++. Sa sličnim problemom se početnici susreću u jeziku C. Naime, prethodni problematični primjer, napisan u duhu jezika C, izgledao bi otprilike ovako:

```
char *p;
std::scanf("%s", p);        // NELEGALNO: Gdje se smješta uneseni tekst???
std::printf("%s", p);
```

Da li ste ikada napisali nešto poput ovoga? Sasvim je vjerovatno da jeste. Da li ste nakon toga bili ubjeđeni da se *smije ovako pisati*? Ukoliko ste već napisali tako nešto, skoro ste bili sigurni da smije. Sad znate ne samo da ne smije, nego da je u pitanju *katastrofalna greška*. Čovjek uči na greškama i uči dok je živ, ali je žalosno što ovakve primjere navode (misleći da su ispravni) čak i neki pojedinci koji bi druge trebali učiti osnovama programiranja...

Postoji jedan specijalni pokazivač koji se naziva *nul-pokazivač* i po konvenciji se smatra da on *ne pokazuje ni na šta*. Pokazivačima se obično dodjeljuje nul-pokazivač onog trenutka kada želimo reći da u tom trenutku pokazivač više ne pokazuje ni na šta korisno. Isto tako, pokazivač se odmah prilikom kreiranja može inicijalizirati na nul-pokazivač (što se toplo preporučuje) ukoliko u tom trenutku još ne znamo na šta on treba pokazivati. Sve do pojave standarda C++11, za nul-pokazivače se prosto koristio simbol *nula* (0), kao za običan broj nula. Drugim riječima, ukoliko smo pokazivaču "`p`" željeli pridružiti nul-pokazivač, prosto smo vršili dodjelu poput "`p = 0`". Isto tako, ukoliko smo željeli inicijalizirati pokazivač "`p`" na nul-pokazivač prilikom deklaracije, morali smo pisati deklarasije poput sljedećih:

```
int *p = 0;                 // U duhu jezika C...
int *p(0);                  // U C++ stilu...
```

Međutim, korištenje nule kao onzake za nul-pokazivač je pomalo konfuzno, jer daje iluziju da se pokazivačima mogu dodjeljivati *brojevi*, a ne mogu (osim *nule*, a i ta nula nije zaista nula nego nul-pokazivač). Štaviše, prema C++ standardu nul-pokazivač uopće ne mora interno biti predstavljen brojem nula (tj. uopće u njemu ne mora biti adresa nula, mada u većini implementacija jeste). Pokušaj da se smanji pomenuta konfuzija bio je uvođenje simboličke konstante "`NULL`" koja je definirana u raznim bibliotekama (poput "`cstring`", "`cstdlib`" itd.) koje čine sastavni dio jezika C i C++ (u jeziku C je ova konstanta *pokazivačkog* a ne *brojčanog tipa*, što je učinjeno sa diljem da se spriječi da se ona koristi u ne-pokazivačkim kontekstima, ali je iz nekih tehničkih razloga u jeziku C++ uzeto da je ona bukvalno sinonim za broj 0). Ipak, mnogi su radije koristili običan broj 0 umjesto konstante "`NULL`", da ne bi pravili nepotrebne pretpostavke o uključenosti pojedinih biblioteka u program. Situacija je konačno raščišćena u standardu C++11 jezika C++, kada je uvedena ova ključna riječ "**`nullptr`**" koja označava nul-pokazivač. Nju bi sada trebalo uvijek koristiti umjesto broja 0 za modeliranje nul-pokazivača. Vrijednost "**`nullptr`**" je *pokazivačke prirode* i ne može se koristiti u ne-pokazivačkim kontekstima. Tako, u skladu sa C++11 standardom, inicijalizaciju pokazivača "`p`" na nul-pokazivač trebamo izvršiti na jedan od sljedećih načina:

```
int *p = nullptr;           // C-ovski stil, ne preporučuje se...
int *p(nullptr);            // Klasični C++ stil
int *p{nullptr};            // Jednoobrazni stil
```

Konvencija da se pokazivač koji ne pokazuje ni na šta eksplicitno postavlja na nul-pokazivač može učiniti neke divlje pokazivače "manje divljim". Oni su i dalje divlji u smislu da ne pokazuju ni na šta smisleno, tako da ih ne smijemo dereferencirati. Međutim, njihov sadržaj nije nepredvidljiv, nego je *tačno određen*, tako da programski možemo testirati (poređenjem sa nul-pokazivačem) da li on pokazuje na nešto smisleno ili ne. Pored očiglednih testova tipa "`p == nullptr`" odnosno "`p != nullptr`" (prihvataju se i testovi "`p == 0`" odnosno "`p != 0`" koji su se koristili prije nego što je ključna riječ "**`nullptr`**" uvedena), jezici C i C++ dozvoljavaju i sâmo navođenje izraza "`!p`" odnosno "`p`" unutar uvjeta "**`if`**" ili "**`while`**" naredbe (pri tome se svi pokazivači automatski tretiraju kao logička vrijednost "tačno", osim nul-pokazivača koji se tretira kao "netačan").

