UNIVERZITET U SARAJEVU
ELEKTROTEHNIČKI FAKULTET SARAJEVO

VEDRAN LJUBOVIĆ, ENIL PAJIĆ

UVOD U PROGRAMIRANJE C I C++

17570198

UNIVERZITETSKO IZDANJE

SARAJEVO, 2018.

Naslov: Uvod u programiranje – C i C++
Autori: Vedran Ljubović, Enil Pajić

WWW.UVODUPROGRAMIRANJE.BA

Izdavač: Elektrotehnički fakultet Univerziteta u Sarajevu

Recenzenti:
V. prof. dr Željko Jurić, Univerzitet u Sarajevu
Doc. dr Boris Milašinović, Sveučilište u Zagrebu

DTP: Vedran Ljubović

Štampa: "Štamparija Fojnica"
Tiraž: 500 primjeraka

SARAJEVO, 2018.
PRVO IZDANJE

Odlukom Senata Univerziteta u Sarajevu, broj 01-1093-16/18 od 28. 11. 2018, ovaj udžbenik je dobio saglasnost da se izda kao univerzitetsko izdanje.

17570198

Sadržaj

1 Prvi koraci 21

1.1 Hello world 22

1.1.1 Redoslijed izvršenja 24

1.1.2 Šta se dešava kada pokrenemo program? 25

1.1.3 Sintaksne i logičke greške 25

1.2 Promjenljive i ulaz/izlaz 27

1.2.1 Šta je to promjenljiva? 27

1.2.2 Inicijalizacija i neinicijalizirane promjenljive 29

1.2.3 Deklaracija i tipovi promjenljivih 29

1.2.4 Formati i escape sekvence 31

1.2.5 Unos promjenljivih pomoću funkcije scanf 33

1.2.6 Dodjela 34

1.2.7 Literali 35

1.2.8 Neki osnovni matematički operatori 36

1.2.9 Ulaz i izlaz u programskom jeziku C++ 37

1.3 Rješavanje programskih zadataka 38

1.3.1 Simboličke konstante 40

1.3.2 Komentari 41

1.3.3 Kamo dalje... 41

2 Uslovi i grananje 43

2.1 Vrste grananja 43

2.1.1 Naredba if 44

17570198

2.2 *Operatori poređenja* 46
2.3 *Alternativa* 48
2.4 *Logički operatori* 49
2.4.1 *Matematička logika u C-u* 50
2.4.2 *Kratkospojna evaluacija* 53
2.5 *Višestruki izbor* 54
2.5.1 *Ugniježđena grananja* 57
2.5.2 *switch-case* 59
2.6 *Kontrola toka naredbom goto* 61

3 *Skok u svijet programiranja* 65
3.1 *Predstavljanje podataka u memoriji računara* 65
3.1.1 *Binarni brojevni sistem* 67
3.1.2 *Oktalni i heksadecimalni brojevni sistem* 68
3.1.3 *Negativni binarni brojevi* 69
3.1.4 *Realni binarni brojevi* 70
3.1.5 *Binarni brojevi sa beskonačno mnogo cifara* 70
3.1.6 *Problem nepreciznosti realnih vrijednosti* 72
3.1.7 *Znakovni tip* 74
3.1.8 *Odsijecanje i zaokruživanje* 76
3.1.9 *Prekoračenje* 78
3.2 *Operatori* 79
3.2.1 *Implicitna konverzija tipa* 81
3.2.2 *Prefiksni i postfiksni operatori* 81
3.2.3 *Ternarni (kondicionalni) operator* 82
3.2.4 *Operator zarez* 83
3.2.5 *Operator sizeof* 83
3.2.6 *Još o nekim operatorima* 84

17570198

4 *Ponavljanje dijelova koda - Petlje* 87

4.1 *for petlja* 88

4.1.1 *Struktura for petlje* 89

4.1.2 *Kako funkcioniše for petlja?* 91

4.1.3 *Zloupotrebe zaglavlja for petlje* 92

4.1.4 *Petlje sa različitim koracima* 94

4.1.5 *Opadajuća for petlja* 96

4.1.6 *Naredbe break i continue* 97

4.1.7 *Višestruke (ugniježđene) petlje* 99

4.2 *while petlja* 105

4.2.1 *Transformacija for u while i obrnuto* 106

4.3 *do-while petlja* 107

4.4 *Primjeri zadataka sa petljama* 110

4.4.1 *Crtanje oblika na ekranu ASCII karakterima* 110

4.4.2 *Sume* 115

4.4.3 *Prosti brojevi, problemi tipa "za svako x"* 120

4.4.4 *Razdvajanje broja na cifre* 124

5 *Nizovi (polja)* 129

5.1 *Rad sa nizovima* 130

5.1.1 *Unos niza pomoću tastature* 132

5.1.2 *Inicijalizacija članova niza* 134

5.1.3 *Neke matematičke operacije nad nizom* 134

5.1.4 *Zadaci tipa "za svako x" i "postoji barem jedan"* 137

5.1.5 *Nizovi proizvoljne dužine* 140

5.2 *Neki algoritmi za rad sa nizovima* 144

5.2.1 *Maksimum i minimum* 145

5.2.2 *Izbacivanje člana iz niza* 150

5.2.3 *Ubacivanje člana u niz* 153

5.2.4 *Prebrojavanje članova niza i histogram* 155

5.2.5 *Sortiranje: Selection Sort* 158

17570198

5.3 Višedimenzionalni nizovi 160
5.3.1 Algoritmi za rad sa matricama 163
5.4 C++ vektori 166
5.4.1 Rad sa vektorima 167
5.4.2 Promjena veličine vektora 168
5.4.3 Osnovne operacije sa vektorima 170
5.4.4 "Siguran" pristup članovima vektora 170
5.4.5 Višedimenzionalni vektori 171

6 Funkcije 175
6.1 Uvod u funkcije u C-u 175
6.1.1 Sintaksa C funkcija 176
6.1.2 Kako pisati funkcije? 178
6.1.3 Korist od funkcija i potreba za njima 181
6.2 Funkcije u C-u 183
6.2.1 Analiza dijelova funkcije 183
6.2.2 Pozivanje funkcija 185
6.2.3 Mjesto pisanja funkcija 187
6.2.4 Prototip funkcije 190
6.2.5 Procedure (funkcije koje ne vraćaju ništa) 193
6.2.6 Funkcije bez parametara 194
6.2.7 Vidljivost promjenljivih deklarisanih unutar funkcije 196
6.2.8 Redoslijed određivanja vrijednosti parametara prilikom poziva 200
6.2.9 Globalne promjenljive 202
6.2.10 Statičke promjenljive 204
6.3 Prosljeđivanje nizova funkcijama 206
6.4 Rekurzivne funkcije 211

7 Pokazivači (pointeri) 221
7.1 Rad sa pokazivačima 224
7.1.1 Stil deklaracije pokazivača 224
7.1.2 Inicijalizacija pokazivača 225
7.1.3 Operator dereferenciranja 226

1757018

7.2 Pokazivači kao parametri funkcija 229

7.2.1 Vraćanje više vrijednosti iz funkcije 230

7.2.2 Pogrešne implementacije `zamijeni` *funkcije* 234

7.3 Pokazivači i nizovi 238

7.3.1 Pokazivačka aritmetika 239

7.3.2 Sličnost između imena niza i pokazivača 240

7.3.3 Prefiksni i postfiksni operatori nad pokazivačima 242

7.3.4 Prosljeđivanje nizova funkcijama 245

7.3.5 Rad sa nizovima koristeći pokazivačku aritmetiku 247

7.4 Pokazivači kao povratni tip funkcije 254

7.4.1 Viseći pokazivači 256

8 Stringovi (znakovni nizovi) 263

8.1 Osnovne operacije nad stringovima 264

8.1.1 Ulaz i izlaz 265

8.1.2 Česte funkcije za rad sa stringovima i njihova implementacija 267

8.1.3 Rješavanje zadataka sa stringovima 279

8.2 Neki algoritmi za rad sa stringovima 280

8.2.1 Pretvaranje broja u string 280

8.2.2 Ekstrakcija broja iz stringa 282

8.2.3 Rad sa rečenicama i riječima 284

8.2.4 Izbacivanje podstringa 285

8.2.5 Ubacivanje podstringa 289

8.3 C++ stringovi 292

8.3.1 Rad sa C++ stringovima 292

8.3.2 Unos stringova u C++u 293

8.3.3 Povećavanje stringova 294

8.3.4 Operacije nad stringovima 295

17570198

9 *Vlastiti tipovi podataka* 299

9.1 *Ključna riječ* `typedef` 299

9.2 *Pobrojani tip (*`enum`*)* 300

9.3 *Strukturni tip podataka (*`struct`*)* 303

9.3.1 *Definicija strukturnog tipa i deklaracija promjenljivih* 304

9.3.2 *Anonimne strukture* 305

9.3.3 *Imenovanje strukturnog tipa pomoću* `typedef` 306

9.3.4 *Rad sa strukturnim promjenljivim* 306

9.3.5 *Strukture i funkcije* 307

9.3.6 *Strukture i nizovi, podstrukture* 310

9.3.7 *Nizovi struktura* 313

9.3.8 *Pokazivači na strukture* 314

9.3.9 *Bitpolja* 315

9.4 *Unija (*`union`*)* 317

10 *Rad sa datotekama* 321

10.1 *O datotekama* 321

10.1.1 *Datotečni sistem* 321

10.1.2 *Putanja* 322

10.1.3 *Standardna hijerarhija i prava pristupa* 324

10.2 *Rad sa datotekama u C-u* 325

10.2.1 *Otvaranje i zatvaranje datoteke* 325

10.2.2 *Rad sa karakterima* 329

10.2.3 *Rad sa formatiranim datotekama* 333

10.3 *Baze podataka* 338

10.4 *Binarne datoteke* 342

10.4.1 *Funkcije za rad sa binarnim datotekama* 343

10.4.2 *Pomjeranje kursora* 346

10.4.3 *Tekstualne vs. binarne datoteke* 347

1757018

A Tablica operatora 351

B Naredbe za ulaz i izlaz 353

B.1 `printf` i `scanf` 353

B.1.1 Formatni stringovi (formati) 353

B.1.2 Podspecifikatori 354

B.1.3 Escape sekvence 355

B.1.4 Povratna vrijednost 356

C Napredne teme o pokazivačima 359

C.1 Dopune uvodnog poglavlja o pokazivačima 359

C.1.1 Napomene o adresnom modelu 359

C.1.2 Pokazivač na element iza kraja niza 359

C.1.3 Tip `ptrdiff_t` i "negativne" adrese 361

C.1.4 Konstantni pokazivači i pokazivači na konstantne objekte 362

C.2 Dinamička alokacija memorije 367

C.2.1 Generički pokazivači (`void *`) 367

C.2.2 Konverzije pokazivača 372

C.2.3 Funkcije `malloc` i `free` 375

C.3 Višestruki pokazivači 381

C.3.1 Neke primjene dvostrukih i trostrukih pokazivača 383

C.3.2 Nizovi pokazivača 386

C.3.3 Pokazivači na višedimenzionalne nizove 391

C.3.4 Pokazivači na matrice kao cjeline 397

C.3.5 Dinamički alocirane i "grbave" matrice 398

C.4 Pokazivači na funkcije 403

C.5 Greške prilikom kompajlerskih optimizacija 417

C.5.1 Ključna riječ `volatile` 418

C.5.2 Ključna riječ `restrict` 419

Indeks 421

17570198

1757018 8

Zahvaljujemo se našim profesorima Harisu Šupiću

i Željku Juriću.

Vedran Ljubović:

Želim da se zahvalim svojoj majci Ćamili Ljubović

koja me je usmjerila u oblast programiranja.

Enil Pajić:

Zahvaljujem se svojim roditeljima koji su mi pružili

podršku u pisanju ove knjige.

17570198

1757018 8

Predgovor

Kako naučiti programirati?

Kao nastavnik koji predaje uvodni kurs programiranja studentima koji u pravilu nemaju nikakvo ili skoro nikakvo predznanje, često sam bio prinuđen da odgovorim na upravo ovo pitanje. Student koji tek želi da zakorači u svijet programiranja nažalost neće uvijek naići na onu vrstu podrške koja mu/joj je potrebna. Tako će mnogi reći da se programiranje ne može tek tako naučiti – ili si se rodio s tim talentom koji samo trebaš otkriti u sebi ili nisi. Drugi pak tvrde da se naučiti programirati ne može bez mnogo godina truda i odricanja, pa oni koji nisu počeli još u osnovnoj školi kada dođu do fakulteta nemaju nikakvih šansi.

A ja sam baš upoznao i vidio mnoge studente koji u tom svom prvom septembru[1] nisu imali ama baš nikakvo predznanje, ni for petlju nisu znali, da bi do januara ili februara zaista naučili sve ono što treba i bili programeri. Tu je važno odgovoriti na pitanje šta tačno znači "biti programer" ili "znati programirati"? Istina je da učenju programiranja nema kraja, čovjek nikada ne može prestati učiti, uvijek postoje stvari koje ne poznajete.

[1] https://en.wikipedia.org/wiki/Eternal_September

Po meni osoba koja "zna programirati" treba *znati rješavati bilo kakav postavljeni problem*, uključujući i probleme kakve nikada ranije nije vidjela. Ovdje nisam slučajno upotrijebio riječ "rješavati" umjesto "riješiti". Neke discipline su pogodne za ljude koji znaju prepoznati taj jedan "trik" koji predstavlja rješenje zadatka. Programiranje nije jedna od njih.

Kada rješavate programski zadatak vaš prvi cilj je da složeni problem razbijete na više jednostavnijih, dok ne dođete do onoga što znate i možete riješiti. Pri tome je najvažnije da ne odustanete, da se ne prepadnete problema, nego da hrabro krenete u rješavanje korak po korak. Ako postoje neki koraci koje ne znate, možete ih preskočiti – neki drugi član vašeg programerskog tima će možda to uraditi umjesto vas.

1/15/01/98

Osoba koja na prvi pogled na tekst zadatka počne planirati, crtati, pisati kôd je osoba koja zna programirati. Ne kažem da zna *dobro* pro-

gramirati – to je vještina koja se vjerovatno uči čitav život. Ali ovo je neki početak.

Posmatrao sam mlade ljude koji su uspjeli naučiti programiranje, ali i mnoge koji nisu, i uočio sam da postoje neka pravila. Da bi se uspješno savladalo programiranje bitno je razumijevanje teorije, naravno. Ali sama teorija je beskorisna ako je ne znate primijeniti. Dalje, vrlo je korisno imati i gotova rješenja zadataka koja možete analizirati i iz njih nešto naučiti. No susreo sam se sa mnogima koji su imali oboje, a opet nisu znali programirati, jer su se fokusirali da napamet uče teoriju, pa čak i gotove komade kôda, bez razumijevanja kada se šta koristi. Ovo razumijevanje se, iz mog iskustva, može steći samo tako što se *samostalno pišu programi*. Baš kao što strani jezik možete naučiti samo tako što pričate sa strancima, kao što bicikl ne možete naučiti voziti tako što gledate druge kako voze, tako i programski jezik možete naučiti samo tako što pričate sa računarom.

Možete početi od toga da gotov program prepravljate da biste vidjeli šta i zašto radi. Po mom mišljenju tu je korisno i da vidite neke primjere koji ne rade, te da razumijete zašto ne rade. Ali, prije ili kasnije, moraćete sjesti pred prazan ekran razvojnog okruženja i početi kucati vaš program od onog prvobitnog "Hello world" programa. Taj korak je kao kada zaplivate prvi put bez zaštitnog pojasa, ili kada počnete voziti bicikl bez pomoćnih točkova. Dok to ne napravite, dok ne dovedete program od ideje koja ne postoji nigdje osim u vašoj glavi do programskog kôda koji radi tačno ono što ste zamislili, vi niste programer.

Pisanje programskog kôda najviše podsjeća na polagano objašnjavanje rješenja nekoj izrazito glupoj osobi. Pri tome moramo vrlo dobro znati koje je predznanje te osobe, koje riječi razumije a koje ne i šta pojedine riječi toj osobi znače kako bismo ih na pravi način upotrijebili.

Recimo da imamo nekog robota i želimo da taj robot ode i kupi hljeb. Izdaćemo mu jasnu naredbu:

1. Kupiti hljeb.

No naš robot ne zna šta znači "kupiti hljeb", pa moramo razraditi to rješenje, recimo ovako:

1. Izaći iz kuće.
2. Otići u prodavnicu.
3. Zatražiti hljeb od prodavača.
4. Uzeti hljeb.
5. Dati pare.
6. Vratiti se kući.

1757018

Ovo bi svaki robot morao znati, zar ne? Ali evo, ovaj naš robot se žali da ne zna šta znači izaći iz kuće, pa da i to objasnimo

1. Izaći iz kuće.
   (a) Doći do izlaznih vrata.
   (b) Pritisnuti šteku prema dolje.
   (c) Povući vrata prema sebi držeći šteku.
   (d) Pustiti šteku.
   (e) ...

itd. No ovdje smo pretpostavili da su vrata otključana. Da li možemo napraviti ovu pretpostavku? Ako ne, trebamo nakon koraka (e) ako vrata nisu otvorena koristiti ključ te ponoviti korake od (b) dalje. Suština je da moramo znati za šta je naš robot isprogramiran a za šta nije, pa koristiti samo one komande koje robot razumije. Također moramo razmisliti koje pretpostavke smijemo napraviti, a koje ne.

Upravo ovaj način razmišljanja je ono što je potrebno programeru. Najprije moramo biti savršeno načisto šta želimo da računar uradi. Računar nije ljudsko biće, nije nam prijatelj niti šef, to je poslušni robot koji obavlja zadatke koje smo mu dali. Zatim moramo taj zadatak saopštiti jezikom koji računar razumije. Na kraju, moramo biti načisto koje pretpostavke smo napravili i zašto.

Jedan ciničan pogled na ovo oličen je u poslovici "računari ne griješe, ljudi griješe". U suštini, računar uvijek radi tačno ono što mu je programer naredio programom. Ako dođe do greške, to je ili greška u programskom kodu tj. u pravilnoj upotrebi odabranog programskog jezika i biblioteka, ili je greška u pretpostavkama npr. o znanju korisnika. Na kraju, možda se ni korisnik ne pridržava uputstva za upotrebu?

Na sličan način možemo razmišljati o razlikama između programskih jezika. U nekom jeziku možda postoji riječ za sortiranje (slaganje nekog skupa po veličini), a u nekom drugom ne postoji, pa morate korak po korak objasniti proces slaganja po veličini. Ipak, kako budete učili različite jezike, vidjećete da su razlike između njih relativno male. Sam osnovni jezik posjeduje prilično mali broj različitih riječi. Za sve ostalo morate koristiti *gotove biblioteke* koje nude posebne naredbe (riječi) za specifične zadatke kao što je pokretanje robota ili 3D grafika.

U praksi programeri ne moraju pamtiti naredbe, nego ih mogu uvijek potražiti u dokumentaciji. Dakle, sposobnost koju programeri moraju imati je da neki konkretan zadatak objasne tj. prepričaju računaru kao da je u pitanju malo dijete, koristeći samo zadati skup riječi.

Kako steći tu sposobnost? Na žalost, na ovo pitanje nema jednostavnog odgovora, jer u suprotnom zanimanje "programer" ne bi

17/5/01 98

bilo jedno od najtraženih i najbolje plaćenih zanimanja danas. Mnogi stručnjaci u oblasti smatraju da se ova sposobnost stiče isključivo iskustvom, kroz rješavanje velikog broja različitih konkretnih programerskih problema. I autori ove knjige podržavaju ovakvo mišljenje. Baš kao što se iskustvo i stručnost pilota mjeri brojem sati leta, tako se i sposobnost programera može mjeriti brojem napravljenih programa. Pri tome je važno da su problemi koje rješavate raznovrsni. Ako napravite stotinu sličnih programa nećete naučiti ništa novo.

Stvar koju ćemo sada izjaviti vjerovatno se neće svidjeti mnogim studentima i učenicima, ali ključan momenat kod učenja programiranja je upravo ono "razbijanje glave" kada rješavate zadatak i ne znate kako da ga riješite. Kroz frustraciju problemom grade se nove sinapse (veze u mozgu), koje će vam sljedeći put olakšati rješavanje sličnog problema. Kada vam neko da gotovo rješenje, ta osoba vas je doslovno opljačkala, jer vam je oduzela priliku da naučite neki bitan koncept samostalnim rješavanjem. Dakle, naravno da trebate tražiti pomoć od kolega, ali bitno je da najprije pokušate sami. Kada zatražite pomoć, insistirajte da vam objasne tačno gdje ste pogriješili, zašto je to pogrešno i kako *vaš program* prepraviti tako da radi, a ne prihvatajte neko potpuno različito rješenje kao odgovor.

Kome je namijenjena ova knjiga?

Moramo ovdje posebno istaknuti jednu stvar: udžbenik koji je pred vama *nije* priručnik za programski jezik C (niti C++)! Ovaj udžbenik vam služi kako biste naučili programirati, a igrom slučaja kao osnovni jezik poduke koristi se programski jezik C. Dakle, ova knjiga je namijenjena totalnim početnicima koji o programiranju ne znaju ništa, mada može biti korisna i onim iskusnijim kako bi osvježili svoje znanje, te onima koji poznaju druge programske jezike ali ne i C. Po toj kombinaciji – za početnike, a koristi se C – smatramo da je ova knjiga jedinstvena.

Mnogi smatraju da se ovakva knjiga ne može napisati, odnosno da programski jezik C nije pogodan za početnike. Prilikom učenja novih stvari bitno je da učenika ne zatrpamo viškom informacija koje se ne mogu odjednom memorisati. Zbog toga se u objašnjavanju prave mnoga pojednostavljenja, detalji se preskaču ili ostavljaju za kasnije, koristi se laički, jednostavan jezik. Pojedinci sa početka pasusa tvrde da programski jezik C "ne nagrađuje aproksimacije" (aproksimacija: u matematici, približna vrijednost).

Autori ove knjige se ne slažu. Smatramo da je programski jezik C jezik kao i svaki drugi, s tim što posjeduje osobinu da programi koji sadrže greške često rade, nekada čak daju i tačne rezultate – ali nisu ispravni. Zato je potrebna određena disciplina da se pri pisanju

1755701198

programa ne prave greške. Nadamo se da čemo čitaoce naučiti toj disciplini kroz savjete, primjere ispravnih i pogrešnih programa, jasne definicije. Ali naš osnovni cilj je da ljudi koji budu čitali ovu knjigu stvore *mentalni model* načina rada računara i izvršavanja programa koji možda nije potpuno ispravan, ali rezultira ispravnim programima u velikoj većini situacija.

Pored toga, važno nam je da čitaoci lakše savladaju druge programske jezike koji nemaju neke od neobičnih osobina C-a. Sve je to u cilju da što prije postanete programeri, makar u prvom periodu i ne budete baš dobri C programeri. Sigurni smo da ćete sve eventualne pogrešne zaključke koje stvorite razjasniti kasnije kako budete čitali napredniju literaturu iz C-a.

Programski jezik C++

Često ćete čuti da se programski jezici C i C++ posmatraju zajedno, pa se koristi fraza "C/C++". Bitno je da znate da su u pitanju dva potpuno različita programska jezika, sa različitim osobinama. Ustvari, programski jezik C++ je prvobitno zamišljen kao *nadskup programskog jezika C*, odnosno plan je bio da se svaki C program bez izmjene može prevesti koristeći C++ prevodilac i pokrenuti. Ipak postoji određen (relativno mali) broj specifičnosti programskog jezika C koje sa C++om neće raditi. Taj broj raste kako koristite novije standarde C-a i C++a. Recimo u novim verzijama C standarda dozvoljeni su nizovi čija je dužina data promjenljivom, koji nisu legalni u izvornom C89 standardu, niti u bilo kojoj verziji C++a. S druge strane, u novijim verzijama C++ standarda konstanta NULL je premještena u posebnu biblioteku i ne koristi se.

Prilikom pisanja ove knjige kao udžbenika za programski jezik C nastojali smo da ona bude korisna i za one koji uče C++. To znači da smo izbjegavali sve osobine C-a koje neće raditi u C++u, kao što su već spomenuti nizovi dužine date promjenljivom. Kod pojava koje bi mogle izazvati probleme (kao što je konstanta NULL) to smo istakli u tekstu.

Ipak, postoje i pored toga neki C programi koji će raditi ako ih kompajlirate sa C prevodiocem, a neće sa C++ prevodiocem, iako ne koriste ništa od onoga što smo naznačili u knjizi. To su programi koji su *u suštini neispravni*, bilo da se ne pridržavaju C standarda ili da se koristi neka osobina koja rezultira nedefinisanim ponašanjem. Zbog toga je jako bitno da se ne oslanjate na to što program radi da biste znali da je ispravan. Kompajler nije alat za provjeru ispravnosti programa! Program da bi bio ispravan treba se pridržavati standarda, a to možete postići tako što koristite samo one osobine jezika koje su navedene u ovoj knjizi.

1777098

Kada već učite programski jezik C++, važno je napomenuti da postoje neke stvari koje su sasvim legalne u tom programskom jeziku, ali se preporučuje da radite stvari na neki drugi način koji možda ne postoji u programskom jeziku C. Krenimo od uključivanja biblioteka: za ulaz i izlaz u C-u koristili smo:

```
#include <stdio.h>
```

i zaista ovakva deklaracija će raditi i u C++u. U pitanju je ista ona biblioteka koju koristimo u našim C programima. Ali zbog raznih problema sa C bibliotekama u programskom jeziku C++ definisane su "C++ verzije" istih tih biblioteka koje se neznatno razlikuju u imenu, te je u novijim verzijama C++ standarda korištenje izvornih C biblioteka proglašeno za "prevaziđeno" (eng. deprecated). U pravilu, naziv C++ verzije date biblioteke dobije se tako što se na početak doda slovo "c", a dio ".h" da se izostavi, ovako:

```
#include <cstdio>
```

Kod uključenja biblioteke na ovaj način, preporučljivo je (mada će program možda raditi i bez toga) da se navodi prefiks `std::` prije pristupa funkcijama:

```
std::printf("Zdravo svijete!");
```

Inače u mnogim udžbenicima i kursevima C++a vidjećete da se u primjerima kôda odmah na početku programa navodi sljedeća naredba:

```
using namespace std;
```

tako da se prilikom pristupa bibliotečnim funkcijama i objektima ne mora svaki put pisati `std::`. Ova praksa ipak *nije preporučljiva* između ostalog i zbog pojave "konflikta imenskih prostora", pa vam preporučujemo da to ne radite i da svaki put navodite `std::`.

Oznake

U ovoj knjizi korištene su sljedeće oznake:

Česta greška. Priloženi kod je pogrešan, a razlog zašto ga navodimo je što je to česta greška koju prave početnici. Savjetujemo da pročitate prateći tekst kako ne biste i vi pravili istu grešku.

Zadatak: *Kroz ove riješene primjere nećete dobiti samo pristup rješavanju sličnih problema. Biće korak-po-korak razrađena i razna alternativna rješenja, pa ćete na taj način naučiti mnoge bitne koncepte iz programiranja.*

17570198

Na mjestu označenom strelicom↑ pozivamo se na nešto što je objašnjeno u nekom drugom dijelu knjige.

↑ Ovdje će biti data oznaka poglavlja ili eventualno stranice.

✓ *Linija označena kvačicom predstavlja* ***definiciju*** *ili* ***pravilo****. Tu je definisan neki novi pojam ili navedeno bitno pravilo kojeg se trebate pridržavati. Sa strane je istaknut pojam koji definišemo.*

✓ Definicija ili pravilo

1757018

1757018 8

1
Prvi koraci

Gotovo sve poznate knjige i priručnici za programiranje počinju sa jednostavnim primjerom programa koji na ekranu ispisuje poruku "Hello, world!" (u prevodu: Zdravo, svijete!). Ovu prastaru tradiciju ne bismo ni mi trebali iznevjeriti, pa hrabro krećemo sa našim prvim programom.

```
1 #include <stdio.h>
2 int main() {
3   printf("Dobro dosli u svijet programiranja!\n");
4   return 0;
5 }
```

Najprije možete uočiti da su sve linije programa označene brojevima radi lakšeg snalaženja. Mi ćemo u našem tekstu pisati npr. "greška se nalazi u liniji 4", ali obratite pažnju da se ovi brojevi ne ukucavaju u okruženje, pa su zato označeni drugačijom bojom i oblikom slova od ostatka programa.

Ovaj program na ekranu ispisuje poruku "Dobro dosli u svijet programiranja!" i završava s radom. Za sada vam možda sve skupa sa znakovima poput raznih vrsta zagrada, hash-tagova i slično djeluje pomalo zbunjujuće. Ne sekirajte se – jedini ustvari bitan dio programa je linija 3. Ostatak programskog koda je šablon koji ćete samo kopirati iz programa u program, otprilike negdje do sredine ove knjige. Nije bitno što za sada i ne razumijete baš najbolje čemu služi.

Kada pokrenete ovaj program, u vašem odabranom okruženju vidjećete *prozor za izvršenje programa*. Kako on tačno izgleda, zavisi od okruženja. U nekim okruženjima otvoriće se novi prozor (najčešće crne boje), a u nekim to je okvir ili dio prozora, obično u donjem dijelu ekrana. Sadržaj prozora za izvršenje će biti:

```
Dobro dosli u svijet programiranja!
```

uz možda još neke poruke. Ove poruke u pravilu signaliziraju da se program izvršio bez grešaka i ne moramo za sada obraćati pažnju na njih.

1/15/01/98

Pokrenuli smo naš prvi program! Sada ga možete i promijeniti. Probajte prepraviti program tako da umjesto dobrodošlice ispisuje vaše ime ili neku smiješnu poruku.

1.1 *Hello world*

Da nas ne bi optužili kako koristimo stvari koje nisu objašnjene, proći ćemo kroz program liniju po liniju i objasniti šta se u svakoj od njih dešava. Ponovo napominjemo, nemojte dozvoliti da vas opisi koje damo ispod prestraše ili zbune. Za sada nije važno da znate šta čemu služi, to će prirodno nadoći kasnije.

1. Linija `#include <stdio.h>` je *preprocesorska direktiva* koja služi za pozivanje eksterne *biblioteke* pod nazivom `stdio.h`.[1] Preprocesorske naredbe prepoznajemo po znaku "hash" ili "povisilica" ili "taraba" na samom početku linije. Programski jezik C koristi biblioteke za funkcionalnosti koje nisu dio samog programskog jezika. Na primjer, biblioteka `stdio.h` (skraćeno od STanDard Input/Output – standardni ulaz/izlaz) sadrži funkcije za unos podataka sa tastature i njihov ispis na ekran. Jedna od tih funkcija je i `printf` koju smo koristili u trećoj liniji.

[1] Ovim je uključena *sistemska* biblioteka, tj. sama datoteka `stdio.h` se nalazi u nekom sistemskom folderu. Za uključivanje biblioteka koje se nalaze u istom folderu kao i C kôd, umjesto znakova veće-od i manje-od treba koristiti dvostruke navodnike: `#include "lokalna.h"`.

2. Linija `int main()` označava početak funkcije pod nazivom `main` (eng. glavna funkcija). Svaki C program se sastoji od jedne ili više funkcija, a jedna od njih mora biti `main` funkcija i to je funkcija koja se ustvari izvršava kada pokrenemo program. Naši prvi programi će za sada imati samo jednu funkciju koju smo mi pisali i to `main` funkciju. Sav sadržaj `main` funkcije nalazi se unutar *velikih* ili *vitičastih* zagrada.[2]

[2] Više o funkcijama u poglavlju 6.

3. Funkcija `printf` služi za ispis teksta na ekran. Dakle u liniji 3 nismo pravili neku našu funkciju, nego smo pozvali već postojeću funkciju `printf` i iskoristili je da obavimo zadatak ispisa na ekran.
   Parametri C funkcija navode se unutar *malih* ili *običnih* zagrada. Parametar funkcije je neka dopunska informacija koja određuje šta funkcija treba konkretno da uradi, npr. parametar funkcije `printf` je tekst (string) koji treba ispisati na ekranu, a stringovi se u C-u označavaju dvostrukim navodnicima.. Ako funkcija nema parametara, mora se navesti otvorena pa zatvorena mala zagrada da bi se znalo da je u pitanju funkcija a ne nešto drugo. Primjer toga možete vidjeti u liniji 2 gdje smo definisali funkciju `main`.
   Obratite pažnju da nakon uzvičnika imamo dva znaka koji se nisu ispisali na ekranu, a to su `\n`. Prvi znak je obrnuta kosa crta (eng. backslash) koji na tastaturi sa našim znakovima ne možemo otkucati direktno nego moramo držati pritisnutu tipku AltGr te pritisnuti

1/15/0198

tipku W. Bitno je da ga razlikujemo od obične kose crte koju dobijemo pritiskom na Shift i broj 7. Ova dva znaka označavaju *prelazak u novi red* (eng. new line). Bez ova dva znaka neka sljedeća naredba `printf` bi nastavila ispisivati tekst u istom redu. U drugim programskim jezicima takav efekat postižemo posebnim naredbama npr. u Pascalu bismo koristili naredbu "println".[3]
Ova linija se završava znakom *tačka-zarez* (eng. semicolon). Svaka naredba u C-u se mora završavati tačka-zarezom. Kako je običaj da se svaka naredba piše u zasebnoj liniji (mada za razliku od nekih drugih programskih jezika C ne insistira na tome), možete zapamtiti da znak tačka-zarez trebate staviti na kraju svake linije uz određene izuzetke. Neke od tih izuzetaka možete vidjeti u našem programu: tačka-zarez se ne postavlja na kraj linija koje sadrže otvorenu ili zatvorenu vitičastu zagradu (linije 2 i 5), kao ni nakon preprocesorskih direktiva (linija 1). Ovo je vrlo pojednostavljeno objašnjenje koje ćemo svakako proširiti u nastavku knjige, ali za sada je dovoljno.

[3] Napomenimo da se u svakom C programu posljednja `printf` naredba treba završavati sa `\n` jer u suprotnom nije garantovano da će se tekst uopšte ispisati na ekran.

4. Naredba `return` prekida rad funkcije i određuje povratnu vrijednost (eng. return value) funkcije.[4] Pošto naš program ima samo jednu funkciju, a to je `main` funkcija, kada se ta funkcija završi, završava se i čitav program. Funkcija `main` mora vratiti jedan cijeli broj kao rezultat svog rada (što smo mogli vidjeti po onome `int` u liniji 2, a više o tome kasnije). Taj cijeli broj po dogovoru je nula ako je program ispravno završio rad, a neki broj različit od nule (u rasponu 1-255) ako je došlo do neke greške.

[4] Pojam povratne vrijednosti će biti jasniji u poglavlju 6.

5. Ne zaboravimo zatvoriti vitičastu zagradu. Ovdje ćemo spomenuti i to da vitičaste zagrade označavaju *blok kôda*, neki skup naredbi koji posmatramo kao jednu cjelinu, a u ovom slučaju ta cjelina je funkcija `main`.

Vjerovatno vam je palo na pamet da u ovom programu imamo pravopisnu grešku, jer smo riječ "dosli" napisali koristeći slovo "s" umjesto slova "š". Program u kojem se naredbom `printf` ispisuju "naša slova" bi radio i vjerovatno i davao izlaz koji izgleda ispravno – ali to nije ispravan C program. Standard za C programski jezik propisuje da se u C programima (bilo van stringova ili u njima) smiju koristiti samo slova engleskog alfabeta, cifre i jedan broj znakova interpunkcije i drugih specijalnih znakova koji svi zajedno čine ASCII skup znakova o kojem ćemo detaljnije pričati kasnije.↑

↑ Poglavlje 3.1.7, str. 74.

Na tom primjeru možemo uočiti još jednu bitnu karakteristiku programskog jezika C: u mnogim situacijama imate programe koji se mogu pokrenuti i čak prividno rade ispravno, ali oni nisu ispravni i postoje situacije u kojima bi mogli dati pogrešan rezultat. C je poznat po tome što vam omogućuje da pravite ovakav tip grešaka kojih ste

1715/01 98

potpuno nesvjesni, tako da morate neprekidno kontrolisati vaš kod i provjeravati da li je ono što ste napisali ispravno.

1.1.1 *Redoslijed izvršenja*

Naš prvi program je imao samo jednu bitnu naredbu: naredbu `printf`. Šta se dešava kada imamo više naredbi? One će se izvršavati jedna po jedna, odozgo prema dolje, osim ako drugačije ne zatražimo.[5] Pri tome izvršenje počinje od prve naredbe unutar bloka koji definiše `main` funkciju. Recimo pogledajmo sljedeći primjer:

[5] Primjere tog tipa ćemo vidjeti u poglavlju 2.

```
1 #include <stdio.h>
2 int main() {
3   printf("Dobro dosli");
4   printf(" u svijet programiranja!\n");
5   return 0;
6 }
```

Najprije će se izvršiti naredba koja na ekranu ispisuje "Dobro dosli", a zatim i naredba koja ispisuje "u svijet programiranja!".

```
Dobro dosli u svijet programiranja!
```

Ova dva stringa se ispisuju jedan pored drugog, a ne jedan ispod drugog, jer nismo zatražili prelazak u sljedeći red. Dakle, ovaj program radi potpuno isto što i prethodni. Ako želimo da prilikom ispisa pređemo u novi red, to moramo zatražiti tako što ćemo u tekstu koji se ispisuje navesti `\n` – znak obrnuta kosa crtica (eng. backslash) nakon kojeg slijedi slovo "n":

```
1 #include <stdio.h>
2 int main() {
3   printf("Dobro dosli\n");
4   printf(" u svijet programiranja!\n");
5   return 0;
6 }
```

Obratite pažnju da se u liniji 4 između znaka navodnika i slova "u" nalazi jedan razmak. Da smo izostavili taj razmak u pretprošlom primjeru, riječi "dosli" i "u" bi bile spojene:

```
Dobro dosliu svijet programiranja!
```

U sljedećem primjeru se taj razmak ispisuje na početku reda, tako da se slovo "u" nalazi ispod slova "o" umjesto ispod slova "D":

```
Dobro dosli
 u svijet programiranja!
```

pa taj razmak možemo i pobrisati.

No, šta će se desiti ako napišemo ovako:

```
1 #include <stdio.h>
2 int main() {
3   printf("Dobro dosli");
4   return 0;
```

1/13/2018

```
5   printf(" u svijet programiranja!\n");
6 }
```

Ovaj program će na ekranu ispisati samo "Dobro dosli", jer se sljedeći `printf` nalazi nakon naredbe `return 0`. Sjetimo se da `return 0` prekida rad main funkcije, a samim time i čitavog programa, tako da izvršenje programa nikada neće doći do linije 5 i tekst "u svijet programiranja!" se nikada neće ispisati.

1.1.2 *Šta se dešava kada pokrenemo program?*

Dugme za pokretanje

Vaše okruženje vjerovatno posjeduje dugme za pokretanje programa u obliku trougla koje podsjeća na "Play" tipku CD ili DVD uređaja. Klikom na tu tipku dobijate već opisani prozor sa izlazom programa. Ali svaki put kada kliknete na tu tipku, u pozadini se dešava niz zasebnih i dosta složenih operacija.

Najprije se poziva *prevodilac* ili *kompajler* (eng. compiler). Prevodilac prevodi program u nekom višem programskom jeziku (npr. C) u mašinski kod koji je direktno razumljiv računaru.

"Hoćete reći da računar ne razumije C?" – sada će se neki pobuniti, "Pa zašto ga onda učimo?"

Viši programski jezici poput C-a su daleko lakši za razumijevanje *ljudima*, dok računari razumiju samo mašinski jezik. Programi pisani u mašinskom jeziku su vrlo teški za čitanje, a pokazuje se da analiza i prepravke programa predstavljaju veliki dio posla jednog programera. Osim toga, na svakoj klasi računara (odnosno računarskoj arhitekturi) mašinski kod je različit, pa kada bi programi bili pisani u mašinskom jeziku, trebalo bi ih ponovo prepravljati za svaku novu arhitekturu. S druge strane, viši programski jezici su uglavnom mašinski neovisni i za popularne programske jezike kao što je C postoje prevodioci za mnoge arhitekture. Npr. Android operativni sistem radi podjednako i na ARM i na Intel baziranim mobilnim uređajima iako je njihov mašinski kod potpuno različit, jer je jezgra Androida pisana u C programskom jeziku a aplikacije i radno okruženje u Java programskom jeziku.

Izlaz prevodioca je izvršna datoteka (koju na Windows operativnom sistemu prepoznajemo po nastavku .exe) koja se zatim može pokrenuti npr. tako što dvokliknemo na nju. Kada pokušamo otvoriti izvršnu datoteku, u pozadini se pokreće još jedan sistemski program koji nazivamo *punjač* (eng. loader) koji učitava program u memoriju, te priprema sve što je potrebno za njegovo izvršenje.

17570198

1.1.3 *Sintaksne i logičke greške*

Probajmo sada u našem programu otkucati nešto besmisleno, npr.:

```
#include <stdio.h>
int main() {
```

```
  printf("Dobro dosli!\n"); blabla
  return 0;
}
```

Vaš program se neće pokrenuti. Umjesto toga ćete negdje u prozoru za izvršenje, ili u donjem dijelu ekrana razvojnog okruženja, vidjeti nešto poput ovoga:

```
/tmp/test/main.c: In function 'main':
/tmp/test/main.c:5:31: error: 'blabla' undeclared (first use in this
     function)
     printf("Dobro dosli!\n"); blabla
                               ^
/tmp/test/main.c:5:31: note: each undeclared identifier is reported only
     once for each function it appears in
/tmp/test/main.c:6:5: error: expected ';' before 'return'
     return 0;
     ^
/tmp/test/main.c:7:1: warning: control reaches end of non-void function [-
     Wreturn-type]
 }
 ^
```

Vaš program sadrži *grešku u sintaksi*. Ukratko, kada napišete neki program i pokušate ga pokrenuti, prevodilac daje sve od sebe da pokuša shvatiti šta ste htjeli reći. Međutim, nekada to jednostavno nije moguće. U tom trenutku proces prevođenja programa pada i prevodilac ispisuje poruku o grešci kao što je ova navedena iznad. U ovom konkretnom programu, prevodilac je pokušao protumačiti `blabla` kao naziv identifikatora (promjenljive, funkcije itd.), ali taj identifikator nije deklarisan,↑ pa smo zato dobili poruku `blabla undeclared`.

↑ Poglavlje 1.2.1, str. 27.

Početak poruke o grešci glasi: `/tmp/test/main.c:5:31: error...` što označava da se greška nalazi u datoteci `main.c`, u liniji 5, kolona 31 (tridesetprvi karakter po redu).