Nul-pokazivači se također često vraćaju kao rezultati iz funkcija koje kao rezultat vraćaju pokazivače, u slučaju da nije moguće vratiti neku drugu smislenu vrijednost. Na primjer, biblioteka "`cstring`" sadrži veoma korisnu funkciju "`strstr`" koja prihvata dva nul-terminirana stringa kao parametre. Ova funkcija ispituje da li se drugi string sadrži u prvom stringu (npr. string "je lijep" sadrži se u stringu "danas je lijep dan"). Ukoliko se nalazi, funkcija vraća kao rezultat *pokazivač na znak* unutar prvog stringa od kojeg počinje tekst identičan drugom stringu (u navedenom primjeru, to bi bio pokazivač na prvu pojavu slova "j" u stringu "danas je lijep dan"). U suprotnom, funkcija kao rezultat vraća *nul-pokazivač*. Sljedeći primjer demonstrira kako se ova funkcija može iskoristiti (obratite pažnju na igre sa pokazivačkom aritmetikom):

```
char recenica[100], fraza[100];
std::cin.getline(recenica, sizeof recenica);
std::cin.getline(fraza, sizeof fraza);
char *pozicija(std::strstr(recenica, fraza));
```

```
if(pozicija != nullptr)
  std::cout << "Navedena fraza se nalazi u rečenici počev od "
    << pozicija - recenica + 1 << ". znaka\n";
else
  std::cout << "Navedena fraza se ne nalazi u rečenici\n";
```

Pored funkcije "strstr", biblioteka "cstring" sadrži i sličnu funkciju "strchr", samo što je njen drugi parametar *znak* a ne string (prihvata se i brojčana vrijednost, sa značenjem ASCII šifre). Ona vraća pokazivač na prvu pojavu navedenog znaka u stringu, ili nul-pokazivač ukoliko takvog znaka nema. Ova funkcija se može zgodno iskoristiti da se dobije pokazivač na "NUL" graničnik stringa (za tu svrhu, funkciji "strchr" treba proslijediti nulu kao drugi parametar). Biblioteka "cstring" sadrži još mnoštvo korisnih funkcija koje kao rezultat vraćaju pokazivače, koje nećemo opisivati s obzirom da bi njihov opis zauzeo previše prostora. Napomenimo da su sve ove funkcije namijenjene isključivo za rad sa klasičnim, nul-terminiranim stringovima, tj. stringovima na način kako ih tretira jezik C. Funkcije slične namjene postoje i za dinamičke stringove (tj. objekte tipa "string" uvedene u jeziku C++), samo što se koriste na nešto drugačiji način i one nikada ne vraćaju pokazivače kao rezultate (pokazivače i dinamičke stringove nije dobro koristiti u istom kontekstu, jer su oni konceptualno različiti). Zainteresirani se upućuju na mnogobrojnu literaturu koja detaljno obrađuje standardnu biblioteku "string" i operacije sa dinamičkim stringovima. Vrijedi još napomenuti da je, u slučaju potrebe, legalno uzeti adresu nekog elementa dinamičkog stringa i sa tako dobijenim pokazivačem (koji je tipa *pokazivač na znak*, isto kao i u slučaju klasičnih nul-terminiranih stringova) koristiti pokazivačku aritmetiku (tj. garantira se da se elementi dina,ičkih stringova također čuvaju u memoriji jedan za drugim). Tako dobijene adrese su garantirano ispravne sve dok se dužina stringa ne promijeni, nakon čega više ne moraju biti (s obzirom da promjena veličine stringa može dovesti do pomjeranja njegovih elemenata u memoriji).

Pošto je cilj ovog dodatka detaljno upoznavanje sa pokazivačima i njihovim svojstvima, recimo još nekoliko riječi o pokazivačima na znakove. Interesantno je da se pokazivači na konstantne znakove mogu inicijalizirati tako da pokazuju na neku stringovnu konstantu, kao da se radi o nizovima znakova, koji se mogu inicijalizirati na sličan način. Na primjer:

```
const char *p = "Ovo je neki string...";
std::cout << p;
```

Ovdje je također moguće (i preporučljivo) koristiti sintaksu sa zagradama (od verzije C++11 dopuštene su i vitičaste zagrade, radi veće jednoobraznosti):

```
const char *p("Ovo je neki string...");
std::cout << p;
```

Ovakvim pokazivačima je čak moguće i naknadno "dodijeliti" stringovne konstante, pri čemu ovakva "dodjela" naravno ne dodjeljuje samu stringovnu konstantu pokazivačkoj promjenljivoj (ona nema "prostora" da primi cijeli niz znakova u sebe), već samo *adresu stringovne konstante*. Međutim, bez obzira na tu činjenicu, ovakva "poludodjela" može, zajedno sa specijalnim tretmanom pokazivača na znakove, proizvesti interesantne efekte. Na primjer, sljedeći primjer je posve legalan:

```
const char *recenica;
recenica = "Ja sam prva rečenica...";
std::cout << recenica << std::endl;
recenica = "A ja sam druga rečenica...";
std::cout << recenica << std::endl;
```

S druge strane, slična konstrukcija *nije moguća* sa običnim nizovima znakova, nego se moraju koristiti ili dinamički stringovi (tj. objekti tipa "string", uvedeni u jeziku C++), ili konstrukcije poput sljedeće:

```
char recenica[100];
std::strcpy(recenica, "Ja sam prva rečenica...");
std::cout << recenica << std::endl;
std::strcpy(recenica, "A ja sam druga rečenica...");
std::cout << recenica << std::endl;
```

Veoma je važno uočiti suštinsku razliku između dva prethodno navedena primjera. U prvom primjeru, "recenica" je *pokazivač*, koji prvo *pokazuje* na jednu stringovnu konstantu, a zatim na drugu. S druge strane, u drugom primjeru, "recenica" je *niz znakova* (za koji je rezerviran *prostor* od 100 mjesta za znakove), u koji se prvo *kopira* jedna stringovna konstanta, a zatim druga. Ova razlika je veoma

važna, mada na prvi pogled djeluje nebitna, jer je krajnji efekat isti. Međutim, kako je u prvom primjeru "recenica" pokazivač na *konstantne znakove*, izmjena sadržaja na koji on pokazuje nije moguća. Stoga bi u primjeru poput sljedećeg prevodilac prijavio grešku:

```
const char *ime;
ime = "Elma";
ime[0] = 'A';              // GREŠKA: "ime" pokazuje na KONSTANTNE znakove!
cout << ime;
```

Na ovome bi sva priča o pokazivačima na znakove mogla da završi da nije jedne nevolje. Naime, počev od standarda C++98 jezika C++ nadalje, kompajleri *ne bi trebali da dozvole* da se običnim pokazivačima na (nekonstantne) znakove dodjeljuju adrese stringovnih konstanti. Kažemo "ne bi trebali", jer je činjenica da gotovo svi raspoloživi kompajleri za C++ *dozvoljavaju* takve dodjele (s obzirom da su one dugi niz godina *načelno* bile dozvoljene, bez obzira što nisu smjele da budu – po ranijim standardima stringovne konstante zapravo *uopće nisu bile konstante*). Stoga će sljedeći primjer "proći nekažnjeno" kroz većinu raspoloživih C++ kompajlera (i praktično *sve* kompajlere za C, s obzirom da jezik C ima znatno "blaži" tretman konstanti, odnosno ne tretira "konstantnost" tako strogo kao jezik C++):

```
char *ime;
ime = "Elma";              // Ovo ne bi trebalo da prođe, međutim...
ime[0] = 'A';              // Ova dodjela je vrlo problematična!
std::cout << "Elma";
```

"Repertoar" mogućih posljedica ove skupine naredbi je raznovrstan. Možda se program "sruši" zbog toga što "ime" pokazuje na stringovnu konstantu koja je možda smještena u dio memorije koji je zabranjen za upis. Ukoliko to nije slučaj, gotovo sigurno da ćemo na ekranu umjesto "Elma" imati ispisano "Alma", bez obzira što naredba ispisa eksplicitno ispisuje tekst "Elma". Naime, obje pojave stringovne konstante "Elma" gotovo sigurno se u memoriji čuvaju na *istom mjestu*, pa će se izmjena sadržaja stringovne "konstante" (izmjena konstante je već sama po sebi apsurdna) odraziti na svaku pojavu te stringovne konstante. Navedimo još jedan katastrofalan primjer koji se može sresti kod početnika (i koji kompajler *ne bi trebao* uopće da dozvoli):

```
char *recenica;
recenica = "Ja sam recenica...";        // Ni ovo ne bi trebalo proći...
...
std::cin >> recenica;                   // Ovo ima kobne posljedice!
```

Postavlja se pitanje *gdje će se u memoriju smjestiti* niz znakova koji unesemo sa tastature? Odgovor je jasan: na najgore moguće mjesto – preko znakova stringovne konstante "Ja sam rečenica..."! Pored toga, ukoliko je uneseni niz znakova *duži* od dužine te stringovne konstante, višak znakova će "pojesti" i sadržaj memorije koji se nalazi iza ove stringovne konstante! Ukoliko nam operativni sistem odmah prekine ovakav program, imamo mnogo sreće. Gora varijanta je da program počne neispravno raditi tek nakon nekoliko sati, kad mu zatreba sadržaj uništenog dijela memorije!