Nakon toga se nastavljaju i druge greške koje prepoznajemo po tome što sadrže ključnu riječ `error`, kao i upozorenja koja su označena sa `warning`. Tako nam recimo prevodilac sada sugeriše da smo zaboravili znak tačka-zarez (*expected ';' before 'return'*), a zatim, pošto nije uspio rastumačiti ostatak programa, da u našoj funkciji nedostaje obavezna naredba `return` (*control reaches end of non-void function*). Iz ovoga zaključujemo da je ustvari prva greška najvažnija, a sve ostale su njena posljedica, tako da je potrebno kod ispravljanja krenuti od prve greške.

✓ *Uvijek prilikom popravljanja grešaka krenite od prve greške, a to je greška na vrhu spiska. Nakon što popravite prvu grešku probajte ponovo pokrenuti program jer su možda i ostale sada ispravljene.*

✓ Kako popravljati greške?

1/55/0198

Ako u programu nema grešaka, ali ima upozorenja, program se može prevesti i pokrenuti. No, većina upozorenja skreće pažnju na neki problem u kodu kojeg možda niste svjesni. Najgori tip grešaka su one za koje ne dobijete čak ni upozorenje, jer istovjetna naredba u

nekom drugom kontekstu može biti ispravna. Ovo nazivamo *logičkom greškom* – greška koja je nastala zbog nerazumijevanja logike programa.

U ostatku ove knjige redovno ćemo skretati pažnju na česte greške i upozorenja koja su vezana za programske strukture koje obrađujemo. Ovdje treba posebno naglasiti da će vam sigurno od velike koristi biti da razumijete poruke koje dobijate od prevodioca i ispravno ih čitate. Poznavanje engleskog jezika je svakako poželjno, ali i prepoznavanje određenih ključnih riječi u poruci greške, koje ćemo također posebno isticati.

1.2 Promjenljive i ulaz/izlaz

Prelazimo odmah na sljedeći primjer nešto složenijeg programa:

```
1  #include <stdio.h>
2  int main() {
3    int a,b,c;
4    printf("Unesite prvi broj: ");
5    scanf("%d", &a);
6    printf("Unesite drugi broj: ");
7    scanf("%d", &b);
8    c = a + b;
9    printf("Suma brojeva %d i %d je %d\n", a, b, c);
10   return 0;
11 }
```

Ovaj program nam omogućuje da unesemo dva broja, a zatim izračunava njihov zbir i ispisuje na ekranu. Korisno, zar ne?

Početak i kraj programa (linije 1, 2, 10 i 11) prepoznajemo od ranije kao šablon za C program, pa ih nećemo ponovo objašnjavati. Krenimo od linije 3 u kojoj se dešavaju interesantne stvari. Linija 3 glasi: `int a,b,c;` a njome su deklarisane tri *cjelobrojne promjenljive* čija su imena `a`, `b` i `c`.

1.2.1 Šta je to promjenljiva?

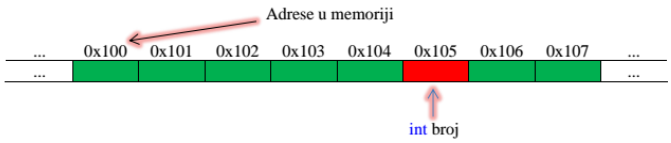
✓ *Promjenljiva ili varijabla (eng. variable) je neka imenovana lokacija u memoriji na kojoj možemo držati neki podatak. U nju možemo pisati i iz nje čitati.*

✓ Promjenljiva

Kao što znamo, računar posjeduje internu (RAM) memoriju u kojoj može držati podatke. Recimo, za neki računar kažemo da ima 4 GB (četiri gigabajta) RAM-a. Memorija računara se sastoji od niza *bajta* (eng. byte). Jedan bajt je memorijska cjelina koja sadrži cijeli broj u opsegu 0-255, a veće brojeve kao i negativne, decimalne itd. možemo predstaviti kombinovanjem više bajta.

Svaka lokacija u memoriji tj. svaki bajt ima svoju adresu koja je ustvari samo redni broj bajta u memoriji, kao što su kuće u nekoj ulici označene brojevima (slika 1.1). Ustvari, adresa ne mora biti (jedan)

1/15/0198



Slika 1.1: Pojednostavljeno predstavljanje memorije i promjenljive

broj, ali je za nivo ovog udžbenika sasvim dovoljna pretpostavka da su u pitanju prirodni brojevi.[6] Dakle, vi preko adrese možete direktno pristupiti bilo kojem bajtu u memoriji, očitati njegovu vrijednost ili zapisati neku drugu.

[6] Kao što stoji u opisu slike, ova grafička predstava memorije je *pojednostavljena*. Priča o organizaciji memorije u C programima je mnogo složenija. No, kako je ovo udžbenik programiranja a ne udžbenik C-a, ta diskusija je ostavljena za dodatak C.

Kada radimo s memorijom, bitno je da naš program unaprijed *zauzme* dio memorije koji mu je potreban za rad kako više programa ne bi koristili iste lokacije istovremeno. Recimo, zamislite kako bi bilo neugodno ako bi npr. jedan program memorisao da je neki učenik dobio ocjenu 5, a drugi program zatim iz nekog svog razloga u istoj lokaciji zapiše broj 1 ili 18 ili -107 !? Možete zaključiti da su adrese u pravilu vrlo veliki brojevi (zamislite, 4 GB je 4 milijarde bajta!) i nepraktični za pamćenje. Osim toga, svaki put kada pokrenemo program stanje memorije je drugačije, moguće da se u međuvremenu pokrenuo neki drugi program koji radi u pozadini (npr. antivirus) i zauzeo određeni dio memorije.

Zbog svega ovoga, mi u pravilu ne radimo direktno s memorijskim adresama. Umjesto toga, mi u našem programskom kodu *deklarišemo promjenljivu*, koja se npr. zove `broj` kao na slici.

✓ *Deklaracija promjenljive je naredba kojom imenujemo memorijsku lokaciju i dodjeljujemo joj tip radi korištenja u programu.*

✓ Deklaracija

Značenje deklaracije promjenljive je sljedeće: u trenutku kada se program pokreće, sistem pronalazi prvu slobodnu lokaciju u memoriji i rezerviše je (zauzima) za naš program, te joj daje ime `broj`.[7] Neka je adresa te lokacije 105 kao na sl. 1.1. Kada god u našem programu napišemo `broj` mi ćemo ustvari pristupati adresi 105. Prilikom sljedećeg pokretanja programa naš `broj` će možda dobiti neku drugu adresu, mi se ne možemo osloniti da će to svaki put biti 105.

[7] Ovisno o tipu to može biti i više uzastopnih lokacija, o čemu čemu pričati ubrzo.

Spomenimo još i da se u standardu C jezika razlikuju pojmovi "deklaracija" i "definicija" pa bi ovo iznad bila *definicija promjenljive*. Mada, u svim drugim programskim jezicima se uvijek govori o deklaraciji promjenljive, to je termin koji se koristi i u mnogim udžbenicima i u svakodnevnoj upotrebi pa ćemo i mi ovdje reći da se promjenljive *deklarišu*, a za neke druge fenomene u programiranju ćemo govoriti o definiciji (o čemu će biti riječi kasnije).

1715701198

1.2.2 *Inicijalizacija i neinicijalizirane promjenljive*

Kada tek deklarišemo promjenljivu, koja je njena vrijednost? U nekim programskim jezicima možemo se osloniti da će npr. cjelobrojna promjenljiva `broj` iz prethodnog primjera imati vrijednost nule, ali C nije jedan od tih jezika. Programski jezik C nam omogućuje da pišemo izuzetno brze i efikasne programe na način da sam jezik ne podrazumijeva da će biti urađeno ništa što bi eventualno usporilo program. Čak i samo postavljanje nove promjenljive na nulu oduzima određeno vrijeme, pa to morate uraditi ručno.

✓ *Postavljanje promjenljive na početnu vrijednost naziva se inicijalizacija promjenljive.*

✓ Inicijalizacija

Npr. u prethodnom primjeru bismo mogli deklarisati promjenljivu `broj` sljedećom naredbom: `int broj=0;`, čime bi ona bila inicijalizirana na nulu. Možemo vrijednost promjenljive promijeniti i nekad kasnije u kodu dodjelom vrijednosti: `broj=0;`

Koju vrijednost promjenljiva ima ako je nismo inicijalizirali? Najjednostavnije rečeno, promjenljiva će dobiti neku vrijednost koja se zatekla na memorijskoj lokaciji koja joj je pridružena. Recimo da ste koristili neki drugi program koji se završio, taj program je sadržavao promjenljivu `x` koja je imala vrijednost 256. Kada pokrenemo naš program, promjenljiva `broj` će možda dobiti istu tu lokaciju te će njena vrijednost biti također 256. Ali to *nije garantovano* – u biti mi ne znamo kakvu vrijednost će imati ovakva promjenljiva. Takve promjenljive se nazivaju **neinicijalizirane promjenljive**. Raditi bilo šta s njima je opasno jer unosi neizvjesnost u naš program, te zbog toga svaku novu promjenljivu najprije morate inicijalizirati. Nije bitno da li to radite odmah prilikom deklaracije (kao iznad) ili nekad kasnije, važno je da je promjenljiva inicijalizirana prije nego što probamo pristupiti njenoj vrijednosti.

Obratite pažnju da, za razliku od nekih drugih jezika, prevodilac programskog jezika C nas često neće upozoriti ako radimo sa neinicijaliziranom promjenljivom. Potrebni su posebni programi koje nazivamo *profileri* koji analiziraju programe za vrijeme rada kako bi nas upozorili na ovakve stvari. Jedan takav program naziva se valgrind.

1.2.3 *Deklaracija i tipovi promjenljivih*

Vratimo se našem programu. Linija 3 je glasila:

```
3  int a,b,c;
```

Ovom naredbom smo deklarisali promjenljive koje se zovu `a`, `b` i `c`, a sve tri su tipa `int` što je skraćeno od *integer* odnosno cijeli broj. Generalno promjenljive se deklarišu naredbom oblika:

1755198

```
TIP var1, var2, var3, var4...;
```

Umjesto `TIP` navodimo jedan od tipova koje podržava programski jezik C. Najčešći tipovi su:

1. *int* – cijeli brojevi u intervalu najčešće $[-2^{31}, 2^{31})$,
2. *float* – realni brojevi jednostruke (male) preciznosti,
3. *char* – jedan znak (slovo, cifra, znak interpunkcije),
4. *double* – realni brojevi dvostruke preciznosti,
5. *long* – cijeli brojevi na (teoretski) većem intervalu.

Postoje i drugi tipovi o kojima ćemo govoriti u poglavlju 3.

✓ Kada koristiti cijele, a kada realne brojeve?

U životu postoje veličine koje su *prebrojive* odnosno predstavljaju broj nekih nedjeljivih cjelina, pa je za njih prirodno koristiti cjelobrojni tip. Npr. broj jabuka u korpi ili broj stanovnika nekog grada možemo predstaviti promjenljivom tipa `int`. No realni svijet uglavnom je sastavljen od veličina koje imaju i decimalni dio, stoga se takvi brojevi i nazivaju *realni brojevi*. Npr. kada govorimo o visini ili težini, tu vrijednost ćemo vjerovatno zaokružiti na cijeli broj centimetara odnosno kilograma, pa ćemo za nekoga reći da ima 57 ili 58 kilograma. Ali nekada kasnije nam može zatrebati da iskažemo težinu od baš 57,5 kilograma, pa je bolje da se za to pripremimo tako što ćemo koristiti realni tip podataka.

Za realne vrijednosti u pravilu trebamo koristiti tip `double` koji je precizniji od tipa `float` ali zauzima dvostruko više memorije.

Nakon oznake tipa slijedi razmak te ime promjenljive ili imena promjenljivih razdvojena zarezom. Promjenljive se mogu odmah i inicijalizirati tako što se nakon naziva promjenljive navede znak jednakosti i željena vrijednost. Na kraju je naravno tačka-zarez.

✓ Identifikator

Imena promjenljivih (i drugih identifikatora kao što su imena funkcija, labela itd.) moraju slijediti određena pravila. Ime može sadržavati samo velika i mala slova, cifre i znak "donja crta" (eng. underscore), pri čemu ime ne može počinjati cifrom (ali može donjom crtom). Kada bi bilo moguće npr. ime `a+b`, kako bismo znali da li je to jedna promjenljiva koja se zove `a+b` ili zbir promjenljivih `a` i `b`? Vidjećete da gotovo svi znakovi na tastaturi osim donje crte u programskom jeziku C imaju posebna značenja, baš kao i plus u prethodnom primjeru.

Obratite pažnju da se u imenima razlikuju velika i mala slova, pa bi tako `broj`, `BROJ` i `Broj` bile tri različite promjenljive. Ipak, nepisano pravilo u C-u je da se promjenljive imenuju isključivo malim slovima, a ukoliko se ime promjenljive sastoji od više riječi koje želite posebno istaknuti, razdvajate ih donjom crtom npr.

```
float ovo_je_vrlo_dugacko_ime_promjenljive;
```

17570198

Određena imena promjenljivih su zabranjena jer se već koriste za ključne riječi programskog jezika C. Ovakva imena se nazivaju *rezervisane riječi*, a primjeri rezervisanih riječi su `return`, `int`, `for` i slično. Primijetite da `printf` nije rezervisana riječ nego funkcija koja je definisana u biblioteci `stdio.h`.

✓ Rezervisana riječ

1.2.4 *Formati i escape sekvence*

Nastavljamo dalje analizirati posljednji primjer. U linijama 4, 6 i 9 nalazi se funkcija `printf` sa kojom smo se već susreli, pa je ne moramo ponovo objašnjavati... ili ipak moramo? Kakvi su ovo postotci i znakovi[8] u liniji 9? Proučićemo detaljnije ovu liniju:

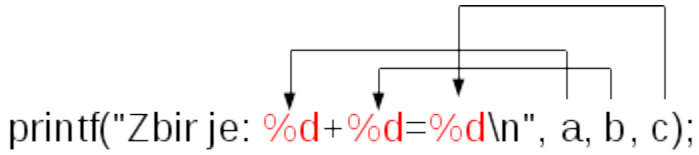
[8] Detaljnije o formatima i escape sekvencama u prilogu B (str. 353).

```
9  printf("Suma brojeva %d i %d je %d\n", a, b, c);
```

Kada u tekstu koji želite ispisati funkcijom `printf` navedete znak postotak, on se neće kao takav ispisati. Umjesto toga, na njegovo mjesto će biti ubačena vrijednost odgovarajuće promjenljive. Slova koja se nalaze iza postotka pobliže određuju kako će ta vrijednost biti ispisana. Vrijednosti koje želimo ispisati navodimo nakon zatvorenih dvostrukih navodnika a prije zatvorene male zagrade, te ih odvajamo znakom zareza.

Ovdje dolazimo do onog slova `f` u `printf`. Vjerovatno znate da "print" na engleskom znači "ispisati", dok "f" dolazi od riječi "format". Oznake koje počinju znakom postotka zovemo *oznake za formatiranje*. Ove oznake se moraju po broju i tipu slagati sa vrijednostima koje ispisujemo.

✓ Oznake za formatiranje



Slika 1.2: Način uvrštavanja formata u pozivu `printf` funkcije

Na slici 1.2 vidimo da će se na mjesto prvog znaka `%d` ubaciti vrijednost promjenljive `a`, druga oznaka `%d` će biti vrijednost promjenljive `b`, a treća će biti njihov zbir – promjenljiva `c`. Slovo `d` označava cijeli broj (tip `int`). Najčešće oznake su:

- `%d` ili `%i` – tip `int`
- `%ld` – tip `long`
- `%f` – tipovi `float` i `double`
- `%c` – tip `char`

17570198

Ako želimo na ekranu ispisati znak postotka, trebamo pisati `%%` jer bi samo jedan znak `%` stvorio zabunu, pošto bi prevodilac pokušao protumačiti sljedeći znak kao oznaku formata.

Između znaka postotka i slova koje označava tip, možemo navesti još određene parametre ispisa koji se nazivaju *podspecifikatori* (eng. subspecifiers).↑ Neki od često korištenih podspecifikatora su:

✓ Podspecifikator

↑ Poglavlje B.1.2, str. 354.

- plus (`+`) – označava da se mora ispisati predznak čak i ako je broj pozitivan,
- broj (npr. `%5d`) – rezerviše određeni broj mjesta na ekranu (u ovom slučaju 5) za ispis, pri čemu su preostala mjesta popunjena razmacima,
- minus (`-`) – poravnava ispis ulijevo, zajedno sa prethodnim podspecifikatorom,
- nula (`0`) – forsira ispis vodećih nula koje se inače ne ispisuju, također u kombinaciji sa brojem,
- tačka (`.`) – zaokružuje realni broj na određeni broj decimala (što će biti objašnjeno uskoro).

Plus, minus, broj i nulu možemo demonstrirati sljedećim kratkim programom:

```
1  #include <stdio.h>
2  int main() {
3    int x=2;
4    printf("%d\n", x);
5    printf("%+d\n", x);
6    printf("%5d\n", x);
7    printf("%-5d!\n", x);
8    printf("%05d\n", x);
9    return 0;
10 }
```

Na ekranu će dati sljedeći izlaz:

```
2
+2
    2
2    !
00002
```

Predznak plus se kod pozitivnih brojeva inače ne ispisuje, ali formatom `%+d` u liniji 5 postigli smo da se na ekranu ispiše `+2`. Format `%5d` je rezervisao 5 mjesta za ispis broja, od čega je broj 2 zauzeo krajnje desno mjesto a ostala mjesta su popunjena razmacima. Kod formata `%-5d` rezervisano je ponovo 5 mjesta ali je broj 2 na krajnje lijevom mjestu, što možemo vidjeti tako što je dodat ispis znaka uzvičnik. Konačno, format `%05d` u rezervisana mjesta upisuje nule.

Kod korištenja znaka tačka, lijevo od tačke navodi se broj rezervisanih mjesta, a desno se nalazi broj decimala. Broj će biti ispisan sa

17/5/10/98

navedenim brojem decimala bez obzira koliko ih ustvari ima, pa ako je broj decimala veći doći će do zaokruživanja, a ako je manji biće na preostalim mjestima ispisane nule. Npr. ako promjenljiva `x` ima vrijednost 2.5, naredba `printf("%5.2f", x);` će ispisati

```
 2.50
```

Ispisane su dvije decimale (druga je nula), a pošto je rezervisano ukupno 5 mjesta, lijevo od cifre 2 se nalazi jedan razmak.

Obratite pažnju da ako nismo rezervisali dovoljno prostora, broj neće biti skraćen, umjesto toga ispis broja će se jednostavno nastaviti u desnu stranu. Tako ćemo za kôd:

```
printf("%3.2f proba\n", 12.345);
printf("%3d proba\n", 6789);
```

dobiti sljedeći rezultat:

```
12.345 proba
6789 proba
```

Isti efekat ćemo dobiti ako uopšte ne navedemo broj rezervisanih mjesta npr. `%.2f`.

Određeni znakovi se ne mogu direktno navesti unutar stringa, pa ih navodimo posebnim oznakama koje se nazivaju *escape sekvence*.↑ Jednu takvu sekvencu smo već upoznali, a to je `\n` koji služi za prelazak u novi red. Često korištene escape sekvence su:

✓ Escape sekvence

↑ Poglavlje B.1.3, str. 355.

- `\n` – novi red,
- `\t` – tabulator, prikazuje određeni broj razmaka kako bi kursor za ispis teksta bio poravnat sa imaginarnom vertikalnom linijom na ekranu (korisno za prikaz tabela),
- `\\` – kada želimo na ekranu ispisati sam znak backslash.

1.2.5 *Unos promjenljivih pomoću funkcije scanf*

Vraćamo se na naš primjer sa početka poglavlja "Promjenljive i ulaz/izlaz".↑ Linije 5 i 7 sadrže poziv funkcije `scanf`:

↑ Str. 27.

```
5 scanf("%d", &a);
```

Funkcija `scanf` služi za unos vrijednosti pomoću tastature u promjenljivu.[9] Kada izvršenje programa dođe do linije 5, program će se zaustaviti, u prozoru za izvršenje će biti prikazan kursor (koji će možda treperiti), čime se signalizira da program čeka da nešto unesete. Naredba u sljedećoj liniji 6 se neće izvršiti sve dok ne unesete neki broj i pritisnete tipku Enter. Inače tipka Enter signalizira da je unos završen i sve dok ne pritisnete ovu tipku neće se desiti ništa, možete popravljati uneseni broj koristeći tipku Backspace sve dok ne potvrdite unos.

[9] Više o funkciji `scanf` u prilogu B (str. 353).

1755011198

Kada konačno pritisnete Enter, uneseni broj će se upisati u promjenljivu `a`. No šta se dešava ako unesete nešto što nije broj npr. slovo? U tom slučaju promjenljiva će ostati nepromijenjena. Ako unesete cifre pa zatim slova, samo cifre će biti unesene a ostatak unesenog teksta će biti zanemaren.

Sintaksa funkcije `scanf` je vrlo slična funkciji `printf`. Formati imaju isto značenje, uz jednu interesantnu razliku: za tip **`double`** u funkciji `printf` koristite format `%f`, ali u funkciji `scanf` trebate pisati `%lf`. Nakon teksta sa formatima navodimo promjenljive kao i u pozivu funkcije `printf`, s tim što prije svake promjenljive moramo navesti znak ampersand `&`. Razlog zašto će nam biti jasan tek kasnije.↑ Za sada zapamtite da ako slučajno izostavite ovaj znak, program će se vjerovatno krahirati.[10]

↑ Poglavlje 7.2, str. 229.

[10] U računarstvu *krahiranje* (eng. crash) označava prijevremeni prekid rada programa uslijed izvršenja nedozvoljene instrukcije. Više o uzrocima ove vrste krahiranja ćemo govoriti u poglavlju 7.1.3 (str. 226).

Obratite pažnju da ono što se nalazi pod navodnicima u funkciji `scanf` se neće ispisati na ekran, nego će se očekivati da korisnik to unese. Npr. mnogi očekuju da će sljedeća naredba:

```
scanf("Unesite broj: %d", &a);
```

proizvesti izlaz:

```
Unesite broj:
```

Ustvari naredba iznad bi očekivala da korisnik sam ukuca tekst "Unesite broj:" i zatim unese broj. Da biste dobili efekat kao u primjeru, morate otkucati kod:

```
printf("Unesite broj: ");
scanf("%d", &a);
```

Naredbom scanf možete istovremeno unijeti više vrijednosti. Npr. ako napišete:

```
scanf("%d,%d", &a, &b);
```

program će očekivati da korisnik unese dva broja razdvojena zarezom. Umjesto zareza možete navesti i neki drugi znak. Ako želite da korisnik razdvoji brojeve razmakom možete navesti format `"%d %d"` ili čak spojeno `"%d%d"`, a u tom slučaju korisnik može razdvojiti brojeve i novim redom.

Čitajući poglavlje 1.2.2 o inicijalizaciji promjenljivih, vjerovatno ste se zapitali da li su naše promjenljive `a` i `b` korektno inicijalizirane. Odgovor je da jesu – iako nemamo karakterističnu dodjelu inicijalne vrijednosti (npr. `a=0`), sama naredba `scanf` će dodijeliti početnu vrijednost promjenljivoj prije nego što je probamo iskoristiti za sabiranje.

17570198

1.2.6 *Dodjela*

Sljedeća linija glasi:

```
8  c = a + b;
```

Ova linija ne opisuje matematičku jednakost. U programskom jeziku C data linija predstavlja *naredbu dodjele*.

✓ *Naredba dodjele funkcioniše na sljedeći način: najprije se izračunava vrijednost izraza navedenog desno od znaka "jednako", a zatim se ta vrijednost upisuje u promjenljivu koja se nalazi lijevo od znaka "jednako".*

✓ Naredba dodjele

Najjednostavnija dodjela je ona kojom promjenljivoj dajemo neku fiksnu vrijednost tj. vršimo njenu inicijalizaciju:

```
x = 2.5;
```

ali vidimo da se tu također može nalaziti i neki matematički izraz koji treba izračunati.

Na sljedećem primjeru možemo vidjeti po čemu se naredba dodjele razlikuje od matematičke jednakosti:

```
a + b = c;
```

Iako bi u matematici ovakav iskaz bio identičan prethodnom, u programskom jeziku C dobili bismo sljedeću poruku greške:

```
Invalid lvalue in assignment
```

Šta bi ovo moglo značiti? Kod čitanja grešaka važno je da ne odustanemo, nego da ih probamo rastumačiti:

- *assignment* na engleskom znači dodjela;
- *lvalue* odnosno *left value* – ono što se nalazi tj. može nalaziti lijevo od dodjele;
- *invalid* – neispravna.

Dakle, ono što smo naveli lijevo od dodjele (znaka "jednako") nije validno jer to nije promjenljiva nego matematički izraz. Izrazu se ništa ne može dodijeliti.

Također obratite pažnju da `a=b` i `b=a` imaju različito značenje. Još jedan interesantan primjer koji često zbunjuje ne-programere je čuvena naredba:

```
x = x + 1;
```

Neka je `x` prije ove naredbe bilo 5. Najprije se izračunava vrijednost desno od dodjele, a to je $5 + 1 = 6$. Zatim se ta vrijednost upisuje u promjenljivu lijevo – `x`. Dakle, efekat naredbe `x=x+1` je da uvećava `x` za jedan.

17570198

1.2.7 *Literali*

U naredbi

```
x = 5;
```

broj 5 nazivamo *doslovce navedena vrijednost* ili *literal*. U primjeru sa kojim smo započeli knjigu

```
printf("Dobro dosli!");
```

string `"Dobro dosli"` je također literal.

Literali u programskom jeziku C imaju tip, pa je literal 5 po definiciji tipa `int`, a literal `5.0` bi bio tipa `double`.[11] Kao što znate, početne i krajnje nule se mogu izostaviti, pa tako i nulu koja se nalazi iza tačke možemo izostaviti. Tako, nekad nam je potrebno da literal ipak bude nekog realnog tipa (jedan takav slučaj ćemo vidjeti ubrzo), pa tada pišemo `5.` što je isto što i `5.0`.

U engleskom jeziku koristi se decimalna tačka umjesto decimalnog zareza, pa se to prenijelo i u sve poznate programske jezike.

[11] Ako želimo literal tipa `float`, trebamo dodati slovo F ovako: `5.0F` ili `5.F`. Slično, za tip `unsigned` se koristi slovo U a slovo L za `long`.

1.2.8 Neki osnovni matematički operatori

U naredbama `x=x+1` i `c=a+b` vidjeli smo da se znak plus može koristiti ako želimo da izračunamo zbir dva broja. Plus je primjer *operatora*. U programskom jeziku C, operator je bilo koji znak koji određuje da je potrebno izvršiti neku (matematičku ili drugu) operaciju. Operator ima jedan, dva ili tri operanda, te ima određenu vrijednost koja se određuje na osnovu operanda. Pa tako, operator plus ima dva operanda, a daje vrijednost zbira ta dva operanda.

Često korišteni operatori su:

- plus (`+`) – sabiranje,
- minus (`-`) – oduzimanje,
- zvjezdica (`*`) – množenje,
- kosa crta (`/`) – dijeljenje,
- postotak (`%`) – ostatak pri cjelobrojnom dijeljenju.

Za razliku od nekih programskih jezika, u C-u ne postoji operator za stepenovanje. Ako želite da stepenujete, morate koristiti funkciju `pow` koja se nalazi u biblioteci `math.h`. Npr. da biste izračunali $y = x^5$ trebali biste pisati

```
y = pow(x,5);
```

Obratite pažnju da operator postotak očekuje da operandi budu cijeli brojevi. Ukoliko probate primijeniti postotak na realne brojeve, dobićete grešku.

Svi nabrojani operatori su *binarni*, odnosno imaju dva operanda. Postoje i *unarni* operatori koji imaju samo jedan operand. Primjer unarnog operatora je unarni minus čija vrijednost je negirana vrijednost operanda:

17570198

```
b = -a;
```

Tu su i *kombinovani operatori* koji istovremeno obavljaju dodjelu i neku drugu operaciju. Najpoznatiji kombinovani operator je `++` koji uvećava vrijednost promjenljive za jedan. Kada napišete `x++` to je ekvivalentno naredbi `x=x+1`, dakle istovremeno se vrši sabiranje jedinice i dodjela tako izračunate vrijednosti promjenljivoj `x`. Tu je i operator `--` koji umanjuje promjenljivu za jedan.

Kombinovani operator `+=` je binarni operator koji uvećava vrijednost promjenljive za neku vrijednost koja se nalazi sa desne strane. `x+=6` je isto što i `x=x+6`. Slični operatori su `-=`, `*=`, `/=` i `%=`.

Ovim smo samo zagrebali površinu operatora u programskom jeziku C, a detaljniju diskusiju koja uključuje ternarni operator (operator sa tri operanda), neke egzotične operatore kao što je operator zarez, i druge, imaćemo u poglavlju 3.2.↑

↑ Str. 79.

1.2.9 *Ulaz i izlaz u programskom jeziku C++*

I u programskom jeziku C++ možemo koristiti funkcije `printf` i `scanf` za ulaz i izlaz. Mada bi primjer iz prethodnih poglavlja trebao raditi bez izmjene, prevešćemo ga u "C++ stil":

```
1  #include <cstdio>
2  int main() {
3    int a,b,c;
4    std::printf("Unesite prvi broj: ");
5    std::scanf("%d", &a);
6    std::printf("Unesite drugi broj: ");
7    std::scanf("%d", &b);
8    c = a + b;
9    std::printf("Suma brojeva %d i %d je %d\n", a, b, c);
10   return 0;
11 }
```

Uočite u liniji 1 da smo umjesto biblioteke `stdio.h` uključili odgovarajuću C++ verziju biblioteke `cstdio`, te da smo ispred svakog poziva funkcija `printf` i `scanf` dodali prefiks `std::`.

Mada je ovaj program sada u potpunosti u skladu sa C++ standardom, možemo reći da nije pisan "u duhu" programskog jezika C++. Programski jezik C++ je jezik "višeg nivoa" od programskog jezika C, te se i funkcije `printf` i `scanf` smatraju funkcijama preniskog nivoa za upotrebu u C++ programima. Identičan program koji bi bio potpuno "u duhu" programskog jezika C++ glasi:

```
#include <iostream>
int main() {
  int a,b,c;
  std::cout << "Unesite prvi broj: ";
  std::cin >> a;
  std::cout << "Unesite drugi broj: ";
  std::cin >> b;
  c = a + b;
```

```
9   std::cout << "Suma brojeva " << a << " i " << b << " je " << c << std::
       endl;
10  return 0;
11 }
```

Da vidimo šta sve novo otkrivamo u ovom primjeru. U liniji 1 uključena je biblioteka `iostream` koja sadrži nove metode rada sa ulazom i izlazom višeg nivoa. Među njih spada i tzv. *objekat izlaznog toka* `cout` koji služi za ispis teksta na ekran. `cout` se koristi tako što nakon navođenja njegovog imena koristimo operator `<<` te slijedi tekst ili promjenljiva čiju vrijednost treba ispisati. Primjer slučaja kada treba ispisivati mješovito tekst i vrijednosti promjenljivih možemo vidjeti u liniji 9: svaki put kada nakon teksta treba ispisati vrijednost promjenljive, zatvaramo navodnike, ponovo navodimo `<<`, te zatim navodimo i naziv promjenljive. Prelazak u novi red možemo postići sa `\n` kao i u C-u, ali za bolju portabilnost preporuka je da koristimo *objekat za prelazak u novi red* – `endl`.

Za unos promjenljivih sa tastature koristimo *objekat ulaznog toka* koji se zove `cin`. Ovdje sada koristimo operator `>>` koji asocira na strelice koje pokazuju sa objekta na promjenljivu, na taj način simbolizujemo da se vrijednost unosi u promjenljivu.

O ulazu i izlazu u programskom jeziku C++ bi se moglo pisati još dosta, no za ovaj početak je dovoljno.

1.3 *Rješavanje programskih zadataka*

Umjetnost programiranja uključuje rješavanje mnogih problema od kojih svaki ima svoju težinu, kao što su:

- specificiranje zahtjeva za program, što u pravilu zahtijeva razgovor sa klijentom;
- analizu problema;
- dizajn rješenja, od hardverske, softverske, mrežne itd. arhitekture do finesa i detalja kao što su korišteni algoritmi i strukture podataka;
- implementacija;
- testiranje.

Kada tek učite programirati, ne možete direktno skočiti na rješavanje problema ovakve složenosti, jer ćete se naprosto izgubiti, da ne govorimo da bi realističan projekat ovog tipa trajao nekoliko mjeseci.

Umjesto toga, mi ćemo krenuti od nekih vrlo jednostavnih primjera koji najčešće i nemaju mnogo dodira sa realnošću. Programi koje budete pravili čitajući ovu knjigu će koristiti naredbe poput `printf` ili

1/1E/7/01/98

`scanf` kako bi se prikazivao tekst na ekranu i čitali ulazi sa tastature, dok su većina modernih aplikacija mobilne ili web aplikacije koje izgledaju i rade potpuno drugačije. Programi će na početku imati svega nekoliko desetina linija, a dovršavaćete ih za 15-ak minuta. Njihov cilj nije da budu kompletne aplikacije koje će neko koristiti nego da demonstriraju neki koncept iz programiranja.

Ovi "školski" primjeri u pravilu izgledaju recimo ovako:

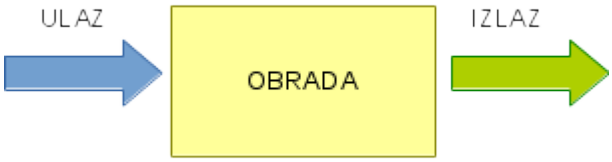
Zadatak 1. *Napisati program koji na osnovu datog poluprečnika kružnice izračunava površinu kruga kojeg opisuje ta kružnica i zapreminu lopte nastale rotacijom kruga oko svoje osi.*

Kako pristupiti rješavanju ovakvog zadatka? Podsjetimo se da svi računarski programi, od web aplikacija do igara, u suštini rade istu stvar: uzimaju neke ulazne vrijednosti i transformišu ih kako bi se dobile izlazne vrijednosti (slika 1.3). Vaš zadatak je da odredite ulaze, izlaze, te matematičku (ili neku drugu) formulu kojom transformišete ulaz u izlaz.

Vratimo se zadatku. Šta bi ovdje bili ulaz, obrada i izlaz?

- *ulaz* – poluprečnik r
- *izlaz* – površina P, zapremina V
- *obrada* – $P = r^2\pi$, $V = \frac{4}{3}r^3\pi$

Preostaje da ovo pretvorimo u C kod.



Slika 1.3: Osnovni princip rada svakog računarskog programa

```
1  #include <stdio.h>
2  #define PI 3.1415926
3  int main() {
4    float r, P, V;
5
6    /* Ulaz */
7    printf("Unesite poluprecnik: ");
8    scanf("%f", &r);
9
10   /* Obrada */
11   P=r*r*PI;
12   V=(4./3)*r*r*r*PI;
13
14   /* Izlaz */
15   printf("Povrsina kruga sa poluprecnikom %.2f je %.2f\n", r, P);
16   printf("Zapremina kugle sa poluprecnikom %.2f je %.2f\n", r, V);
17   return 0;
18 }
```

U liniji 12 vidimo da je korišten literal↑ realnog tipa `4.` Razlog za to je što kada bismo pisali `(4/3)` dobili bismo rezultat 1 jer su oba operanda tipa `int` pa bi i rezultat bio tipa `int`. U ovom slučaju zagrade oko izraza `4./3` su navedene radi jasnoće, ali nisu potrebne odnosno rezultat bi bio isti i bez njih.[12]

↑ Poglavlje 1.2.7, str. 35.

[12] Operatori `*` i `/` su istog prioriteta i lijevo su asocijativni. Pojmovi *prioritet* i *asocijativnost* objašnjeni su u poglavlju 3.2, str. 79.

1.3.1 *Simboličke konstante*

U ovom programu možemo uočiti nekoliko novih i interesantnih stvari.

U liniji 2 definisali smo *simboličku konstantu*. Konstanta je vrijednost koja se ne može promijeniti za vrijeme rada programa. U programskom jeziku C konstante definišemo `#define` preprocesorskom direktivom nakon koje slijedi naziv konstante, te njena vrijednost razdvojena razmakom. Obratite pažnju da u preprocesorskim direktivama nema znaka tačka-zarez, te da nema dodjele (znaka "jednako").

Preprocesorske direktive su u suštini instrukcije za prevodilac koji će ih iskoristiti na određeni način kako bi proizveo izvršnu datoteku. U slučaju direktive `#include` biće uključena biblioteka koja se nalazi u zasebnoj datoteci, a u slučaju `#define` prevodilac će doslovce proći kroz kod i zamijeniti svaku pojavu konstante sa onim što je navedeno nakon razmaka.

Ovo možemo provjeriti sljedećim interesantnim primjerom:

```
1 #include <stdio.h>
2 #define pisi printf
3 int main() {
4   pisi("Ovo je primjer #define direktive");
5   return 0;
6 }
```

Dakle bilo gdje u programu da navedemo riječ `pisi` (osim naravno unutar navodnika), ona će biti zamijenjena sa `printf`. Kao da smo definisali vlastitu naredbu!

Ova osobina zamjene često ima neočekivane posljedice. Recimo, mnogi bi očekivali da će ovaj program ispisati broj 25:

```
#define pet 2+3
printf("%d", pet*pet);
```

No, na ekranu će se ispisati 11, jer nakon zamjene dobijamo ustvari sljedeću liniju:

```
printf("%d", 2+3*2+3);
```

pa pošto množenje ima veći prioritet od sabiranja, najprije će se pomnožiti 3*2 što daje 6, a zatim imamo 2+6+3. Zbog toga se praktikuje da, ako u vrijednostima simboličkih konstanti imamo ikakve izraze sa operatorima, taj izraz stavimo u zagrade, npr. ovako:

```
#define pet (2+3)
```

17570198

Kao što je običaj da se promjenljive imenuju malim slovima, konstante se u pravilu imenuju svim velikim slovima npr.
`NEKA_SIMBOLICKA_KONSTANTA`

1.3.2 *Komentari*

Nakon što smo ovo razjasnili, program bi nam trebao postati očigledan. U linijama 7-8 nalazi se ulaz, u linijama 11-12 obrada, a u linijama 15-16 izlaz. No ovo smo svakako daleko lakše uočili zahvaljujući *komentarima* koji se nalaze u linijama 6, 10 i 14.

Komentar je dio koda koji prevodilac ignoriše, dakle koji ne obavlja nikakvu korisnu funkciju. Zašto ih onda pišemo? Zbog čovjeka koji će čitati kod. Generalno, mnoge stvari koje radimo u programiranju služe boljoj čitljivosti programa.

U kodu možemo vidjeti komentare u C-stilu koji počinju znakovima kosa crta i zvjezdica `/*` a završavaju znakovima zvjezdica i kosa-crta `*/`. Pošto imaju definisan početak i završetak, ovi komentari se mogu prostirati preko više linija.

Pored ovog tipa komentara postoje i komentari u C++-stilu koji izgledaju ovako:

```
int a; // a je stranica kvadrata
scanf("%d", &a); // unos
```

Komentari u C++-stilu počinju sa dvije kose crte a završavaju prelaskom u novi red, tako da ako želite nastaviti komentar u sljedećem redu morate ponovo navesti dvije kose crte. Komentare u C++ stilu možete koristiti i u C programima, mada ćete ponekad dobiti upozorenje zbog toga.

1.3.3 *Kamo dalje...*

Sada posjedujete znanje dovoljno da pišete vaše prve programe. Ovi programi za sada se svode na različite računske operacije nad brojevima, što znači da ne koristite niti djelić mogućnosti programskog jezika. Ipak, već u ovom trenutku se možete zainteresirati za ovakav način rada sa računarom.

Tek sa poznavanjem programiranja vi zaista možete koristiti vaš računar. Sve do sada ste isključivo koristili programe koje je pisao neko drugi i bili ste ograničeni idejama tih drugih osoba. Znanje programiranja vam omogućuje da realizirate vlastite ideje.

Mi se, naravno, nadamo da smo vas zainteresirali ovim uvodom i da ćete nastaviti dalje sa učenjem programiranja, koristeći ovu knjigu ili neku drugu. Naš odabir redoslijeda lekcija je posljedica višegodišnjeg iskustva u podučavanju i sigurni smo da će vas uputiti u ono što je bitno, bez obzira kojom vrstom programiranja se planirate baviti.

1755701198

Pitanja za razumijevanje gradiva

1. Čemu služi funkcija `printf`? U kojoj biblioteci se nalazi?

2. Čemu služi naredba `return`?

3. Šta je to promjenljiva (varijabla)?

4. Koja je razlika između tipova `int` i `double`? Kada koristiti jedan a kada drugi?

5. Koja je razlika između tipova `float` i `double`? Kada koristiti jedan a kada drugi?

6. Šta označava format `%d`?

7. Za koji tip promjenljive koristimo različit format u funkcijama `printf` i `scanf`?

8. Šta je to literal?

9. Za koju čestu matematičku operaciju ne postoji operator u C-u i šta se koristi umjesto njega?

10. Šta je to unarni operator? Navedite jedan primjer.

11. Kojom preprocesorskom direktivom se kreiraju simboličke konstante?

17570198

2
Uslovi i grananje

U prethodnom poglavlju smo spomenuli da se programi u pravilu izvršavaju odozgo prema dolje, naredbu po naredbu, ali da posebnim naredbama možemo promijeniti redoslijed izvršenja.↑ Takve naredbe se nazivaju *naredbe za kontrolu toka*.

↑ Poglavlje 1.1.1, str. 24.

U ovom poglavlju ćemo se baviti jednom vrstom naredbi za kontrolu toka koje se zajedničkim imenom nazivaju **grananje**. Pomoću grananja možemo odrediti da se neki blok naredbi hoće ili neće izvršiti ovisno o nekim vrijednostima npr. nečemu što je korisnik unio sa tastature.

2.1 *Vrste grananja*

Kada učimo o grananju obično govorimo o tri vrste grananja. To su:

- opcija,
- alternativa i
- višestruki izbor.

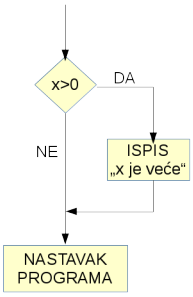
Slike 2.1-2.3 prikazuju ove tri vrste grananja koristeći **dijagram toka** (eng. flow chart). Na ovom dijagramu, mogući pravci izvršenja programa su označeni linijama sa strelicama. Prateći linije možete vidjeti kojim redom se naredbe mogu izvršavati ovisno o vrijednostima promjenljivih.