Iz svega što je rečeno bitno je shvatiti da se dodjela stringovnih konstanti pokazivačima na znakove nipošto ne smije shvatiti kao lijepa i elegantna zamjena za funkciju "strcpy" (kako se propagira u nekim udžbenicima za jezik C), nego kao nešto što treba koristiti sa izuzetnim oprezom, i to obavezno sa kvalifikatorom "**const**". Na taj način će nas kompajler upozoriti pokušamo li izvršiti nedozvoljeni "napad" na memoriju. Napomenimo da su ovi komentari uglavnom namijenjeni onima koji imaju određena predznanja u jeziku C i žele da ta znanja nastave koristiti u jeziku C++. S druge strane, onima kojima programski jezik C nije "jača strana", kao i onima kojima je C++ prvi programski jezik sa kojim se susrećemo, savjetuje se da za bilo kakav ozbiljniji rad sa stringovima ne koristite niti nul-terminirane nizove znakova niti pokazivače na znakove, već dinamičke stringove, odnosno tip "string"!

Mada je rečeno da pokazivačima nije moguće eksplicitno dodjeljivati brojeve, niti je moguće pokazivaču na jedan tip dodijeliti pokazivač na drugi tip, postoji indirektna mogućnost da se ovakva dodjela ipak izvrši, pomoću operatora za *pretvorbu tipova* (*typecasting* operatora). Na primjer, neka imamo sljedeće deklaracije:

```
int *pok_na_int;
double *pok_na_double;
```

Tada pretvorba tipova dozvoljava, da na sopstveni rizik, izvršimo i ovakve dodjele:

```
pok_na_int = (int *)3446;                          // VRLO RIZIČNO!
pok_na_double = (double *)pok_na_int;              // VRLO RIZIČNO!
```

Prikazane konverzije tipova koriste sintaksu i stil jezika C. Ova sintaksa radi i u jeziku C++, mada se u jeziku C++ preporučuje druga sintaksa, u kojoj se javlja ključna riječ "**reinterpret_cast**" (koja djeluje slično kao i "**static_cast**", samo služi za drastične pretvorbe pokazivačkih tipova):

```
pok_na_int = reinterpret_cast<int *>(3446);                // VRLO RIZIČNO!
pok_na_double = reinterpret_cast<double *>(pok_na_int);    // VRLO RIZIČNO!
```

Bez obzira koja se sintaksa koristi, ovakve dodjele su veoma opasne. Prvom dodjelom smo eksplicitno postavili pokazivač "pok_na_int" da pokazuje na adresu 3446 (uz pretpostavku da računar na kojem radimo koristi linearni memorijski model), mada ne znamo šta se tamo nalazi. Drugom dodjelom smo postavili pokazivač "pok_na_double" da pokazuje na istu adresu na koju pokazuje "pok_na_int", što je također veoma opasno. Naime, ako se na nekoj lokaciji nalazi cijeli broj, tamo ne može u isto vrijeme biti realni broj (ne smijemo zaboraviti da se cijeli i realni brojevi zapisuju na posve različite načine u računarskoj memoriji, čak i kada imaju *istu vrijednost*). Pored toga, na različite tipove pokazivača pokazivačka aritmetika ne djeluje isto. Stoga su ovakve pretvorbe ostavljene samo onima koji *veoma dobro znaju šta njima žele da postignu*. Treba napomenuti da se pretvorbom broja u pokazivač dobija *pokazivač*, koji se dalje može dereferencirati kao i svaki drugi pokazivač. Stoga su naredbe poput sljedeće sasvim legalne (obje su ekvivalentne, s tim što druga radi samo u jeziku C++):

```
*(int *)3446 = 50;                                 // VRLO RIZIČNO!
*reinterpret_cast<int *>(3446) = 50;               // VRLO RIZIČNO!
```

Ove naredbe će na adresu 3446 u memoriju smjestiti broj 50 (posljedice ovakvih akcija preuzima programer, i u "normalnim" programima ih ne treba koristiti). Jezici C i C++ su rijetki jezici koji omogućavaju ovakvu slobodu u pristupima resursima računara, što ih čini idealnim za pisanje sistemskog softvera (naravno, za tu svrhu treba izuzetno dobro poznavati arhitekturu računara i operativnog sistema za koji se piše program). Stoga, ukoliko želite koristiti ovakve konstrukcije, provjerite da li ste *sigurni* da znate šta radite. Ukoliko utvrdite da ste sigurni, provjerite da li ste *sigurno sigurni*. Na kraju, ukoliko ste sigurni da ste sigurno sigurni, opet *ne radite to ako zaista ne morate*!