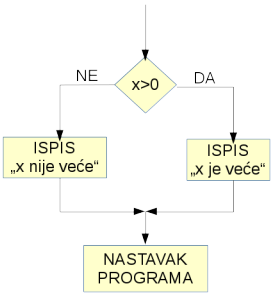
Prva vrsta grananja o kojoj govorimo je *opcija* kod koje se neki dio koda može ali ne mora izvršiti. Recimo, na slici 2.1 možemo vidjeti da će se ispis "x je veće" desiti samo ako je ispunjen uslov $x > 0$, a ako je $x \leqslant 0$ onda se na ekranu neće ispisati ta poruka.

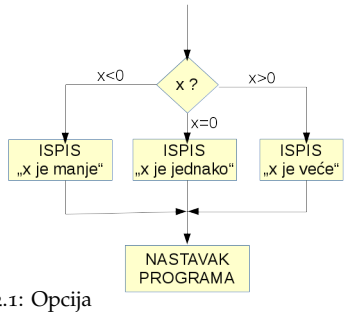
Alternativa je vrsta grananja kod koje postoje dvije grane koda (sl. 2.2). Jedna grana će se izvršiti ako je uslov ispunjen, a druga ako nije. Sigurno je da će se uvijek izvršiti ili lijeva ili desna grana, ne može se desiti da se ne izvrši niti jedna od njih ili obje.

17570198

Konačno, *višestruki izbor* (sl. 2.3) sadrži više grana koda od kojih će se izvršiti jedna ovisno o vrijednosti nekog izraza. U nekim varijantama je moguće i da se ne izvrši niti jedna grana. U primjeru sa slike, promjenljiva `x` je ili veća od nule, ili je jednaka nuli, ili je pak manja od nule, pa se ne može desiti da nijedna od tih varijanti nije ispunjena ili da postoji više mogućnosti koje istovremeno važe.







Slika 2.1: Opcija
Slika 2.2: Alternativa
Slika 2.3: Višestruki izbor

2.1.1 *Naredba if*

U programskom jeziku C grananje se izvodi koristeći ključnu riječ `if` (što na engleskom znači "ako"), a ponekad je prati i `else` (eng. u suprotnom). Način upotrebe ove naredbe je

```
if (uslov)
  blok;
```

Recimo da želimo ispisati poruku "x je pozitivno" ako je promjenljiva `x` veća od nule kao na sl. 2.1. To bismo napisali ovako:

```
if (x > 0) {
  printf("x je pozitivno\n");
}
```

Najprije navodimo ključnu riječ `if`, zatim u malim (običnim) zagradama navodimo uslov koji se ispituje, te nakon toga slijedi blok kôda koji će se izvršiti ako je uslov ispunjen, a ako nije onda će taj blok biti samo preskočen.

✓ *Blok kôda je skup sastavljen od jedne ili više C naredbi koje su grupisane koristeći vitičaste (velike) zagrade kako bismo ih mogli posmatrati kao cjelinu.*

✓ Blok kôda

Već smo se do sada susretali sa blokovima koda, recimo sama funkcija `main` se sastoji od zaglavlja i tijela funkcije koje je jedan blok.

Ako se unutar bloka koda nalazi samo jedna naredba, vitičaste zagrade možemo i izostaviti, pa smo tako u prethodnom primjeru mogli pisati

```
if (x>0)
  printf("x je pozitivno\n");
```

1/15/01/98

što bi značilo isto. Ali zagrade svakako nisu nepotrebne jer smanjuju zabunu, pa preporučujemo početnicima da za sada koriste vitičaste zagrade za svaki blok koda, uključujući i one koji sadrže samo jednu naredbu.

Obratite pažnju na poziciju znaka tačka-zarez u gornjim primjerima. On se navodi nakon cjelovite naredbe koja se nalazi unutar bloka (a to je naredba `printf`), a ne navodimo ga nakon uslova, otvorenih ili zatvorenih vitičastih zagrada. Sjetite se pravila koje smo spomenuli u prvom poglavlju↑ da znak tačka-zarez ne treba stavljati tamo gdje imamo otvorene ili zatvorene vitičaste zagrade. Pogledajte sljedeći pogrešan primjer koda:

↑ Str. 23.

```
if (x>0); {
  printf("x je pozitivno\n");
}
```

Veliki problem ovdje je što prevodilac neće ispisati nikakvu poruku greške, pa čak niti upozorenje! Ovo je sasvim validan C kod iz razloga što znak tačka-zarez, naveden sam za sebe, označava *prazan blok koda*, odnosno možemo reći da je to naredba koja ne radi ništa. Tumačenje primjera koji smo dali iznad bi bilo sljedeće: blok koda koji se nalazi unutar uslova je znak tačka-zarez, a blok koda sa naredbom `printf` se nalazi poslije uslova i izvršava se u svakom slučaju, pa će za svaku unesenu vrijednost `x` program ispisati "x je pozitivno".

Inače, dozvoljeno je da u programu imate blokove koda bez ikakvog posebnog razloga, samo radi grupisanja naredbi.

Navešćemo i jedan primjer kompletnog programa koji sadrži grananje tipa "opcija".

```
1  #include <stdio.h>
2  #define GODINA 2018
3  int main() {
4    int gr;
5    printf("Unesite godinu rodjenja: ");
6    scanf("%d", &gr);
7    if (GODINA - gr < 18) {
8      printf("Roditeljska paznja!\n");
9    }
10   printf("Sada slijedi sadrzaj...\n");
11   return 0;
12 }
```

U liniji 2 je definisana konstanta `GODINA` koja označava trenutno aktuelnu godinu, pa je po potrebi možemo vrlo lako popraviti. U liniji 6 korisnik unosi svoju godinu rođenja u promjenljivu `gr`. U liniji 7 provjeravamo da li korisnik ima minimalno 18 godina tj. da li je razlika trenutne i godine rođenja manja od 18, pa ako jeste na ekranu se ispisuje poruka "Roditeljska paznja!".

Pošto se u bloku koda nalazi samo naredba `printf` u liniji 8, mogli smo izostaviti vitičaste zagrade. No obratite pažnju da ovo može biti opasno ako ikada poželimo dodati još jednu naredbu u blok. Recimo u

1755701198

sljedećem kodu programer je pokušao razdvojiti ispis u dvije zasebne naredbe:

```
#include <stdio.h>
#define GODINA 2018
int main() {
  int gr;
  printf("Unesite godinu rodjenja: ");
  scanf("%d", &gr);
  if (GODINA - gr < 18)
    printf("Roditeljska ");
    printf("paznja!\n");
  printf("Sada slijedi sadrzaj...\n");
  return 0;
}
```

Iako je poravnanje koda razmacima (indentacija) korišteno tako da sugeriše da se naredba `printf("paznja!\n");` nalazi unutar uslova, tj. da će se izvršiti samo ako je uslov ispunjen, nisu korištene vitičaste zagrade tako da će se uvijek ispisati riječ "paznja" čak i ako je korisnik stariji od 18 godina.

✓ `if` nije petlja

Često ćete pri razgovoru čak i sa iskusnijim programerima čuti termin "if petlja". Ovaj termin je pogrešan jer *if nije petlja!* Naredba `if` u svim nama poznatim programskim jezicima je *naredba grananja* (eng. branch) kojom označavamo blok koda koji se može ali i ne mora izvršiti ovisno o nekom uslovu. S druge strane, *petlja* (eng. loop) je blok koda koji se može ponavljati više puta, što je sasvim različita stvar. Sa petljama u programskom jeziku C ćemo se detaljnije upoznati u poglavlju 4.↑

↑ Str. 87.

2.2 *Operatori poređenja*

U prethodna dva primjera vidjeli smo kako se može provjeriti da li je neka vrijednost manja od neke druge vrijednosti koristeći znak "<". Znak "<" je jedan od **operatora poređenja**. U prvom poglavlju smo već spominjali matematičke operatore↑ i rekli smo da npr. izraz `a+b` ima vrijednost zbira promjenljivih `a` i `b`. Šta bi onda bili operatori poređenja?

↑ Poglavlje 1.2.8, str. 36.

✓ *Operator poređenja upoređuje dvije vrijednosti i daje logičku vrijednost istine ili neistine.*

✓ Operator poređenja

U programskom jeziku C ne postoji logički tip podataka (boolean, bool) nego se za tu svrhu koriste cijeli brojevi (tip `int`). Broj 0 označava neistinu (false), a bilo koji broj različit od nule (npr. 1) označava istinu (true).[1] S druge strane, u jeziku C++ postoji tip `bool` koji je interno izveden kao cjelobrojni tip tako da sva razmatranja data ovdje važe i za C++. Dakle, naredba `if` očekuje da će se u zagradama nalaziti neka logička vrijednost – tj. broj 0 ili 1 – a takvu vrijednost možemo dobiti koristeći operator poređenja.

[1] Možemo se osloniti da će operator poređenja uvijek dati vrijednost 1 za istinu, mada naredba `if` prihvata i bilo koju drugu vrijednost različitu od nule kao istinu.

1755/0198

Operatori poređenja u programskom jeziku C su:

- < manje od
- > veće od
- <= manje ili jednako
- >= veće ili jednako
- == jednako
- != različito

Obratite pažnju da jedan znak = označava dodjelu, dok dva znaka == označavaju poređenje (provjeru jednakosti). Dodatnu zabunu može izazvati činjenica da je dodjela u programskom jeziku C ujedno i operator čija je vrijednost ona vrijednost koja je dodijeljena. Tako npr. u C-u možete napisati sljedeći izraz:

```
1  a = 1;
2  b = 2;
3  c = a + (b=3);
4  printf("a=%d, b=%d, c=%d", a, b, c);
```

Značenje koda u liniji 3 bi bilo sljedeće: pošto je izraz `(b=3)` u zagradama, promjenljiva `b` će dobiti vrijednost 3. Nakon toga će se sabrati `a` i vrijednost izraza u zagradi, a naredba dodjele ima vrijednost dodijeljenog (3), tako da će `c` biti 4, pa će se na ekranu ispisati `a=1, b=3, c=4`.

Posljedica ovoga su česte i krajnje podmukle greške sljedećeg tipa:

```
printf("Unesite x: ");
scanf("%d", &x);
if (x=0) {
  printf("Unijeli ste nulu!");
  return 0;
}
a = b / x;
```

Cilj programera u ovom programu je bio da spriječi dijeljenje s nulom u posljednjoj liniji u slučaju da korisnik unese 0. No kakvu god vrijednost za `x` da unesete (čak i nulu) doći će do dijeljenja s nulom! Naime, pošto je u uslovu `x=0` naveden samo jedan znak jednakosti, ovo nije poređenje nego je dodjela, tako da će `x` postati nula bez obzira na prethodnu vrijednost. Osim toga, dodjela ima vrijednost dodijeljenog, a broj 0 u programskom jeziku C označava neistinu, pa uslov neće biti ispunjen i program se neće prekinuti. Nakon toga ćemo preći na liniju `a=b/x` gdje dolazi do dijeljenja s nulom.

Pogledajmo i suprotan primjer:

```
1 #include <stdio.h>
2 int main() {
3   int suma=0, x, y;
```

1/15/01198

```
4   printf("Unesite cijeli broj: ");
5   scanf("%d", &x);
6   printf("Unesite cijeli broj: ");
7   scanf("%d", &y);
8   suma == x + y;
9   printf("Suma je: %d\n", suma);
10  return 0;
11 }
```

Ovaj program će uvijek ispisivati da je suma nula, iz razloga što naredba u liniji 8 ne radi ništa. Naime, `suma==x+y` je savršeno validan C izraz koji provjerava da li promjenljiva `suma` ima istu vrijednost kao i izraz `x+y`, te daje 1 ako su jednaki a 0 ako nisu. Bez obzira koju od tih vrijednosti da smo dobili, mi s njom ne radimo apsolutno ništa, odnosno ovakav kod ima isti efekat kao da smo u našem kodu napisali samo `x+y;` – to je validan C izraz koji ima neku vrijednost ali nema nikakvog efekta na promjenljive. Osim toga, vrijednost ovog izraza nismo nigdje iskoristili (uzaludno smo računali zbir), pa će nas neki prevodioci i upozoriti na to sa porukom "Statement with no effect".

2.3 *Alternativa*

Nakon što smo se upoznali sa operatorima poređenja, vrijeme je da pogledamo i drugu vrstu grananja a to je *alternativa* (sl. 2.2). Za implementaciju alternative u programskom jeziku C koristimo ključnu riječ `else` na sljedeći način:

```
if (x>0) {
  printf("x je pozitivno\n");
} else {
  printf("x NIJE pozitivno\n");
}
```

Dakle, nakon naredbe `if`, uslova i bloka koda koji se izvršava ako je uslov ispunjen slijedi ključna riječ `else` te blok koda koji se izvršava ako uslov nije ispunjen. Obratite pažnju da će se sigurno izvršiti jedan od ova dva bloka. Neko bi recimo mogao umjesto "x NIJE pozitivno" napisati "x je negativno", ali taj program ne bi davao tačan rezultat za slučaj kada korisnik unese broj nula, jer nula nije ni pozitivan ni negativan broj. Rekli smo da se obavezno mora desiti jedan od ova dva slučaja, a ako bismo željeli da pokrijemo i treći slučaj kada je `x` jednako nuli morali bismo koristiti višestruki izbor.

Pogledaćemo i primjer cjelovitog programa sa alternativom:

```
1 #include <stdio.h>
2 #define DPULS 40
3 #define GPULS 120
4 int main() {
5   int puls;
6   printf("Unesite puls: ");
7   scanf("%d", &puls);
8   if (puls >= DPULS && puls <= GPULS) {
```

```
9     printf("Puls je normalan\n");
10  } else {
11    printf("Puls nije normalan\n");
12  }
13  return 0;
14 }
```

Definisali smo dvije konstante koje označavaju gornju i donju granicu normalnog pulsa. Kada korisnik unese puls provjeravamo da li je taj puls u opsegu normalnih vrijednosti, te ispisujemo jednu od dvije poruke u tim slučajevima.

2.4 *Logički operatori*

I u prethodnom programu uočavamo nešto novo, a to je dvostruki znak ampersanda && koji označava logičku konjunkciju (logičko "i"). Operator && je jedan od **logičkih operatora** u programskom jeziku C. Ti operatori su:

- && konjunkcija (logičko "i"), daje vrijednost istine ako su oba operanda istiniti, u suprotnom daje neistinu, primjer: `x>0 && x<3`
- `||` disjunkcija (logičko "ili"), daje vrijednost istine ako je bilo koji od operanda istinit, a ako su oba neistinita daje neistinu, primjer: `x<0 || x>3`
- ! negacija, unarni operator daje vrijednost istine ako je operand neistinit, i obrnuto, primjer: `!(y<1)` (što je isto što i `y>=1`).

Više o operatorima govorićemo u poglavlju 3.2.↑ ↑ Str. 79.

Rekli smo da u C-u logičku istinu označava bilo kakva vrijednost različita od nule, pa tako i logičke operatore možemo direktno primjenjivati na cijele brojeve mada to najčešće i nema baš smisla. Obratite pažnju da logičko "i" && nije isto što i binarno (eng. bit-wise) "i" koje označavamo jednim znakom ampersanda &, mada se često dešava da imaju istu vrijednost!

Zadatak 1. *Koristeći logičke operatore C-a izraziti uslov* $n \in (0,3) \cup (5,100)$.

U matematici kada govorimo o intervalima koristimo male (obične) zagrade da označimo da rubni članovi ne pripadaju intervalu, a uglaste (srednje) zagrade ako pripadaju. Najprije ćemo napisati uslov $n \in (0,3)$. Ovaj uslov važi ako je n veće od nula a manje od tri, dakle oba ova poduslova moraju biti ispunjena, tako da u C-u pišemo `n>0 && n<3`. Slično tome, uslov $n \in (5,100)$ bismo mogli zapisati kao `n>5 && n<100`.

Ostaje da izrazimo operator unije. Broj n pripada uniji dva intervala ako pripada bilo kojem od tih intervala (zasigurno ne može biti

1/5/0198

istovremeno u dva nepreklapajuća intervala), a u slučaju presjeka n bi moralo pripadati u oba intervala istovremeno. Iz toga zaključujemo da za uniju koristimo logički operator disjunkcije, a za presjek operator konjunkcije, tako da kompletan uslov u C-u pišemo kao `if ((n>0 && n<3) || (n>5 && n<100))`. Dodali smo po jedan par zagrada kako bismo označili da se operator disjunkcije odnosi na dva intervala, a ne na pojedinačne izraze `n<3` i `n>5`, ali u C-u operator konjunkcije ima veći prioritet od operatora disjunkcije,↑ pa smo mogli pisati ovaj izraz i bez zagrada a sa istim značenjem:

↑ Poglavlje 3.2, str. 79.

```
if (n>0 && n<3 || n>5 && n<100)
```

Skrenućemo sada pažnju na vrlo čestu grešku kod početnika. Često ćemo naime provjeru pripadnosti intervalu vidjeti ovako opisanu:

```
if (0 < n < 3)
```

Ovo *nije ispravno*! Kako će C protumačiti ovakav izraz? Prije svega, reći ćemo da je operator < *lijevo asocijativan* što znači da se tumači s lijeva na desno, odnosno prethodni izraz je ekvivalentan sa `(0<n)<3`.[2] Dakle prvo će se odrediti vrijednost izraza `0<n`. Npr. ako je korisnik unio broj 10 ovaj uslov će biti ispunjen, pa je vrijednost ovog podizraza logička istina tj. 1. Zatim se poredi `1<3` što je također istina!

[2] Više o asocijativnosti operatora ćemo pričati u poglavlju 3.2, str. 79.

Dakle, za `n=10` izraz `0<n<3` će imati vrijednost istine što zasigurno niste očekivali.[3]

[3] Ustvari imaće vrijednost istine za svaku vrijednost `n`, pa i negativnu.

2.4.1 *Matematička logika u C-u*

Ako ste pažljivo čitali prethodno poglavlje, mogli ste uočiti da programski jezik C ustvari implementira jednu vrstu matematičke logike, tj. *Booleove algebre*. Naime, britanski matematičar George Boole (Džordž Bul) u 19. stoljeću pokušao je da formalizuje oblast *logike* koju su utemeljili još stari Grci, ali koristeći principe poznate iz matematike. Kao što se u nama poznatoj algebri radi sa brojevima, Booleova algebra koristi dvije osnovne *istinosne vrijednosti*, a to su istina $\top$ i neistina $\bot$.

✓ Booleova algebra

Nad istinosnim vrijednostima mogu se primjenjivati *logički operatori* među koje spadaju: konjunkcija $\wedge$, disjunkcija $\vee$ i negacija $\neg$. Vidimo da programski jezik C ustvari sadrži jednu vrstu Booleove algebre pri čemu je:

- $\top$ = `1`
- $\bot$ = `0`
- $\wedge$ = `&&`
- $\vee$ = `||`
- $\neg$ = `!`

17570198

Princip rada logičkog operatora dat je *istinosnom tablicom*, koja navodi vrijednost operacije za sve moguće vrijednosti operanada. Pošto postoje samo četiri moguće kombinacije, ove tablice nisu prevelike. Npr. ovako glase tablice za operatore konjunkcije, disjunkcije i negacije:

Tablica 2.1: Istinosna tablica za konjunkciju $\wedge$, disjunkciju $\vee$ i negaciju $\neg$

A	B	$A \wedge B$
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

A	B	$A \vee B$
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	$\top$	$\top$

A	$\neg A$
$\bot$	$\top$
$\top$	$\bot$

Kada ovo znamo, možemo provjeriti da li neke od zakonitosti matematičke logike važe u programskom jeziku C.

✓ *Zakon neutralnog elementa: Neutralni element za konjunkciju je istina, a za disjunkciju je neistina.*

✓ Zakon neutralnog elementa

Neutralni element za neki operator je onaj koji kada se nađe kao operand, vrijednost operacije je jednaka vrijednosti drugog operanda. Npr. neutralni element za sabiranje je nula (svaki broj sabran sa nulom je isti taj broj), a za množenje je jedan. Recimo neka imamo neki logički iskaz (npr. `x>0`), koji ćemo radi kraćeg pisanja označiti slovom A. Možemo vidjeti na istinosnoj tablici da je `if (A && 1)` istina ako je A istina, a neistina ako je A neistina. Dakle, ovaj uslov je identičan sa `if (A)`, odnosno 1 je neutralni element za operator &&. Također `if (A || 0)` je identično sa `if (A)`, odnosno 0 je neutralni element za operator ||.

✓ *Zakon idempotencije: Konjunkcija ili disjunkcija nekog logičkog iskaza sa samim sobom je isti taj iskaz.*

✓ Zakon idempotencije

Slično kao iznad uočavamo da je `if (A && A)` identično sa `if (A)`, a isto važi i za `if (A || A)`.

✓ *Zakon komutacije: Konjunkcija i disjunkcija su obje komutativne operacije.*

✓ Zakon komutacije

Kada kažemo da je neka operacija komutativna, to znači da operandi mogu zamijeniti mjesta a da značenje kompletnog izraza nije promijenjeno. Uočavamo da bi `if (A && B)` trebalo biti ekvivalentno sa `if (B && A)` (no nije zbog kratkospojne evaluacije, koju ćemo objasniti kasnije). Također i `if (A || B)` bi trebalo biti ekvivalentno sa `if (B || A)`.

✓ *Zakon asocijacije: Konjunkcija i disjunkcija su obje asocijativne operacije.*

✓ Zakon asocijacije

Kažemo da je operacija asocijativna ako redoslijed obavljanja operacija sa više od dva operanda nije bitan sve dok poredak tih operanada

1757/0198

ostaje nepromijenjen. U programskom jeziku C operatori && i || su binarni, dakle uvijek imaju dva operanda, ali možemo ovo pravilo primijeniti tako da grupisanje operanada pomoću zagrada ne mijenja značenje. Recimo posmatrajmo uslov
`if (A && B && C)`. Pošto je operator && lijevo asocijativan, znamo da je ovo isto što i `if ((A && B) && C)`. Ali možemo se uvjeriti da isto značenje ima i uslov `if (A && (B && C))` odnosno važi zakon asocijacije za operator &&. Isto tako su ekvivalentni uslovi `if ((A || B) || C)` i `if (A || (B || C))`.

✓ *Zakon distribucije: Konjunkcija i disjunkcija su međusobno distributivne.* ✓ Zakon distribucije

Distributivnost nam je poznata kod operacija množenja i sabiranja, npr. znamo da matematički izraz $x \cdot (y + z)$ ima istu vrijednost kao i $x \cdot y + x \cdot z$. Ali isto to važi i za konjunkciju i disjunkciju, i to u oba pravca. Uslov `if (A || (B && C))` je ekvivalentan uslovu `if ((A || B) && (A || C))`. Distributivnost za disjunkciju bi dala sljedeće: uslov `if (A && (B || C))` je ekvivalentan uslovu
`if ((A && B) || (A && C))`.

✓ *De Morganov zakon: Negiranje operacije konjunkcije (ili disjunkcije) vrši se tako što se svaki od operanada zasebno negira, a operator konjunkcije se zamjenjuje disjunkcijom (i obrnuto, disjunkcija konjunkcijom).* ✓ De Morganov zakon

Ovaj zakon je najlakše shvatiti kada ga predstavimo programskim kodom. Dakle, ako imamo `if (!(A && B))` i želimo da se riješimo unutrašnjih zagrada, ovaj uslov možemo zamijeniti ekvivalentnim uslovom `if (!A || !B)`. Vidimo da su operandi A i B negirani, a operator && je zamijenjen sa ||. De Morganov zakon za disjunkciju kaže da je `if (!(A || B))` ekvivalentno sa `if (!A && !B)`.

Da pogledamo par primjera kako ova pravila možemo iskoristiti u svakodnevnom programerskom životu.

Zadatak 2. *Pojednostaviti uslov:*
`if (mjesec == 2 && dan < 20 || mjesec == 2 && dan >= 20)`.

Kako bismo lakše shvatili kako se matematička logika primjenjuje u programiranju, sve poduslove u kojima se ne koriste logički operatori zamijenićemo slovima A, B, C itd. Dakle, ako kažemo da A predstavlja `mjesec==2`, B je `dan<20`, a C je `dan>=20`, uslov iznad glasi `if (A && B || A && C)`.

Ovaj uslov prepoznajemo kao rezultat primjene distributivnosti za disjunkciju, pa ovo pravilo možemo primijeniti unazad da dobijemo ekvivalentan uslov `if (A && (B || C))` odnosno
`if (mjesec==2 && (dan<20 || dan>=20)`

1775 0198

Sada možemo uočiti da je uslov u zagradi `(dan<20 || dan>=20)` tautologija: svaka vrijednost promjenljive dan je manja, veća ili jednaka 20. Tautologiju možemo zamijeniti logičkom istinom 1 pa imamo `if (mjesec==2 && 1)`, a pošto je 1 neutralni element za konjunkciju, još kraće se može pisati `if (mjesec==2)`. Uslov se dalje ne može pojednostavljivati.

Pojednostavljivanje uslova je korisno raditi kad god je moguće, jer tako uočavamo logičke greške, a i kod će biti lakši za razumijevanje.

Zadatak 3. *Napisati kod koji na ekranu ispisuje poruku "Neispravan mjesec" ako cjelobrojna promjenljiva mjesec sadrži neispravnu vrijednost.*

Mjesec je ispravan ako se nalazi na intervalu [1,12]. Vidjeli smo ranije da provjeru pripadnosti intervalu pišemo ovako:

`if (mjesec>=1 && mjesec<=12)`

Obratite pažnju da su korištene uglaste zagrade, što znači da vrijednosti 1 i 12 pripadaju intervalu, pa smo u uslovu koristili operatore veće ili jednako i manje ili jednako.

Ovo bi bio uslov koji provjerava da li je mjesec ispravan, no nama treba provjera da li je mjesec *neispravan*. Mogli bismo samo negirati prethodni uslov u zagradi

```
if (!(mjesec>=1 && mjesec<=12))
  printf("Neispravan mjesec");
```

Ovakav kod bi bio tačan, ali ružno izgleda, pa možemo primijeniti De Morganovo pravilo: svaki od uslova za sebe negiramo, a konjunkcija prelazi u disjunkciju:

```
if (!(mjesec>=1) || !(mjesec<=12))
  printf("Neispravan mjesec");
```

Broj koji nije veći ili jednak od jedan je manji od jedan, te slično i za drugi uslov, tako da imamo:

```
if (mjesec<1 || mjesec>12)
  printf("Neispravan mjesec");
```

Ovo je ujedno i primjer provjere da li neka vrijednost *ne pripada* intervalu.

2.4.2 *Kratkospojna evaluacija*

Vratimo se na zakon komutacije koji kaže da je `if (A && B)` ekvivalentno `if (B && A)`.

Ovako bi bilo prema Booleovoj algebri, no u programerskoj praksi ovo nije potpuno tačno. Naime, kada napišemo uslov `if (A && B)`, vrijednosti poduslova A i B se određuju s lijeva na desno, pa ako se odredi da je A neistina (nula), uslov B se neće ni provjeravati. Prema istinosnoj tablici vidimo da ako je jedan od operanada konjunkcije neistina, rezultat je uvijek neistina bez obzira na drugi operand. Na ovaj

1/15/01/98

način računar štedi vrijeme izvršenja programa jer je rezultat već poznat i bez potrebe da određujemo B. Slično važi i za disjunkciju: ako uslov glasi `if (A || B)`, ako se odredi da je A istina (ne-nula) uslov B se neće ni provjeravati. Ovo ponašanje se naziva *kratkospojna evaluacija* (eng. short-circuit evaluation).

Na prvi pogled zaključujemo da nema razlike u rezultatu, ali ova pojava može biti značajna ako neki od operatora ima *sporedne efekte*. Pod pojmom sporednog efekta podrazumijevamo ustvari promjenu vrijednosti promjenljivih koje izaziva operator,[4] o čemu ćemo više govoriti u poglavlju 3.2.↑

[4] Također sporedni efekat može biti i ispis na ekran. Ukratko, sve što ostavlja posljedice može se smatrati sporednim efektom.
↑ Str. 79.

Kao primjer, uzećemo operator dodjele za koji smo rekli da ujedno dodjeljuje vrijednost promjenljivoj na lijevoj strani i vraća dodijeljenu vrijednost. Recimo da uslov glasi:

```
if (x>1 || y=0)
```

Najprije uočimo da se kod `y` nalazi samo jedan znak =. Dakle, ovo je operator dodjele a ne poređenja. Ako je `x` zaista veće od jedan, drugi uslov se neće ni provjeravati (jer već znamo da je kompletan uslov ispunjen) pa će `y` zadržati svoju staru vrijednost. Ali ako je `x` bilo manje ili jednako jedan, `y` će postati nula! U ovom slučaju blok naredbi u uslovu se neće izvršiti jer je nula logička neistina. Za razliku od toga, da smo napisali:

```
if (y=0 || x>1)
```

promjenljiva `y` bi uvijek postala nula.

Naglašavamo da je ovakav kod izuzetno loš i navodi na pogrešne zaključke, pa se početnicima savjetuje da izbjegavaju sporedne efekte u uslovima i po mogućnosti da takve operatore izdvoje u zasebne naredbe radi bolje čitljivosti.

2.5 *Višestruki izbor*

Mogli ste primijetiti da se kroz mnoge primjere u ovom poglavlju provlače dani i mjeseci. Naime, sve ovo je bila priprema za program koji omogućuje korisniku da unese datum svog rođenja te na ekran ispisuje pripadajući horoskopski znak! Sada ćemo se na primjeru programa za horoskop upoznati i sa trećom vrstom grananja, a to je višestruki izbor.

Započeli smo program za ispis horoskopskog znaka ovako:

```
#include <stdio.h>
int main() {
  int dan, mjesec;
  printf("Unesite dan i mjesec rodjenja: ");
  scanf("%d%d", &dan, &mjesec);
  if (mjesec < 1 || mjesec > 12) {
    printf("Mjesec nije ispravan");
  }
```

```
  if (dan < 1 || dan > 31) {
    printf("Dan nije ispravan");
  }
  printf("Vas horoskopski znak je: ");
  ...
  printf("Kraj programa.\n");
  return 0;
}
```

Za sada ćemo zanemariti to što neki mjeseci imaju manje od 31 dana.

Nakon što korisnik unese neke cijele brojeve kao dan i mjesec rođenja, najprije trebamo provjeriti da li su ti brojevi ispravni, o čemu smo govorili kod De Morganovog zakona.↑ Program koji je dat iznad nije zadovoljavajući, jer kakve god brojeve korisnik da unese ispisaće se poruka "Mjesec nije ispravan" ili "Dan nije ispravan", a zatim će se nastaviti ispis horoskopskog znaka. Želimo da se u slučaju neispravnog unosa program prekine.

↑ Str. 52.

Prvo što bi početnik mogao pokušati bi bilo ovo:

```
#include <stdio.h>
int main() {
  int dan, mjesec;
  printf("Unesite dan i mjesec rodjenja: ");
  scanf("%d%d", &dan, &mjesec);
  if (mjesec < 1 || mjesec > 12) {
    printf("Mjesec nije ispravan");
  }
  if (dan < 1 || dan > 31) {
    printf("Dan nije ispravan");
  } else {
    printf("Vas horoskopski znak je: ");
    ...
  }
  printf("Kraj programa.\n");
  return 0;
}
```

Ispis horoskopskog znaka je sada u *else bloku*, što znači da ako dan nije ispravan, ispisuje se odgovarajuća poruka, a ako jeste, ispisuje se horoskopski znak. No, i dalje ostaje problem sa mjesecom. U programu iznad, ako korisnik unese neispravan mjesec a ispravan dan, na ekranu će ipak biti ispisan horoskopski znak.

U ovom programu ustvari postoje tri varijante koje treba predvidjeti:

1. Mjesec nije ispravan.
2. Dan nije ispravan.
3. I mjesec i dan su ispravni, treba ispisati horoskopski znak.

Dakle ovdje nam treba *višestruki izbor*.

Višestruki izbor u programskom jeziku C realizuje se *if-else if-else if...* strukturom. Najprije se provjerava uslov u prvom if-u. Ako je ispunjen, izvršava se odgovarajući blok. U suprotnom se provjerava uslov u drugom if-u itd. Na kraju ove strukture se može nalaziti finalni

17157018 98

else koji se izvršava ako niti jedan od uslova nabrojanih iznad nije ispunjen. No, moguće je i da nema finalnog else-a, što znači da ako nijedan od nabrojanih uslova nije ispunjen, neće se izvršiti ništa.

Ispravna provjera unesenog dana i mjeseca glasila bi:

```
#include <stdio.h>
int main() {
  int dan, mjesec;
  printf("Unesite dan i mjesec rodjenja: ");
  scanf("%d%d", &dan, &mjesec);
  if (mjesec < 1 || mjesec > 12) {
    printf("Mjesec nije ispravan");
  } else if (dan < 1 || dan > 31) {
    printf("Dan nije ispravan");
  } else {
    printf("Vas horoskopski znak je: ");
    ...
  }
  printf("Kraj programa.\n");
  return 0;
}
```

Obratite pažnju da ako su i dan i mjesec neispravni, ovaj program će ispisati samo poruku "Mjesec nije ispravan", jer se drugi uslov uopšte ne provjerava ako je prvi uslov ispunjen. Ako želimo da ispišemo obje poruke, mogli bismo obraditi četvrtu varijantu "Ni mjesec ni dan nisu ispravni", te bismo u tom slučaju prije svih ostalih uslova dodali uslov koji provjerava obje vrijednosti istovremeno.

Nastavljamo program dalje. Recimo da ćemo ispisivati horoskopske znakove po kalendaru, krećući od januara. Osobe rođene između 21. decembra i 20. januara imaju znak Jarac, a rođene između 21. januara i 19. februara znak Vodolija. Ako pokušate napisati po jedan uslov za svaki znak, stvari će se zakomplikovati jer imamo dvije promjenljive mjesec i dan i trebamo provjeriti obje. Mogli bismo recimo pisati ovako:

```
#include <stdio.h>
int main() {
  int dan, mjesec;
  printf("Unesite dan i mjesec rodjenja: ");
  scanf("%d%d", &dan, &mjesec);
  if (mjesec < 1 || mjesec > 12) {
    printf("Mjesec nije ispravan");
  } else if (dan < 1 || dan > 31) {
    printf("Dan nije ispravan");
  } else {
    printf("Vas horoskopski znak je: ");
    if (mjesec == 1 && dan <= 20)
      printf("Jarac");
    else if (mjesec == 1 && dan > 20)
      printf("Vodolija");
    else if (mjesec == 2 && dan <= 19)
      printf("Vodolija");
    else if ...
    ...
  }
```

```
21    printf("Kraj programa.\n");
22    return 0;
23  }
```

Obratite pažnju da uslov u liniji 14 neće uopšte biti provjeravan ako važi uslov u liniji 12, dakle u liniji 14 možemo pretpostaviti da uslov iz linije 12 nije ispunjen. Ako je mjesec januar, a dan nije manji ili jednak 20, dan je sigurno veći od 20 pa to ne moramo posebno provjeravati, tako da u liniji 14 možemo staviti samo `else if (mjesec==1)`.

Naravno, rješenje se moglo i kraće napisati tako što spojimo uslove u jedan npr.

```
14      else if (mjesec == 1 || mjesec == 2 && dan <= 19)
15        printf("Vodolija");
```

Ostatak programa za horoskop možete za vježbu napisati sami po istom principu.

Kod višestrukog izbora strukturom *if-else if-else* važna je upotreba ključne riječi `else`. Početnici često izostave ovu ključnu riječ čime dobiju program koji radi za mnoge (ili čak sve!) slučajeve, ali ovo nije višestruki izbor nego smo tek naveli dvije ili više opcije zaredom. Razlika je u tome što kod višestrukog izbora znamo da će se izvršiti tačno jedna od navedenih grana, ili nijedna ako nemamo finalni else. Kod uzastopnih opcija moguće je da će se desiti obje.

✓ Višestruki izbor vs. niz uzastopnih opcija

2.5.1 *Ugniježđena grananja*

Nakon što smo opisali *if-else if-else* strukturu, važno je da je razlikujemo od ugniježđenih (eng. nested) grananja. Naime, rekli smo da se u bloku koda koji je unutar grananja može nalaziti bilo kakav validan C kod. To znači da se tu može nalaziti i drugo grananje, također i petlja ili bilo šta drugo što vam padne na pamet. Pogledajmo sljedeći primjer iz kojeg je uklonjeno poravnanje (indentacija) radi otežane analize:

```
if (x>0)
if (y>0)
printf("A");
else
printf("B");
```

Kada će se u ovom slučaju ispisati "A" a kada "B"?

Najprije, uočavamo da u primjeru iznad uopšte nema vitičastih zagrada, pošto one nisu obavezne ako se u bloku koda nalazi samo jedna naredba. Jasno je da naredba `printf("B");` čini blok koda koji pripada naredbi `else`, a naredba `printf("A");` čini blok koji pripada uslovu `if (y>0)`.

Kojoj od naredbi `if` pripada `else`? Pravilo je sljedeće: naredba `else` pripada onoj naredbi `if` koja se nalazi neposredno ispred bloka koda, koji se pak nalazi neposredno ispred naredbe `else`. Recimo, sljedeći program bi dao sintaksnu grešku:

✓ Kojem `if`-u pripada `else`?

1/15/2018

```
if (x>0)
if (y>0)
printf("A");
printf("C");
else
printf("B");
```

Greška doslovce glasi "else without if" jer blok koda neposredno prije naredbe `else` čini naredba `printf("C");` (pošto nema vitičastih zagrada, samo jedna naredba je u bloku). Neposredno iznad ovog bloka tj. naredbe se ne nalazi `if` nego naredba `printf("A");`. Dakle, `else`-u nije pridružen `if`. U prethodnom primjeru možemo zaključiti da `else` pripada uslovu `if (y>0)`.

Ostaje da se upitamo šta se tačno nalazi u uslovu `if (x>0)` odnosno gdje su granice tog bloka koda? Naime, naredba `if` zajedno sa pripadajućim blokom koda, naredbom `else` i njenim pripadajućim blokom koda, čini iz perspektive C prevodioca *jednu naredbu* tako da ona kompletna predstavlja jedan blok koda! Drugim riječima, ako bismo ovaj primjer napisali sa svim vitičastim zagradama i poravnanjem, on bi izgledao ovako:

```
if (x>0) {
  if (y>0) {
    printf("A");
  } else {
    printf("B");
  }
}
```

Slovo "A" će se ispisati ako su i `x` i `y` pozitivni. Slovo "B" će se ispisati ako je `x` pozitivno a `y` nije, dok ako `x` nije pozitivno neće se desiti ništa. Pa, kada znamo da `if`, `else` i njima pripadajući blokovi svi zajedno čine jedan blok, možemo pogledati i ovaj primjer:

```
if (x>0)
if (y>0)
printf("A");
else
printf("B");
else
printf("C");
```

Ovo je ponovo potpuno validan C kod, a ostaje da odredite kada će se ispisati slovo "C".

Radi bolje čitljivosti, toplo preporučujemo da kod ugniježđenih grananja koristite vitičaste zagrade bez obzira na to što se u bloku nalazi samo jedna naredba. Ugniježđena grananja su često samo zamjena za logičku konjunkciju, npr. sljedeći kod:

```
if (x>0)
  if (y>0)
    printf("Oba su veca od nule");
```

se kraće i preglednije može zapisati

1778701198

```
if (x>0 && y>0)
  printf("Oba su veca od nule");
```

2.5.2 *switch-case*

Ako u svakom od uslova višestrukog izbora provjeravamo vrijednost jedne te iste promjenljive, taj višestruki izbor se ponekad može kraće i preglednije zapisati koristeći blok **switch-case**. Pored toga, switch-case je i efikasniji jer se uslovni izraz testira samo jednom, a ne kod svakog uslova. Pogledajmo recimo sljedeći program koji na ekranu ispisuje naziv mjeseca urađen na način kako smo radili do sada:

```
#include <stdio.h>
int main() {
  int mjesec;
  printf("Unesite mjesec rodjenja: ");
  scanf("%d", &mjesec);
  if (mjesec == 1) {
    printf("Januar\n");
  } else if (mjesec == 2) {
    printf("Februar\n");
  } else if (mjesec == 3) {
    printf ("Mart\n");
  ......
  } else {
    printf ("Neispravan mjesec!\n");
  }
  return 0;
}
```

Ako korisnik unese broj 12 ovdje će se izvršiti čak 11 operacija poređenja. Možemo isti ovaj program napisati koristeći switch-case na sljedeći način:

```
#include <stdio.h>
int main() {
  int mjesec;
  printf("Unesite mjesec rodjenja: ");
  scanf("%d", &mjesec);
  switch(mjesec) {
    case 1:
      printf("Januar\n");
      break;
    case 2:
      printf("Februar\n");
      break;
    case 3:
      printf("Mart\n");
      break;
    ....
    default:
      printf("Neispravan mjesec!\n");
  }
  return 0;
}
```

Sintaksa *switch-case bloka* je sljedeća:

- Najprije se navodi ključna riječ `switch` te se u običnim zagradama navodi *izraz* čiju vrijednost provjeravamo. Taj izraz može biti samo pojedinačna promjenljiva (kao `mjesec` u prethodnom primjeru), ili to može biti složeniji izraz npr. `switch(x+y-3)`.
- Zatim se otvara vitičasta zagrada te se navodi ključna riječ `case` nakon koje bez ikakvih zagrada slijedi jedna od mogućih vrijednosti promjenljive iz `switch`-a, npr. 1. Nakon toga se navodi *dvotačka* a ne tačka-zarez!
- Ispod `case` navode se naredbe koje se trebaju izvršiti ako kontrolna promjenljiva ima datu vrijednost.
- Ključna riječ `break` označava kraj slučaja.
- Nakon `break` slijedi `case` za sljedeći slučaj itd.
- Ključna riječ `default` (nakon koje se navodi dvotačka) označava kôd koji će se izvršiti ako nijedan od slučajeva nije ispunjen. U našem primjeru to će se desiti ako promjenljiva mjesec nema nijednu od vrijednosti 1,2,...,12. Ovaj `default` blok se može i izostaviti.

Obratite pažnju da kada naiđe na ključnu riječ `break`, prevodilac preskače ostatak switch-case bloka i izvršenje se nastavlja na prvoj naredbi nakon zatvorene vitičaste zagrade (u našem slučaju to bi bila naredba `return 0`). Ova ključna riječ se može i izostaviti, u kojem slučaju će se izvršenje programa nastaviti na sljedećem slučaju. Na primjer, da smo napisali:

```
#include <stdio.h>
int main() {
  int mjesec;
  printf("Unesite mjesec rodjenja: ");
  scanf("%d", &mjesec);
  switch(mjesec) {
    case 1:
      printf("Januar");
    case 2:
      printf("Februar");
    case 3:
      printf("Mart");
    ....
    default:
      printf("Neispravan mjesec!\n");
  }
  return 0;
}
```

17/57/01198

Šta bi se desilo ako korisnik unese npr. broj 2? Na ekranu bi se ispisalo `FebruarMart...Pogresan mjesec`. Drugim riječima, izvršavanje bi korektno skočilo na `case 2`, ali pošto u ovom slučaju nema `break` naredbe, nastavilo bi se sa izvršavanjem slučaja 3, 4 itd. sve dok se ne naiđe na naredbu `break` ili dok se ne dođe do kraja switch-case bloka.

2.6 Kontrola toka naredbom goto

Pošto u ovom poglavlju pričamo o kontroli toka programa, red je da spomenemo i naredbu `goto`. Ova naredba se koristi kako bi se izvršio "skok" na određenu lokaciju u kodu čime se izvršavanje nastavlja od te lokacije. Jedan od načina kako se može iskoristiti ova naredba je da tražimo ponavljanje unosa u slučaju da unesena vrijednost nije ispravna:

```
1  #include <stdio.h>
2  int main() {
3    int mjesec;
4  greska:
5    printf("Unesite mjesec rodjenja: ");
6    scanf("%d", &mjesec);
7    if (mjesec<1 || mjesec>12) {
8      printf("Mjesec je neispravan!\n");
9      goto greska;
10   }
11   switch(mjesec) {
12     case 1:
13       printf("Januar\n");
14       break;
15       ...
16 return 0;
17 }
```

U liniji 4 uočavamo riječ `greska` koja se završava znakom dvotačka. Ako se ne nalazimo u switch-case bloku, dvotačka označava **labelu**. Labela je imenovana lokacija u programu, što znači da kada prevodilac naiđe na labelu on će je samo preskočiti. Svrha labele je da na nju skočimo naredbom `goto`, što možemo vidjeti u liniji 9. Iza naredbe `goto` navodi se labela na koju želimo skočiti (bez dvotačke). Ako je uneseni mjesec manji od 1 ili veći od 12, ispisaće se poruka "Mjesec je neispravan!" nakon čega će izvršavanje ponovo skočiti na liniju 4 u kojoj se nalazi labela `greska`, te će se ponovo ispisati "Unesite mjesec rodjenja:", tražiti unos mjeseca itd. Drugim riječima, unos mjeseca se ponavlja sve dok korisnik ne unese ispravan mjesec.

Početnicima u programiranju naredba `goto` je privlačna, jer je dosta lako shvatiti šta će se desiti. Međutim, ovu naredbu treba izbjegavati, jer rezultira pojavom **špageti-koda**. Razlog zašto se koristi termin "špageti-kod" je to što takav program podsjeća na špagete, nemoguće ga je raspetljati i zaključiti tačno kada će se šta desiti. Morate shvatiti da je za programere važnije da je program lagan za čitanje nego da on radi ispravno, jer neispravan program neko drugi može popraviti. U ovom trenutku, čitanje tuđeg koda djeluje skoro kao nemoguća misija, ali kako budete gradili iskustvo, shvatićete da ono ne samo da je potrebno, nego i da ćete znatno više vremena provoditi čitajući tuđi ili vlastiti kod nego pišući ga.

1755/01198

U svim situacijama, kod koji koristi `goto` naredbu može se prepraviti

tako da se ne koristi ta naredba, pri čemu kod postaje razumljiviji, ukazuje se logika koja stoji u pozadini. Npr. gornji primjer se može ljepše uraditi koristeći petlje.↑ Ipak, postoji par situacija u C-u kada je `goto` najbolje rješenje, npr. za izlazak iz višestruke ugniježđene petlje. Ovo je dosta kontroverzna tema. Mnogi smatraju da je `goto` potpuno nepotreban i da ga treba ukinuti, te u mnogim programskim jezicima ova naredba uopšte ne postoji. Kao i ranije, jezik C slijedi filozofiju da programeru treba dati sve potrebne alate, a on ih zatim treba koristiti odgovorno, pa se isto odnosi i na `goto`. U početnim programima koje budemo obrađivali, praktično nikada se neće desiti da je `goto` ispravan izbor.

↑ Poglavlje 4.3, str. 107.

Pitanja za razumijevanje gradiva

1. Koje vrste grananja poznajemo?

2. Koja naredba se koristi za grananje?

3. Šta su to operatori poređenja (relacioni operatori) i možete li ih nabrojati?

4. Kako označavamo logičku istinu i neistinu u C-u?

5. Pomoću kojih ključnih riječi realiziramo alternativu?

6. Šta su to logički operatori i možete li ih nabrojati?

7. Koje od pravila Booleove logike ne važi u potpunosti u C-u i zašto?

8. Zašto radimo pojednostavljivanje uslova?

9. Kako provjeravamo da li neka vrijednost x *ne pripada* datom opsegu?

10. Kada napišemo ključnu riječ `else`, na koji `if` se ona odnosi?

11. Koje su prednosti korištenja switch-case strukture?

12. Da li je i kada je opravdano koristiti naredbu `goto`?

17570198

Zadaci za vježbu

Zadatak 4. *Napisati program koji učitava tri cijela broja iz intervala od 50 do 200. Ako svi uneseni brojevi nisu iz tog intervala, program treba ispisati poruku "Svi brojevi nisu iz intervala 50 do 200" i završiti s radom. Ako su brojevi unutar tog intervala program treba ispisati učitane brojeve po veličini od najmanjeg prema najvećem, te odrediti koliko različitih brojeva je uneseno. Primjer ulaza i izlaza:*

```
Unesite tri broja: 150 100 150
Brojevi poredani po velicini glase: 100,150,150
Unesena su 2 razlicita broja
```

Zadatak 5. *Napisati program koji učitava koeficijente a, b i c realnog tipa. Ti koeficijenti definiraju funkciju* $f(x) = ax^2 + bx + c$. *Program zatim provjerava da li su ti koeficijenti u rasponu -10 do 10. Ako bilo koji od koeficijenata nije u rasponu, program ispisuje poruku "Koeficijenti a, b i c nisu u zadanom rasponu". Ako koeficijenti a, b i c jesu u zadanom rasponu, program kao izlaz daje vrijednost prve derivacije u tački x. Tačka x se također unosi sa standardnog ulaza. Primjer ulaza i izlaza programa:*

```
Unesite koeficijente a, b i c: 1 2 1
Unesite tacku x: 1
Prva derivacija u tacki x=1 je 4.
```

Zadatak 6. *Napisati program koji će izračunati presjek skupova S1 i S2. Skupovi S1 i S2 definisani su intervalima realnih brojeva:* $S1 = [a, b]$, $S2 = [c, d]$. *Na primjer, ako se za ulaz uzmu skupovi* $S1 = [3, 9]$ *i* $S2 = [7, 11]$, *program kao izlaz treba dati: "Rezultantni interval je [7,9]".*

17570198

1757018 8

3
Skok u svijet programiranja

U dosadašnjem dijelu knjige , nastojali smo da gradivo pojednostavimo kako bismo ga približili početnicima u programiranju. To znači da smo neke stvari morali prešutjeti, jer bi u tom trenutku za čitaoca to moglo biti "previše informacija odjednom".

Kako ulazimo sve dublje u svijet programiranja, pokazuje se da ti prešućeni detalji mogu biti važni, pogotovo kod razumijevanja neobičnih situacija koje nastaju kada pravimo naizgled bezazlene greške u kodu.

Zbog toga je potrebno da se s vremena na vrijeme vratimo na ono što je već rečeno, ali da ovaj put proširimo priču i sa svim onim što ranije nismo rekli. Do sada smo tek vrhovima prstiju opipavali vodu programiranja, sada je vrijeme da napravimo skok. Nadamo se da se nećete udaviti :)

3.1 Predstavljanje podataka u memoriji računara

Ranije smo govorili da u memoriju računara možemo pohranjivati cijele, decimalne brojeve, slova itd. Vidjeli smo i da ti brojevi nisu neograničene veličine.↑ Da bismo shvatili šta se dešava npr. kada prekoračimo dozvoljenu veličinu broja, potrebno je da znamo kako su podaci ustvari predstavljeni u memoriji računara.

↑ Poglavlje 1.2.3, str. 29.

Kod razumijevanja principa rada računara, korisno je shvatiti da je osnovna gradivna jedinica računara *prekidač*. Računar je u suštini električni aparat, pa su dva osnovna stanja sa kojima operiše stanje *zatvoren prekidač* (struja teče) i stanje *otvoren prekidač* (struja ne teče; Slika 3.1).[1] Numeričke vrijednosti (brojevi) se mogu predstaviti pomoću niza prekidača, odnosno pomoću niza nula i jedinica, koristeći *binarni brojevni sistem*.

[1] U praksi najčešće su nula i jedinica predstavljene sa dva različita naponska nivoa, npr. nula je $0V$ a jedinica $5V$.

Memorija računara se sastoji od niza binarnih cifara (bita eng. BInary digiT), ali ovim bitovima u pravilu ne možemo direktno pristupati. Najmanja jedinica memorije koja se direktno može adresirati se zove bajt (eng. byte), a njena veličina je 8 bita na svim modernim raču-

1775701198

Slika 3.1: Prekidačka logika

narima.[2] Kada na raspolaganju imamo 8 bita, tj. 8 nula ili jedinica, pomoću njih možemo predstaviti ukupno 256 različitih kombinacija, što nije puno. U našim programima često radimo sa brojevima većim od toga.

[2] Ova knjiga teži da vas nauči da pišete programe koji će raditi na svim teoretski mogućim računarima, te će sva pravila i primjeri biti u skladu s tim. Ali kod objašnjavanja pojedinih detalja ćemo pretpostavljati da koristite PC računar proizveden u 21. vijeku, te da na njemu koristite operativni sistem Windows ili Linux i podjednako svjež prevodilac. Ove pretpostavke bi trebale obuhvatiti veliku većinu naših čitalaca.

Tip	Veličina (po standardu)	Veličina (na PC-u)
`char`	8	8
`short`	najmanje 16	16
`int`	najmanje 16	32
`long`	najmanje 32	32
`long long`	najmanje 64	64
`float`	32	32
`double`	64	64
`long double`	najmanje 64	64, 80, 128

Tablica 3.1: Numerički tipovi u programskom jeziku C

Prije svega, potrebno je da nabrojimo numeričke tipove koje programski jezik C poznaje (tablica 3.1). Prvih pet tipova su cjelobrojni, a preostala tri su realni. Pored ovih, C poznaje razne specifične tipove o kojima ćemo govoriti kasnije.

U tablici 3.1 date su količine memorije (u bitima) koje svaki tip zauzima. Obratite pažnju da za mnoge tipove standard programskog jezika C ne definiše preciznu veličinu nego je dozvoljeno da ona varira, a mi smo dali veličinu na PC (x86) arhitekturi koja se koristi na savremenim Intel-baziranim računarima. Tako recimo u C programskom jeziku tip `long` može koristiti više memorije od tipa `int`, ali na PC platformi oni koriste istu količinu memorije, pa među njima ustvari i nema nikakve razlike.[3]

[3] U izvornom C89 standardu tip `long long` uopšte ne postoji, a veličine realnih tipova nisu date, pa smo se u tablici 3.1 pozivali na noviji C99 standard.

Svaki od cjelobrojnih tipova (prvih pet tipova u tablici 3.1) može biti predznačni (eng. signed) ili nepredznačni (eng. unsigned), što određujete tako što ispred tipa navedete odgovarajuću ključnu riječ za tip npr. `signed short` ili `unsigned long`. Kod predznačnog tipa prvi bit se koristi za predznak, što znači da je najveći broj koji njime možemo predstaviti dva puta manji nego kod nepredznačnog tipa. O ovome ćemo detaljnije govoriti ubrzo.

Ako ne navedete niti `signed` niti `unsigned` podrazumijeva se `signed`,

1157098

osim za tip `char` koji u osnovnoj varijanti nije niti predznačni niti nepredznačni (o tome detaljnije u poglavlju 3.1.7). Ako napišemo samo npr. `signed x` ili `unsigned a`, podrazumijeva se bazni tip `int`.

3.1.1 *Binarni brojevni sistem*

Uobičajeni sistem brojeva sa kojim radimo naziva se *dekadski brojevni sistem*, odnosno sistem brojeva čija je baza 10. To znači da se za predstavljanje brojeva u ovom sistemu koristi deset cifara: 0, 1, 2... 9. Recimo, kada napišemo broj 519, znamo da se on sastoji od devet jedinica, jedne desetice i pet stotica. Odnosno, čitajući cifre zdesna nalijevo, svaka sljedeća cifra se množi sa težinskim koeficijentom koji je 10 puta veći od prethodnog.

Zašto baš 10? Bazu 10 koristimo zato što imamo deset prstiju na obje ruke. Naši preci su brojali stvari koristeći prste, baš kao što djeca često rade i danas, pa je prirodno iz toga nastao način bilježenja brojeva na bazi broja 10. Ali da ljudi recimo nemaju 10 prstiju, moglo se desiti da u upotrebi bude brojevni sistem sa nekom drugom bazom, većom ili manjom od deset. Recimo, oktalni brojevni sistem bi bio vrlo pogodan za hobotnice (oktopuse) jer je njegova baza 8, što znači da se koriste cifre 0-7 dok cifre 8 i 9 u takvom sistemu uopšte ne bi postojale. Broj 8 (odnosno količina od 8 predmeta nekog tipa npr. 8 jabuka) bi se predstavljalo sa $10_{(8)}$ jer bi hobotnici bilo potrebno svih osam njenih krakova da predstave taj broj, pa bi za sljedeći veći broj morala nekako zapisati prvu cifru.[4]

[4] Oznakom $_{(8)}$ obilježavamo da je u pitanju broj u oktalnom brojevnom sistemu.

Kako izgleda onda binarni brojevni sistem? U binarnom brojevnom sistemu baza je 2, pa gledajući zdesna nalijevo, prva cifra ima težinu 1, sljedeća težinu 2, zatim 4, 8, 16 itd. Ove težine se množe sa nula ili jedan, odnosno težine uz jedinicu sabiramo a one uz nulu jednostavno preskačemo. Recimo, spomenuti broj 519 u binarnom brojevnom sistemu glasi: 1000000111. Čitajući zdesna nalijevo, prve tri cifre imaju težine 1, 2 i 4, pa je njihov zbir 7. Zatim preskačemo nule te dolazimo do početne jedinice koja ima težinu 512 (2^9 pošto je u pitanju deseta cifra zdesna). Ukupna suma je $512 + 7 = 519$.

Vidimo da je konverzija iz binarnog u dekadski sistem vrlo jednostavna. Kako vršimo konverziju u obrnutom pravcu, iz dekadskog u binarni? Najlakši način je da dijelimo broj sa dva, te u svakom koraku sa strane bilježimo ostatak pri dijeljenju, a za sljedeće dijeljenje koristimo cijeli dio količnika. Kada tako dođemo do nule, pročitamo zapisane ostatke odozdo prema gore. Recimo neka je dat broj 47. Kada ga podijelimo sa dva dobijamo 23 i ostatak 1, tako da je prva cifra zdesna našeg binarnog broja jedinica:

$$47/2 = 23 \quad (1)$$
$$23/2 = 11 \quad (1)$$

1/5/0198

$$
\begin{aligned}
11/2 &= 5 \quad (1)\\
5/2 &= 2 \quad (1)\\
2/2 &= 1 \quad (0)\\
1/2 &= 0 \quad (1)
\end{aligned}
$$

Čitajući odozdo prema gore cifre ostataka dobijamo 101111 što je broj 47 predstavljen binarnim brojevnim sistemom.

3.1.2 *Oktalni i heksadecimalni brojevni sistem*

U računarstvu su čestu u upotrebi još i *oktalni brojevni sistem* čija je baza 8, kao i *heksadecimalni brojevni sistem* koji ima bazu 16. Pošto heksadecimalni sistem ima više od deset cifara, nakon što potrošimo poznate cifre koristimo velika slova abecede, pa će tako cifra A u heksadecimalnom sistemu imati dekadsku vrijednost 10, B je 11 itd. dok najveća cifra u ovom sistemu F ima vrijednost 15 u dekadskom sistemu.[5]

[5] U vašem C programu možete navoditi i literale u oktalnom i heksadecimalnom sistemu. Svaki literal koji počinje cifrom nula smatra se za broj u oktalnom sistemu, pa tako `015` nema dekadsku vrijednost 15 nego 13. Za heks. literal navodite cifru 0 i slovo x, npr. `0xABBA`.

Ova dva brojevna sistema se prije svega koriste jer je vrlo jednostavna konverzija iz njih u binarni sistem i nazad, a zapis u binarnom sistemu često može biti vrlo dugačak i nezgrapan za korištenje. Razlog za tu jednostavnost je jasan iz tablice 3.3 koja sadrži prvih 16 brojeva u sva četiri opisana brojevna sistema. Vidimo da jedna cifra u oktalnom brojevnom sistemu može opisati sve moguće kombinacije trocifrenih brojeva u binarnom sistemu. Također, jedna cifra heksadecimalnog sistema opisuje sve četverocifrene binarne brojeve. Radi bolje preglednosti, naveli smo početne nule, koje se inače mogu izostaviti.

Tablica 3.3: Četiri često korištena brojevna sistema

Dec.	Bin.	Okt.	Heks.	Dec.	Bin.	Okt.	Heks.
0	0000	00	0	8	1000	10	8
1	0001	01	1	9	1001	11	9
2	0010	02	2	10	1010	12	A
3	0011	03	3	11	1011	13	B
4	0100	04	4	12	1100	14	C
5	0101	05	5	13	1101	15	D
6	0110	06	6	14	1110	16	E
7	0111	07	7	15	1111	17	F

Princip konverzije vidjećemo na primjeru broja $1111000_{(2)} = 120_{(10)}$. Najprije podsjetimo se da dodavanjem nula na početak broja (s lijeve strane) ne mijenjamo njegovu vrijednost. Da bismo konvertovali ovaj broj u oktalni brojevni sistem, najprije dodajemo dovoljno nula da ukupan broj cifara bude djeljiv sa tri. U slučaju navedenog broja, treba dodati dvije nule. Zatim broj podijelimo u blokove od po tri cifre: 001 111 000. Za svaki od ovih blokova jednostavno pročitamo odgovarajuću cifru oktalnog brojevnog sistema iz tabele iznad. Na ovaj način

17/03/2018

dobijemo broj $170_{(8)}$.

Slično je i sa heksadecimalnim sistemom: najprije dodamo jednu nulu kako bi ukupan broj cifara bio djeljiv sa četiri. Broj u vidu blokova od po četiri cifre dakle glasi 0111 1000. Iz tabele iznad možemo pročitati da je vrijednost broja $78_{(16)}$.

Što se tiče konverzije iz dekadskog u oktalni ili heksadecimalni sistem i nazad, postoje metode za direktnu konverziju, ali uvijek možete najprije broj pretvoriti u binarni sistem a zatim u željeni brojevni sistem na jednostavne načine koji smo dali iznad.

3.1.3 *Negativni binarni brojevi*

Kako predstaviti negativne brojeve u binarnom sistemu? Na papiru bismo dodali znak minus ispred broja: -101111, ali u memoriji računara na raspolaganju su nam samo nule i jedinice. Dva načina koji su najčešće korišteni ćemo opisati u nastavku. Oba polaze od sljedećeg cilja: da se oduzimanje broja y od x može predstaviti kao sabiranje broja x sa brojem $-y$: $x - y = x + (-y)$.

✓ Komplement jedinice (1kk)

Recimo da u negiranom broju jednostavno zamijenimo sve jedinice nulama a nule jedinicama. Kao prvu cifru broja postavljamo jedinicu kako bismo označili da je u pitanju negativan broj, a nulu koristimo za pozitivne brojeve. Dakle, broj $+47$ bismo pisali kao 0101111, a -47 kao 1010000. Ovakav zapis naziva se *komplement jedinice ili 1kk*.[6] Problem je što ako ova dva broja saberemo po binarnom brojevnom sistemu nećemo dobiti nulu nego broj čije su sve cifre jedinice (uključujući i početnu cifru koja označava predznak). Kakvu vrijednost ima ovaj broj? Primjenjujući logiku 1kk, unazad zaključujemo da je vrijednost ovog broja -0 (minus nula).

[6] Obratite pažnju da ovdje koristimo sedam binarnih cifara, odnosno 7 bita. U praksi broj bita je u pravilu neki stepen dvojke npr. 8 bita, 16 bita itd.

Uzmimo da za predstavljanje naših binarnih brojeva koristimo 8 binarnih cifara (8 bita). Pošto prva cifra predstavlja predznak, najveći pozitivan broj koji možemo ovako zapisati je 127, a najmanji negativan broj je -127. Dakle, ukupno možemo predstaviti 255 različitih brojeva (uključujući i nulu), zato što postoje dva različita kodiranja (sve nule i sve jedinice) koja predstavljaju pozitivnu i negativnu nulu, a one su u matematici jednake. Zbog toga se u računarstvu za binarne brojeve u pravilu koristi *komplement dvojke ili 2kk* kod kojeg su negativni brojevi "pomaknuti za jedan". Vrijednost u 2kk dobijamo tako što na vrijednost negativnog broja u 1kk saberemo 1, pa će broj -47 imati vrijednost 1010001.

✓ Komplement dvojke (2kk)

1/15/01/98

U 2kk kodiranju nula se predstavlja sa brojem čije su sve cifre nule: 00000000. Broj čije su sve cifre jedinice ima vrijednost -1, a broj koji počinje jedinicom i zatim slijedi 7 nula bi imao vrijednost -128. Dakle, sa 8 bita koristeći 2kk možemo predstaviti brojeve na opsegu $[-128, 127]$ odnosno ukupno 256 različitih vrijednosti. Obratite paž-

nju i da u 2kk vrijedi ranije data jednakost, odnosno oduzimanje se izvodi kao sabiranje sa negiranom vrijednošću umanjitelja. Npr. ako saberemo $47_{(10)} = 0101111_{(2)}$ i $-47_{(10)} = 1010001_{(2)}$ koristeći obično binarno sabiranje (za pozitivne brojeve) dobićemo nulu. Naravno, ako ovako matematičkim operacijama dođemo do broja izvan navedenog opsega, trebali bismo to nekako naznačiti što se označava posebnim bitom ili flagom (zastavicom) prenosa (eng. carry).

3.1.4 *Realni binarni brojevi*

Šta je sa vrijednostima koje imaju decimalni dio? Pomoću binarnog brojevnog sistema možemo predstaviti i realne brojeve na isti način kao i u dekadskom sistemu. Recimo, posmatrajmo broj $123,456$ u dekadskom sistemu. Kao što se cifra tri množi sa 10^0, dva sa 10^1, a jedan sa 10^2, možemo nastaviti niz i iza zareza pa se četvorka množi sa $10^{-1} = 0,1$, petica sa $10^{-2} = 0,01$ itd.

Slično čitamo i realne brojeve u binarnom sistemu. Recimo, broj $1101,01101$ pretvaramo u dekadski brojevni sistem tako što najprije vršimo pretvorbu dijela ispred zareza na način kao i do sada: $1101_{(2)} = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13_{(10)}$. Zatim istu logiku primijenimo i na dio iza zareza: $0,01101_{(2)} = 2^{-2} + 2^{-3} + 2^{-5} = 0,25 + 0,125 + 0,03125 = 0,40625_{(10)}$. Dakle kompletan broj ima vrijednost $13,40625$.

Konverzija iz dekadskog u binarni sistem se radi na sljedeći način: Dio ispred zareza konvertujemo kao i do sada, pa ćemo dobiti da je $13_{(10)} = 1101_{(2)}$. Za dio iza zareza, umjesto dijeljenja, *množimo* broj sa dva u svakom koraku, te uzmimamo dio ispred decimalnog zareza (koji će imati vrijednost nule ili jedinice) i pišemo ga sa strane. U sljedećem koraku odbacujemo taj dio te ponavljamo postupak sve dok ne dobijemo rezultat 1. Cifre koje smo pisali sa strane čitamo odozgo prema dolje.

$$
\begin{aligned}
0,40625 \cdot 2 &= 0,8125 \quad (0)\\
0,8125 \cdot 2 &= 1,625 \quad (1)\\
0,625 \cdot 2 &= 1,25 \quad (1)\\
0,25 \cdot 2 &= 0,5 \quad (0)\\
0,5 \cdot 2 &= 1 \quad (1)
\end{aligned}
$$

Konačan rezultat je $1101,01101$.

3.1.5 *Binarni brojevi sa beskonačno mnogo cifara*

Ako biste vježbali sada pretvaranje različitih realnih brojeva iz dekadskog u binarni sistem, uočili biste jednu prilično neintuitivnu činjenicu: neke od često korištenih vrijednosti imaju beskonačno mnogo cifara nakon decimalnog zareza kada ih pretvorimo u binarni brojevni sistem.

Recimo, poznati su nam mnogi racionalni brojevi koji, kada ih pro-

1/15/01/98

bamo predstaviti u decimalnom obliku, dobijemo broj sa beskonačno mnogo cifara iza decimalnog zareza. Jedna trećina je primjer takvog broja: njena vrijednost je 0,33333... Osobina svih takvih racionalnih brojeva je da se cifre nakon decimalnog zareza nakon određenog broja decimala počinju ponavljati. Pokazuje se da mnogi brojevi koji u dekadskom sistemu imaju konačan broj decimala, u binarnom sistemu imaju beskonačno mnogo decimala.

Probaćemo pretvoriti broj $0,1$ iz dekadskog u binarni brojevni sistem na ranije opisani način:

$$
\begin{aligned}
0,1 \cdot 2 &= 0,2 \quad (0)\\
0,2 \cdot 2 &= 0,4 \quad (0)\\
0,4 \cdot 2 &= 0,8 \quad (0)\\
0,8 \cdot 2 &= 1,6 \quad (1)\\
0,6 \cdot 2 &= 1,2 \quad (1)\\
0,2 \cdot 2 &= 0,4 \quad (0)\\
0,4 \cdot 2 &= 0,8 \quad (0)\\
0,8 \cdot 2 &= 1,6 \quad (1)\\
0,6 \cdot 2 &= 1,2 \quad (1)\\
0,2 \cdot 2 &= 0,4 \quad (0)\\
0,4 \cdot 2 &= 0,8 \quad (0)\\
&\ldots
\end{aligned}
$$

Vidimo da se proces nakon petog koraka ponavlja i da ćemo, koliko god dugo nastavili sa računicom, dobijati redom cifre 0011 nakon čega se ponovo vraćamo na dekadski broj $0,2$. Dakle, $0,1_{(10)} = 0,0001100110011...{}_{(2)}$, odnosno to je u binarnom sistemu broj sa beskonačno mnogo cifara iza decimalnog zareza.

Posljedica ovog efekta je da se mnogi realni brojevi (npr. $0,1$) uopšte ne mogu predstaviti u memoriji računara, jer bi bila potrebna beskonačna količina memorije! Kada u vašem računarskom programu unesete broj 0,1, u memoriju će ustvari najvjerovatnije biti pohranjena vrijednost $0,09999849...$ Ako ovaj broj odmah zatim ispišete na ekran, računar će zaokružiti rezultat tako da ćete najvjerovatnije ponovo vidjeti vrijednost $0,1$, čime je od vas sakriveno da postoji greška. Međutim ta greška će se pokazati ako budete radili različite računske operacije.[7] Zbog toga je najsigurniji način rada da uvijek pretpostavljamo kako realne vrijednosti u memoriji računara nisu egzaktne, nego su približno tačne.

Npr. za tip `float` koristi se 32 bita od čega se 1 bit koristi za predznak, 8 bita za eksponent, a 23 bita za binarne cifre.[8] Dakle, svi brojevi sa više od 23 binarne cifre će sadržavati grešku, što nakon prevođenja u dekadski sistem odgovara otprilike 6 decimala.

[7] Interesantno je spomenuti da je u famoznom incidentu 1991., američka navođena raketa Patriot promašila metu i izazvala pogibiju 28 osoba, jer su programeri koji su pisali softver za navođenje rakete pogrešno predstavili da će realne vrijednosti biti egzaktno predstavljene u memoriji.

[8] Način predstave realnih brojeva u memoriji propisan je standardom IEEE-754.

1757019

3.1.6 *Problem nepreciznosti realnih vrijednosti*

Opšti princip rješavanja ovog problema je da kod realnih vrijednosti ne možemo direktno koristiti operatore poređenja kao što je == nego moramo uračunati i određenu grešku. No programeri su često skloni da ovaj problem zanemare jer se on u mnogim situacijama ne vidi. Pogledajte sljedeći jednostavan programčić:

```
#include <stdio.h>
int main() {
  float a, b;
  printf("Unesite dva realna broja: ");
  scanf("%f%f", &a, &b);
  if (a == b) printf("Jednaki su\n");
  return 0;
}
```

Čak i ako unesete brojeve $0,1$ program će korektno ispisati da su ta dva broja jednaka. Razlog je što je greška koja je unesena kod obje vrijednosti ista, odnosno obje promjenljive će imati vrijednosti $0,09999849...$ koja je i dalje jednaka. Dakle, u ovakvim situacijama gdje nad brojevima nisu vršene nikakve računske operacije, sigurno je njihovo međusobno poređenje. No dovoljno je da vrlo malo zakomplikujemo situaciju, recimo ovako:

```
#include <stdio.h>
int main() {
  float a, b, c;
  printf("Unesite dva realna broja: ");
  scanf("%f%f", &a, &b);
  c = a + b;
  if (c == 0.1)
    printf("Zbir je 0.1");
  return 0;
}
```

U ovom programu ako unesete dva puta vrijednost 0.05 program neće ispisati poruku "Zbir je 0.1", jer zbog greške u načinu predstavljanja ovih vrijednosti u memoriji zbir neće biti tačan.

Kako popraviti ovaj program? Prvi korak ka ispravci bi bilo koristiti tip `double` koji koristi više memorije za decimalne cifre, pa je greška manje izražena. Generalno, preporučujemo da *uvijek* koristite tip `double` umjesto `float` jer je ušteda u memoriji na savremenim računarima zanemarljiva, a potencijal za greške ovog tipa ogroman. Ipak, u prethodnom primjeru čak i sa tipom `double` greška se i dalje može desiti ovisno o unesenim brojevima, npr. ako unesete brojeve 0.01 i 0.09 nećemo dobiti ispis tačan čak ni sa tipom `double`. Potpuno ispravan program bi glasio:

✓ Uvijek koristite tip `double` umjesto `float`.

```
#include <stdio.h>
#include <math.h>
#define EPSILON 0.00001
int main() {
  float a, b, c;
```

17157198

```
6     printf("Unesite dva realna broja: ");
7     scanf("%f%f", &a, &b);
8     c = a + b;
9     if (fabs(c - 0.1) < EPSILON)
10      printf("Zbir je 0.1\n");
11    return 0;
12  }
```

✓ Epsilon

Za dva realna broja predstavljena u memoriji računara možemo tvrditi da su "jednaki" ako je njihova razlika po apsolutnoj vrijednosti manja od neke odabrane male vrijednosti ε. Ovu vrijednost ε biramo na osnovu procjene greške, npr. možemo vidjeti da vrijednost $0,09999849 \approx 0,1$ počinje značajnije odstupati nakon šeste decimale, pa je u ovom konkretnom programu zadovoljavajuća vrijednost $\varepsilon = 0.00001$. Ali, da smo kojim slučajem zbir množili sa deset pa poredili sa brojem 1, i sama greška bi postala deset puta veća pa bi i vrijednost epsilona trebala biti deset puta veća.[9] S druge strane, korištenjem tipa `double` greška bi se mogla smanjiti toliko da je početna vrijednost ε i dalje dobra.

[9] Kada poredite dva broja možete koristiti i *relativni epsilon* kod kojeg se tolerancija definiše u postotcima veće od poređenih veličina.

U prethodnom kodu iskoristili smo funkciju, `fabs` za određivanje apsolutne vrijednosti koja se nalazi u biblioteci `math.h`. Obratite pažnju da u C-u postoji i funkcija `abs`, ali ona radi isključivo sa cijelim brojevima, te će zaokružiti vrijednost koju joj pošaljete kao parametar. Naravno, mogli ste izbjeći ovu funkciju tako što ručno razdvojite pozitivni i negativni slučaj:

```
9  if (c - 0.1 < EPSILON && 0.1 - c < EPSILON)
```

Generalno, kada radimo sa realnim brojevima, ako je nad njima vršena bilo kakva manipulacija, ne možemo direktno koristiti poznate operatore poređenja. Umjesto toga uvijek gledamo da li je razlika veća ili manja od greške. Operatore >= i <= posmatramo kao kombinaciju veće-od (manje-od) i jednakosti.

Umjesto...	*Pišemo...*
`a==b`	`fabs(a-b)<EPSILON`
`a!=b`	`fabs(a-b)>EPSILON`
`a>b`	`a-b>EPSILON`
`a<b`	`b-a>EPSILON`
`a>=b`	`a>b || fabs(a-b)<EPSILON`
`a<=b`	`a<b || fabs(a-b)<EPSILON`

✓ Problem sa funkcijom `pow`

Još jedna interesantna posljedica ovog problema je i preporuka da ne koristite funkciju `pow` za stepenovanje kada radite isključivo sa cijelim brojevima. Npr. kada napišete `int x = pow(5,2);` očekivali biste da će `x` imati vrijednost 25. No, funkcija `pow` vraća realnu vrijednost, te obavlja različite operacije koje mogu izazvati pojavu da se za ove konkretne brojeve dobije rezultat 24,9999.... Nakon dodjele u promjenljivu tipa `int`, dio iza zareza će biti odbačen, pa će promjenljiva `x` dobiti vrijednost 24! Srećom, stepenovanje cijelog broja cijelim brojem

1755798

se vrlo lako da izvesti koristeći for petlju (poglavlje 4), a nakon što naučite pisati funkcije (poglavlje 6), moći ćete napraviti vlastitu funkciju za cjelobrojno stepenovanje koja ne pati od ovih problema.

3.1.7 *Znakovni tip*

Memorija računara može sadržavati isključivo nule i jedinice. Ove nule i jedinice mi tumačimo na određeni način koji nam je potreban za rješavanje nekog zadatka. Recimo, neki niz od četiri bajta (32 bita) mi možemo posmatrati kao cijeli broj u binarnom brojevnom sistemu koji predstavlja broj jabuka u nekoj korpi. Računar "ne zna" da su u pitanju jabuke – programer mu je dao instrukciju da spomenute bite drži u nekoj promjenljivoj tipa `int` pod imenom `j`, te da ih zatim ispiše na ekran naredbom `printf("%d jabuka", j);` čime smo naglasili da ćemo bite transformisati u dekadski brojevni sistem (format `%d`), te ispisati riječ `jabuka` kako bi korisnik znao da su u pitanju jabuke.

Sve ovo su *naredbe za transformaciju podataka* koji su u memoriji i dalje u vidu niza nula i jedinica kojima mi dajemo značenje po želji.

Slično je i sa slovima. Slova se kao takva ne mogu držati u memoriji računara, ali mi možemo posmatrati jedan bajt (blok od 8 bita) kao slovo tj. znak. Za svaku moguću kombinaciju bita definisan je određen znak (slovo, cifra...) koji se tim bitima predstavljaju. Tabela ovih vrijednosti data je standardom koji se zove ASCII (američko standardno kodiranje za razmjenu podataka), a možete je vidjeti u tablici 3.4. Pošto binarne vrijednosti zauzimaju dosta prostora u tabeli, obično se konvertuju u neki drugi brojevni sistem (dekadski, oktalni ili heksadecimalni).

✓ ASCII standard

Spominjali smo tip `char` u programskom jeziku C.↑ Ovaj tip nije ništa drugo nego cjelobrojni tip, ali za razliku od tipa `int` koji tipično zauzima 4 bajta, ili `short` koji zauzima 2 bajta, tip `char` zauzima jedan bajt i definisan je samo za vrijednosti na intervalu 0-127 koje se nalaze u ASCII tabeli. Dok god su vrijednosti pozitivne i manje od 128, sa tipom `char` radimo kao sa bilo kojim cjelobrojnim tipom.

↑ Poglavlje 1.2.3, str. 29.

```
char a=1, b=2, c;
c = a + b;
printf("%d", c);
```

Kao što smo rekli, format `%d` određuje da će vrijednost promjenljive `c` biti *tumačena* kao cijeli broj u dekadskom brojevnom sistemu. S druge strane, format `%c` određuje da će cjelobrojna vrijednost promjenljive biti potražena u ASCII tabeli i prikazan odgovarajući znak.

```
char slovo=33;
slovo *= 2;
printf("%c", slovo); /* Slovo B ima ASCII kod 66 */
```

Kada u programskom kodu koristimo jednostruke navodnike (apostrofe), to je u biti kratica koja znači da će znak pod navodnicima biti

✓ Upotreba jednostrukih navodnika (apostrofa)

1.1.2.1.98

Tablica 3.4: ASCII tabela

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	NULL	32	20	' '	64	40	@	96	60	'
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	"	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BELL	39	27	'	71	47	G	103	67	g
8	8	BKSP	40	28	(	72	48	H	104	68	h
9	9	TAB	41	29	)	73	49	I	105	69	i
10	A	LF	42	2A	*	74	4A	J	106	6A	j
11	B	VTAB	43	2B	+	75	4B	K	107	6B	k
12	C	FF	44	2C	,	76	4C	L	108	6C	l
13	D	CR	45	2D	-	77	4D	M	109	6D	m
14	E	SO	46	2E	.	78	4E	N	110	6E	n
15	F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CANC	56	38	8	88	58	X	120	78	x
25	19	EOM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

potražen u ASCII tabeli i umjesto njega će biti uvrštena njegova numerička vrijednost.

```
int x='2', y='1';
x = x - y * 2;
printf("%d", x); /* -48 */
```

Obratite pažnju u primjeru iznad da su brojevi `'2'` i `'1'` navedeni pod apostrofima. To znači da će biti uvrštena ASCII vrijednost ovih znakova. Cifre su također znakovi koji se mogu prikazati na ekranu, pa u ASCII tabeli pronalazimo cifru nula na kodu 48, jedan na 49, dva na 50 itd. Dakle, `x` će imati vrijednost $50 - 49 \cdot 2 = -48$. Pošto je promjenljiva `x` tipa `int`, nije problem ni to što je ona dobila negativnu vrijednost, sve dok takvu vrijednost ne pokušamo pridružiti promjenljivoj tipa `char`.

✓ Neispisivi znakovi

Znakovi sa ASCII vrijednostima 0-31 te 127 spadaju u tzv. neispisive (eng. nonprintable) znakove, što znači da pokušaj njihovog ispisivanja na ekranu najvjerovatnije neće dati nikakav rezultat, ili ćemo dobiti neki neočekivani znak. Ovi znakovi imaju specijalna značenja kao što je prelazak u novi red (`\n`) ali ne biste ih trebali koristiti direktno, nego možete napisati npr. `char c='\n';`.

Što se tiče vrijednosti 128-255, one nisu definisane osnovnom ASCII tabelom, ali su tradicionalno korištene za internacionalne znakove koji ne postoje u engleskom alfabetu. Ova primjena je danas napuštena, te se za prikaz nacionalnih karaktera obično koristi Unicode koji koristi

1777017198

više od jednog bajta. Za sam tip `char` nije definisano da li se prvi bit koristi za predznak ili ne. Ako želite promjenljive veličine jednog bajta za koje je precizno definisano da li imaju ili nemaju predznak, trebate koristiti tipove `signed char` odnosno `unsigned char`.

3.1.8 *Odsijecanje i zaokruživanje*

Šta će se desiti ako cjelobrojnoj promjenljivoj probamo pridružiti realnu vrijednost?

```
int n = 3.999;
printf("%d", n);
```

Na ekranu će se ispisati broj 3. Pošto se promjenljivom tipa `int` ne mogu predstaviti cifre iza decimalnog zareza, one će jednostavno biti odbačene, što nazivamo **odsijecanje**. Ovaj fenomen možemo iskoristiti kada god želimo da dobijemo cjelobrojni dio nekog realnog broja, npr. mogli bismo deklarisati pomoćnu promjenljivu tipa `int`:

```
float x = 4.2;
int pomocna;
...
pomocna = x; /* pomocna je 4 */
x = pomocna; /* x je sada takodjer postalo 4 */
```

Postoji kraći način da dobijemo isti efekat kao iznad. Ako tip navedemo unutar običnih zagrada, vrijednost koja se nalazi desno od zagrada će biti *posmatrana kao vrijednost tog tipa*, pa je sljedeći program ekvivalentan prethodnom:

```
float x = 4.2;
...
x = (int) x; /* x je sada 4 */
```

Staru vrijednost 4.2 smo privremeno pretvorili u vrijednost tipa `int` kako bi ona bila zaokružena, a zatim smo je ponovo dodijelili promjenljivoj `x`. Obratite pažnju da je promjenljiva `x` i dalje promjenljiva tipa `float`! Recimo, ako nakon ovog koda napišemo:

```
x /= 3; /* x je 1.333... */
```

vrijednost neće biti zaokružena.

Ova konstrukcija sa tipom u zagradama se naziva **kalup** (eng. cast). ✓ Kalup (cast)

Odsijecanje se dešava i sa drugim tipovima. Recimo ako bismo napisali:

```
unsigned char broj = 500;
printf("%d", broj);
```

Na ekranu će se ispisati 244. Naime, tip `unsigned char` je veličine osam bita, tako da će biti uzeto krajnje desnih osam bita broja 500. Broj 500 u binarnom sistemu je 111110100, dakle nakon odsijecanja dobijamo $11110100_{(2)} = 244_{(10)}$.

U matematici kada kažemo "zaokruživanje" obično mislimo na za- ✓ Zaokruživanje

17157/01198

mjenu najbližim cijelim brojem, što znači da se 3.9 zaokružuje na 4, a 3.1 na 3. Što se tiče broja 3.5 koji je podjednako udaljen od 3 i 4, po konvenciji on se zaokružuje na 4. Ovakvo zaokruživanje postižemo funkcijom `round` koja se nalazi u biblioteci `math.h`. Isti efekat kod pozitivnih možemo postići tako što na vrijednost saberemo 0.5 prije odsijecanja:

```
y = (int) (x + 0.5);
```

no kod negativnih brojeva bismo morali oduzeti 0.5. Pored ove funkcije postoje još funkcija `floor` koja uvijek zaokružuje na manji cijeli broj, te funkcija `ceil` koja uvijek zaokružuje na veći cijeli broj. Razlike između nabrojanih načina zaokruživanja možete vidjeti na primjerima u tablici 3.5.