Već smo vidjeli da je, zahvaljujući pokazivačkoj aritmetici, funkcije koje barataju sa nizovima, poput ranije napisane funkcije "IspisiNiz", moguće iskoristiti tako da se izvrše samo nad *dijelom niza*, koji ne počinje nužno od prvog elementa. Međutim, sve takve funkcije bile su *heterogene* po pitanju svojih parametara, odnosno njihovi parametri bili su različitih tipova (na primjer, jedan parametar funkcije "IspisiNiz" bio je sam niz, dok je drugi parametar bio broj elemenata koje treba ispisati). Za mnoge primjene je praktičnije imati funkcije sa *homogenom strukturom parametrara*, odnosno funkcije čiji su parametri *istog tipa*. Na primjer, funkcija "IspisiNiz" mogla bi se napisati tako da njeni parametri budu pokazivači na *prvi* i *posljednji* element koji će se ispisati. Ovako napisane funkcije su često ne samo jednostavnije za korištenje, već i jednostavnije za implementaciju, jer se na taj način može efikasno koristiti pokazivačka aritmetika. Zbog nekih tehničkih razloga, bolje je umjesto pokazivača na posljednji element koristiti pokazivač *iza posljednjeg elementa*. Na primjer, upravo takva je sljedeća verzija funkcije "IspisiNiz", u kojoj parametri "pocetak" i "iza_kraja" predstavljaju respektivno pokazivač na prvi element niza koji treba ispisati i pokazivač koji pokazuje iza posljednjeg elementa koji treba ispisati:

```
void IspisiNiz(double *pocetak, double *iza_kraja) {
  for(double *p = pocetak; p < iza_kraja; p++)
    std::cout << *p << std::endl;
}
```

Alternativno, mogli smo čak i izbjeći korištenje dodatne promjenljive "p", recimo ovako:

```
void IspisiNiz(double *pocetak, double *iza_kraja) {
  while(pocetak < iza_kraja) std::cout << *pocetak++ << std::endl;
}
```

U svakom slučaju, ukoliko želimo na primjer ispisati elemente niza realnih brojeva "niz" sa indeksima 3, 4, 5, 6 i 7, možemo koristiti sljedeći poziv:

```
IspisiNiz(niz + 3, niz + 8);
```

Također, ukoliko isti niz ima "n" elemenata, tada čitav niz možemo ispisati pozivom poput

```
IspisiNiz(niz, niz + n);
```

Iz posljednjeg primjera je ujedno vidljivo zbog čega je praktično da drugi parametar pokazuje iza posljednjeg elementa, a ne tačno na njega (ovo nije jedini razlog). Istu funkciju bismo mogli iskoristiti i za ispis elemenata vektora (ovdje se misli na tip "vector" uveden u jeziku C++), ukoliko eksplicitno uzmemo adrese pojedinih elemenata pomoću operatora "&". Na primjer, sljedeća dva poziva su analogna prethodnim pozivima za slučaj kada umjesto niza "niz" treba ispisati elemente vektora "v" (u istom opsegu kao u prethodnim primjerima):

```
IspisiNiz(&v[3], &v[8]);
IspisiNiz(&v[0], &v[n]);
```

Kasnije ćemo vidjeti da standardna biblioteka "algorithm" posjeduje čitavo mnoštvo funkcija za manipulaciju nizovima i vektorima (i drugim srodnim strukturama podataka koje ćemo kasnije obrađivati) koje koriste upravo ovakvu konvenciju o parametrima koja je iskorištena u posljednjoj verziji funkcije "IspisiNiz".

Interesantno je napomenuti da postoje ne samo pokazivači na jednostavne objekte, kao što su cijeli i realni brojevi ili znakovi, već je moguće napraviti pokazivače na praktično *bilo koje objekte*. Tako npr. postoje *pokazivači na nizove*, *pokazivači na funkcije* pa i *pokazivači na pokazivače* (tzv. *dvojni pokazivači*). Također se mogu napraviti *nizovi pokazivača*, *reference na pokazivače* (samo u jeziku C++) i razne druge komplicirane strukture. Sa nekim od ovakvih složenijih pokazivačkih tipova upoznaćemo se kasnije u okviru ovog kursa.