	`(int)`	`floor`	`ceil`	`round`
1.3	1	1	2	1
1.7	1	1	2	2
-1.3	-1	-2	-1	-1
-1.7	-1	-2	-1	-2
1.5	1	1	2	2

Tablica 3.5: Razlike između načina zaokruživanja u C-u

✓ Zaokruživanje realnog broja na n decimala

A šta znači "zaokružiti broj na n decimala"? Recimo da imamo broj $\pi = 3,1415926...$ i traži se da ga zaokružimo na tri decimale. U tom slučaju trebamo dobiti broj $3,142$ iz razloga što je to broj sa tri decimale koji je najbliži broju π; $(n+1)$-va decimala je 5, što znači da za n-tu decimalu uzimamo sljedeći veći broj a to je 2.

Funkcija `round` nema mogućnost da joj zadamo željeni broj decimala, no ovaj efekat možemo postići vrlo jednostavnim trikom: najprije pomnožimo broj sa 10^n, zatim pozovemo funkciju `round` kako bismo dobili najbliži cijeli broj, te konačno podijelimo broj sa 10^n kako bismo dobili broj sa n decimala.

```
1  #include <stdio.h>
2  #include <math.h>
3  int main() {
4    float x;
5    printf("Unesite x: ");
6    scanf("%f", &x);
7    x = round(x * 1000) / 1000;
8    printf("%g\n", x);
9    return 0;
10 }
```

Ovdje smo koristili format `%g` koji ispisuje onoliko decimala koliko broj stvarno ima, dakle nakon zaokruživanja promjenljiva `x` će imati 3 decimale i toliko će biti i ispisano. Za razliku od toga, format `%f` će uvijek ispisati fiksno 6 decimala (ako ne zatražimo drugačije), pa čak i ako su u pitanju nule.

Obratite pažnju da smo isti ispis na ekranu mogli postići bez linije 7 tako što bismo jednostavno koristili format `%.3f`. No ovo rješenje nije univerzalno: na ovaj način možemo samo *ispisati na ekran*

1715701198

zaokružen broj, on se ne može koristiti da dobijemo vrijednost koju možemo uvrstiti u neke druge formule.

Recimo, moglo se tražiti da korisnik unese tri broja, a da se zatim izračuna srednja vrijednost tih brojeva zaokruženih na jednu decimalu. Npr. ako je korisnik unio 3.141, 5.888 i 2.55, ispravno rješenje je $(3.1 + 5.9 + 2.6)/3 = 3.8666....$ Vidimo da samo rješenje nije zaokruženo na jednu decimalu, niti možemo prvo izračunati srednju vrijednost unesenih brojeva pa je onda nekako manipulisati do željenog rezultata.

Za vježbu možete prepraviti program naveden iznad tako da korisnik pored broja x unosi i broj tačnih decimala n.

3.1.9 *Prekoračenje*

U primjeru:

```
unsigned char broj = 500;
printf("%d", broj);
```

imali smo slučaj *prekoračenja* (eng. overflow). Prekoračenje nastupa kada nekoj promjenljivoj pokušamo pridružiti vrijednost koja je izvan opsega dozvoljenih vrijednosti za tu promjenljivu. Kao i u mnogim drugim slučajevima, C prevodilac nam neće dati upozorenje ako napravimo ovu vrstu greške, nego to moramo sami provjeriti.

Standard programskog jezika C kaže da je prekoračenje jedan slučaj *nedefinisanog ponašanja*, odnosno da se u tom slučaju može desiti bilo šta i da takve situacije trebamo izbjegavati. No, kada pišemo programe za PC računare, mi tačno znamo šta će se desiti u slučaju prekoračenja iz razloga što se međusobna dodjela memorijskih lokacija u pravilu dešava u samom procesoru računara.

Kada imamo promjenljivu *nepredznačnog cjelobrojnog tipa*, prekoračenje će dovesti do odsijecanja binarnih cifara kao što smo vidjeli iznad. Kod *predznačnih cijelih brojeva* situacija je slična, s tim što ne smijemo zaboraviti da se prvi bit koristi za predznak. Na primjer

```
signed char broj = 500;
printf("%d", broj);
```

Ovaj program će ispisati vrijednost -12, što možemo jednostavno zaključiti sljedećom logikom: najprije dolazi do odsijecanja binarnih cifara tako da se u promjenljivu upisuje vrijednost 11110100. Pošto je prvi bit jedinica, znamo da je ovo negativan broj koji čitamo po 2kk. Dakle, naprije konvertujemo u pozitivan broj tako što od ovog broja oduzmemo 1, pa imamo 1110011 (prvu cifru predznaka smo izdvojili), zatim zamijenimo jedinice nulama a nule jedinicama tako da dobijamo 0001100, te konačno pretvaramo ovaj broj iz binarnog u dekadski sistem tako da dobijamo broj 12.

Kod realnih brojeva prekoračenje je znatno složenije i u ovoj knjizi se nećemo njime baviti.

17570198

3.2 *Operatori*

U ranijim poglavljima smo navodili neke osnovne matematičke operatore (poglavlje 1), te logičke↑ i operatore poređenja↑ (koji se još zovu i relacioni operatori, poglavlje 2). Vrijeme je da pogledamo kakvih još operatora imamo na raspolaganju. U programskom jeziku C, mnoge funkcionalnosti su realizirane pomoću operatora, čak i neke za koje ne važi uobičajena intuicija.

↑ Poglavlje 2.4, str. 49.
↑ Poglavlje 2.2, str. 46.

✓ *Operator u programskom jeziku C je simbol koji označava da je potrebno izvršiti matematičku, relacionu, logičku ili neku drugu operaciju i dati rezultat. Operator može (ali ne mora) promijeniti i vrijednost jednog ili oba operanda.*

✓ Operator

Kada govorimo o operatorima u programskom jeziku C, kažemo da oni određuju **vrijednost izraza**, te eventualno imaju neke **sporedne efekte** koje izazivaju.

✓ Sporedni efekat operatora

Recimo, poznat nam je operator sabiranja (plus). Ako napišemo `x+y` vrijednost ovog izraza je zbir vrijednosti promjenljivih x i y. Operator + nema sporednih efekata, odnosno sam kod `x+y` neće promijeniti vrijednost niti x niti y. Ali ako napišemo recimo `x=x+y`, promijeniće se vrijednost x. Vidimo da se ovdje koriste dva simbola – `=` i `+` – što znači da koristimo dva različita operatora.

Znak jednakosti `=` predstavlja *operator dodjele*. Vrijednost ovog operatora je vrijednost onoga što se nalazi sa desne strane, a sporedni efekat je da se ta vrijednost upisuje u promjenljivu s lijeve strane. Odnosno, ako napišemo `a=b`, vrijednost ovog izraza je jednaka vrijednosti b, a sporedni efekat je da će promjenljiva a poprimiti tu vrijednost. Vidimo da ovdje termin "sporedni efekat" (eng. side effect) koristimo za nešto što uopšte nije sporedno nego je suština samog operatora =, ali je to jednostavno uobičajeni termin koji koristimo za ovu pojavu.

Operatore razvrstavamo u skupine po više kriterija. Već smo spomenuli da se operatori mogu razvrstati po njihovom *značenju* (semantici) na:

✓ Vrste operatora

- matematičke operatore, kao što je sabiranje;
- logičke operatore, kao što je logička konjunkcija `&&` ili disjunkcija `||`;
- relacione operatore tj. operatore poređenja kao što su veće od `>`, manje od `<`, jednakost `==` itd.;
- operatore dodjele, u koje ubrajamo osnovnu dodjelu kao i razne kombinovane operatore↑ npr. `+=`;
- operatore koji predstavljaju različite jezičke konstrukcije npr. male zagrade u kojima navodite parametre funkcije su također operator;

↑ Poglavlje 1.2.8, str. 36.

17570198

neke od tih operatora ćemo objasniti u nastavku teksta, a za neke za sada ne posjedujete dovoljno znanja da biste ih shvatili, ali kada dođe pravi trenutak objasnićemo i njih.

Po *broju operanda* razlikujemo:

- **unarne operatore** koji imaju jedan operand; npr. u kodu `y=-x` imamo poznati operator dodjele te unarni operator minus, njegov jedini operand je x, vrijednost je negirana vrijednost x-a, a sporednih efekata nema;
- **binarne operatore** koji imaju dva operanda, kao što su već spomenuti operatori sabiranja i dodjele;
- te **ternarni operator**↑ koji ima tri operanda.

↑ Poglavlje 3.2.3, str. 82.

✓ Prioritet operatora

Sljedeći kriterij je **prioritet** i **asocijativnost operatora**. Recimo da u našem kodu imamo izraz `x=2+3*4;` – da li će x imati vrijednost 20 ili 14? Ako ukucate ovaj izraz u neki jednostavan kalkulator dobićete vrijednost 20, no ipak u matematici i programskom jeziku C pouzdano znamo da rezultat treba biti 14 iz razloga što operator množenja * ima *veći prioritet* od operatora sabiranja +. U ovom slučaju to znači da operand 3 koji se nalazi između dva operatora pridružujemo onom operatoru koji ima veći prioritet a to je operator množenja.

Drugi način razmišljanja o prioritetu operatora je da oko operatora sa većim prioritetom (i njegovih operanada) možemo postaviti zagrade a da se vrijednost izraza ne promijeni. Drugim riječima, x će i dalje biti 14 ako napišemo `x=2+(3*4);` dok će se vrijednost promijeniti i postati 20 ako napišemo `x=(2+3)*4;`.

Također znamo da operator logičke konjunkcije `&&` ima veći prioritet od logičke disjunkcije `||`.

✓ Asocijativnost operatora

No, šta se dešava ako u nekom izrazu imamo dva operatora istog prioriteta, ili čak dva puta isti operator? Kod sabiranja ovo nije bitno jer je sabiranje asocijativno, odnosno sasvim je svejedno da li pišemo `(x+y)+z` ili `x+(y+z)` jer će rezultat biti isti. Ali uzmimo operator koji nije asocijativan kao što je operator poređenja `==`. Ovaj operator daje vrijednost logičke istine 1 ako su operandi međusobno jednaki, a 0 ako nisu. Dakle, ako napišemo `(x==y)==z`, najprije će se međusobno uporediti x i y, pa ako su jednaki biće uvrštena vrijednost 1 a ako nisu 0. Zatim se vrijednost promjenljive z upoređuje sa 1 ili 0. Vidimo da `x==(y==z)` ima sasvim drugačije značenje. Šta bi onda značio izraz `x==y==z`?

17557098

Odgovor na ovo pitanje glasi da je operator poređenja `==` *lijevo asocijativan*, što znači da je izraz `x==y==z` ekvivalentan izrazu `(x==y)==z` odnosno prilikom pridruživanja operanda operatoru čitamo izraz slijeva nadesno. Postoje i *desno asocijativni* operatori kod kojih pridružujemo operande zdesna nalijevo.

Inače, ako se pitate kako uporediti da li su promjenljive x, y i z sve tri međusobno jednake, jedini ispravan način da to provjerite je da napišete uslov `x==y && y==z`.

Spisak svih operatora koje posjeduje programski jezik C dat je u prilogu.↑ Napominjemo još jednom da bi vam neki od njih mogli biti nejasni jer ne posjedujete dovoljno znanja programiranja, ali ovo će se s vremenom razjasniti. U prvoj koloni nalazi se prioritet, a u četvrtoj je asocijativnost, pa na osnovu ove tabele možete znati tačnu vrijednost svakog izraza.

↑ Poglavlje A, str. 351.

U nastavku poglavlja ćemo se pozabaviti nekim operatorima i njihovim osobinama koje smo u dosadašnjem tekstu preskočili.

3.2.1 *Implicitna konverzija tipa*

U poglavlju o zaokruživanju↑ govorili smo o eksplicitnoj konverziji tipa vrijednosti do koje dolazi kada vrijednost jednog tipa dodijelimo promjenljivoj drugog tipa ili kada koristimo kalup.

↑ Poglavlje 3.1.8, str. 76.

No šta se dešava ako imamo neki binarni operator, npr. operator sabiranja +, čiji operandi su različitog tipa? Recimo, ako napišemo `x+y`, a promjenljive `x` i `y` su različitog tipa, kojeg tipa će biti njihov zbir?

Tip vrijednosti izraza se određuje tako što redom čitamo sljedeća pravila i primijenimo prvo od njih koje važi (kao da imamo strukturu if-else if-else...):

1. Ako je jedan od operanada tipa **`long double`** (bilo koji), rezultat će biti tipa **`long double`**.
2. Ako je jedan od operanada tipa **`double`**, rezultat je tipa **`double`**.
3. Ako je jedan od operanada tipa **`float`**, rezultat je tipa **`float`**.
4. Ako je jedan od operanada tipa **`long`**, rezultat je tipa **`long`**.
5. U suprotnom rezultat je tipa **`int`**, čak i ako su oba operanda tipa **`char`** ili **`short`**.

Ovu konverziju tipa nazivamo *implicitnom konverzijom* i možemo vidjeti da kod nje nikada ne dolazi do odsijecanja.

3.2.2 *Prefiksni i postfiksni operatori*

Uočili smo da operatori `++` i `--` postoje u dvije varijante: prefiksnoj (`++a`) i postfiksnoj (`a++`). Obje varijante operatora `++` imaju isti *sporedni efekat*, a to je da će se vrijednost promjenljive `a` uvećati za jedan. Razlikuje se njihova *vrijednost*: prefiksna varijanta ima uvećanu vrijednost, a postfiksna vrijednost prije uvećavanja. Slično je i za operatore `--`.

1755/0198

Lakše ćemo shvatiti o čemu se radi na sljedećem primjeru:

```
1 #include <stdio.h>
2 int main() {
3   int x=5, y, z;
4   y=x++;
5   z=++x;
6   printf("%d %d %d\n", x, y, z);
7   return 0;
8 }
```

Šta će ovaj program ispisati na ekranu? Promjenljiva `x` će sigurno imati vrijednost 7, jer smo dva puta primijenili operator `++` na nju. U liniji 4 koristimo *postfiksnu verziju* operatora `++`, što znači da izraz `x++` ima vrijednost promjenljive x *prije uvećanja*, dakle izraz ima vrijednost 5, iako će x postati 6. Zbog toga će `y` dobiti vrijednost 5.

Početnicima je često lakše da razumiju naredbe poput `y=x++` na sljedeći način: "*prvo* se vrijednost iksa dodijeli u y, *pa se tek onda* x poveća za jedan".[10]

[10] Ovdje je važno napomenuti da to nije potpuno tačno: postoje jezičke konstrukcije u C-u kod kojih zaista govorimo o redoslijedu izvršenja koraka npr. prilikom pozivanja funkcija. Ovdje redoslijed nije bitan ustvari, ono što je bitno je da kompletan izraz ima neuvećanu vrijednost.

Da vidimo još i šta se dešava u liniji 5. Rekli smo da će nakon linije 4 `x` imati vrijednost 6, iako će `y` biti 5. U liniji 5 sada imamo *prefiksnu varijantu* operatora čija je vrijednost jednaka uvećanoj vrijednosti promjenljive. Dakle, pošto je `x` bilo 6 ono će sada postati 7, te će ova uvećana vrijednost 7 biti pridružena promjenljivoj `z`. Na ekranu se ispisuje: `7 5 7`.

3.2.3 *Ternarni (kondicionalni) operator*

Operator `?:` je interesantan jer je to jedini operator u programskom jeziku C koji ima tri operanda. Ovaj operator je ekvivalentan alternativi o kojoj smo govorili u poglavlju 2.3↑ tj. možemo ga zamijeniti if-else strukturom. Izraz `a?b:c` je izraz koji ima vrijednost `b` ako je `a` logička istina, a vrijednost `c` ako nije. Sam operator nema sporednih efekata, ali ako `b` ili `c` imaju sporednih efekata, oni će se desiti samo ako je `a` istina ili neistina, respektivno. Korištenjem ternarnog operatora dobijamo kod koji je kraći ali često slabije razumljiv.

↑ Str. 48.

Npr. pogledajmo ovaj kod:

```
if (x>0)
  printf("Pozitivan");
else
  printf("Nije pozitivan")
```

Koristeći ternarni operator mogli smo pisati:

```
x>0 ? printf("Pozitivan") : printf("Nije pozitivan");
```

Ovdje smo iskoristili poziv funkcije `printf` kako bi se posao ispisa obavio unutar samog ternarnog operatora. Funkcija `printf` ima i povratnu vrijednost koja nije pretjerano korisna,↑ pa smo je ovdje zanemarili kao što smo to radili i do sada. No, ovaj kod nije u potpunosti u duhu ternarnog operatora. Mogli smo iskoristiti samu vrijednost string literala↑ pa pisati ovako:

↑ Poglavlje B.1.4, str. 356.

↑ Poglavlje 1.2.7, str. 35.

1715170198

```
printf(x>0 ? "Pozitivan" : "Nije pozitivan");
```

Ternarni operator se često koristi u konstrukcijama poput sljedeće:

```
prosjek = (broj > 0) ? suma / broj : 0;
```

Ovdje smo najprije provjerili da li je broj članova veći od nule kako bismo izbjegli dijeljenje s nulom. Ako jeste, računamo prosjek na uobičajen način (`suma/broj`) te će promjenljiva `prosjek` dobiti tu vrijednost, a ako nije postavljamo prosjek na nulu. Upotrebu ternarnog operatora smo mogli izbjeći npr. ovako:

```
prosjek = 0;
if (broj > 0) prosjek = suma / broj;
```

Upotreba ovog operatora je stvar ukusa. Programeri su podijeljeni po pitanju da li je prošli ili pretprošli isječak koda lakši za razumijevanje.

3.2.4 *Operator zarez*

Operator zarez je čest uzrok grešaka kod programiranja. Ovaj operator je u suštini vrlo jednostavan: njegova vrijednost je uvijek jednaka desnom operandu, a sporednih efekata nema. Dakle `a,b` će imati vrijednost `b`. Pri tome interesantna osobina je da ovaj operator ima najmanji mogući prioritet, manji čak i od dodjele. Recimo ako zaboravite da se u C-u koristi decimalna tačka a ne zarez pa napišete `x=5,6;` promjenljiva `x` će imati vrijednost 5 iz razloga što zarez ima manji prioritet od dodjele, pa će se ovaj izraz posmatrati kao `(x=5),6`.

Gdje se može koristiti ovakav neobičan operator? Njegova glavna primjena je kada želimo da se unutar jednog iskaza izvrši više zadataka koje možemo razdvojiti zarezom. Recimo sljedeća naredba je potpuno validna

```
if (x > 0) x--, y++;
```

Još jedna česta greška koju vidimo kod početnika je kôd poput ovoga:

```
if (a,b,c > 5)
```

Pošto operator zarez ima vrijednost desnog operanda, ovaj uslov će imati isto značenje kao `if(c>5)`, a vrijednosti `a` i `b` će biti zanemarene. Ako želite provjeriti da li su sve tri promjenljive veće od pet, jedini ispravan način da to uradite je:

```
if (a > 5 && b > 5 && c > 5)
```

17570198

3.2.5 *Operator sizeof*

Iako izgleda kao funkcija, ključna riječ `sizeof` ustvari označava *operator* programskog jezika C. Ovaj unarni operator daje količinu memorije u bajtima koju zauzima neki tip ili promjenljiva. `sizeof` možemo

koristiti na dva načina: ako se koristi bez zagrada, `sizeof` daje količinu memorije koju zauzima promjenljiva:

```
double x;
int y = sizeof x;
```

`y` će imati vrijednost 8 jer tip `double` na PC arhitekturi zauzima 8 bajta. Ako iza sizeof koristimo zagrade, možemo direktno navesti tip.

```
1 int a = sizeof(int);
2 printf("%d", sizeof a)
```

Veličina tipa `int` će biti 4.

No primjeri iznad nisu striktno tačni jer operator `sizeof` ne daje vrijednost tipa `int` nego tipa `size_t`. Tip `size_t` je nepredznačni cjelobrojni tip čija veličina je dovoljna da se njime mogu predstaviti sve memorijske veličine koje se mogu javiti u nekom C programu (npr. na PC arhitekturi to je `unsigned long`). Dodjela promjenljivoj tipa `int` u liniji 1 će uraditi eksplicitnu konverziju tipa, dok bismo u funkciji `printf` trebali koristiti format `%zu` za tip `size_t`.

3.2.6 *Još o nekim operatorima*

Operator % često se naziva *modulo operator* ali ovo ustvari nije tačno. Operator % je ustvari operator ostatka pri dijeljenju (eng. remainder). Razlika se očituje kod primjene na negativne brojeve. Matematski modulo daje razliku do množitelja koji je manji po apsolutnoj vrijednosti pa je tako $-19 \mod 4 = 1$ ali je $-19\%4 = -3$.

Činjenica da je ostatak negativan je čest uzrok grešaka kod rastavljanja brojeva na cifre. Obično se pretpostavlja da se posljednja cifra broja može dobiti tako što radimo ostatak od dijeljenja sa deset: `x%10`, ali ako je `x` negativno ovaj ostatak će biti negativan što nas može iznenaditi, pa je bolje da uzimamo apsolutnu vrijednost: `abs(x%10)`.

Obratite pažnju da kombinovani operatori koji istovremeno obavljaju neku računsku operaciju i dodjelu imaju vrlo nizak prioritet. Ovo se odražava na primjere kao što je npr.:

```
y %= x - 4;
```

Početnici često pretpostavljaju da je ovaj kôd ekvivalentan sljedećem:

```
y = y % x - 4;
```

Međutim, ovo nije tačno, jer zbog malog prioriteta operatora `%=` trebamo pretpostaviti da se oko operatora minus nalazi zagrada, pa će kôd biti ustvari ekvivalentan sa

```
y = y % (x - 4);
```

1757011198

Pitanja za razumijevanje gradiva

1. Pretvorite $101101011_{(2)}$ u dekadski brojevni sistem.

2. Pretvorite $3A8FC_{(16)}$ u binarni brojevni sistem.

3. Pretvorite $588_{(10)}$ u heksadecimalni brojevni sistem tako što ćete ga najprije pretvoriti u binarni sistem, pa iz binarnog u heksadecimalni.

4. Pretvorite $58,334_{(10)}$ u binarni brojevni sistem.

5. Odredite vrijednost ε za program u kojem korisnik unosi dva realna broja sa najviše dvije decimale, a zatim se izračunava njihov proizvod.

6. Koliko je `c` za `char c='A' + '0';` ?

7. Šta je to kalup (eng. cast)?

8. Koja je razlika između funkcija `floor`, `ceil` i `round`?

9. Kako zaokružiti realnu promjenljivu `x` na dvije decimale?

10. Koliko operanada mogu imati operatori u programskom jeziku C?

11. Kada je važan prioritet operatora a kada asocijativnost?

12. U kojim situacijama dolazi do implicitne konverzije tipa?

13. Koja je razlika između prefiksne i postfiksne verzije operatora `++`?

14. Koju vrijednost ima operator zarez?

15. Šta je `sizeof` i za šta se koristi?

17570198

1757018 8

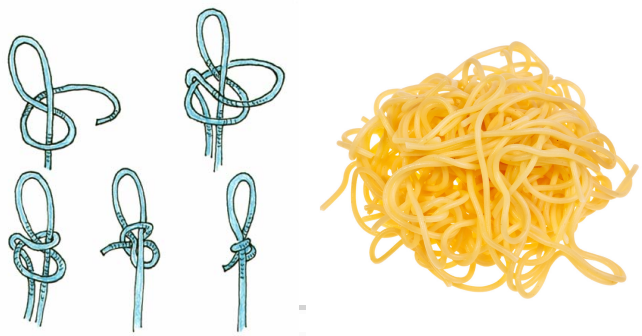
4
Ponavljanje dijelova koda - Petlje

Petlja (eng. loop) je dio programa čije se izvršavanje ponavlja određeni broj puta. Najjednostavniji primjer petlje bi bio sljedeći: recimo da želimo na ekran ispisati svoje ime 100 puta, možemo napisati sljedeći kod koji će detaljno biti objašnjen kasnije:

```
#include <stdio.h>
int main() {
  int i;
  for (i=0; i<100; i++) {
    printf("Svoje ime\n");
  }
  return 0;
}
```

Da bismo znali da li je u našem programu potrebna petlja, trebamo sebi postaviti pitanje: postoji li neki zadatak koji se ponavlja? U tom smislu prevod "petlja" možda i nije baš najsretnije odabran. Na našem jeziku kažemo da je nešto zapetljano ako se ne može razaznati šta je šta, ako je komplikovano. Primjer zapetljanog koda smo vidjeli kada smo govorili o `goto` naredbi i špageti-kodu.↑

↑ Poglavlje 2.6, str. 61.



Slika 4.1: Petlja ili špageti kod?

1757/0198

Upravo suprotno, petlje u programiranju služe baš tome da razjasne stvari i olakšaju prepoznavanje logike koja stoji stoji iza nekog ponavljajućeg koda. Po tome petlje su sličnije čvorovima. Mornari, izviđači i drugi poznaju nekoliko tipova čvorova, pri čemu dobro znaju kada se

koji koristi, koje su mu prednosti i mane itd. Slično, iskusan programer prepoznaje petlju, ona mu olakšava da shvati šta je drugi programer želio postići (slika 4.1).

U programskom jeziku C postoje tri tipa petlji: **for petlja**, **while** i **do-while petlja**. Svakoj od njih ćemo posvetiti po jedno poglavlje. Neki početnici u programiranju su skloni da sve što se nalazi u vitičastim zagradama nazovu petljom (a to je ustvari blok koda↑), pa tako često čujete frazu "if-petlja". *if nije petlja*. if je kontrolna struktura pomoću koje se postiže uslovno izvršenje i grananje kôda. switch-case također nije petlja, to je kontrolna struktura koja služi za višestruki izbor. Riječ "petlja" može se koristiti isključivo za dio koda koji se ponavlja.

↑ Poglavlje 2.1.1, str. 44.

4.1 *for petlja*

Prvi tip petlje koji ćemo upoznati je **for petlja**. for petlja se pretežno koristi kada želimo ponoviti neki kod određeni unaprijed poznati broj puta.

Pogledajmo jedan jednostavan primjer for petlje:

```
1 #include <stdio.h>
2 int main() {
3   int i;
4   for (i=0; i<5; i++) {
5     printf("if nije petlja\n");
6   }
7   return 0;
8 }
```

Kada pokrenemo ovaj program, na ekranu dobijamo:

```
if nije petlja
if nije petlja
if nije petlja
if nije petlja
if nije petlja
```

Petlja se izvršila tačno 5 puta. Kontrolna promjenljiva `i` petlje dobijala je vrijednosti od 0 do 5 ne uključujući peticu i svaki put se povećala za 1, dakle $[0, 5)$ što je ukupno pet vrijednosti. Kako bismo postigli da se petlja izvrši 8 puta umjesto 5?

Obratite pažnju na liniju sa pozivom `printf` funkcije:

```
5     printf("if nije petlja\n");
```

Čemu služi oznaka `\n`? Šta bi se desilo da smo je izostavili?

Možemo uočiti da je tijelo petlje zatvoreno u vitičaste zagrade:

```
4   for (i=0; i<5; i++) {
5     printf("if nije petlja\n");
6   }
```

Ove vitičaste zagrade označavaju blok koda o čemu smo pričali iznad. Rekli smo i to da ako se samo jedna naredba nalazi u bloku, zagrade

1.13.04.98

se mogu izostaviti. Koliko ovdje imamo naredbi u bloku? Možemo li izostaviti zagrade?

4.1.1 *Struktura for petlje*

Na prethodnom primjeru možemo uočiti da se program u kojem se koristi `for` petlja sastoji od tri dijela: *deklaracija kontrolne promjenljive*, *zaglavlje petlje* i *tijelo petlje*. Tamo gdje su boje vidljive, na slici 4.2 možete vidjeti deklaraciju kontrolne promjenljive označenu žutom bojom, zaglavlje plavom, a tijelo crvenom.

```
int i;
for (i=0; i<8; i++) {
  printf("if nije petlja!\n");
}
```

Slika 4.2: Struktura for petlje

`for` petlja najčešće ima *kontrolnu promjenljivu* ili *brojač*. Kontrolna promjenljiva, kao i svaka druga, može imati bilo kakvo ime koje želite, ali iz historijskih razloga uobičajeno je da se kontrolne promjenljive za for petlje imenuju `i`, `j`, `k`... Kontrolna promjenljiva pomaže da prebrojimo koliko puta se petlja izvršila. Ona će najprije poprimiti vrijednost 0, zatim 1, 2 itd. a kada dostigne vrijednost 5 petlja će se prekinuti, pa se na taj način osigurava da se `for` petlja izvrši tačno 5 puta.

✓ Kontrolna promjenljiva petlje (brojač)

Inače u originalnom C89 standardu predviđeno je da se sve promjenljive moraju deklarisati na početku funkcije, dok je u C++u čak preporučeno da se deklarišu tamo gdje se koriste. C++ omogućuje da se promjenljiva deklariše u samom zaglavlju for petlje, ovako:

```
for (int i=0; i<5; i++) {
```

U C-u ovo nije dozvoljeno, dakle primjer iznad ne predstavlja validan C kod i neće se kompajlirati (osim ako niste greškom kreirali C++ projekat umjesto C projekta).

Zaglavlje petlje definiše koliko puta će se petlja izvršiti, na način da se određuje početna i krajnja vrijednost kontrolne promjenljive `i`, kao i korak za koji će se `i` povećati u svakom prolazu kroz petlju. U ranijem primjeru vidimo da je početna vrijednost `i` nula, da se petlja prekida kada `i` postane 5 (što znači da 5 nije uključeno u petlju), te da se u svakom prolazu `i` poveća za jedan (`i++`). Kako bismo izmijenili ovu for petlju tako da je i broj 5 uključen?

✓ Zaglavlje petlje

Tijelo petlje je blok naredbi koje se ponavljaju. Sve ono što je u

✓ Tijelo petlje

1715770198

tijelu petlje će se ponavljati, ono što nije – neće. Prije i poslije tijela petlje ne navodi se znak tačka-zarez. Ako napišemo:

```
for (int i=0; i<5; i++) ; {
  printf ("i je %d\n", i);
}
```

ispisaće se samo jednom "i je 5" iz sljedećeg razloga: tijelo petlje je prvi blok naredbi koji se nalazi nakon zaglavlja petlje, a znak tačka-zarez je validan blok koji je prazan (naredba koja ne radi ništa). Dio u vitičastim zagradama ustvari se nalazi izvan (poslije) for petlje.

Posvetimo se malo detaljnije zaglavlju petlje. Zaglavlje for petlje (slika 4.3) nalazi se unutar običnih (malih) zagrada, a sastoji se opet od tri dijela razdvojena znakom tačka-zarez (eng. semicolon). Generalno, da bi for petlja bila validna moraju se u zaglavlju javiti tačno dva tačka-zareza. Ta tri dijela su: *inicijalizacija* (označena žutom bojom, tamo gdje su boje vidljive), *uslov* (označen plavom bojom) i *ažuriranje* (označeno crvenom bojom).

```
for (i=0; i<8; i++) {
```

Slika 4.3: Zaglavlje for petlje

Inicijalizacija je neka naredba koja će se izvršiti samo jednom prije ulaska u petlju. Ona služi za postavljanje početne vrijednosti kontrolne promjenljive. Npr. u primjeru iznad `i=0` je postavilo kontrolnu promjenljivu na njenu početnu vrijednost nula.

✓ Inicijalizacija petlje

Sve dok je **uslov** ispunjen petlja će se ponavljati. Kada uslov prestane biti ispunjen, petlja se završava a izvršavanje se nastavlja poslije petlje (tj. iza tijela petlje). U pravilu, uslovom se definiše krajnja vrijednost kontrolne promjenljive. Npr. iznad je stavljeno `i<5`, što znači da će `i` rasti dok ne dostigne vrijednost pet.

✓ Uslov petlje

Konačno, **ažuriranje** je neka naredba koja će se izvršiti prilikom svakog prolaza kroz petlju, poslije tijela petlje, a služi kako bi se definisao korak sa kojim će kontrolna promjenljiva rasti. U našem primjeru `i++` označava da se `i` pri svakom prolazu kroz petlju povećava za jedan.

✓ Ažuriranje petlje

Pogledajmo neke primjere.

Koliko puta će se izvršiti sljedeće petlje:

```
for (i=1; i<8; i++) { ...
for (i=100; i<200; i++) { ...
for (i=1; i<=8; i++) { ...
for (i=8; i<1; i++) { ...
```

U posljednjem slučaju petlja se neće izvršiti nijednom, jer uslov petlje nije ispunjen odmah na početku. Postavili smo `i` na 8 što nije manje od 1, tako da uslov već u startu nije ispunjen. To znači da se tijelo petlje neće izvršiti niti jednom, odnosno neće se uopšte na ekranu ispisati da "if nije petlja".

1/157/0198

Za kraj ove diskusije o strukturi `for` petlje pogledaćemo ovaj programčić:

```
1  #include <stdio.h>
2  int main() {
3    int i,n;
4    printf("Unesite broj n: ");
5    scanf("%d", &n);
6    for (i=0; i<n; i++) {
7      printf("if nije petlja!\n");
8    }
9    return 0;
10 }
```

Ovdje korisnik najprije sa tastature unosi cijeli broj `n`, a zatim se na ekranu ispiše tekst "if nije petlja" `n` puta. Tu možemo vidjeti primjer `for` petlje koja se izvršava neki proizvoljan broj puta koji korisnik definiše svojim unosom, a ako korisnik unese negativan broj petlja se naravno neće izvršiti niti jednom.

4.1.2 *Kako funkcioniše for petlja?*

Sada pogledajmo ovaj jednostavni program koji je vrlo sličan prethodnom.

```
1  #include <stdio.h>
2  int main() {
3    int i,n;
4    printf("Unesite broj n: ");
5    scanf("%d", &n);
6    for (i=0; i<n; i++) {
7      printf("i je %d\n", i);
8    }
9    printf("poslije petlje %d\n", i);
10   return 0;
11 }
```

Probajte odgonetnuti šta će ovaj program ispisati na ekran ako korisnik unese `n=5`. Možete li?

Sada pogledajte rješenje:

```
i je 0
i je 1
i je 2
i je 3
i je 4
poslije petlje 5
```

Da bismo shvatili zašto je program ispisao baš ovo, trebamo razumjeti precizno kako funkcioniše `for` petlja, odnosno kojim redom se izvršavaju naredbe. Proći ćemo kroz program korak po korak.

U liniji 3 programa deklarisane su promjenljive `i` i `n`. U ovom trenutku one su neinicijalizirane.↑ U liniji 5 korisnik unosi `n` sa tastature, a rekli smo da će unijeti broj 5. Da vidimo šta se dešava dalje.

↑ Poglavlje 1.2.2, str. 29.

```
6    for (i=0; i<n; i++) {
```

1/15/01/98

U liniji 6 programa nalazi se zaglavlje for petlje koje se izvršava ovako: najprije se dešava *inicijalizacija*, tj. postavlja se `i` na 0. Zatim se provjerava uslov tj. da li je `i<n` odnosno da li je $0 < 5$. Pošto je uslov ispunjen prelazi se na *tijelo petlje*. Tijelo petlje u liniji 7 ispisuje poruku `i je ...`, u ovom trenutku promjenljiva i još uvijek ima vrijednost 0 pa će se ispisati `i je 0`.

Ovim je tijelo petlje završeno. Tek kada se izvrše sve naredbe u tijelu petlje izvršava se *ažuriranje kontrolne promjenljive* koje se nalazi u *zaglavlju for petlje*. Naredba za ažuriranje glasi `i++`, pa će se `i` povećati za 1 i postaće 1.

✓ Ažuriranje se izvršava nakon tijela petlje

Sada se ponovo provjerava uslov, da li je `i<n` tj. 1<5, što je ispunjeno. Ponovo se izvršava tijelo petlje, tj. ispisuje se `i je 1`. Ponovo se izvršava ažuriranje, pa `i` postaje 2 itd.

Preskočimo sada na posljednji prolaz `for` petlje u kojem se ispisala poruka `i je 4`. Nakon izvršenja tijela petlje (ispisa poruke `i je 4`) izvršava se ažuriranje petlje `i++`, `i` se povećava za 1 i postaje 5. Zatim se provjerava uslov `i<n`, pošto je `i=5` i `n=5` uslov *nije* ispunjen (5 nije manje od 5) i petlja se prekida. U ovom trenutku izvršavanje for petlje *skače* na kraj petlje tj. na prvu naredbu koja se nalazi poslije tijela petlje. To je naredba:

```
9  printf("poslije petlje %d\n", i);
```

U ovom trenutku `i` je 5 pa se ispisuje poruka `poslije petlje 5`.

Šta iz ovog primjera možemo zaključiti?

✓ Vrijednost kontrolne promjenljive po završetku petlje

1. Kontrolna promjenljiva će u prvom prolazu kroz petlju imati vrijednost koja je dodijeljena u polju *inicijalizacije*.
2. U posljednjem prolazu kroz petlju uslov je još uvijek ispunjen, pa ako smo stavili uslov `i<n`, `i` će imati vrijednost `n-1`.
3. Po izlasku iz petlje, kontrolna promjenljiva će imati vrijednost `n` odnosno imaće *prvu vrijednost za koju uslov više nije ispunjen*.

Ovo nam ostavlja razne zgodne mogućnosti pri radu sa petljama odnosno da možemo znati kako i kada se petlja završila.↑

↑ Poglavlje 4.1.6, str. 97.

4.1.3 *Zloupotrebe zaglavlja for petlje*

Ako znamo tačno kada se izvršava koje polje zaglavlja, to nam ostavlja mogućnost da ih koristimo na razne druge načine osim onog koji smo vidjeli naprijed. Ovo smo nazvali "zloupotrebom" iz razloga što se na taj način umanjuje čitljivost čitavog programa. Naravno, takav program će i dalje raditi ispravno.

Prije svega, svako od polja zaglavlja može biti prazno. `for` petlja je validna dok god u zaglavlju imamo dva tačka-zareza, a između njih se ne mora nalaziti ništa. Recimo, pogledajte ovaj program:

17/11/0198

```
#include <stdio.h>
int main() {
  int i=0,n;
  printf("Unesite broj n: ");
  scanf("%d", &n);
  for (; i<n; i++) {
    printf("i je %d\n", i);
  }
  printf("poslije petlje %d\n", i);
  return 0;
}
```

U odnosu na prethodni primjer, samo smo inicijalizaciju promjenljive `i` pomakli na mjesto gdje je ona i deklarisana, čime je polje za inicijalizaciju u zaglavlju for petlje ostalo prazno. Često ćete vidjeti i ovakav kod:

```
  for (i; i<n; i++) {
```

Ovo nije greška striktno govoreći, ali ćete u odgovarajućoj liniji dobiti upozorenje prevodioca

```
Statement with no effect
```

koje označava da imate kod koji ne radi ništa. Naime, ako navedete samo naziv promjenljive, sam taj naziv je ujedno i izraz koji ima određenu vrijednost, ali mi sa tom vrijednošću nismo uradili ništa, baš kao da ste u zasebnoj liniji napisali naredbu: "`i;`". Zapamtite da je prvo polje zaglavlja `for` petlje *naredba koja se izvršava prije ulaska u petlju*. Prevodilac vas tu upozorava da niste nešto slučajno zaboravili pošto je unesena naredba beskorisna. Da bismo izbjegli poplavu upozorenja (koja je generalno loša jer se mogu slučajno preskočiti ozbiljne greške), bolje je da pobrišemo ovo `i` koje je ujedno i nepotrebno jer je dozvoljeno imati prazno polje zaglavlja.

Također i ažuriranje može biti prazno. Gdje bismo trebali u prethodnom primjeru prebaciti `i++` kako bi polje zaglavlja za ažuriranje postalo prazno, a da program zadrži istu funkciju?

Na kraju, i sam uslov može biti prazan. Petlja koja glasi ovako:

✓ Beskonačna petlja

```
  for (;;) {
```

naziva se *beskonačna petlja*. Beskonačna petlja će se izvršavati sve dok korisnik ne prekine program. No moguće je i da programer prijevremeno prekine izvršavanje na neki način.↑ Dakle beskonačne petlje nisu potpuno beskorisne.

↑ Poglavlje 4.1.6, str. 97.

Nakon što smo potpuno ispraznili zaglavlje, u njega možemo dodati neke druge stvari. Npr. pogledajte ovaj program:

```
#include <stdio.h>
int main() {
  int i=0,n;
  printf("Unesite broj n: ");
  for (scanf("%d", &n); i<n; printf("i je %d\n", i++));
  printf("poslije petlje %d\n", i);
```

```
7   return 0;
8 }
```

Ovaj program i dalje radi potpuno isto što i prethodnih nekoliko primjera. Da biste prepoznali inicijalizaciju, uslov i ažuriranje for petlje, samo tražite znak tačka-zarez. Radi lakšeg razumijevanja obojićemo ih bojama kao i ranije (slika 4.4).

```
for (scanf("%d", &n); i<n; printf("i je %d\n", i++));
```

Slika 4.4: Zloupotreba zaglavlja for petlje

Dakle, pošto je inicijalizacija naredba koja se izvršava samo jednom prije početka petlje, tu smo ubacili poziv funkcije `scanf` jer nam je to potrebno samo jednom. Uslov je ostao isti kao i ranije. Operacija ažuriranja nam je sada kompletno ranije tijelo petlje sa svojom `printf` funkcijom. Možete uočiti da se unutar poziva `printf` funkcije `i` uvećava za jedan, pri čemu smo koristili postfiksni operator `++`↑ kako bismo osigurali da se na ekranu ispiše neuvećana vrijednost.

↑ Poglavlje 3.2.2, str. 81.

✓ Prazan blok koda

Konačno, zelenom bojom je označeno tijelo petlje koje se sastoji samo od znaka tačka-zarez! Naime, pošto smo kompletno bivše tijelo petlje uspjeli uvući u zaglavlje, tijelo nam je sada prazno. Ponekad u kodu se dešava da želimo označiti blok koda koji je prazan tj. koji ne sadrži niti jednu naredbu. To možemo uraditi na dva ravnopravna načina. Jedan je naravno da stavimo samo otvorenu i zatvorenu vitičastu zagradu

```
for (scanf("%d", &n); i<n; printf("i je %d\n", i++)) { }
```

a drugi je da navedemo znak tačka-zarez. Znak tačka-zarez bez drugih oznaka predstavlja naredbu koja ne radi ništa (u programiranju su takve naredbe poznate kao nul-naredbe).

4.1.4 *Petlje sa različitim koracima*

U svim dosadašnjim primjerima korak u petlji je bio jedan, što naravno ne mora biti. Probajmo recimo u našem ranijem primjeru staviti korak 3:

```
1  #include <stdio.h>
2  int main() {
3    int i,n;
4    printf("Unesite broj n: ");
5    scanf("%d", &n);
6    for (i=0; i<n; i+=3) {
7      printf("i je %d\n", i);
8    }
9    printf("poslije petlje %d\n", i);
10   return 0;
11 }
```

Obratite pažnju na polje ažuriranja u zaglavlju petlje. `i+=3` je naredba koja koristeći kombinovani operator↑ += uvećava `i` za 3. Isto bi značenje imala i naredba `i=i+3`. Obratite pažnju da je ažuriranje *naredba*. Da ste napisali samo `i+3` ovako:

↑ Poglavlje 1.2.8, str. 36.

```
for (i=0; i<n; i+3) {
```

u pitanju bi bila beskonačna petlja jer se `i` nikada ne bi mijenjalo, program bi do u beskonačnost ispisivao `i je 0`.

✓ *Naredba `i+3` ne mijenja vrijednost `i`, dok `i+=3` mijenja.*

✓ `i+3` vs. `i+=3`

Neka je korisnik unio broj 10. Program će sada ispisati:

```
Unesite broj n: 10
i je 0
i je 3
i je 6
i je 9
poslije petlje 12
```

Vidimo da se u svakom koraku `i` povećalo za tri. Odakle sada 12 na kraju? U posljednjem prolasku kroz petlju `i` je bilo 9. Zatim se izvršilo ažuriranje, `i` se povećalo za 3 i postalo 12. Sada će program evaluirati uslov `i<n`, pa pošto 12 nije manje od 10, petlja će se prekinuti i izvršavanje nastaviti u naredbi koja ispisuje "poslije petlje".

Drugim riječima, 12 je prva vrijednost promjenljive `i` za koju uslov više nije bio ispunjen. Još o ovome ćemo naučiti kroz neke primjere.

Zadatak 1. *Napisati program koji omogućuje korisniku da unese cijeli broj n, a zatim na ekranu ispisuje sve parne brojeve na intervalu $[n, n^2]$.*

Naravno, mogli bismo ovo riješiti tako što napišemo petlju koja prolazi kroz sve brojeve na intervalu $[n, n^2]$ a zatim provjeravamo da li je svaki od tih brojeva paran.

```
1  #include <stdio.h>
2  int main() {
3    int i,n;
4    printf("Unesite broj n: ");
5    scanf("%d", &n);
6    for (i=n; i<=n*n; i++) {
7      if (i%2 == 0)
8        printf("%d\n", i);
9    }
10   return 0;
11 }
```

Možemo li iskoristiti korak petlje kako bismo uzimali svaki drugi broj?

```
1  #include <stdio.h>
2  int main() {
3    int i,n;
4    printf("Unesite broj n: ");
5    scanf("%d", &n);
6    for (i=n; i<=n*n; i+=2) {
7      printf("%d\n", i);
```

```
  }
  return 0;
}
```

Problem je što ne znamo da li je broj n paran ili nije. Ako broj n nije paran, program iznad će ispisati sve neparne brojeve na intervalu, a ne sve parne. Rješenje je da vrijednost `i` bude u tom slučaju uvećana za jedan:

```
#include <stdio.h>
int main() {
  int i,n;
  printf("Unesite broj n: ");
  scanf("%d", &n);
  i=n;
  if (n%2 == 1) i++;
  for (; i<=n*n; i+=2) {
    printf("%d\n", i);
  }
  return 0;
}
```

ili kraće ovako:

```
  for (i=n+n%2; i<=n*n; i+=2) {
```

Zadatak 2. *Napisati program koji omogućuje korisniku da unese cijeli broj x, a zatim provjerava da li je x stepen dvojke (takav broj da je $x = 2^n$ gdje je n neki prirodan broj) te ispisuje poruku JESTE ako je uslov ispunjen.*

Ovaj zadatak bi se mogao riješiti tako što koristimo funkciju logaritma ili da se pokuša tražiti ostatak pri dijeljenju x sa dva više puta uzastopno. A možemo probati i ovako: proći ćemo kroz sve stepene dvojke koji su manji ili jednaki `x` i ako je na kraju rezultat jednak `x`-u ispisati traženu poruku.

```
#include <stdio.h>
int main() {
  int i,x;
  printf("Unesite broj n: ");
  scanf("%d", &x);
  for (i=1; i<=x; i*=2) {
    if (i==x) printf("JESTE\n");
  }
  return 0;
}
```

4.1.5 *Opadajuća for petlja*

17570198

Prirodno je postaviti pitanje: može li korak biti negativan? Odgovor je naravno da može, a takvu petlju nazivamo **opadajuća**. Primjer korištenja opadajuće petlje se može vidjeti ovdje:

```
#include <stdio.h>
int main() {
```

```
3   int i,n;
4   printf("Unesite broj n: ");
5   scanf("%d", &n);
6   for (i=n; i>0; i--) {
7     printf("i je %d\n", i);
8   }
9   printf("poslije petlje %d\n", i);
10  return 0;
11 }
```

Ako korisnik unese broj 5, izlaz programa će izgledati ovako:

```
i je 5
i je 4
i je 3
i je 2
i je 1
poslije petlje 0
```

Šta možemo zaključiti o opadajućim petljama iz ovog primjera?

- početna vrijednost za `i` je 5;
- u svakom prolazu `i` se smanjuje za 1;
- petlja se izvršava sve dok je `i` veće od 0; pošto je početna vrijednost 5, uslov mora biti ispunjen za tu početnu vrijednost, a u suprotnom se petlja neće niti jednom izvršiti;
- pošto smo stavili uslov `i>0`, u posljednjem prolazu `i` će biti 1;
- po izlasku iz petlje, kontrolna promjenljiva dobija prvu vrijednost za koju uslov više nije ispunjen, a to je 0.

4.1.6 *Naredbe break i continue*

Došlo je vrijeme da predstavimo dvije nove naredbe C programskog jezika koje služe specifično za kontrolu rada petlji, a to su naredbe *break* i *continue*.

Naredba **break** služi za prekid petlje. Ustvari, ovu naredbu smo već spominjali kada smo govorili o switch-case strukturi.↑ Zaista, `break` naredba ima dva značenja u programskom jeziku C. Unutar switch-case strukture, naredba `break` označava završetak slučaja, a izvršavanje programa se nastavlja iza završetka switch-case bloka. Unutar petlje, `break` označava prekid petlje, a izvršavanje programa će "skočiti" iza završetka petlje.

↑ Poglavlje 2.5.2, str. 59.

> Ako se naredba `break` ne nalazi niti u switch-case bloku niti u petlji, dobićete sintaksnu grešku! Često se kod početnika može vidjeti da pokušavaju prekinuti čitav program naredbom `break`.

Prekid programa se može postići naredbom `return` unutar `main` funkcije, a u drugim funkcijama može se koristiti funkcija `exit` iz biblioteke `stdlib.h`.

Primjer upotrebe naredbe `break` možete vidjeti u ovom programu:

```
#include <stdio.h>
int main() {
  int i=0,n;
  printf("Unesite broj n: ");
  scanf("%d", &n);
  for (;;) {
    printf("i je %d\n", i);
    i++;
    if (i>=n) break;
  }
  printf("poslije petlje %d\n", i);
  return 0;
}
```

Ovaj program je ekvivalentan primjeru koji smo proučavali ranije. Sada se koristi beskonačna for petlja, iz koje se izlazi naredbom `break`. Uslov je invertovan↑ (sjetimo se da je >= negacija od <) jer se u ovom programu petlja *prekida* kada je uslov ispunjen, a ranije smo u zaglavlju for petlje navodili uslov za koji će se petlja *ponavljati* sve dok je taj uslov ispunjen.

↑ Poglavlje 2.4, str. 49.

Naredba **continue** preskače sve preostale naredbe do kraja petlje i prelazi na sljedeću iteraciju petlje. Ovo znači da će se nakon naredbe `continue` izvršiti ažuriranje i ponovo krenuti od početka sa tijelom petlje. Zbog toga je sam naziv ove naredbe često zbunjujući, jer nije najjasnije šta se to nastavlja. Pogledajmo sljedeći primjer upotrebe naredbe `continue`:

✓ Naredba `continue`

```
#include <stdio.h>
#include <math.h>
int main() {
  int i;
  double x,y;
  for (i=0; i<100; i++) {
    x = sin(i/3.);
    if (x<=0) continue;
    y = i + sqrt(x);
    printf("i=%d x=%f y=%f\n", i, x, y);
  }
  return 0;
}
```

U ovom programu, za svaku vrijednost `i` na intervalu [0,99] računa se određeni matematički izraz, pod uslovom da takav izraz ima smisla, odnosno ako je potkorijena veličina x pozitivan broj. Za vrijednosti $i = 0, 1, 2...9$, `x` će biti pozitivno. Već za $i = 10$ važi $i/3 = 3.3333...$, pa je $x = \sin 3.333 = -0.187 \leqslant 0$. U ovom slučaju će se izvršiti naredba `continue`, što znači da se neće računati `y=i+sqrt(x)` (odnosno neće se računati korijen negativnog broja), niti će se ispisati poruka na ekran. Izvršavanje programa preskače na naredbu `i++` čime `i` postaje 11 i zatim se izvršava sljedeća iteracija petlje, tj. računa se `sin(11/3.)` što će opet biti negativan broj itd. Za `i=30` će `x` ponovo biti pozitivno itd.

Da bismo dodatno utvrdili razumijevanje naredbi `break` i `continue`,

1755198

pogledajmo još jedan primjer:

```
1  #include <stdio.h>
2  int main() {
3    int k,h=0;
4    for (k=0; k<70; k++) {
5      if(k%6==0) continue;
6      h++;
7      if (k==6 || k==12)
8        break;
9    }
10   printf("h=%d\n",h);
11   return 0;
12 }
```

Ovaj program će na ekranu ispisati broj 58. Da bismo razumjeli zašto, proći ćemo kroz njega liniju po liniju.

Kao što znamo, ova `for` petlja bi se trebala izvršiti 70 puta. Brojač `h` će se uvećati za jedan u svakom prolazu kroz petlju (linija 6), osim kada se izvrši naredba `continue` u liniji 5, budući da ona preskače naredbu `h++`. Ova naredba `continue` će se izvršiti za svako `k` čiji je ostatak pri dijeljenju sa 6 jednak nuli, odnosno za sve brojeve djeljive sa 6.[1] Na intervalu [0,70) takvih brojeva ima ukupno 12:

[1] Naglašavamo da je i nula djeljiva sa šest, te važi `0%6=0`.

```
0, 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66.
```

Dakle, naredba `h++` će se izvršiti 70-12=58 puta. Što se tiče naredbe `break`, ona se neće nikada izvršiti, jer su brojevi 6 i 12 djeljivi sa 6 (vidimo ih na prethodnom spisku), pa će u oba slučaja naredba `continue` preskočiti naredbu `break`.

Ako program modifikujemo tako da stoji:

```
7    if (k==6 || k==12 || k==17)
```

vidimo da 17 nije djeljivo sa 6, pa će se program prekinuti kada `k` bude 17. Petlja će se izvršiti 18 puta (za vrijednosti k=0,1,2,...,17 – vrijednost 17 moramo uključiti jer će `break` nastupiti tek nakon naredbe `h++`). Na ovom intervalu će se tri puta desiti `continue`, tako da će program ispisati broj 15.

4.1.7 *Višestruke (ugniježđene) petlje*

U bloku naredbi koje čine tijelo petlje mogu se nalaziti bilo kakve validne naredbe programskog jezika C. Dakle, u bloku se može nalaziti i druga petlja. Šta bi takav program mogao značiti? Pa ako znamo da pomoću `for` petlje ponavljamo neki blok n puta, ako se u tom bloku nalazi druga `for` petlja koja ponavlja neki drugi blok k puta, onda će taj unutrašnji blok biti ponovljen ukupno $n \cdot k$ puta.

Napravićemo jedan školski primjer programa koji radi sa dvostrukom petljom kako bismo bolje razumjeli šta se u takvoj petlji dešava sa kontrolnim promjenljivim.

✓ Dvostruka petlja

1777198

```
#include <stdio.h>
int main() {
  int i,j;
  for (i=0; i<5; i++) {
    for (j=0; j<5; j++) {
      printf("i=%d,j=%d\n", i, j);
    }
  }
  return 0;
}
```

Prvo što moramo uočiti je da se u ugniježđenoj petlji ne može koristiti ista kontrolna promjenljiva. Da smo u unutrašnjoj petlji koristili promjenljivu `i` ovako:

```
  for (i=0; i<5; i++) {
    for (i=0; i<5; i++) {
      ...
```

ona bi u samoj toj unutrašnjoj petlji dostigla vrijednost 5. Dakle, po završetku unutrašnje petlje `i` bi bilo 5, pa bi se odmah zatim i vanjska petlja završila jer uslov `i<5` više nije ispunjen, dakle dio označen sa tri tačke bi se izvršio sve skupa 5 puta. Inače, možete slobodno koristiti istu kontrolnu promjenljivu za veći broj petlji dok god te petlje nisu ugniježđene.

Dakle, da vidimo šta će ispisati prethodni program u kojem smo korektno koristili promjenljive `i` i `j`:

```
i=0,j=0
i=0,j=1
i=0,j=2
i=0,j=3
i=0,j=4
i=1,j=0
i=1,j=1
...
i=4,j=3
i=4,j=4
```

Ukupno biće ispisano 25 poruka. Najprije će `i` postati 0. Zatim se izvršava kompletna `j`-petlja (obično se tako skraćeno naziva petlja čija je kontrolna promjenljiva `j`). Ta petlja će proći kroz sve vrijednosti `j` od 0 do 4 i završiti se, ispisujući prvih pet poruka. Po završetku ove petlje `j` ima vrijednost 5 (što nam nije bitno), a `i` i dalje ima vrijednost 0. Zatim se izvršava naredba `i++` kojom `i` postaje 1. Pošto je `i` i dalje manje od 5, ponovo se izvršava kompletna `j`-petlja. U zaglavlju `j`-petlje `j` se vraća na polaznu vrijednost 0 i zatim prolazi kroz vrijednosti [0,5) itd. Ovaj ciklus se završava kada `i` postane 5, a `j` je također 5.

Dvostruku petlju je najlakše shvatiti kroz neke primjere kao što je sljedeći.

Zadatak 3. *Napraviti program koji omogućuje korisniku da unese prirodan broj n, a zatim na ekranu iscrtava kvadrat ispunjen znakovima zvi-*

17570198

jezda (*) čije su sve stranice dugačke po n zvijezda.

```
Unesite broj n: 5
*****
*****
*****
*****
*****
```

Kod rješavanja ovog zadatka možemo primijeniti *opštu strategiju rješavanja programskih zadataka*, koja je inspirisana latinskom uzrečicom "podijeli pa vladaj":

✓ *Ako imate složen problem koji ne znate (isprve) riješiti, podijelite ga na jednostavnije probleme koje znate riješiti.*

✓ Strategija rješavanja problema

Čak i ako ne uspijete riješiti sve dijelove zadatka, barem imate nešto što je u praksi često bolje nego ništa. Ostatak problema možete podijeliti sa vašim kolegama, ili naći neko polu-gotovo rješenje pretraživanjem interneta.

U ovom slučaju možemo zadatak podijeliti ovako:

- Podzadatak 1: Nacrtati jedan red od n zvjezdica.
  - Da bismo ovo uradili, trebamo ponoviti zadatak ispisa jedne zvjezdice n puta.
- Podzadatak 2: Ponoviti podzadatak 1 (ispis jednog reda) n puta.

Važno je zapamtiti da računar obavlja zadatke koje smo mu dali jedan po jedan, pa je dovoljno strpljivo navoditi rješenja i ponavljati ih potreban broj puta dok ne dođemo do konačnog rezultata.

Dakle, kako ćemo ispisati jednu zvjezdicu na ekranu? Kôd koji obavlja taj zadatak glasi `printf("*");` – ovom naredbom smo ispisali jednu zvjezdicu. Sada nju trebamo ponoviti n puta. Za ponavljanje nekog zadatka n puta koristimo for petlju:

```
for (i=0; i<n; i++)
  printf("*");
```

Time smo dobili n zvjezdica, što čini jedan red. Ostaje da ovaj zadatak ispisa jednog reda ponovimo n puta. Možemo li kompletan kod naveden iznad staviti u još jednu petlju, ovako?

```
for (i=0; i<n; i++)
  for (i=0; i<n; i++)
    printf("*");
```

1777098

Ovo nije ispravno jer u ugniježđenoj petlji koristimo istu kontrolnu promjenljivu. Moramo preimenovati jednu od njih, npr. u ugniježđenoj petlji ćemo koristiti `j`:

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    printf("*");
```

Podsjetimo se da blok koda ne moramo ograditi vitičastim zagradama ako se u njemu nalazi samo jedna naredba. `printf("*");` je samo jedna naredba, tako da u `j`-petlji nemamo vitičastih zagrada. No sama `j`-petlja u svojoj cjelosti također predstavlja samo jednu naredbu! Iz perspektive C-a, naredbe koje podrazumijevaju blok kao što su `for` i `if` i same čine jednu jedinstvenu naredbu, pa ako je to jedina naredba u bloku, taj blok ne moramo ograditi vitičastim zagradama. Generalno, veća je vjerovatnoća da ćete isključiti neku naredbu iz bloka jer niste stavili zagrade, nego da ćete izazvati sintaksnu grešku.

Prethodni primjer već liči na funkcionalan kod. Napišimo i ostatak programa da provjerimo kako radi.

```
1  #include <stdio.h>
2  int main() {
3    int i,j,n;
4    printf("Unesite broj n: ");
5    scanf("%d", &n);
6    for (i=0; i<n; i++)
7      for (j=0; j<n; j++)
8        printf("*");
9    return 0;
10 }
```

Probajmo pokrenuti ovaj program i unijeti recimo broj 3:

```
Unesite broj n: 3
*********
```

Program je ispisao devet zvjezdica, ali se sve te zvjezdice nalaze u istom redu. Zašto? Da li smo negdje zatražili od računara da pređe u novi red? Nismo. Potrebno je nakon svakog reda zvjezdica ispisati po jedan znak `\n`. Gdje trebamo dodati odgovarajući `printf`? Ako to uradimo ovako:

```
1  #include <stdio.h>
2  int main() {
3    int i,j,n;
4    printf("Unesite broj n: ");
5    scanf("%d", &n);
6    for (i=0; i<n; i++)
7      for (j=0; j<n; j++)
8        printf("*");
9    printf("\n");
10   return 0;
11 }
```

1/13/01198

nećemo imati nikakvog efekta jer će se novi red ispisati na kraju programa, gdje je nevidljiv. Ovakva popravka:

```
1  #include <stdio.h>
2  int main() {
3    int i,j,n;
4    printf("Unesite broj n: ");
```

```
5   scanf("%d", &n);
6   for (i=0; i<n; i++) {
7     for (j=0; j<n; j++) {
8       printf("*");
9       printf("\n");
10    }
11  }
12  return 0;
13 }
```

će pak proizvesti sljedeći rezultat:

```
Unesite broj n: 3
*
*
*
*
*
*
*
*
*
```

Drugim riječima, dobijamo po jedan novi red nakon svake zvjezdice. Ispravno rješenje je sljedeće: `j`-petlja predstavlja ispis jednog reda, pa znak `\n` treba ispisati nakon `j`-petlje, ali unutar `i`-petlje koja odbrojava redove, tj. ovako:

```
1  #include <stdio.h>
2  int main() {
3    int i,j,n;
4    printf("Unesite broj n: ");
5    scanf("%d", &n);
6    for (i=0; i<n; i++) {
7      for (j=0; j<n; j++) {
8        printf("*");
9      }
10     printf("\n");
11   }
12   return 0;
13 }
```

Zadatak 4. *Napraviti program koji na ekranu ispisuje tablicu množenja prvih 10 prirodnih brojeva.*

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  ...
```

1/15/01/98

Tablica množenja zahtijeva da svaki broj [1,10] pomnožimo sa svakim brojem [1,10]. Ukupno na ekranu trebamo ispisati 100 brojeva. U prvom redu nalaze se brojevi 1,2,3,...,10 pomnoženi brojem 1, u drugom redu se nalaze isti ti brojevi pomnoženi brojem 2 itd. Ovdje možemo uočiti sličnost sa prethodnim programom. Jedan red tablice ispisujemo petljom:

```
  for (j=1; j<=10; j++)
    printf("%d", i*j);
```

pri čemu je `i` ovdje neki broj sa kojim množimo sve članove tog reda. Možemo uočiti da petlja prolazi kroz sve vrijednosti $j \in [1, 10]$, te da smo `printf` funkciji direktno proslijedili rezultat množenja brojeva `i` i `j`. Ostaje da uočimo da ova promjenljiva `i` također poprima sve vrijednosti na intervalu [1,10] te da na kraju svakog reda ispišemo `\n`.

```
1  #include <stdio.h>
2  int main() {
3    int i,j;
4    for (i=1; i<=10; i++) {
5      for (j=1; j<=10; j++) {
6        printf("%d", i*j);
7      }
8      printf("\n");
9    }
10   return 0;
11 }
```

Ovo će na ekranu proizvesti dosta zbrkan ispis:

```
12345678910
2468101214161820
...
```

Problem je što između pojedinačnih brojeva nemamo nikakvog razmaka. Možemo jednostavno dodati jedan razmak u `printf` nakon formata `%d`:

```
6      printf("%d ", i*j);
```

To bi nam dalo sljedeći ispis:

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
...
```

Ovo već počinje ličiti na tablicu množenja, ali brojevi nisu lijepo poravnati. Da bismo dobili lijepo poravnanje kao u primjeru, uočavamo da će najveći broj u ovoj tablici biti 100, dakle trocifren, tako da je dovoljno da za svaki broj rezervišemo po 4 mjesta. Preostaje da prepravimo poziv funkcije printf na sljedeći način:

```
6      printf("%4d", i*j);
```

i time ćemo dobiti ispis kao u postavci zadatka.

Spomenućemo još i da, kod ugniježđenih petlji, broj iteracija unutrašnje petlje može zaviseti od indeksa vanjske petlje, odnosno unutrašnja petlja se neće svaki put izvršiti isti broj puta. Takve primjere ćemo vidjeti u poglavlju 4.4.

1755/0198

4.2 *while petlja*

Nakon što smo se detaljno upoznali sa for petljom, možemo proučiti i drugi tip petlje u C-u, a to je `while` petlja. Za razliku od for petlje, while petlja se koristi kada broj izvršenja nije unaprijed poznat, tj. ne zavisi od neke konkretne konstante ili promjenljive.

Riječ **while** se prevodi kao "sve dok" – "sve dok je uslov ispunjen, ponavljaj tijelo petlje". Sintaksa while petlje je sljedeća:

```
while (uslov) tijelo;
```

Sve dok je uslov ispunjen, tijelo petlje se ponavlja. Obratite pažnju da je sa while petljom vrlo lako napraviti beskonačnu petlju! To će se desiti ako se u tijelu petlje ne dešava ništa što bi promijenilo istinitost uslova. Npr. ako napišemo:

```
while (x>0) y++;
```

ovo će sigurno biti ili beskonačna petlja ili petlja koja se ne izvrši nijednom. Ako je uslov `x>0` ispunjen prvi put, biće ispunjen i svaki sljedeći put jer se `x` ne mijenja nigdje u petlji. Pogledajmo malo složeniji primjer:

```
1  #include <stdio.h>
2  int main() {
3    int x, suma=0;
4    while (suma <= 100) {
5      printf("Unesite broj: ");
6      scanf("%d", &x);
7      suma += x;
8    }
9    printf("suma je %d\n", suma);
10   return 0;
11 }
```

Značenje while petlje ovdje je "sve dok je suma manja ili jednaka od 100 ponavljaj unos broja `x` i dodaj ga na sumu". U ovom primjeru ne možemo znati kakve brojeve će unositi korisnik, suma može odmah prekoračiti broj 100 a može se unositi beskonačno mnogo brojeva (ako korisnik bude unosio i negativne brojeve ili nule).

for petlja je istovremeno jednostavnija i složenija od while petlje. Jednostavnija je zato što – ako se ispravno koristi – na prvi pogled možemo vidjeti koliko će se puta izvršiti. Složenija je zato što se for petlja sastoji od nekoliko dijelova koji se ne izvršavaju onim redom kojim su navedeni, nego se sa uslova skače na tijelo petlje, pa nazad na ažuriranje.

✓ Da li je jednostavnija for ili while petlja?

Kod while petlje, analiza je vrlo jednostavna. Ako je uslov ispunjen, izvršava se tijelo petlje, te se program vraća na uslov. Ako nije, izvršenje skače na prvu naredbu poslije tijela petlje.

Ako želimo namjerno napraviti beskonačnu petlju pomoću petlje while, samo se podsjetimo da je u C-u jedinica logička vrijednost istine.↑ Pa ako napišemo:

↑ Poglavlje 2.2, str. 46.

1755798

```
  while(1)
```

ova petlja će biti beskonačna, pošto će uslov u petlji uvijek imati vrijednost "istina".

4.2.1 *Transformacija for u while i obrnuto*

Možemo uočiti da je while petlja ustvari ekvivalentna for petlji. Svaka for petlja se može pretvoriti u while petlju i obrnuto. Zašto onda postoje različiti tipovi petlji? Odgovor leži u nečemu što smo do sada ponovili više puta: većina programskih struktura postoji prije svega zato da olakšaju čitanje i razumijevanje programa, odnosno da smanje pojavu grešaka i olakšaju njihovu popravku.

for petlja olakšava razumijevanje programa u kojem se može tačno znati broj ponavljanja nekog koda i raspon vrijednosti koje treba poprimiti kontrolna promjenljiva. Tamo gdje nam ovaj broj nije unaprijed jasan, ili gdje nam nije bitna vrijednost kontrolne promjenljive, koristimo while petlju.

Transformacija for petlje u while petlju i obrnuto je korisna vježba za razumijevanje obje vrste petlji. Krenućemo od jednostavnog primjera koji smo koristili više puta ranije:

```
for (i=0; i<n; i++) {
  printf("i je %d", i);
}
```

Kao što znamo, možemo dobiti program koji obavlja istu funkciju ako inicijalizaciju prebacimo ispred petlje, a ažuriranje postavimo u tijelo petlje nakon ostalih naredbi:

```
i=0;
for (; i<n; ) {
  printf("i je %d", i);
  i++;
}
```

A for petlja koja u zaglavlju sadrži samo uslov je ustvari while petlja! Odnosno možemo napisati:

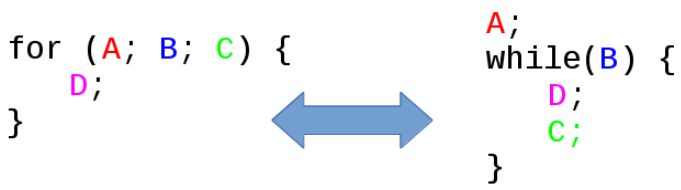
```
i=0;
while (i<n) {
  printf("i je %d", i);
  i++;
}
```

Generalno, svaku for petlju (koja ne sadrži `continue`)[2] možemo pretvoriti u while petlju na sljedeći način (slika 4.5): inicijalizaciju (označenu crvenom bojom) prebacimo ispred petlje, ažuriranje (označeno zelenom bojom) prebacimo u tijelo petlje nakon ostalih naredbi, a uslov (označen plavom bojom) ostaje gdje jeste i sada postaje uslov while petlje.

Moguća je i transformacija u obrnutom smjeru, pri čemu tražimo u okolnom kodu naredbe koje bi mogle biti kandidati za inicijalizaciju i

[2] Ako petlja sadrži naredbu `continue`, onda ćemo imati efekat da će u while varijanti biti preskočena i naredba ažuriranja. I to se može riješiti ali zahtijeva malo više analize.

17570198



Slika 4.5: Transformacija for petlje u while petlju i obrnuto

ažuriranje, pa ako ih ne nađemo, možemo ta polja for petlje ostaviti i praznim.

4.3 *do-while petlja*

Mnogi početnici u programiranju ne razlikuju **do-while** petlju od **while** petlje. Ustvari, u pitanju je potpuno različit tip petlje koja igrom slučaja koristi istu ključnu riječ `while`. Da bi se izbjegla ovakva zabuna, u mnogim jezicima se koristi *repeat-until* petlja koja je ekvivalentna do-while s tim što je uslov negiran.

✓ *Razlika između while i do-while petlje je u tome što se while petlja ne mora izvršiti niti jednom, dok će se do-while petlja uvijek izvršiti barem jednom.*

✓ while vs. do-while

Recimo možemo napisati ovakav kod:

```
int x=5;
while (x<3) {
  printf("Lozinka glasi: califragilisticexpialidocius\n");
}
```

Ova petlja se neće izvršiti niti jednom, korisnik nikada neće vidjeti lozinku jer uslov `x<3` u datom trenutku sigurno nije ispunjen. Izvršenje programa preskače kompletno tijelo petlje i nastavlja sa prvom naredbom nakon petlje. Da smo umjesto toga stavili do-while petlju:

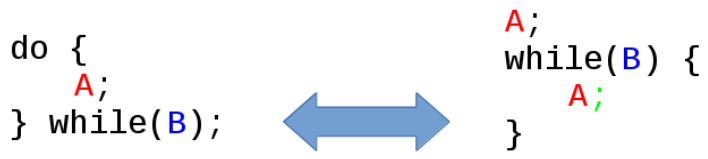
```
1 int x=5;
2 do {
3   printf("Lozinka glasi: califragilisticexpialidocius\n");
4 } while (x<3);
```

korisnik bi sigurno ugledao lozinku, bez obzira da li je uslov ispunjen ili ne. Ispunjenost uslova utiče isključivo na to koliko puta će se petlja ponoviti.

Iz ovoga proizlazi da ne možemo jednostavno pretvoriti do-while petlju u while petlju, nego bismo morali ponoviti kompletan kod petlje (slika 4.6).

Ovaj dupli kod možemo izostaviti samo ako znamo da je uslov B sigurno ispunjen kada prvi put ulazimo u while petlju. do-while petlja upravo služi da bismo izbjegli nepotrebne inicijalizacije promjenljivih koje osiguravaju barem jedno izvršenje petlje.

1/15/01/98



Slika 4.6: Transformacija do-while petlje u while petlju

Obratimo pažnju na sintaksu do-while petlje koja se može vidjeti na primjeru iznad: nakon ključne riječi `do` slijedi tijelo petlje (blok naredbi), zatim ključna riječ `while`, uslov naveden u običnim (malim) zagradama, te na kraju znak tačka-zarez. Treba naglasiti da se znak tačka-zarez javlja tek na kraju! Nema ga poslije riječi `do` ili prije riječi `while`.

Da bismo pokazali kako se koristi do-while petlja, podsjetimo se jednog programa koji smo koristili za demonstraciju rada switch-case strukture.↑ Srezaćemo program na interesantni dio:

↑ Poglavlje 2.5.2, str. 59.

```
1  #include <stdio.h>
2  int main() {
3    int mjesec;
4    printf("Unesite mjesec: ");
5    scanf("%d", &mjesec);
6    if (mjesec < 1 || mjesec > 12) {
7      printf("Neispravan mjesec\n");
8      return 0;
9    }
10   if (mjesec == 1) ...
```

U programu smo morali osigurati da se promjenljiva `mjesec` nalazi na intervalu [1,12]. To smo tada mogli uraditi tako što u slučaju neispravnog mjeseca prekinemo program ili možemo zatražiti ponovni unos naredbom `goto`. Sada smo naučili petlje, pa možemo iskoristiti while petlju da bismo ponavljali unos mjeseca sve dok je on neispravan (dok je izvan intervala [1,12]).

```
#include <stdio.h>
int main() {
  int mjesec;
  while (mjesec < 1 || mjesec > 12) {
    printf("Unesite mjesec: ");
    scanf("%d", &mjesec);
  }
  ...
```

Ovaj program sadrži grešku. Naime, uporedili smo vrijednost promjenljive `mjesec` sa brojevima 1 i 12 prije nego što je promjenljiva dobila ikakvu vrijednost. Ovo je *pristup neinicijaliziranoj promjenljivoj*! Iako će u velikoj većini slučajeva ovaj program raditi ispravno, postoji određena vjerovatnoća da će promjenljiva slučajno dobiti vrijednost na intervalu [1,12], tako da korisnik nikada neće ni dobiti priliku da unosi

1/17/01/98

mjesec.

Ovaj problem možemo riješiti na nekoliko načina. Možemo staviti još jedan unos promjenljive `mjesec` prije petlje koji će se sigurno izvršiti:

```
#include <stdio.h>
int main() {
  int mjesec;
  printf("Unesite mjesec: ");
  scanf("%d", &mjesec);
  while (mjesec < 1 || mjesec > 12) {
    printf("Unesite mjesec: ");
    scanf("%d", &mjesec);
  }
  ...
```

Međutim, ovo bi dovelo do *ponavljanja koda*. Svako ponavljanje koda smatra se štetnim. Nekada je lakše riješiti problem tako što uradimo copy-paste određenog koda onoliko puta koliko vam je potreban, ali to treba izbjegavati kada god je moguće. Pored toga što nepotrebno produžuje program i otežava čitljivost, ponavljanje koda može voditi ka greškama jer kada uočite grešku u kodu, sada tu istu grešku morate popraviti na više mjesta, a velika je vjerovatnoća da ćete preskočiti neko od tih mjesta.

Možete i postaviti inicijalnu vrijednost promjenljive `mjesec` na neku vrijednost koja osigurava da će se while petlja izvršiti barem jednom:

```
#include <stdio.h>
int main() {
  int mjesec=0;
  while (mjesec < 1 || mjesec > 12) {
    printf("Unesite mjesec: ");
    scanf("%d", &mjesec);
  }
  ...
```

Pošto smo postavili `mjesec` na nulu, znamo da će se petlja izvršiti barem jednom. Ovo je također jedna vrsta koda koji je ispravan i daje tačan rezultat, ali je primjer loše prakse u programiranju koja se naziva *magična konstanta*.

✓ *Magična konstanta je vrijednost čija svrha nije jasna sama po sebi.* ✓ Magična konstanta

Naime, ovaj primjer je dovoljno jednostavan da je jasno šta se ovdje dešava, ali u složenijim programima programer koji analizira taj program može postaviti pitanje: šta predstavlja ta nula? Zašto baš nula? Zašto ne 1000 ili -1? Navođenje bilo kakve konstantne vrijednosti u kodu u pravilu zahtijeva komentar koji objašnjava zašto je baš ta vrijednost odabrana.

Ako malo razmislimo, u oba ova primjera mi smo ustvari simulirali rad do-while petlje. U primjeru sa ponavljanjem koda, mi smo upravo obavili transformaciju while u do-while petlju. U primjeru sa `mjesec=0`, postavili smo početnu vrijednost promjenljive `mjesec` da bismo osigurali da se while petlja izvrši barem jednom. Pravo pitanje koje će

1757019 98

iskusniji programer postaviti početniku koji koristi magične konstante glasi "šta želiš postići?" Ako želimo postići da se petlja izvrši barem jednom, za to se koristi do-while petlja. Pa ćemo prepraviti program tako da se koristi upravo ta petlja:

```
#include <stdio.h>
int main() {
  int mjesec;
  do {
    printf("Unesite mjesec: ");
    scanf("%d", &mjesec);
  } while (mjesec < 1 || mjesec > 12);
  if (mjesec == 1) {
    ...
```

Vidimo da u ovom primjeru nema potrebe za magičnim konstantama niti za ponavljanjem kôda. Inače, preporuka je da koristimo do-while kada god uslov petlje zavisi od neke promjenljive koja prvu smislenu vrijednost dobija tek unutar tijela petlje.

4.4 *Primjeri zadataka sa petljama*

U nastavku ćemo proći korak po korak kroz različita rješenja nekih interesantnih zadataka iz programiranja u kojima se koriste isključivo petlje. Na ovaj način ćemo kroz primjere naučiti veći broj različitih "trikova" koji nam u nastavku našeg učenja programiranja mogu biti vrlo korisni.

4.4.1 *Crtanje oblika na ekranu ASCII karakterima*

Prilikom učenja programiranja susrešćete se sa određenim brojem primjera gdje se na ekranu iscrtava određeni oblik sastavljen od pluseva, zvjezdica, tačaka i sl. pri čemu su možda dimenzije oblika unesene putem tastature. Koja je svrha ovakvih zadataka, ako znamo da se već dugo vremena nikakvi ozbiljni programi niti igre ne rade na ovaj način?

Crtanje 2D oblika je jedan vrlo praktičan primjer kroz koji možemo razumjeti način rada programa, a pogotovo dvostrukih petlji. Vidjećemo kasnije da će nam to biti korisno kada budemo učili matrice i druge višedimenzionalne strukture. Mnogi lakše razumiju apstraktne pojmove na vizualan način, ako ih vide ilustrovane nekim crtežom.

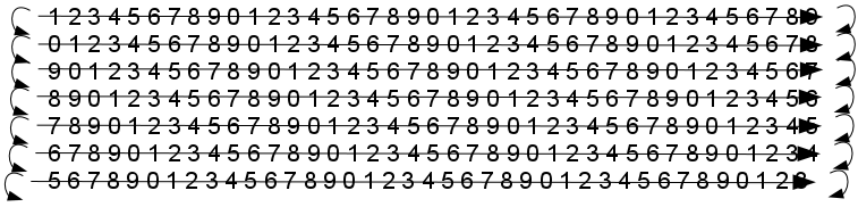
Prilikom rješavanja ovih zadataka, vrlo je važno shvatiti da se mi iz C programa ne možemo (na standardan način) pozicionirati na proizvoljnu lokaciju na ekranu. Ne možemo ići prema lijevo niti prema gore. Moguće je koristiti znak `\b` za brisanje jednog karaktera, ali ovo nam u pravilu nije previše korisno. Ako želimo pomjeriti kursor prema dolje ispisujemo znak za novi red `\n`, ali obratite pažnju da se nakon ispisa `\n` više ne možemo vratiti nazad prema gore. Kretanje u desno

1.1.5.7.0.1.9.8

postižemo ispisom znaka razmaka: `printf(" ");`

Dakle, kada razmišljamo o rješenju, uvijek moramo imati na umu da se kursor kreće red po red, odozgo prema dolje, a unutar jednog reda s lijeva na desno. Ovaj smjer kretanja odgovara tzv. scan-linijama koje se koriste inače za osvježavanje sadržaja na ekranima (slika 4.7).

✓ Redoslijed kretanja kursora na ekranu



Slika 4.7: Redoslijed kretanja po scan-linijama

Zadatak 5. *Nacrtati trougao sastavljen od zvjezdica kao na primjeru čije su katete dugačke po n zvjezdica (broj n unosi korisnik).*

```
Unesite broj n: 5
*
**
***
****
*****
```

Ovdje ćemo se podsjetiti opšteg principa rješavanja programskih zadataka datog u uvodnom poglavlju knjige ,↑ pa ćemo dati jednu njegovu varijantu:

↑ Poglavlje 1.3, str. 38.

✓ *Ako ne znate uraditi ono što se traži, uradite dio koji znate a zatim probajte prepravljati program prema ciljanom rješenju.*

✓ Strategija rješavanja problema II

Ono što znamo uraditi je nacrtati kvadrat čije stranice čini n zvjezdica, jer smo to uradili u zadatku 3.↑

↑ Poglavlje 4.1.7, str. 99.