Izlaganje o prostim pokazivačkim tipovima završićemo kratkim opisom tzv. *generičkih* odnosno *netipiziranih* (engl. *typeless*) *pokazivača*. Ovi pokazivači se jako mnogo koriste u jeziku C, dok je njihova upotreba u jeziku C++ danas svedena na najnužniji minimum (uglavnom samo za potrebe pravljenja objekata koji u sebi mogu čuvati druge objekte različitih tipova). Ipak, njihov opis može pomoći razumijevanju prave prirode pokazivačkih tipova. Generički pokazivači se definiraju kao pokazivači na tip "**void**", i za njih se smatra da je *nepoznato* na koji tip tačno pokazuju. Stoga se oni ne mogu dereferencirati, niti se sa njima može vršiti pokazivačka aritmetika. Zapravo, ovi pokazivači su još ograničeniji: sa njima se ne može raditi *praktično ništa* sve dok se pomoću operatora za pretvorbu tipa ne pretvore u pokazivač na neki konkretan tip. Međutim, generičkom pokazivaču se može dodijeliti *ma koji drugi pokazivač*, i funkcija koja kao formalni parametar očekuje generički pokazivač prihvatiće kao stvarni parametar *bilo koji pokazivač*. Stoga su se generički pokazivači intenzivno koristili prije nego što su u jezik C++ uvedene generičke (šablonske) funkcije, o kojima ćemo govoriti kasnije u toku ovog kursa. Naime, oni su ranije bili *jedini način* da se naprave funkcije koje mogu prihvatati nizove *različitih tipova* kao parametre (u jeziku C oni su i dalje jedini način da se naprave funkcije koje, makar u izvjesnoj mjeri, posjeduju neka svojstva generičkih funkcija).

Razmotrimo koncept generičkih pokazivača na jednom konkretnom primjeru. Neka je potrebno napisati funkciju "KopirajNiz" koja posjeduje tri parametra "odredisni", "izvorni" i "br_elemenata" i koja kopira "br_elemenata" elemenata iz niza "izvorni" u niz "odredisni". Neka je dalje tu funkciju potrebno napisati tako da radi sa nizovima čiji su elementi proizvoljnog tipa. Uz pomoć generičkih funkcija, kakve danas postoje u jeziku C++, takvu funkciju je posve lako napisati, što ćemo vidjeti uskoro u okviru ovog kursa. Međutim, na ovom mjestu nas zanima kako se isti problem može donekle riješiti bez upotrebe generičkih funkcija. Za tu svrhu će nam poslužiti *generički pokazivači*. Naime, ako bismo formalne parametre "odredisni" i "izvorni" deklarirali tako da budu generički pokazivači (odnosno, pokazivači na "**void**"), takva funkcija bi kao odgovarajuće stvarne parametre prihvatala bilo kakav pokazivač, pa samim tim (zbog automatske konverzije nizova u pokazivače) i *bilo kakav niz*. Dakle, takva funkcija bi zaista prihvatala *nizove proizvoljnog tipa* kao stvarne parametre. Međutim, sa generičkim pokazivačima se ne može raditi ništa dok se ne izvrši njihova pretvorba u pokazivač na neki konkretan tip. Sad se javlja prirodno pitanje *u koji tip izvršiti njihovu pretvorbu*. Naime, nijedna klasična funkcija ne može ni na kakav način saznati kojeg su tipa njeni stvarni parametri (generičke funkcije to *mogu*, ali ovdje ne govorimo o njima). Stoga je najprirodnije izvršiti njihovu konverziju u pokazivače na tip "**char**", s obzirom da je tip "**char**" *najelementarniji tip podataka* (svaki podatak tipa "**char**" uvijek zauzima *tačno jedan bajt*), a nakon toga obaviti kopiranje niza kao da se radi o običnom *nizu bajtova* (odnosno, obaviti kopiranje nizova *bajt po bajt*, onako kako su oni pohranjeni u memoriji). Međutim,

tada se umjesto *broja elemenata* koje treba kopirati, kao parametar mora zadavati *broj bajtova* koje želimo kopirati. Tako napisana funkcija "`KopirajNiz`" mogla bi izgledati recimo ovako:

```
void KopirajNiz(void *odredisni, void *izvorni, int br_bajtova) {
  for(int i = 0; i < br_bajtova; i++)
    ((char *)odredisni)[i] = ((char *)izvorni)[i];
}
```

Sad, ako na primjer želimo kopirati 10 elemenata niza realnih brojeva "`niz1`" u niz "`niz2`", mogli bismo koristiti sljedeći poziv:

```
KopirajNiz(niz2, niz1, 10 * sizeof(double));
```

Primijetimo kako je ovdje upotrijebljen operator "**`sizeof`**", sa ciljem pretvaranja broja elemenata u broj bajtova (podsjetimo se da operator "**`sizeof`**" daje kao rezultat broj bajtova koji zauzima određeni tip ili izraz). Naravno, umjesto izraza "**`sizeof(double)`**" može se koristiti i izraz "**`sizeof niz1[0]`**" čime poziv funkcije postaje neovisan od stvarnog tipa elemenata niza. U suštini, izraz "`10 * sizeof(double)`" bi se čak mogao prosto zamijeniti sa "**`sizeof niz1`**", jer ono što nas zanima je zapravo broj bajtova koje zauzima niz "`niz1`". Vidimo da smo, na izvjestan način, uz pomoć generičkih pokazivača uspjeli napraviti funkciju koja kao argumente prihvata nizove proizvoljnih tipova, ali po cijenu da umjesto broja elemenata koji se kopiraju moramo zadavati broj bajtova koji se kopiraju. Uskoro ćemo vidjeti da to nije jedini nedostatak ovakvog pristupa.

Radi boljeg razumijevanja, navedimo i još jedan nešto složeniji primjer koji koristi generičke pokazivače. Pretpostavimo da treba napisati funkciju "`IzvrniNiz`" sa dva parametra "`niz`" i "`br_elemenata`" koja izvrće "`br_elemenata`" elemenata niza "`niz`", čiji su elementi proizvoljnog tipa, u obrnuti poredak. Ponovo, ovaj problem je vrlo lako rješiv uz pomoć generičkih funkcija, ali na ovom mjestu nas ne zanima takvo rješenje. Ovdje je cilj pokazati kako se ovaj problem može riješiti uz pomoć generičkih pokazivača. Jasno je da parametar "`niz`" treba biti generički pokazivač. Međutim, ovdje se javlja dodatni problem u odnosu na funkciju iz prethodnog primjera. Da bismo izvrnuli elemente niza u obrnuti poredak, ne smijemo prosto izvrnuti *bajtove* od kojih je niz formiran u obrnuti poredak. Naime, ukoliko to učinimo, izvrnućemo i bajtovsku strukturu *svakog individualnog elementa niza*, što naravno ne smijemo uraditi. Dakle, bajtovska struktura svakog individualnog elementa niza *ne smije se narušiti*. Međutim, funkcija ne može ni na kakav način saznati tip elemenata niza, odakle slijedi da ne može saznati ni *koliko bajtova zauzima svaki od elemenata niza*. Ukoliko malo bolje razmislimo o svemu, zaključićemo da se problem ne može riješiti ukoliko u funkciju ne uvedemo *dodatni parametar* (nazvan recimo "`vel_elementa`") koji govori upravo *koliko bajtova zauzima svaki od elemenata niza*. Uz pomoć ovakvog dopunskog parametra, tražena funkcija se može napraviti uz malo "igranja" sa pokazivačkom aritmetikom. Jedno od mogućih rješenja moglo bi izgledati recimo ovako:

```
void IzvrniNiz(void *niz, int br_elemenata, int vel_elementa) {
  for(int i = 0; i < br_elemenata / 2; i++) {
    char *p1((char *)niz + i * vel_elementa);
    char *p2((char *)niz + (br_elemenata - i - 1) * vel_elementa);
    for(int j = 0; j < vel_elementa; j++) {
      char pomocna(p1[j]);
      p1[j] = p2[j]; p2[j] = pomocna;
    }
  }
}
```

Ukoliko sada želimo izvrnuti elemente niza "`niz1`" od 10 elemenata u obrnuti poredak, mogli bismo koristiti poziv poput sljedećeg:

```
IzvrniNiz(niz1, 10, sizeof niz1[0]);
```

Opisani primjeri *nisu nimalo jednostavni*, i traže izvjesno poznavanje načina na koji su organizirani podaci u računarskoj memoriji. Najbolje je da probate ručno pratiti izvršavanje napisanih funkcija na nekom konkretnom primjeru. Napomenimo da su ovi primjeri ubačeni čisto sa ciljem pojašnjenja prave prirode pokazivača. Naime, u jeziku C++ ovakve primjere ne treba pisati, s obzirom da se razmotreni problemi mnogo lakše rješavaju pomoću *pravih* generičkih funkcija (koje ćemo uskoro upoznati). Ovakve kvazi-generičke funkcije tipično zahtijevaju čudne ili dopunske parametre (poput parametra "`vel_elementa`" u posljednjem primjeru). Pored toga, nemoguće je na ovaj način napraviti funkciju koja bi, recimo, našla *najveći element niza*, ili *sumu svih elemenata u nizu* (koja bi radila za nizove

proizvoljnih tipova elemenata), s obzirom da na ovaj način mi uopće ne baratamo sa elementima niza, već sa *bajtovima* koje tvore elementi niza. Stoga je takav pristup moguć jedino za primjene u kojima tačna priroda elemenata niza nije ni bitna (kakva je npr. kopiranje).