```
Unesite broj n: 5
*****
*****
*****
*****
*****
```

Programski kod glasi:

```c
#include <stdio.h>
int main() {
  int i,j,n;
  printf("Unesite broj n: ");
  scanf("%d", &n);
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      printf("*");
    }
    printf("\n");
```

```
11   }
12   return 0;
13 }
```

Na koji način se ovaj program razlikuje od željenog? U slučaju kvadrata, u svakom redu imamo n zvjezdica, dok kod trougla u prvom redu imamo 1 zvjezdicu, u drugom 2 zvjezdice itd. Možemo reći da u nekom opštem `i`-tom redu imamo `i` zvjezdica. Uočavamo da je unutrašnja `j`-petlja (linije 7 i 8) zadužena za ispis jednog reda zvjezdica, te da je njen krajnji opseg n. Ostaje da nađemo to `i` koje nam definiše broj zvjezdica – a to je upravo ista promjenljiva `i` koju imamo u liniji 6! Naime, u prvom redu kvadrata `i` će biti 0, u drugom će biti 1 itd. Nama je potrebna u prvom redu jedna zvjezdica, u drugom dvije itd. pa možemo pisati:

```
7     for (j=0; j<i+1; j++) {
```

ili

```
7     for (j=0; j<=i; j++) {
```

Zadatak 6. *Napraviti program koji na ekranu iscrtava trougao sastavljen od zvjezdica kao na primjeru, čije su katete dugačke po n zvjezdica (broj n unosi korisnik).*

```
Unesite broj n: 5
*****
 ****
  ***
   **
    *
```

Tekst zadatka je isti, ali se oblik razlikuje tj. trougao je rotiran. Na koji način možemo ispisati ovako rotiran trougao?

Krenimo opet analizirati svaki pojedinačni red trougla. U prvom redu imamo 5 zvjezdica, dakle n zvjezdica, u drugom redu imamo 4 zvjezdice (n-1), u trećem 3 (n-2) itd. Dalje uočavamo da je svaki red pomaknut u desnu stranu. Da se ne bi desilo da su zvjezdice pomaknute ulijevo, prije svakog reda trebamo ispisati određeni broj razmaka. Koji je to broj?

1. U prvom redu ispisujemo 0 razmaka i n zvjezdica
2. U drugom redu ispisujemo 1 razmak i n-1 zvjezdica
3. U trećem redu ispisujemo 2 razmaka i n-2 zvjezdica
4. ...

17570198

Možemo uočiti da u nekom `i`-tom redu treba ispisati `i` razmaka i `n-i` zvjezdica, pri čemu `i` kreće od 0, a to je upravo ono što nam treba. Rješenje glasi:

```
1  #include <stdio.h>
2  int main() {
3    int i,j,n;
4    printf("Unesite broj n: ");
5    scanf("%d", &n);
6    for (i=0; i<n; i++) {
7      for (j=0; j<i; j++)
8        printf(" ");
9      for (j=0; j<n-i; j++)
10       printf("*");
11     printf("\n");
12   }
13   return 0;
14 }
```

Petlja u linijama 7 i 8 ispisuje `i` razmaka, a petlja u linijama 9 i 10 ispisuje `n-i` zvjezdica, sve skupa se ponavlja `n` puta petljom koja počinje u liniji 6.

Zadatak 7. *Napraviti program koji na ekranu iscrtava šuplji kvadrat kao na slici sastavljen od znakova tačka pri čemu je svaka stranica kvadrata dugačka n znakova (korisnik unosi broj n sa tastature).*

```
Unesite broj n: 5
.....
.   .
.   .
.   .
.....
```

Znamo da se ispis vrši red po red, pa uočavamo najprije da su prvi i posljednji red međusobno jednaki, i sastoje se od n tačaka. Ako pogledamo redove "u sredini", i oni su međusobno jednaki. Svaki od ovih srednjih redova sastoji se od: jedne tačke, `n-2` razmaka te na kraju još jedna tačka. Broj redova u sredini oblika je `n-2`. Ova dvica potiče od činjenice da smo isključili prvi i posljedni red. Dakle rješenje zadatka bi glasilo ovako:

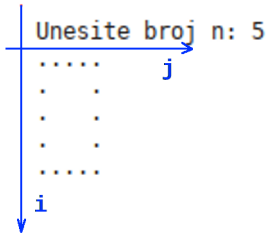
```
#include <stdio.h>
int main() {
  int i,j,n;
  printf("Unesite broj n: ");
  scanf("%d", &n);

  for (i=0; i<n; i++)
    printf(".");
  printf("\n");

  for (i=0; i<n-2; i++) {
    printf(".");
    for (j=0; j<n-2; j++)
      printf(" ");
    printf(".\n");
  }

  for (i=0; i<n; i++)
    printf(".");
```

```
20  printf("\n");
21
22  return 0;
23 }
```

Kôd u linijama 7-9 ispisuje prvi red tačaka koji se završava prelaskom u novi red (`\n`). Linije 18-20 predstavljaju posljednji red i možemo vidjeti da su identične linijama 7-9. U linijama 11-16 ispisuje se sredina, pri čemu jedan red sredine ispisuju linije 12-15 (tačka, `n-2` razmaka, tačka i `\n`), a taj red se ponavlja `n-2` puta. Radi lakšeg čitanja koda ostavili smo prazne redove u linijama 6, 10, 17 i 21.

✓ Rješenje zadatka pomoću koordinatnog sistema

Postoji i drugačiji pristup rješavanju zadataka sa crtanjem oblika na ekranu. Možemo na prostor za ispis postaviti koordinatni sistem (slika 4.8). Ishodište koordinatnog sistema je u gornjem lijevom uglu,



Slika 4.8: Crtanje oblika pomoću koordinatnog sistema

jer ispis kreće slijeva nadesno, odozgo prema dolje. Vodoravna koordinata je `j`, a uspravna je `i`. Sada zadatak postaje da kreiramo logičku (Booleovu) funkciju $f(i,j)$ koja ima vrijednost *istina* ako treba na koordinatama (i,j) ispisati tačku, a *neistina* ako ne treba tj. ako treba ispisati razmak. Ne smijemo zaboraviti `\n` na kraju svakog reda.

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    if (f(i,j))
      printf(".");
    else
      printf(" ");
  }
  printf("\n");
}
```

Pošto još uvijek nismo radili funkcije, logički uslov za ispis tačke ćemo uglaviti u kod umjesto `f(i,j)`. Vidimo da se tačka treba ispisati ako je `i` nula (gornja stranica) ili ako je `i=n-1` (donja stranica) ili `j=0` (lijeva stranica) ili `j=n-1` (desna stranica). Prevedeno u C, to nam daje sljedeći kompletan program:

```
1 #include <stdio.h>
2 int main() {
3   int i,j,n;
4   printf("Unesite broj n: ");
5   scanf("%d", &n);
6
7   for (i=0; i<n; i++) {
```

```
    for (j=0; j<n; j++) {
      if (i==0 || i==n-1 || j==0 || j==n-1)
        printf(".");
      else
        printf(" ");
    }
    printf("\n");
  }

  return 0;
}
```

Zadatak 8. *Napisati program koji omogućuje unos pozitivnog cijelog broja n>4 (zabraniti unos brojeva manjih ili jednakih broju 4), a zatim na ekranu ispisuje veliko štampano slovo N sastavljeno od znakova * čije je širina i visina n znakova. Primjeri za n=5 i n=6:*

```
*   *                          *    *
**  *                          **   *
* * *                          * *  *
*  **                          *  * *
*   *                          *   **
                               *    *
```

Kao u prethodnom primjeru sa koordinatnim sistemom, uočavamo da se znak zvjezdica ispisuje ako je $j = 0$, $j = n - 1$ ili $j = i$ (dijagonala). Zabranu unosa broja $n \leqslant 4$ postižemo do-while petljom.

```
#include <stdio.h>
int main() {
  int i,j,n;
  do {
    printf("Unesite broj n: ");
    scanf("%d", &n);
  } while (n<=4);
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      if (j==0 || j==n-1 || j==i)
        printf("*");
      else
        printf(" ");
    }
    printf("\n");
  }
  return 0;
}
```

4.4.2 *Sume*

1/5/0198

Sljedeći čest tip zadataka sa petljama su programi koji trebaju izračunati neku *matematičku sumu*. Pa pogledajmo na primjeru kako se rješavaju ovakvi zadaci.

Zadatak 9. *Napisati program koji izračunava* arctanh x *– hiperbolički arkus tangens nekog broja x – koristeći sljedeću aproksimativnu formulu:*

$$\operatorname{arctanh} x = x + \frac{x^3}{3} + \frac{x^5}{5} + \dots \tag{4.1}$$

Računari ne mogu predstaviti brojeve sa beskonačnom preciznosti. Zbog toga se iracionalne vrijednosti (kakve su u pravilu rezultati trigonometrijskih funkcija) izračunavaju tako što se pronađe neki red koji konvergira ka željenoj vrijednosti, a zatim se izračunava njegova konačna suma.[3] Broj n članova sume određuje se tako da krajnji rezultat bude dovoljno tačan za potrebe zadatka.

[3] Treba spomenuti da ovaj red konvergira samo za $|x| < 1$.

Dakle, ako je zadatak definisan kao iznad, korisnik ustvari treba unijeti dvije vrijednosti: broj `x` koji je u pravilu realan broj, te prirodan broj `n` koji predstavlja broj članova sume.

Opšti princip rješavanja zadataka sa sumama je sljedeći: najprije moramo zapisati sumu u obliku:

$$\operatorname{arctanh} x \approx \sum_{i=1}^{n} \frac{x^{2i-1}}{2i-1} \tag{4.2}$$

Ovakav zapis nam je potreban zato što je njime izražen opšti "i-ti" član sume. Zatim trebamo taj opšti član zapisati koristeći C sintaksu:

```
clan = pow(x, 2*i-1) / (2*i-1)
```

Ovdje moramo obratiti pažnju na nekoliko stvari. Najprije, kada u matematici napišemo $2i - 1$, između 2 i i se nalazi operacija množenja koja je izostavljena. U C-u operator množenja je zvjezdica i ne možemo ga preskočiti, tj. ako napišemo `2i-1` u našem programu dobićemo sintaksnu grešku. Dalje, razlomak kao takav je operacija dijeljenja, ali moramo paziti da brojnik i nazivnik stavimo u zagrade, jer bismo u suprotnom dobili

$$\frac{x^{2i-1}}{2} i - 1 \tag{4.3}$$

Stepenovanje u C-u ne postoji kao operacija, nego trebamo koristiti funkciju `pow` definisanu u biblioteci `math.h`. Funkcija `pow` uvijek vraća vrijednost tipa **`double`**, čak i kad su oba operanda cjelobrojne vrijednosti, tako da ovdje neće doći do zaokruživanja. Ovo sve bi nam trebalo biti poznato od ranije.↑

↑ Poglavlje 3.1.6, str. 72.

Treći i posljednji korak je izračunavanje sume n ovakvih članova. To postižemo tako što kreiramo for petlju u kojoj za svako $i \in [1, n]$ izračunavamo i-ti član i dodajemo ga na promjenljivu `suma`:

```
for (i=1; i<=n; i++) {
  clan = pow(x, 2*i-1) / (2*i-1);
  suma += clan;
}
```

1 // 2 // 3 // 4 // 5 // 6 // 7 // 8 // 9 // 10 // 11 // 12 // 13 // 14 // 15 // 16 // 17 // 18

Kada pogledamo sumu datu u zadatku, možemo uočiti da se u njoj javljaju isključivo neparni brojevi. Ustvari, $2i - 1$ je samo matematički način da izrazimo činjenicu da je neki broj neparan. Naš program

možemo učiniti bržim i lakšim za čitanje tako što izračunavanje izraza `2*i-1` ne radimo u svakom prolazu kroz petlju:

```
for (i=1; i<2*n; i+=2) {
  clan = pow(x,i) / i;
  suma += clan;
}
```

U ovoj varijanti u svakom prolazu kroz petlju `i` se povećava za 2, pa pošto smo počeli od broja 1, `i` će poprimati neparne vrijednosti. Broj članova sume treba biti `n`, što smo osigurali uslovom `i<2*n`.

Dodatno ubrzanje programa možemo postići tako što ne koristimo funkciju `pow`. Naime, u svakom članu sume, brojnik možemo dobiti tako što prethodnu vrijednost brojnika pomnožimo sa x:

```
  clan = stepen / i;
  stepen *= x;
  suma += clan;
```

Potrebno je inicijalizirati promjenljivu `stepen`, pa iz postavke zadatka vidimo da je početna vrijednost brojnika x.

Kompletno rješenje zadatka glasi ovako:

```
1  #include <stdio.h>
2  int main() {
3    int i,n;
4    double suma=0, x, clan, stepen;
5    printf("Unesite broj x: ");
6    scanf("%lf", &x);
7    do {
8      printf("Unesite broj clanova sume: ");
9      scanf("%d", &n);
10   } while (n<=0);
11   stepen = x;
12   for (i=1; i<2*n; i+=2) {
13     clan = stepen / i;
14     stepen *= x;
15     suma += clan;
16   }
17   printf("Suma iznosi: %g\n", suma);
18   return 0;
19 }
```

Možemo uočiti da su promjenljive `suma`, `x` i `clan` tipa **`double`**, jer za velike vrijednosti `n` povećana preciznost može imati efekta na rezultat, te smo koristili format `%lf` za unos. Petljom do-while osigurali smo da je vrijednost `n` pozitivna pošto C nema tip podataka za prirodne brojeve. Na kraju koristimo format `%g` za ispis čime postižemo da će se ispisati onoliko decimala koliko bude potrebno.

17570198

Zadatak 10. *Napisati program koji izračunava broj π na k tačnih decimala, po sljedećoj aproksimativnoj formuli:*

$$\pi = 4 \cdot (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \dots) \tag{4.4}$$

Koliko članova sume n nam je potrebno da bismo imali k tačnih decimala? Uočavamo da je svaki sljedeći član sume date iznad manji od prethodnog. Dakle kako raste naše `i` dolazimo do članova koji su toliko mali da ne bi trebali imati efekta na prvih k decimala. Konkretno, kada član sume postane manji od 10^{-k} možemo prekinuti petlju.

Ovdje treba napomenuti da čak ni sa tipom **`double`** ne možemo predstaviti beskonačno precizne brojeve. Kada radimo sa brojevima manjim od 0.000001 greška zbog nepreciznosti predstavljanja realnih brojeva u memoriji postaje tolika da se lako može desiti da upadnemo u beskonačnu petlju. No, za potrebe rješavanja ovog zadatka, pretpostavićemo da je tip **`double`** precizniji nego što realno jeste.

Suma u skraćenom obliku glasi:

$$\pi \approx 4 \cdot \sum_{i=1}^{n} (-1)^i \frac{1}{2i-1} \tag{4.5}$$

Ovdje $(-1)^i$ je matematička notacija koja označava da se mijenja predznak razlomka ($(-1)^0 = 1$, $(-1)^1 = -1$, $(-1)^2 = 1$, $(-1)^3 = -1$ ...). Opšti član možemo izračunati direktno pomoću funkcije `pow`:

```
clan = pow(-1,i) / (2*i-1)
```

ali ovo bi bilo prilično neefikasno rješenje, jer bismo u svakom prolasku kroz petlju radili stepenovanje, što nije potrebno. Umjesto toga možemo pisati ovako:

```
clan = predznak / (2*i-1);
predznak = -predznak;
```

pri čemu ćemo početnu vrijednost promjenljive `predznak` staviti na 1. Dakle u svakom prolasku kroz petlju ova promjenljiva će mijenjati vrijednost u -1, pa 1, -1, 1... Ostaje još da kao u prethodnom zadatku uočimo da je $2i-1$ oznaka za neparan broj. Konačno rješenje glasi:

```
1  #include <stdio.h>
2  #include <math.h>
3  int main() {
4    int k, i=1, predznak=1;
5    double suma=0, clan, limit;
6    do {
7      printf("Unesite broj decimala: ");
8      scanf("%d", &k);
9    } while (k<=0 || k>7);
10   limit = pow(10, -k);
11   do {
12     clan = (double)predznak/i;
13     predznak = -predznak;
14     i += 2;
15     suma += clan;
16   } while (clan>=limit);
17   suma *= 4;
18   printf("Suma iznosi: %g\n", suma);
19   return 0;
20 }
```

Kada član sume postane manji od vrijednosti `limit` (a to je 10^{-k}), prekidamo petlju. Za centralni dio koda koristili smo do-while petlju jer nije unaprijed poznat broj članova sume, a besmisleno je da se petlja ne izvrši barem jednom. Moramo paziti da su `predznak` i `i` cijeli brojevi, pa u liniji 12 dodajemo `(double)` da bismo izbjegli odsijecanje.↑ Ne zaboravite da se izračunata suma na kraju treba pomnožiti sa 4.

↑ Poglavlje 3.1.8, str. 76.

Zadatak 11. *Izračunati vrijednost sume:*

$$S(x) = x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \tag{4.6}$$

Ova suma konvergira jer funkcija faktorijela raste brže od eksponencijalne funkcije. Faktorijel se definiše ovako:

✓ Faktorijel

$$k! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot k \tag{4.7}$$

Dakle, $k!$ možemo izračunati sljedećim C kodom:

```
fakt=1;
for (i=2; i<=k; i++)
  fakt *= i;
```

Postavljanje promjenljive `fakt` na 1 je bitno, jer je jedinica neutralna za množenje. Da smo pisali `fakt=0;` rezultat bi uvijek bio nula. Množenje broja sa 1 je nepotrebno pa petlja može početi od dvojke.

Iz postavke zadatka je vrlo lako odrediti opšti član $\frac{x^i}{i!}$ pa bi rješenje glasilo:

```
1  #include <stdio.h>
2  int main() {
3    int i,j,n,fakt;
4    double suma=0, x, clan;
5    printf("Unesite broj x: ");
6    scanf("%lf", &x);
7    do {
8      printf("Unesite broj clanova sume: ");
9      scanf("%d", &n);
10   } while (n<=0);
11   for (i=1; i<=n; i++) {
12     /* Faktorijel broja i (i!) */
13     fakt = 1;
14     for (j=2; j<=i; j++)
15       fakt *= j;
16     clan = pow(x,i) / fakt;
17     suma += clan;
18   }
19   printf("Suma iznosi: %g\n", suma);
20   return 0;
21 }
```

1/15/7/01198

Ovo rješenje je potpuno tačno, ali ga možemo uraditi znatno efikasnije. Pogledajmo sljedeće rješenje:

```
1  #include <stdio.h>
2  int main() {
3    int i,j,n;
```

```
4   double suma=0, x, clan=1;
5   printf("Unesite broj x: ");
6   scanf("%lf", &x);
7   do {
8     printf("Unesite broj clanova sume: ");
9     scanf("%d", &n);
10  } while (n<=0);
11  for (i=1; i<=n; i++) {
12    clan *= x/i;
13    suma += clan;
14  }
15  printf("Suma iznosi: %g\n", suma);
16  return 0;
17 }
```

Pogledajte u liniji 12 kako izračunavamo opšti član. Svaki član sume jednak je prethodnom članu pomnoženom sa x/i:

$$\frac{x^{i-1}}{(i-1)!} \cdot \frac{x}{i} = \frac{x^i}{i!} \tag{4.8}$$

Sada u petlji imamo jedno sabiranje, jedno množenje i jedno dijeljenje, dok smo ranije imali još i stepenovanje i petlju za računanje faktorijela. Početna vrijednost člana je 1 (linija 4) kako bi u prvom prolazu bio pomnožen sa $x/1$, tako da dobijamo prvi član sume x.

Na ovom programu možemo vidjeti primjer nečega što zovemo *iterativno razmišljanje* – umjesto da rješavamo čitav problem odjednom, pokušavamo zamisliti transformaciju koja se treba izvršiti u nekom *"i minus prvom"* koraku da bismo dobili *i-ti korak*. Ovakva rješenja su često vrlo efikasna i brza, te štede memoriju računara.

Prilikom rješavanja zadataka u kojima se pojavljuje faktorijel, obratite pažnju da funkcija faktorijela raste izuzetno brzo, pa već za vrijednosti oko 13! prekoračujemo opseg vrijednosti koje možemo predstaviti tipom `int`. Činjenica da se to neće desiti u posljednjem rješenju je još jedna od njegovih bitnih prednosti.

4.4.3 *Prosti brojevi, problemi tipa "za svako x"*

Prosti brojevi se često javljaju u zadacima iz programiranja, a pogotovo u raznim algoritmima iz oblasti kriptografije, jer je provjera da li je broj prost jedan od matematičkih problema koji je (za sada) nemoguće riješiti računarski bez korištenja petlje. Ako otkrijete rješenje bez korištenja petlje, u svakom slučaju se javite jer biste se mogli izuzetno obogatiti! Uobičajena definicija prostog broja kakvu nalazimo u školskim udžbenicima glasi:

✓ *Broj (veći od 1) je prost ako je djeljiv isključivo brojem 1 i samim sobom.* ✓ Prost broj

Pregledajući studentske radove redovno možemo vidjeti ovakva rješenja:

1/5/0198

```
if (n%1 == 0 && n%n == 0)
  print("Prost\n");
else
  print("Slozen\n");
```

Problem ovdje je što ste previdjeli riječ *isključivo* u tekstu definicije. Svaki broj je djeljiv sa 1 i sa samim sobom, ali prosti brojevi nisu djeljivi ni sa kojim drugim brojem, pa bi sa stanovišta programiranja bolja bila sljedeća definicija: *Broj je prost ako NIJE djeljiv niti sa jednim brojem između 1 i tog broja.*

Često se može vidjeti i ovakvo rješenje:

```
if (n%2 != 0 && n%3 != 0 && n%5 != 0 && n%7 != 0)
  print("Prost\n");
else
  print("Slozen\n");
```

Ponekad se uslovi nastavljaju, pa studenti znaju ispisati i više od pola stranice teksta nabrajajući brojeve sa kojima prost broj nije djeljiv. Studentu svaka čast na trudu, ali ovakvo rješenje ponovo nije tačno. To možemo vidjeti iz sljedećeg primjera: uzmimo sve brojeve koje je student nabrojao u svom uslovu i nađimo neki broj koji je veći od svih tih brojeva i koji je prost (u primjeru iznad to bi bilo 11). Nazovimo taj broj x. Broj x^2 je složen broj jer je djeljiv sa x, no on nije djeljiv niti sa jednim od brojeva navedenih u kôdu! Dakle, ako bi korisnik unio broj $n = 11^2 = 121$, kôd naveden iznad bi za taj broj ispisao da je prost što nije tačno.

Pa će zatim student probati popraviti svoj kod na sljedeći način:

```
if (n%2 != 0 && n%3 != 0 && n%5 != 0 && n%7 != 0 && n%(int)sqrt(n) != 0)
  print("Prost\n");
else
  print("Slozen\n");
```

Posljednji poduslov provjerava da li je broj djeljiv sa svojim kvadratnim korijenom, koristeći operator kalupa za zaokruživanje.↑ Ovo ponovo nije tačno, jer u našem primjeru iznad umjesto x^2 možemo uzeti x^3, pa x^5 itd. Ukratko: *nemoguće* je provjeriti da li je neki broj n prost a da ne provjerimo da li je n djeljivo sa svim brojevima na intervalu $(1, n)$, odnosno najmanje na intervalu $[2, \sqrt{n}]$.

↑ Poglavlje 3.1.8, str. 76.

Naoružani ovim znanjem, možemo probati da realizujemo program koji provjerava da li je broj prost:

```
#include <stdio.h>
int main() {
  int i,j,n;
  printf("Unesite broj n: ");
  scanf("%d", &n);

  for (i=2; i<n; i++) {
    if (n%i == 0)
      printf("Slozen\n");
    else
      printf("Prost\n");
```

```
  }

  return 0;
}
```

Na osnovu izgleda programa već znate da je i on pogrešan. Da bismo razumjeli zašto, pokrenimo ovaj program i unesimo npr. broj 8:

```
Unesite broj n: 8
Slozen
Prost
Slozen
Prost
Prost
Prost
```

Problem ovdje je što je program nastavio provjeravati djeljivost za brojeve 3, 4 itd., što nije potrebno, jer već kada ustanovimo da je 8 djeljivo sa 2, znamo da je taj broj složen i ne moramo provjeravati dalje.

Prepravićemo program tako da se on prekine čim ustanovimo da je broj složen:

```
  for (i=2; i<n; i++) {
    if (n%i == 0) {
      printf("Slozen\n");
      break;
    } else {
      printf("Prost\n");
    }
  }
```

Sada će za broj 8 program dati tačan rezultat:

```
Unesite broj n: 8
Slozen
```

No za broj 9 imamo:

```
Unesite broj n: 9
Prost
Slozen
```

Šta se ovdje desilo? Prođimo kroz program korak po korak. Najprije je `i` imalo vrijednost 2, pa smo provjeravali da li je 9 djeljivo sa 2. Pošto nije, uslov nije ispunjen, pa se na ekranu ispisalo `Prost`. Vidimo da ovo nije ispravno jer je 9 složen broj, pa se tekst `Prost` ne treba ispisivati u petlji za svaku vrijednost `i` sa kojom `n` nije djeljivo, nego samo na kraju petlje. Možemo ovako prepraviti program:

```
  for (i=2; i<n; i++) {
    if (n%i == 0) {
      printf("Slozen\n");
      break;
    }
  }
  printf("Prost\n");
```

No sada za broj 9 imamo:

```
Unesite broj n: 9
Slozen
Prost
```

jer se tekst `Prost` ispisuje za svako uneseno `n`, bilo ono prost ili složen broj. Želimo da se tekst `Prost` ispiše samo ako broj `n` nije djeljiv niti sa jednim `i`, odnosno ako se **`break`** nikada nije izvršilo. Kako to možemo znati? Pa podsjetimo se poglavlja o naredbi **`break`**↑ gdje smo rekli da vrijednost kontrolne promjenljive `i` poslije petlje može biti korisna da znamo da se petlja završila bez prekida. Time dolazimo do kompletnog tačnog rješenja zadatka:

↑ Poglavlje 4.1.6, str. 97.

```
1  #include <stdio.h>
2  int main() {
3    int i,j,n;
4    printf("Unesite broj n: ");
5    scanf("%d", &n);
6
7    for (i=2; i<n; i++) {
8      if (n % i == 0) {
9        printf("Slozen\n");
10       break;
11     }
12   }
13   if (i == n)
14     printf("Prost\n");
15
16   return 0;
17 }
```

Provjera da li je broj prost spada u klasu problema *"za svako x"* – u matematici se ovo označava sa $\forall x$. Prirodan broj n je prost ako i samo ako za svako x na intervalu (1,n) važi da n nije djeljivo sa x. Matematički to možemo zapisati ovako:

✓ Problemi tipa "za svako x"

$$\text{prost}(n) \iff \forall x \in (1, n), x \in \mathbb{N}, x \nmid n \tag{4.9}$$

Zadatke tipa "za svako x" rješavamo na sljedeći način:

1. Pretpostavimo da uslov važi za svako x.
2. Koristeći for petlju prolazimo kroz sve moguće vrijednosti x.
3. Ako nađemo takvo x da uslov ne važi, zaključujemo da uslov zadatka nije ispunjen i prekidamo petlju.

Prvi korak se obično realizuje koristeći *logičku promjenljivu*. U programskom jeziku C ne postoji logički (boolean) tip podataka, nego umjesto njega koristimo **`int`** pri čemu 1 označava istinu a 0 neistinu.↑

↑ Poglavlje 2.2, str. 46.

Dakle, program koji provjerava da li je neki broj `n` prost možemo napisati i ovako:

```
7    prost = 1; /* logicka promjenljiva */
8    for (i=2; i<n; i++)
```

175/0198

```
    if (n % i == 0) {
      prost = 0;
      break;
    }
  }
  if (prost)
    printf("Prost\n");
  else
    printf("Slozen\n");
```

Ovo rješenje je bolje od prethodnog jer ga lakše možemo iskoristiti u nekom većem programu, npr. ako se traži da prebrojimo sve proste brojeve na nekom intervalu. No obratite pažnju da broj 1 nije niti prost niti složen, pa bi se trebao dodati poseban uslov za to (ovisno šta se tačno tražilo u zadatku).

✓ Problemi tipa "postoji barem jedan"

Također često imamo i zadatke tipa *"postoji barem jedan x"* – a matematička oznaka je $\exists x$. Recimo provjera da li je broj složen spada u ovaj tip zadatka. Prirodan broj n je složen ako i samo ako na intervalu (1,n) postoji barem jedno x takvo da je n djeljivo s njim. Matematički to možemo zapisati

$$\text{slozen}(n) \iff \exists x \in (1, n), x \in \mathbb{N}, x \mid n \tag{4.10}$$

Zadatak tipa $\exists x$ je invertovana verzija zadatka $\forall x$, odnosno za brojeve veće od 1 važi da je broj složen ako nije prost i obrnuto. Drugim riječima, ako za svako x važi suprotan uslov od traženog, to znači da nije tačno da postoji barem jedan x za koji važi traženi uslov. Algoritam je u biti isti samo se mijenja značenje logičke istine i neistine (0 i 1):

1. Pretpostavimo da ne postoji nijedno x koje zadovoljava uslov.
2. Koristeći for petlju prolazimo kroz sve moguće vrijednosti x.
3. Ako nađemo takvo x za koje uslov važi, zaključujemo da je uslov zadatka ispunjen i prekidamo petlju.

Ukratko možemo reći i ovako: ako smo pretpostavili da neka tvrdnja važi, uvijek provjeravamo suprotan uslov, onaj koji ruši tvrdnju.

4.4.4 *Razdvajanje broja na cifre*

Zadatak 12. *Napisati program koji izračunava sumu cifara cijelog broja.*

```
Unesite broj: 28147
Suma njegovih cifara je: 22
```

Nekoliko napomena vezanih za ovaj zadatak: Ovdje ćemo raditi isključivo sa tipom `int`, pošto nepreciznost realnih tipova↑ pravi probleme sa primjenom algoritma koji ćemo opisati u nastavku. Neki početnici pokušavaju ovaj zadatak riješiti tako što koriste nizove ili

↑ Poglavlje 3.1.6, str. 72.

1/15/2011 98

stringove, pa ako u primjeru iznad umjesto cijelog broja unosimo string dobijamo neki sasvim različit problem, no nije teško promijeniti tekst zadatka tako da rješenje sa stringom postaje izuzetno komplikovano.

Da bismo riješili zadatak, moramo napraviti petlju koja prolazi kroz uneseni broj cifru po cifru. Da bismo dobili posljednu cifru broja možemo koristiti ostatak pri dijeljenju (modulo) sa 10. S druge strane, ako radimo cjelobrojno dijeljenje sa 10, doći će do odsijecanja↑ svih cifara osim posljednje. Ove dvije operacije možemo postaviti u petlju. Npr. ako je broj `x=28147`, imaćemo:

↑ Poglavlje 3.1.8, str. 76.

- `c=x%10 -> 7`, `x=x/10 -> 2814`
- `c=x%10 -> 4`, `x=x/10 -> 281`
- `c=x%10 -> 1`, `x=x/10 -> 28`
- `c=x%10 -> 8`, `x=x/10 -> 2`
- `c=x%10 -> 2`, `x=x/10 -> 0`

Promjenljiva `c` je redom poprimala vrijednosti cifara broja 28147 od manje vrijednih ka većim, tj. zdesna nalijevo. Petlja se završava kada `x` postane nula. Dakle rješenje zadatka glasi:

```
1  #include <stdio.h>
2  int main() {
3    int x, c, suma=0;
4    printf("Unesite broj: ");
5    scanf("%d", &x);
6
7    while (x != 0) {
8      c = x % 10;
9      suma += c;
10     x = x / 10;
11   }
12   printf ("Suma njegovih cifara je: %d\n", suma);
13
14   return 0;
15 }
```

Obratimo pažnju na par stvari. Za početnu vrijednost promjenljive `suma` smo uzeli 0. Uslov petlje glasi `x!=0` a ne `x>0`, jer bi ova druga varijanta radila netačno ako korisnik unese negativan broj. No u tom slučaju će i `c` biti negativno, pa bismo u liniji 8 trebali uzeti:

```
8      c = abs(x % 10);
```

te uključiti potrebnu biblioteku `stdlib.h`. U ovom konkretnom zadatku, ako korisnik unese `x=0` petlja se neće nijednom izvršiti i rješenje će biti tačno. Međutim u nekim zadacima (npr. gdje je potrebno prebrojati cifre), petlja se treba izvršiti jednom i kada je unesena nula, pa u tom slučaju trebamo koristiti petlju do-while.

1/45/0198

Zadatak 13. *Napisati program koji okreće cijeli broj naopako.*

Za ovaj zadatak ćemo dati rješenje, pa ćemo ga zatim prodiskutovati:

```
1  #include <stdio.h>
2  int main() {
3    int x, c, naopako=0;
4    printf("Unesite broj: ");
5    scanf("%d", &x);
6
7    while (x != 0) {
8      c = abs(x % 10);
9      naopako = naopako * 10 + c;
10     x = x / 10;
11   }
12   printf ("Broj okrenut naopako glasi: %d\n", naopako);
13
14   return 0;
15 }
```

U liniji 9 smo u svakom prolazu staru vrijednost promjenljive `naopako` množili sa 10, a zatim dodavali `c`. U prvom prolazu biće $0 \cdot 10 = 0$, pa će `naopako` imati vrijednost posljednje cifre. U sljedećem prolazu se ta posljednja cifra množi sa 10 i dodaje se pretposljednja cifra itd. Recimo ako je broj glasio 28147 kao u prethodnom primjeru, biće $7 \cdot 10 + 4 = 74$. U trećem prolazu ćemo imati $74 \cdot 10 + 1 = 741$ itd. Na kraju ćemo imati 74182. Vidimo kako se množenjem sa 10 ustvari dodaje nula, koju ćemo zatim zamijeniti cifrom `c`.

Obratite pažnju da ako je korisnik unio broj koji se završava nulama, npr. 83000, rezultat će biti 38 a ne 00038 jer se početne nule ne mogu predstaviti promjenljivom tipa `int`. Također možemo primijetiti da ako se unese negativan broj, rezultat će biti pozitivan zbog poziva funkcije `abs`. Jedan način da ovo riješimo je da u nekoj pomoćnoj promjenljivoj sačuvamo informaciju da li je uneseni broj bio negativan ili pozitivan.

Pitanja za razumijevanje gradiva

1. Šta je to "petlja" u programiranju?

2. Kojim redoslijedom se izvršavaju elementi for petlje?

3. Šta je to beskonačna petlja? Navedite primjer.

4. Navedite primjer for petlje koja na ekranu ispisuje sve brojeve između 100 i 1000 (uključivo) koji su djeljivi sa 3. Kakvu vrijednost će imati kontrolna promjenljiva po završetku te petlje? Navedite petlju koja ispisuje iste brojeve u opadajućem redoslijedu.

5. Čemu služi naredba `break` a čemu `continue`? Navedite primjere korištenja.

1755/0198

6. Ako se u tijelu petlje koja se izvršava 3 puta nalazi petlja koja se izvršava 8 puta, koliko će se ukupno puta izvršiti tijelo unutrašnje petlje?

7. Koja je razlika između for i while petlje, kada se koristi koja?

8. Koja je razlika između while i do-while petlje?

9. Kako možemo transformisati for u while petlju i obrnuto?

Zadaci za vježbu

Zadatak 14. *Napisati program koji će ispisati brojeve koji su djeljivi sa 5, 7 ili 11, a koji su manji ili jednaki od nekog broja n koji je učitan sa tastature. Brojeve treba ispisati od najvećeg prema najmanjem. Obavezno koristiti for petlju i if-else strukturu.*

Zadatak 15. *Napravite program "Snijeg" koji, za uneseni broj n, na ekranu iscrtava uzorak od n×n zvjezdica koja su naizmjenično razmaknute sa po jednim razmakom. Primjer ulaza i izlaza:*

```
Unesite broj n: 5
* * * * *
 * * * * *
* * * * *
 * * * * *
* * * * *
```

Zadatak 16. *Napisati program koji će izračunavati vrijednost konačnog reda:*

$$S = \sum_{i=1}^{n} \frac{(-1)^{i-1}}{i} \tag{4.11}$$

Broj n se unosi sa tastature, a rezultat sume treba ispisati na ekranu zaokružen na tri decimale.

Zadatak 17. *Napisati program koji određuje i na ekranu ispisuje najveću i najmanju cifru broja. Savjet: Za rješavanje ovog zadatka, vrlo je korisno da najprije pročitate poglavlje o određivanju najvećeg broja u nizu (5.2.1).*

Zadatak 18. *Napisati program koji sve parne cifre u broju zamjenjuje cifrom* 0*. Pošto cjelobrojni tipovi u C-u ne mogu predstaviti brojeve koji počinju nulama, sve parne cifre koje se nalaze s lijeve strane broja trebaju biti obrisane. Ako su sve cifre broja parne, rezultat treba biti broj* 0*.*

Zadatak 19. *Igra pogađanja brojeva. Računar memoriše neki slučajno izabrani broj između 1 i 100, a zatim korisnik pogađa taj broj. Za svaki*

1/5/0198

pokušaj, računar treba ispisati VECI ako je memorisani broj veći od unesenog, MANJI ili JEDNAK. Kada je broj jednak, igra se završava. Igrač može izaći iz igre prije kraja unosom broja -1.

1757018

5
Nizovi (polja)

Često se dešava da nam je u nekom programu koji razvijamo potreban vrlo veliki broj promjenljivih. Recimo da je zadatak unijeti 100 brojeva, te ih ispisati na ekran u obrnutom redoslijedu. Jedini potpuno ispravan način za rješavanje ovog problema je da deklarišemo 100 promjenljivih! Mogli bismo to pokušati ovako rješavati:

```
#include <stdio.h>
int main() {
  int a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,aa,ab,...;
  printf("Unesite prvi broj: ");
  scanf("%d", &a);
  printf("Unesite drugi broj: ");
  scanf("%d", &b);
  printf("Unesite treci broj: ");
  scanf("%d", &c);
  printf("Unesite cetvrti broj: ");
  scanf("%d", &d);
  ...

  return 0;
}
```

Ovo je naravno suludo. Ovakav program bi bio ogroman i krajnje nepraktičan za rad i održavanje.

Da se ne bismo mučili na ovaj način, programski jezik C – kao i većina drugih jezika – omogućuje nam da *odjednom deklarišemo 100 promjenljivih*. Dovoljno je da napišemo:

```
int niz[100];
```

i deklarisali smo 100 cjelobrojnih promjenljivih! Naredba koju smo naveli opisuje programsku strukturu koja se naziva **niz** ili **polje** (eng. array).

✓ *Niz je posebna vrsta promjenljive koja u sebi sadrži određeni broj promjenljivih istog tipa označenih rednim brojevima (indeksima).* ✓ Niz

17570198

Deklaracija niza ima sljedeći oblik:

```
TIP ime[duzina];
```

Najprije navodimo osnovni `TIP`. Svi članovi niza će biti tog (istog) tipa. Zatim navodimo ime niza koje zadovoljava iste uslove kao ime bilo koje druge promjenljive.↑ Konačno, u uglastim zagradama dajemo veličinu niza odnosno broj članova niza. Veličina niza mora biti konstanta ili literal (vrijednost koju navodimo direktno u kodu kao u primjeru `int niz[100];`). Navođenje promjenljive za veličinu niza je dozvoljeno u standardu C99 i novijim, ali nije podržano standardom C90, kao ni u C++ programskom jeziku (mada će raditi na nekim prevodiocima), pa nije preporučljivo koristiti ovaj način deklarisanja niza.

↑ Poglavlje 1.2.3, str. 29.

Nakon što smo naredbom `int niz[100];` deklarisali 100 promjenljivih tipa `int`, kako sada pristupamo tim promjenljivim? Njihova imena su:

```
niz[0], niz[1], niz[2], ..., niz[99]
```

Za razliku od nekih drugih jezika, u programskom jeziku C najniži indeks u nizu je nula.[1] Pošto elemenata ima ukupno 100, posljednji element ima indeks 99, a `niz[100]` nije član niza.

[1] Razlozi za ovo će biti jasni kada budemo radili sa nizovima preko pokazivača (poglavlje 7.3.2, str. 240).

Ovi nazivi promjenljivih mogu stajati bilo gdje u kodu gdje smo mogli postaviti promjenljivu tog tipa. Za provjeru razumijevanja probajte odgovoriti na sljedećih par pitanja. Pretpostavimo da je deklarisan niz naredbom `double x[10]`.

- Kojom naredbom ćemo postaviti vrijednost petog člana niza na 3.4?
- Kojom naredbom ćemo ispisati osmi član niza na ekran?
- Kojom naredbom ćemo omogućiti korisniku da pomoću tastature unese drugi član niza?

5.1 *Rad sa nizovima*

Rekli smo da dužina niza mora biti konstanta ili literal, no prilikom pristupa nekom članu niza kao indeks možemo koristiti promjenljivu, pa čak i izraz, pod uslovom da je ta promjenljiva/izraz cjelobrojnog tipa.[2] Pogledajte sljedeći primjer programa.

[2] Ustvari tipa `ptrdiff_t` što je objašnjeno u prilogu C.1.3 (str. 361).

```
1  #include <stdio.h>
2  int main() {
3    double a[20];
4    int n;
5    printf("Unesite indeks u nizu: ");
6    scanf("%d", &n);
7    printf("Unesite %d-ti clan niza: ", n+1);
8    scanf("%lf", &a[n]);
9    ...
10
11   return 0;
12 }
```

Najprije smo deklarisali niz pod imenom `a`, tipa `double`, sa 20 elemenata. Zatim smo deklarisali i jednu promjenljivu `n` tipa `int`. Korisnik

najprije unosi vriijednost `n` a zatim unosi član niza na indeksu `n`. U liniji 7 ispisujemo da korisnik unosi (`n+1`)-vi član niza iz razloga što indeksi nizova u C-u kreću od nule, a u našem govornom jeziku kažemo da je to prvi član niza, pa treba sve pomaći za jedan.

Ovaj program će raditi sasvim ispravno ako kao `n` unosimo brojeve [0,19]. No šta će se desiti ako korisnik unese broj van tog opsega? Time smo pristupili memorijskoj lokaciji koja ne pripada našem nizu, što je greška! Pošto dobro dizajniran program ne smije pristupati ilegalnim memorijskim lokacijama, potrebno je da prepravimo naš program tako da korisniku ne bude dozvoljeno da unosi nelegalne vrijednosti za `n`, na sljedeći način:

```
1  #include <stdio.h>
2  int main() {
3    double a[20];
4    int n;
5    do {
6      printf("Unesite indeks u nizu (0-19): ");
7      scanf("%d", &n);
8    } while (n<0 || n>19);
9
10   printf("Unesite %d-ti clan niza: ", n+1);
11   scanf("%lf", &a[n]);
12   ...
13
14   return 0;
15 }
```

Inače, interesantno je da izlazak van opsega niza često rezultira programom koji radi, pa čak i daje tačne rezultate, ali u određenim situacijama krahira. Recimo, u Code::Blocks okruženju na Windows XP operativnom sistemu sljedeći program će doći do vrlo velikog broja prije nego što se konačno krahira:

✓ Izlazak iz opsega niza

```
#include <stdio.h>
int main() {
  int i, niz[100];
  for (i=0; ; i++) {
    niz[i] = i;
    printf("%d\n", i);
  }
  return 0;
}
```

Ovdje imamo jednu beskonačnu petlju u kojoj `i` neprekidno raste za 1 tako da redom pristupamo svakom elementu niza. Na našem računaru program se krahira tek kada `i` dostigne vrijednost nekoliko miliona.

✓ *Program koji (često) radi nije nužno ispravan.*

✓ Ispravnost programa

17570198

Zašto se ovo dešava? Kada deklarišemo niz od 100 elemenata mi smo u memoriji rezervisali prostor za 100 promjenljivih tipa `int`. Ako zatim probamo pristupiti 101. elementu, mi ćemo ustvari pristupiti memorijskoj lokaciji koja se nalazi neposredno nakon kraja niza.[3] Ova

[3] Koristeći *relativno adresiranje*, što će biti objašnjeno u poglavlju 7.3.2 (str. 240).

Slika 5.1: Izlazak iz opsega niza lako može dovesti do ovakvih situacija

lokacija može biti slobodna ili može pripadati nekoj drugoj promjenljivoj u našem programu, a u tim situacijama najčešće možemo normalno koristiti tu lokaciju iako ona ne pripada nizu. Ako je pak ovu lokaciju rezervisao neki drugi program, operativni sistem će spriječiti pristup toj memorijskoj lokaciji iz sigurnosnih razloga i prisilno prekinuti naš program, što nazivamo "krahiranje programa". Ovo čak možemo smatrati sretnijom varijantom, jer program koji se ne krahira ali podmuklo mijenja vrijednosti drugih promjenljivih je mnogo opasniji.

✓ Krahiranje

Ukratko, programski jezik C vas neće zaštititi od izlaska iz opsega niza. Morate jako dobro paziti da ne napravite takvu grešku i u svakom trenutku znati šta se u programu dešava, u suprotnom se možete naći u situacijama kao na slici 5.1.

Još neke razlike između nizova u C-u i sličnih struktura u nekim drugim programskim jezicima:

- Veličina deklarisanog niza se ne može mijenjati za vrijeme rada programa, ali je moguća jedna simulacija tog efekta koju ćemo opisati u poglavlju 5.1.5.
- Jedan niz ne možete dodijeliti drugom. Ako želite da kopirate sve elemente jednog niza u drugi, morate koristiti for petlju.[4]
- Nije moguće ispisati na ekran sve članove niza jednom naredbom.

[4] Postoji i bibliotečna funkcija `memcpy` koja obavlja taj zadatak.