Bez obzira na brojna ograničenja funkcija koje koriste generičke pokazivače, one se mnogo koriste u jeziku C (jer u njemu bolja alternativa i ne postoji). Stoga mnoge funkcije iz biblioteka koje dolaze uz jezik C koriste ovaj pristup. Kako je jezik C++ naslijedio *sve biblioteke* koje su postojale u jeziku C, one postoje i u jeziku C++. Na primjer, funkcija "`memcpy`" iz biblioteke "`cstring`" identična je posljednjoj napisanoj verziji funkcije "`KopirajNiz`". Drugim riječima, kopiranje niza "`niz1`" od 10 realnih brojeva u niz "`niz2`" može se, u principu, obaviti i na sljedeći način:

```
std::memcpy(niz2, niz1, 10 * sizeof niz1[0]);
```

Također, ni funkcije koje kao jedan od parametara zahtijevaju veličinu elemenata niza u bajtovima (poput posljednje verzije funkcije "`IzvrniNiz`"), nisu nikakva rijetkost u standardnim bibliotekama. Takva je, na primjer, funkcija "`qsort`" iz biblioteke "`cstdlib`", koja služi za sortiranje elemenata niza u određeni poredak. Programeri u jeziku C intenzivno koriste ovakve funkcije. Mada se one, u načelu, mogu koristiti i u jeziku C++, postoje jaki razlozi da u jeziku C++ ove funkcije *ne koristimo* (ova napomena namijenjena je najviše onima koji imaju dobro prethodno programersko iskustvo u jeziku C). Naime, sve funkcije zasnovane na generičkim pokazivačima svoj rad baziraju na činjenici da je manipulacije sa svim tipovima podataka moguće izvoditi *bajt po bajt*. Ta pretpostavka je tačna za sve tipove podataka koji postoje u jeziku C, ali ne i za neke novije tipove podataka koji su uvedeni u jeziku C++ (kao što su svi tipovi podataka koji koriste tzv. *vlastiti konstruktor kopije*). Na primjer, mada će funkcija "`memcpy`" bespijekorno raditi za kopiranje nizova čiji su elementi tipa "`double`", njena primjena na nizove čiji su elementi tipa "`string`" može imati fatalne posljedice. Isto vrijedi i za funkciju "`qsort`", koja može napraviti pravu zbrku ukoliko se pokuša primijeniti za sortiranje nizova čiji su elementi dinamički stringovi. Tipovi podataka sa kojima se može sigurno manipulirati pristupanjem individualnim bajtima koje tvore njihovu strukturu nazivaju se *POD tipovi* (od engl. *Plain Old Data*). Na primjer, prosti tipovi kao što su "`int`", "`double`" i klasični nizovi *jesu* POD tipovi, dok vektori i dinamički stringovi *nisu*.

Opisani problemi sa korištenjem funkcija iz biblioteka naslijeđenih iz jezika C koje koriste generičke pokazivače ne treba mnogo da nas zabrinjava, jer za svaku takvu funkciju u jeziku C++ postoje bolje alternative, koje garantirano rade u svim slučajevima. Tako, na primjer, u jeziku C++ umjesto funkcije "`memcpy`" treba koristiti funkciju "`copy`" iz biblioteke "`algorithm`", o kojoj ćemo govoriti kasnije, dok umjesto funkcije "`qsort`" treba koristiti funkciju "`sort`" (također iz biblioteke "`algorithm`", koju ćemo kasnije detaljno obraditi). Ove funkcije, ne samo da su univerzalnije od srodnih funkcija naslijeđenih iz jezika C, već su i jednostavnije za upotrebu. Na kraju, ,oramo dati još samo jednu napomenu. Programeri koji posjeduju bogatije iskustvo u jeziku C, teško se odriču upotrebe funkcije "`memcpy`" i njoj srodnih funkcija (poput "`memset`"), jer im je poznato da su one, zahvaljujući raznim trikovima na kojima su zasnovane, *vrlo efikasne* (mnogo efikasnije nego upotreba obične *for*-petlje). Međutim, bitno je znati da odgovarajuće funkcije iz biblioteke "`algorithm`", uvedene u jeziku C++, poput funkcije "`copy`" (ili "`fill`", koju treba koristiti umjesto "`memset`"), tipično nisu *ništa manje efikasne*. Zapravo, u mnogim implementacijama biblioteke "`algorithm`", poziv funkcije poput "`copy`" automatski se prevodi u poziv funkcije poput "`memcpy`" ukoliko se ustanovi da je takav poziv *siguran*, odnosno da tipovi podataka sa kojima se barata predstavljaju POD tipove. Stoga, programeri koji sa jezika C prelaze na C++ ne trebaju osjećati nelagodu prilikom prelaska sa upotrebe funkcija poput "`memcpy`" na funkcije poput "`copy`". Jednostavno, treba prihvatiti činjenicu: funkcije poput "`memcpy`", "`memset`", "`qsort`" i njima slične *zaista ne treba koristiti* u C++ programima (osim u vrlo iznimnim slučajevima). One su zadržane pretežno sa ciljem da omoguće kompatibilnost sa već napisanim programima koji su bili na njima zasnovani!