17570198

5.1.1 *Unos niza pomoću tastature*

Pošto je moguće koristiti cjelobrojnu promjenljvu za pristup elementu niza, to nam ukazuje da je za rad sa nizovima izuzetno pogodna for petlja. Pogledajmo sljedeći program za unos niza:

```
1  #include <stdio.h>
2  #define BROJ_EL 10
3  int main() {
4    int niz[BROJ_EL], suma, i;
5
6    for (i=?; i<?; i??) {
7      printf("Unesite %d-ti clan niza: ", i+1);
8      scanf("%d", &niz[i]);
9    }
10   ...
```

Prije svega, u liniji 2 definisali smo konstantu `BROJ_EL` da bi nam kasnije bilo lakše mijenjati dimenzije niza. Ako u zadatku treba raditi sa nizom od 100 elemenata, često se pokaže kao vrlo zamorno da svaki put kada testiramo program unosimo 100 brojeva putem tastature. Zato ćemo privremeno postaviti dužinu niza na 10, pa kada osiguramo da program radi ispravno za 10 elemenata, lako je promijeniti dužinu na 100, jer je dužina dana na samo jednom mjestu u programu.

Vidimo da smo tu konstantu iskoristili i u deklaraciji niza (linija 4).

Sada na osnovu ranijih primjera ne bi trebalo biti teško pretpostaviti šta u zaglavlju for petlje treba staviti umjesto upitnika. Početna vrijednost indeksa u nizu je 0, krajnja je `BROJ_EL-1` pa ćemo staviti da je `i<BROJ_EL`, a u svakom koraku `i` se povećava za 1. U pozivu funkcije `printf` koristimo vrijednost `i+1` iz razloga koje smo već objasnili.

Dakle program za unos niza treba glasiti ovako:

```
#include <stdio.h>
#define BROJ_EL 10
int main() {
  int niz[BROJ_EL], suma, i;

  for (i=0; i<BROJ_EL; i++) {
    printf("Unesite %d-ti clan niza: ", i+1);
    scanf("%d", &niz[i]);
  }

  ...

  return 0;
}
```

Mnogi početnici u programiranju se pitaju da li postoji neki lakši način unosa, može li se konstruisati neki format u `scanf` funkciji tako da unesemo čitav niz odjednom.[5]

[5] Npr. ovo je moguće u programskom jeziku FORTRAN.

```
scanf("%???", niz);
```

Odgovor je: ne, u C-u nije moguće unijeti niz bez petlje. Praktično ništa korisno se ne može raditi sa nizom bez korištenja petlje. Na ovom primjeru vidite kako svaka lekcija ustvari zahtijeva poznavanje svih prethodnih lekcija, pa nas učenje na preskok neće nigdje dovesti.

1/15/01/98

5.1.2 *Inicijalizacija članova niza*

Kao i sve ostale promjenljive, članovi niza nakon deklaracije imaju neku nepoznatu vrijednost koja se zatekla na odgovarajućoj memorijskoj lokaciji, odnosno kažemo da su neinicijalizirani.↑ Potrebno ih je inicijalizirati, odnosno postaviti sve članove niza na neku početnu vrijednost, recimo 0.

↑ Poglavlje 1.2.2, str. 29.

Inicijalizaciju možemo uraditi koristeći for petlju:

```
for (i=0; i<BROJ_EL; i++) {
  niz[i]=0;
```

no programski jezik C nudi nam lakši način da postavimo početne vrijednosti članova niza. Recimo možemo kucati ovako:

✓ Inicijalizacijska lista

```
double a[5] = { 0.1, 0.2, 0.3, 0.4, 0.5 };
```

Sintaksa je sljedeća: nakon deklaracije niza navodi se znak jednakosti (=), zatim se u vitičastim zagradama nabrajaju inicijalne vrijednosti članova niza razdvojene znakom zarez.

Obratite pažnju da je ovaj tip inicijalizacije dostupan samo na mjestu gdje deklarišemo niz. Nije dozvoljeno pisati recimo ovako:

```
double a[5];
a = { 0.1, 0.2, 0.3, 0.4, 0.5 };
```

Jedna jako zgodna osobina ove inicijalizacije niza jeste da ne moramo navesti onoliko vrijednosti kolika je veličina niza. Ako navedemo manji broj vrijednosti, ostali članovi niza se inicijaliziraju na nulu. Ako želimo postaviti sve članove niza na nulu, dovoljno je navesti početnu vrijednost prvog elementa:

```
int niz[100] = {0};
```

No obratite pažnju da sljedeća naredba neće kreirati niz petica:

```
int niz[100] = {5};
```

Samo prvi član niza će imati vrijednost 5, a ostali će biti nule! U programskom jeziku C++ ne moramo navesti ni taj prvi element:

```
int niz[100] = {};
```

ali u C-u to nije dozvoljeno i ovo nije validan C kod.

Kada koristimo ovakvu inicijalizaciju (ali samo tada!), možemo deklarisati niz bez navođenja dimenzije:

```
double a[] = { 0.1, 0.2, 0.3, 0.4, 0.5 };
```

U tom slučaju dimenzija se određuje na osnovu broja elemenata u inicijalizaciji. Dakle, niz `a` u primjeru iznad će imati veličinu 5, jer smo naveli pet elemenata u vitičastim zagradama.

1/15/0198

5.1.3 *Neke matematičke operacije nad nizom*

Vratimo se na primjer sa početka poglavlja. Program koji smo do sada napisali još uvijek ne radi ništa, samo smo unijeli članove niza i

zatim smo stavili tri tačke kao ono što slijedi dalje. Sada da vidimo kako ćemo izračunati sumu svih članova niza.

✓ Suma članova niza

Da bismo sumirali članove niza, najprije trebamo deklarisati neku promjenljivu koja će se zvati `suma` i koju ćemo postaviti na nulu iz razloga što je nula vrijednost neutralna za operaciju sabiranja. Zatim ćemo napraviti petlju koja uzima jedan po jedan član niza i uvećava promjenljivu `suma` za vrijednost tog člana niza. Na kraju ćemo ispisati sumu.

Programski kod koji obavlja ovu operaciju je sljedeći:

```
1  #include <stdio.h>
2  #define BROJ_EL 10
3  int main() {
4    int niz[BROJ_EL], suma, i;
5
6    for (i=0; i<BROJ_EL; i++) {
7      printf("Unesite %d-ti clan niza: ", i+1);
8      scanf("%d", &niz[i]);
9    }
10
11   suma = 0;
12   for (i=0; i<BROJ_EL; i++)
13     suma += niz[i];
14
15   printf("Suma clanova niza je: %d\n", suma);
16   return 0;
17 }
```

Prvi dio programa (do linije 10) je potpuno identičan kao i ranije i nije ga potrebno objašnjavati. Zatim u liniji 12 možemo uočiti for petlju koja kontrolnoj promjenljivoj `i` daje vrijednosti od 0 do `BROJ_EL-1`. Ova petlja nam omogućuje da pristupimo svim članovima niza, te se svaki član niza dodaje na sumu.

Ovaj program se može drastično skratiti. Računanje sume može se odvijati u istoj petlji koja služi za unos niza. Ustvari, ako je ovo sve što program treba raditi, nema nikakve potrebe da koristimo niz! Svaka unesena vrijednost će nam poslužiti samo da bismo je dodali na sumu, pa možemo koristiti običnu promjenljivu:

```
#include <stdio.h>
#define BROJ_EL 10
int main() {
  int x, suma=0, i;
  for (i=0; i<BROJ_EL; i++) {
    printf("Unesite %d-ti clan niza: ", i+1);
    scanf("%d", &x);
    suma += x;
  }

  printf("Suma clanova niza je: %d\n", suma);
  return 0;
}
```

Ovo je program vrlo sličan programu koji smo vidjeli kada smo se upoznavali sa while petljom.↑ Ali, radi vježbe za sada ćemo se držati

↑ Poglavlje 4.2, str. 105.

rješenja u kojem se koristi niz i u kojem je svaka operacija u zasebnoj petlji.

Da bismo izračunali srednju vrijednost (prosjek) članova niza, dovoljno je da ovako izračunatu sumu podijelimo sa brojem elemenata:

✓ Srednja vrijednost (prosjek) članova niza

```
prosjek = (double) suma / BROJ_EL;
```

Obratite pažnju da je suma deklarisana kao promjenljiva tipa `int`, a `BROJ_EL` je cjelobrojna konstanta, pa da bismo izbjegli zaokruživanje moramo izvršiti konverziju tipa koristeći operator konverzije tj. kalup.↑

↑ Poglavlje 3.1.8, str. 76.

Još jedan zadatak koji je često potrebno obaviti nad članovima niza jeste prebrojati članove niza koji zadovoljavaju neki uslov. Recimo da se u zadatku traži da prebrojimo koliko niz ima članova koji su parni brojevi. Ovakav problem se rješava na sljedeći način: najprije deklarišemo jednu cjelobrojnu promjenljivu u kojoj ćemo držati broj članova koji zadovoljavaju uslov iz zadatka. Ovakva promjenljiva se naziva **brojač** (eng. counter). Početna vrijednost brojača treba biti 0 (niti jedan član niza ne zadovoljava uslov). Zatim ulazimo u petlju u kojoj prolazimo kroz članove niza i provjeravamo da li član niza zadovoljava uslov. Ako član niza zadovoljava uslov uvećavamo brojač za jedan.

✓ Prebrojavanje članova niza koji zadovoljavaju neki uslov

Dakle, dio programa koji određuje koliko niz ima članova koji su parni brojevi izgledao bi ovako:

```
br_parnih = 0;
for (i=0; i<BROJ_EL; i++)
  if (niz[i] % 2 == 0)
    br_parnih++;
```

Nakon ove petlje, promjenljiva `br_parnih` imaće vrijednost broja članova niza koji su parni brojevi.

Obratite pažnju da nas interesuju *članovi niza* koji su parni brojevi. Program poput ovog ne bi bio ispravan:

```
for (i=0; i<BROJ_EL; i++)
  if (i % 2 == 0)
    br_parnih++;
```

jer se ovdje provjerava da li je indeks u nizu paran broj, a ne član niza. Rezultat ovakvog koda bi bila promjenljiva `br_parnih` koja bi uvijek imala vrijednost polovice broja `BROJ_EL`.

Zadatak 1. *Napisati program koji omogućuje korisniku da unese niz od 100 elemenata, a zatim izračunava srednju vrijednost članova niza koji su parni brojevi.*

1/5770198

Srednja vrijednost članova niza koji su parni brojevi je suma članova niza koji su parni brojevi podijeljena sa brojem takvih članova niza. Dakle, sada možemo povezati dijelove koda koje smo ranije razvili u program koji rješava postavljeni zadatak.

```
1  #include <stdio.h>
2  #define BROJ_EL 10
3  int main() {
4    int niz[BROJ_EL], suma_parnih=0, br_parnih=0, i;
5    double prosjek;
6
7    for (i=0; i<BROJ_EL; i++) {
8      printf("Unesite %d-ti clan niza: ", i+1);
9      scanf("%d", &niz[i]);
10   }
11
12   for (i=0; i<BROJ_EL; i++) {
13     if (niz[i] % 2 == 0) {
14       suma_parnih += niz[i];
15       br_parnih++;
16     }
17   }
18
19   prosjek = (double) suma_parnih / br_parnih;
20   printf("Srednja vrijednost parnih clanova niza je: %g\n", prosjek);
21   return 0;
22 }
```

Najvažniji dio koda je petlja koja se nalazi na linijama 12-17. Ovdje provjeravamo svaki član niza, te ako je taj član (a ne njegov indeks!) paran broj, dodajemo ga na sumu. Time osiguravamo da se izračunava suma samo parnih članova niza, a ne svih. U istom uslovu određujemo i broj članova niza koji su parni brojevi. Na kraju te dvije vrijednosti podijelimo.

Šta će se desiti ako korisnik ne unese niti jedan paran broj? Program će ispisati poruku

```
Srednja vrijednost parnih clanova niza je: -nan
```

NaN (Not-a-Number) je specijalna vrijednost koju mogu dobiti realne promjenljive u C-u u slučaju pokušaja izvođenja matematičke operacije čiji je rezultat nedefinisan, kao što je $0/0$. Postavlja se pitanje: šta želimo da nam program ispiše u ovom slučaju? Jednostavno dodajte uslov kojim provjeravate da li je broj parnih brojeva nula i u tom slučaju ispišite poruku po želji.

✓ NaN

5.1.4 *Zadaci tipa "za svako x" i "postoji barem jedan"*

U poglavlju o prostim brojevima↑ govorili smo o problemima tipa "za svako x" ($\forall x$) i varijantom "postoji barem jedan x" ($\exists x$), pa smo dali opšti algoritam za rješavanje takvih problema. Veliki broj zadataka sa nizovima se može predstaviti kao neka varijanta ovog problema. Kao primjer rješavanja problema $\forall x$ na nizu uzmimo sljedeći zadatak.

↑ Poglavlje 4.4.3, str. 120.

Zadatak 2. *Potrebno je provjeriti da li je unesen takav niz da su svi njegovi članovi djeljivi sa 3.*

Podsjetimo se opšteg algoritma za rješavanje problema tipa $\forall x$:

1757018

1. Pretpostavimo da uslov važi za svako x.
2. Koristeći for petlju prolazimo kroz sve moguće vrijednosti x.
3. Ako nađemo takvo x da uslov ne važi, zaključujemo da uslov zadatka nije ispunjen i prekidamo petlju.

```
1  #include <stdio.h>
2  #define BROJ_EL 5
3  int main() {
4    int i, djeljivi_sa_3, niz[BROJ_EL];
5    for (i=0; i<BROJ_EL; i++)
6      scanf("%d", &niz[i];
7
8    djeljivi_sa_3 = 1; /* 1. */
9    for (i=0; i<BROJ_EL; i++) /* 2. */
10     if (niz[i] % 3 != 0) { /* 3. ... */
11       djeljivi_sa_3 = 0;
12       break;
13     }
14
15   if (djeljivi_sa_3)
16     printf("Svi clanovi niza su djeljivi sa 3\n");
17   else
18     printf("Svi clanovi niza NISU djeljivi sa 3\n");
19
20   return 0;
21 }
```

U ovom primjeru imamo niz od 5 elemenata (što je određeno konstantom `BROJ_EL`) koji je u liniji 4 deklarisan, a u linijama 5 i 6 unesen, što bi trebalo da nam je već dobro poznato. U liniji 4 smo također deklarisali i logičku promjenljivu `djeljivi_sa_3` koja će imati vrijednost 1 ako su svi uneseni brojevi djeljivi sa 3, a 0 ako nisu. Ovaj problem se naravno mogao riješiti kompletan u jednoj for petlji, ali radi boljeg demonstriranja koncepta $\forall x$, razdvojili smo ga na zasebne petlje.

Najprije u liniji 8 postavljamo logičku promjenljivu na vrijednost 1, odnosno pretpostavljamo da su svi članovi niza djeljivi sa 3. U linijama 9-13 nalazi se petlja koja prolazi kroz sve članove niza. Petlja ne posjeduje vitičaste zagrade, jer if naredbu (i prateći blok naredbi u linijama 10-13) posmatramo kao jednu naredbu. Ako naiđemo na takav član niza koji nije djeljiv sa tri (uslov u liniji 10), logička promjenljiva će biti postavljena na vrijednost 0 što znači neistina (linija 11). U tom slučaju možemo i prekinuti petlju naredbom `break` (linija 12) jer nema potrebe da analiziramo ostale članove niza, znamo da uslov zadatka nije ispunjen.

Pogledajmo primjer niza:

```
6 3 9 8 12 6 7 3
```

Za ovaj niz uslov nije ispunjen jer članovi 8 i 7 nisu djeljivi sa tri. Petlja u linijama 9-13 će prolaziti jedan po jedan kroz elemente niza 6, 3 i 9 za koje uslov `if (niz[i]%3 != 0)` neće biti ispunjen. Zatim ćemo naići

1/15/01/98

na član niza na indeksu 3 a to je broj 8 za koji uslov jeste ispunjen. Logička promjenljiva će postati 0 i petlja će se prekinuti jer u tom trenutku više nije bitno da li su ostali članovi poslije osmice djeljivi sa 3 ili ne, riješili smo problem iz zadatka. Ako pak niz prepravimo tako da su svi brojevi djeljivi sa tri:

```
6 3 9 9 12 6 6 3
```

Uslov u liniji 10 se neće nikada ispuniti i logička promjenljiva će zadržati svoju prvobitnu vrijednost 1. Pogrešno rješenje bi bilo:

```
for (i=0; i<BROJ_EL; i++)
  if (niz[i] % 3 != 0) {
    djeljivi_sa_3 = 0;
  } else {
    djeljivi_sa_3 = 1;
  }
```

jer bi u tom slučaju logička promjenljiva zadržala vrijednost koja se tiče posljednjeg člana niza. Recimo, u primjeru prvog niza:

```
6 3 9 8 12 6 7 3
```

petlja se ne bi prekinula break-om, pa bi došla do posljednjeg člana broja 3, za koji bi se izvršio else blok pa bi promjenljiva `djeljivi_sa_3` dobila vrijednost 1, iako u nizu ima članova koji nisu djeljivi sa 3! Da smo u `if` dodali `break`:

```
for (i=0; i<BROJ_EL; i++)
  if (niz[i] % 3 != 0) {
    djeljivi_sa_3 = 0;
    break;
  } else {
    djeljivi_sa_3 = 1;
  }
```

ovaj program bi sada uvijek davao tačne rezultate, ali je u tom slučaju `else` blok suvišan jer sve što on radi je da postavlja promjenljivu `djeljivi_sa_3` na vrijednost 1 za početnih nekoliko članova koji su djeljivi sa 3, a ova promjenljiva svakako ima tu vrijednost i nema potrebe da je ponovo postavljamo.

Dodajmo još da je za ovaj konkretni zadatak (ali ne i za neke buduće zadatke) potpuno ispravno rješenje koje uopšte ne koristi logičku promjenljivu, nego se isključivo posmatra da li se for petlja prekinula prije kraja ili ne.↑

↑ Poglavlje 4.1.6, str. 97.

```
1  #include <stdio.h>
2  #define BROJ_EL 5
3  int main() {
4    int i, djeljivi_sa_3, niz[BROJ_EL];
5    for (i=0; i<BROJ_EL; i++)
6      scanf("%d", &niz[i];
7
8    for (i=0; i<BROJ_EL; i++)
9      if (niz[i] % 3 != 0)
10       break;
11
```

```
12  if (i == BROJ_EL)
13    printf("Svi clanovi niza su djeljivi sa 3\n");
14  else
15    printf("Svi clanovi niza NISU djeljivi sa 3\n");
16
17  return 0;
18 }
```

Kao varijantu problema tipa $\exists x$ uzmimo program koji provjerava da li uneseni niz sadrži neki broj `k` koji je ranije unesen

```
sadrzi=0; /* 1. Pretpostavimo da ne postoji nijedno x */
for (i=0; i<BROJ_EL; i++) { /* 2. Prolazimo kroz sve x */
  if (niz[i] == k) { /* 3. Ako postoji x za koje vazi uslov... */
    sadrzi=1; /* ...zakljucujemo da je uslov ispunjen... */
    break; /* ...i prekidamo petlju */
  }
}
```

Nakon ove petlje logička promjenljiva `sadrzi` označava da li niz sadrži `k` ili ne. Za vježbu probajte izmijeniti ovaj program tako da provjerava da li niz sadrži barem jedan broj na intervalu (50,100).

5.1.5 *Nizovi proizvoljne dužine*

Često u udžbenicima programiranja, pogotovo onim koji nisu pisani za C, možemo vidjeti zadatke ovog tipa:

Zadatak 3. *Korisnik najprije sa tastature unosi veličinu niza n, a zatim unosi cjelobrojne članove niza...*

Kako ovakav zadatak riješiti? Mnogi će to uraditi ovako:

```
#include <stdio.h>
int main() {
  int n,i;
  printf("Unesite velicinu niza n: ");
  scanf("%d", &n);
  int niz[n];
  for (i=0; i<n; i++) {
    printf("Unesite %d-ti clan niza: ", i+1);
    scanf("%d", &niz[i]);
  }
  ...

  return 0;
}
```

Što je najgore, ovo rješenje će sa mnogim prevodiocima raditi.[6] No ovo nije dobro rješenje!

[6] Još gore rješenje (koje će također često raditi) je ono koje sadrži liniju `int n, niz[n];`.

✓ *To što program radi ne znači da je ispravan.*

✓ Ispravan program

Morate zapamtiti da prevodilac nije alat za provjeru ispravnosti koda. Prevodilac je program koji prevodi izvorni kod u izvršni. Pri tome, prevodilac pokušava "pročitati misli" programera i proizvesti

17570198

izvršni kod čak i iz neispravnog izvornog koda ako je moguće, a greške se ispisuju tek ako prevodilac baš ne zna kako nastaviti dalje. Vaša odgovornost kao programera je da prekontrolišete da li u programu postoje sintaksne ili logičke greške, uključujući i one koje prevodilac nije prijavio. Ovo podrazumijeva da morate *znati šta se dešava* u svakoj liniji koda, ili kako neki kažu trebate "imati prevodilac u glavi". Da, zaista je to potrebno.

Osim toga, autori prevodilaca dodaju mnoge nestandardne funkcionalnosti sa ciljem da olakšaju život programerima, ali korištenje tih mogućnosti je opasno jer se dobija kod koji nije portabilan odnosno ne može se pokrenuti na nekom drugom prevodiocu. Kao budući inženjeri koji će sutra razvijati kod profesionalno, vaša odgovornost je da pišete kod koji je usklađen sa standardima.

U prethodnom primjeru smo u liniji 6 deklarisali niz čija je veličina data promjenljivom `n`, no standard C90 propisuje da veličina niza mora biti konstanta (kreirana ključnom riječi `#define`) ili literal (neki broj npr. 100). Noviji standard C99 dozvoljava i promjenljive, ali one nisu podržane u jeziku C++ pa ne preporučujemo njihovo korištenje. U C90 i C++u ispravan pristup je *dinamička alokacija*.↑ Dalje, ovaj program ne radi nikakvu kontrolu ulaza, pa ako korisnik unese negativan broj ili vrlo veliki pozitivan broj, program će se krahirati.

↑ Poglavlje C.2.3, str. 375.

Da bi zadatak bio rješiv u programskom jeziku C sa našim trenutnim nivoom znanja možemo ga dodefinisati, ovako:

Zadatak: *Korisnik najprije sa tastature unosi veličinu niza n* ***(najviše 100)****, a zatim unosi cjelobrojne članove niza...*

Ovo nam omogućuje da deklarišemo niz čija je veličina 100, ali pri tome ne koristimo svih 100 elemenata nego samo prvih `n` koje je korisnik unio:

```
1  #include <stdio.h>
2  int main() {
3    int n, i, niz[100];
4    do {
5      printf("Unesite velicinu niza n: ");
6      scanf("%d", &n);
7    } while (n<1 || n>100);
8    for (i=0; i<n; i++) {
9      printf("Unesite %d-ti clan niza: ", i+1);
10     scanf("%d", &niz[i]);
11   }
12   ...
13
14   return 0;
15 }
```

do-while petljom u linijama 4-7 osigurali smo da je $n \in [1, 100]$. Petlja u liniji 8 ide samo do `n`, te moramo paziti i da sve buduće petlje u programu idu do `n`. Naime, recimo da smo u liniji 12 nastavili pisati kod koji računa sumu članova niza:

1 // D // 0 // 1 // 98

```
  for (i=0; i<100; i++)
    suma += niz[i];
```

Ovaj program bi sadržavao grešku, jer pristupa *neinicijaliziranim članovima niza*. Ako je korisnik unio npr. `n=5`, petlja u linijama 8-11 će unijeti elemente `niz[0]`, `niz[1]`, ..., `niz[4]` i tu će se prekinuti. Vrijednost elementa `niz[5]` nije poznata, jer je nismo inicijalizirali. Ako bismo niz deklarisali kao `int niz[100]={0}` suma članova niza bi bila tačna, ali bilo koji drugi algoritam bi mogao imati problem, i generalno naš program bi bio zbunjujući jer bi ostalo nejasno koliko članova ustvari ima niz. Najbolje rješenje je pisati ovako:

```
12  for (i=0; i<n; i++)
13    suma += niz[i];
```

Dakle, da ponovimo, u liniji 3 smo deklarisali niz maksimalne veličine 100 elemenata. Tu veličinu nazivamo **fizička veličina niza**. No u našem programu nećemo koristiti svih 100 elemenata nego ćemo koristiti neki manji broj n koji korisnik unosi sa tastature. Ovo je **logička veličina**.

✓ Fizička i logička veličina niza

Poznavaoci drugih programskih jezika bi se mogli ovdje pobuniti i postaviti pitanje: ali kako se u C-u deklarišu nizovi *stvarno* proizvoljne veličine? Problem je u tome što nizovi stvarno proizvoljne veličine *ne postoje*, a C je jezik koji ne sakriva tu činjenicu od vas. Memorija u stvarnom svijetu nije neograničena. Npr. ako imate u vašem računaru 4 GB RAM-a, najveći niz koji možete kreirati ne može biti veći od toga. Vi uvijek imate *neko* ograničenje na količinu podataka, samo ga ponekad ne znate ili vam je bilo mrsko da ga projektujete.

U svakom slučaju mora postojati neko ograničenje na veličinu niza, a u našem programu to ograničenje je 100. Procijenili smo da korisnik nikada neće imati potrebu da unese više od 100 elemenata, što treba biti zabilježeno i u dokumentaciji programa. Ustvari, ako malo razmislimo, jedini problem rješenja datog iznad je što je zauzeto potencijalno više memorije nego što je potrebno. Trebate odgovoriti na pitanje: koliko je maksimalno `n` koje nam može zatrebati?

Konačno, ako je stvarno potrebno da radite sa neograničenom količinom podataka, razmislite da li ima potrebe da sve podatke držite istovremeno u memoriji? Pri tome ne mislimo samo o radu sa datotekama (poglavlje 10). Pogledajmo recimo primjer programa koji zaista radi sa neograničenom količinom podataka, bez korištenja datoteka:

Zadatak 4. *Napisati program koji omogućuje korisniku da unese neograničeno velik niz cijelih brojeva koji se završavaju nulom, a zatim na ekranu ispisuje sumu tih brojeva.*

Rješenje:

1/5/0198

```
1 #include <stdio.h>
2 int main() {
```

```
3   int n, suma=0;
4   do {
5     printf("Unesite broj: ");
6     scanf("%d", &n);
7     suma += n;
8   } while (n != 0);
9   printf("Suma je: %d\n", suma);
10  return 0;
11 }
```

U ovom programu ustvari nemamo niz, jer za rješenje zadatka niz nije potreban. Naime u govornom jeziku riječ "niz brojeva" ima više značenja, ne mora se odnositi na programsku strukturu "array". Ako je u nekom zadatku obavezno koristiti baš niz, sastavljač zadatka to treba naglasiti riječima "obavezno koristiti nizove". Istina, u ovom rješenju može doći do prekoračenja promjenljive `suma` tipa `int`, ali to bi važilo i da je korišten niz. Pogledajmo sljedeći zadatak.

Zadatak 5. *Korisnik najprije sa tastature unosi niz realnih brojeva koji se završava nulom, a zatim određuje broj članova niza koji su veći od prosjeka.*

Ovdje ipak moramo koristiti niz, pošto ovakav program radi u dva prolaza: u prvom prolazu se izračunava prosjek, a zatim se traži broj članova niza koji ispunjavaju uslov. No, u programskom jeziku C veličina niza mora biti ograničena, tako da ćemo dopuniti zadatak:

Zadatak: *Korisnik najprije sa tastature unosi niz realnih brojeva* ***(najviše 100)*** *koji se završava nulom, a zatim određuje broj članova niza koji su veći od prosjeka.*

Pogledajmo rješenje ovog zadatka:

```
1  #include <stdio.h>
2  int main() {
3    double niz[100], prosjek=0;
4    int i, n, broj=0;
5    printf("Unesite clanove niza (0 za kraj)\n");
6    for (i=0; i<100; i++) {
7      scanf("%lf", &niz[i]);
8      if (niz[i]==0) break;
9    }
10   n=i;
11
12   for (i=0; i<n; i++)
13     prosjek += niz[i];
14   prosjek /= n;
15
16   for (i=0; i<n; i++)
17     if (niz[i] > prosjek)
18       broj++;
19
20   printf("Broj clanova niza vecih od prosjeka je: %d\n", broj;
21   return 0;
22 }
```

Petlja u linijama 6-9 omogućuje unos najviše 100 realnih brojeva koji se završava kada korisnik unese nulu. Šta će se desiti ako korisnik

nikada ne unese nulu? Unos će se prekinuti nakon stotog broja. Isti efekat smo mogli postići i ovakvom petljom:

```
6  do {
7    scanf("%f", &niz[i]);
8    if (++i == 100) break;
9  } while (niz[i-1] != 0);
```

ali prethodno rješenje izgleda ljepše.

Kako ćemo znati koliko elemenata je korisnik unio? Kada korisnik unese nulu izvršiće se naredba `break`, pa će poslije petlje `i` biti broj unesenih elemenata. Ako korisnik nije unio nulu, `i` će biti 100 što je također tačno. No sljedeća petlja u liniji 12 će vratiti `i` na 0 odnosno izgubićemo njegovu vrijednost. Jedan način da to spriječimo je da ubuduće u programu ne koristimo `i` kao kontrolnu promjenljivu petlje, ali to bi moglo biti zbunjujuće, pa ćemo umjesto toga broj elemenata držati u promjenljivoj `n` (linija 10) što je uobičajen naziv za promjenljivu koja drži broj elemenata niza.

U linijama 12-14 računamo prosjek. Pošto je ovaj dio kôda prilično očigledan, nećemo uvoditi još jednu promjenljivu `suma`, nego koristimo promjenljivu `prosjek`. Promjenljiva `broj` nam predstavlja brojač elemenata koji zadovoljavaju uslov iz zadatka.

5.2 *Neki algoritmi za rad sa nizovima*

Riječ *algoritam* nastala je od imena arapskog naučnika Al-Horezmija. U računarskim naukama, algoritam predstavlja jedan opšti pristup rješenju problema, niz koraka koji su potrebni da bi se riješio dati problem. Tokom godina neki problemi u programiranju su se vrlo često ponavljali, pa su naučnici pokušali odrediti najbolji (najbrži, najlakši...) način kojim se pouzdano i potpuno tačno dolazi do rješenja. Ove metode dolaska rješenja objašnjene korak po korak nazivamo algoritmima.

Obično je algoritam opisan na apstraktan način, koristeći govorni jezik, dijagrame toka ili pseudokod. Ako je algoritam odgovarajući za problem, posao programera ostaje da prevede algoritam u konkretni programski jezik. U popularnim programskim jezicima možete naći gotove implementacije mnogih poznatijih algoritama. Međutim, one često nisu dovoljne. Često se dešava da programer mora u svom projektu iskoristiti neku blago modificiranu varijantu poznatog algoritma, odnosno da se algoritam mora prilagoditi problemu.

Jedan školski primjer ovakvog prilagođavanja je algoritam sortiranja koji slaže neki niz po nekom kriteriju koji nije direktno poređenje članova, npr. niz studenata se može sortirati i po prezimenu i po datumu rođenja. Pošto je ovaj slučaj vrlo čest, u pravilu se kod bibliotečnih funkcija sortiranja može definisati i funkcija kriterija.↑

↑ Str. 414.

1/15/2018

Sa nekim algoritmima smo se već susreli. Recimo algoritam izračunavanja aritmetičke sredine (poznatije kao prosjek ili srednja vrijednost) glasi:

- izračunati sumu svih članova niza;
- sumu podijeliti sa brojem članova.

Na sličan način definišu se algoritmi kao što su geometrijska sredina, harmonijska sredina, medijan i drugi. Ove programe možete probati napraviti za vježbu, ne razlikuju se mnogo od programa za aritmetičku sredinu.

5.2.1 *Maksimum i minimum*

Vrlo često je potrebno pronaći najveći (maksimum) ili najmanji (minimum) član niza. U suštini postoji samo jedan potpuno ispravan način dolaska do najvećeg člana pa možemo taj način nazvati algoritmom za pronalazak maksimuma. Minimum se određuje na vrlo sličan način.

Na vrlo visokom nivou rješenje možemo opisati ovako: da bismo provjerili da li je neki broj najveći član niza, moramo uporediti sve članove niza sa tim brojem. Kada se ovako opiše problem, mnogi početnici u programiranju pišu rješenje koje izgleda otprilike ovako:

```
max=0;
for (i=0; i<BROJ_EL; i++) {
  if (niz[i] > niz[i-1])
    max = niz[i];
}
```

U ovom rješenju imamo čak tri greške! Možete li ih uočiti?

Greške su:

1. Pristup izvan opsega niza, pristupa se elementu na indeksu `i-1` a petlja je počela od 0.
2. Šta ako su svi članovi niza negativni? Nula nije dobra početna vrijednost za promjenljivu `max`.
3. Ovaj program pronalazi lokalni maksimum a ne maksimum, jer se porede samo susjedni elementi niza.

Da bismo došli do potpuno tačnog rješenja, popravićemo jednu po jednu grešku.

Pristup izvan opsega niza dešava se zato što smo u trećoj liniji problematičnog koda pristupili elementu `niz[i-1]` a početna vrijednost za `i` je 0. Dakle u prvom prolazu kroz petlju pristupili smo elementu na indeksu -1 što može dovesti do krahiranja programa. Ovaj problem možemo riješiti tako što ćemo u petlji za početnu vrijednost uzeti 1, ili ćemo pak prepraviti petlju tako da se porede elementi na indeksima

17577198

`i` i `i+1` a uslov petlje bi glasio `i<BROJ_EL-1`. Kod sa prvom popravkom sada glasi:

```
max=0;
for (i=1; i<BROJ_EL; i++) {
  if (niz[i] > niz[i-1])
    max = niz[i];
}
```

Sada prelazimo na grešku broj 3. Da bismo shvatili uzrok ove greške, uzmimo primjer niza od pet elemenata koji glase:

```
4 -22 151 0 88
```

Najprije će se porediti -22 sa 4. Uslov neće biti ispunjen i neće se desiti ništa. U drugom prolasku kroz petlju `i` će biti 2, pa će se porediti 151 sa -22. Uslov je ispunjen pa će promjenljiva `max` dobiti vrijednost 151. Za sada izgleda da program radi kako treba. U trećem prolasku promjenljiva `i` ima vrijednost 3, poredi se 0 sa 151. Uslov nije ispunjen i ne dešava se ništa. Konačno, u posljednjem prolasku poredi se 88 sa 0, uslov je ispunjen i promjenljiva `max` dobija vrijednost 88, što je netačno.

Greška je nastupila u posljednjem prolasku kroz petlju kada su upoređeni `i`-ti član niza 88 i `i-1`-ti član niza 0. Broj 88 je trebalo uporediti sa 151, do sada pronađenim maksimumom, pa pošto 88 nije veće od 151 promjenljiva `max` je trebala zadržati tu vrijednost. Da bismo postigli ovaj efekat mijenjamo uslov na sljedeći način:

```
max=0;
for (i=0; i<BROJ_EL; i++) {
  if (niz[i] > max)
    max = niz[i];
}
```

Primjećujete i da smo početnu vrijednost `i` vratili na nulu jer više nemamo problem sa pristupom elementu na indeksu `i-1`. No ovaj program i dalje nije tačan jer smo pogrešno odabrali početnu vrijednost promjenljive max. Da to ilustrujemo, pogledajmo slučaj niza koji se sastoji isključivo od negativnih brojeva:

```
-4 -22 -151 -2 -88
```

Kod ovakvog niza, uslov u trećoj liniji koda nikada neće biti ispunjen, jer niti jedan član niza nije veći od nule. Neko bi mogao predložiti da se za početnu vrijednost `max` uzme -100 ili -1000, ali opet je moguć slučaj da su svi članovi niza manji i od tog broja. Mogli bismo za početnu vrijednost uzeti broj -2^{31}, što je najmanja moguća vrijednost koja se može pohraniti u promjenljivu tipa `int`. Takav program bi svakako radio ispravno na našem PC računaru. No, to opet nije potpuno ispravno rješenje jer smo tu napravili pretpostavku o tipu i opsegu promjenljivih, npr. taj program neće raditi ispravno ako ga probamo pokrenuti na nekom mikrokontroleru gdje je `int` 16-bitni cijeli broj, ili ako se odlučimo promijeniti tip niza.

1/1/2/7/0/1/9/8

Jedino rješenje koje ne pravi nikakve pretpostavke oko tipa promjenljivih u nizu je ono gdje se za početnu vrijednost uzima prvi član niza. Tu imamo dvije mogućnosti: ili je taj prvi član niza maksimum, ili se u nizu nalazi član koji je veći od njega pa će se sigurno nekada izvršiti uslov iz zadatka.

Kod za ovakvo rješenje glasi:

```
max = niz[0];
for (i=1; i<BROJ_EL; i++) {
  if (niz[i] > max)
    max = niz[i];
}
```

i ovo je jedino do sada potpuno ispravno rješenje. Možete uočiti da smo početnu vrijednost `i` ponovo vratili na 1, ali ovaj put zato što nema potrebe da prvi član niza poredimo sa samim sobom. Rješenje sa `i=0` ne bi bilo pogrešno, ali je ovako neznatno efikasnije.

Kako bi glasilo rješenje koje traži najmanji član u nizu? Izmjena je neznatna:

```
min = niz[0];
for (i=1; i<BROJ_EL; i++) {
  if (niz[i] < min)
    min = niz[i];
}
```

Da bi priča bila kompletna, daćemo čitav program koji možete ukucati i pokrenuti:

```
1  #include <stdio.h>
2  #define BROJ_EL 10
3  int main() {
4    int niz[BROJ_EL], max, i;
5
6    for (i=0; i<BROJ_EL; i++) {
7      printf("Unesite %d-ti clan niza: ", i+1);
8      scanf("%d", &niz[i]);
9    }
10
11   max = niz[0];
12   for (i=0; i<BROJ_EL; i++)
13     if (niz[i] > max)
14       max = niz[i];
15
16   printf("Najveci clan niza je: %d\n", max);
17   return 0;
18 }
```

Podsjetimo se sada priče o algoritmima. Ako je ovo jedan, glavni, jedini način pronalaska maksimuma i minimuma, zašto onda ne postoje u standardnoj biblioteci nekakve gotove funkcije "max" i "min" koje samo pozovemo i koristimo? Zato što problemi iz realnog života u pravilu nisu tako jednostavni, pa u vašem programu morate napraviti i određene preinake koje zahtijevaju dobro poznavanje i razumijevanje algoritma za traženje maksimuma/minimuma.

1777577098

Recimo, šta ako trebamo naći na kojem mjestu u nizu se nalazi najveći element, odnosno tražimo *indeks u nizu* najvećeg elementa. Jedno rješenje koje početnicima često padne na pamet je da nakon petlje koja pronalazi najveći član napravimo novu petlju koja traži taj član u nizu:

```
11  max = niz[0];
12  for (i=1; i<BROJ_EL; i++) {
13    if (niz[i] > max)
14      max = niz[i];
15  }
16
17  for (i=0; i<BROJ_EL; i++) {
18    if (niz[i] == max) {
19      printf("Najveci clan se nalazi na %d. mjestu\n", i+1);
20      break;
21    }
22  }
```

Obratite pažnju da smo morali u drugoj petlji postaviti naredbu `break` u uslov, za slučaj da u nizu postoji više jednakih članova koji su najveći. Generalno, da bismo ispravno riješili ovaj zadatak, moramo precizno definisati šta se traži u slučaju da u nizu ima više jednakih članova, ali jedno logično rješenje je da ispišemo poziciju prvog takvog člana.

Dakle, ovakvo rješenje je naravno tačno prema postavci zadatka, ali da li se to moglo riješiti odmah u prvoj petlji? Pogledajte sljedeće rješenje

```
11  imax = 0;
12  for (i=1; i<BROJ_EL; i++) {
13    if (niz[i] > niz[imax])
14      imax = i;
15  }
16  printf("Najveci clan se nalazi na %d. mjestu\n", imax + 1);
```

Ovaj program je vrlo sličan prvobitnom tačnom rješenju za traženje maksimuma, ali sada promjenljiva `imax` ima drugačije značenje od prethodne `max` (što smo označili tako što smo joj promijenili ime). Promjenljiva `imax` umjesto najvećeg člana niza čuva indeks u nizu najvećeg člana (i max = indeks najvećeg). Najprije postavljamo `imax=0`, odnosno pretpostavljamo da je najveći član na indeksu 0. Zatim krećemo od indeksa 1 i svaki član poredimo sa članom na indeksu `imax`. Ako je `i`-ti član niza veći od člana na indeksu `imax`, `imax` postaje `i` odnosno od sada najveći član niza je član na trenutnom indeksu.

Zadatak 6. *Napisati program koji omogućuje korisniku da unese niz od 100 elemenata, a zatim pronalazi i na ekranu ispisuje najveći član niza koji je paran broj.*

17570198

Prije svega, moramo se odlučiti šta ćemo uraditi u slučaju da se u nizu ne nalazi niti jedan paran broj. Recimo da u tom slučaju treba ispisati poruku "Niste unijeli niti jedan paran broj".

Recimo mogli bismo prepraviti naš program za traženje najvećeg člana ovako:

```
max=niz[0];
for (i=1; i<BROJ_EL; i++) {
  if (niz[i] > max && niz[i] % 2 == 0)
    max = niz[i];
}
```

Ovaj program će vrlo često dati tačan rezultat, ali problem se javlja ako je prvi uneseni član niza neparan broj i usto je veći od svih parnih članova niza. Recimo, neka je korisnik unio niz:

```
11 1 2 3 4 5 6 7 8 9
```

Promjenljiva `max` će nakon prethodne petlje imati vrijednost 11, što je netačan rezultat. Mogli bismo napraviti još jednu petlju koja najprije pronalazi nekakav paran broj u nizu (bilo kakav, nije bitno da li je najveći ili ne). Ovaj broj će nam isključivo poslužiti kao početna vrijednost promjenljive `max`. Zatim u sljedećoj petlji tražimo stvarni maksimum.

```
max=niz[0];
for (i=0; i<BROJ_EL; i++)
  if (niz[i] % 2 == 0)
    max = niz[i];
for (i=0; i<BROJ_EL; i++) {
  if (niz[i] > max && niz[i] % 2 == 0)
    max = niz[i];
}
```

Ovo rješenje je prije svega vrlo neefikasno. Mogli bismo naredbom `break` prekinuti prvu petlju čim nađemo neki paran broj, jer nama ustvari nije ni bitno koji član niza smo odabrali dok god je to član niza koji je paran broj. Ipak, ovo rješenje ćemo označiti kao netačno jer ono nije odgovorilo na prvo pitanje koje smo postavili, a to je šta ako su svi članovi niza neparni brojevi. Kôd iznad će u tom slučaju vratiti prvi (neparan) član niza što je netačan odgovor, mada bismo mogli nekim dodatnim uslovom provjeriti da li se to desilo i ispisati odgovarajuću poruku kao što je urađeno u sljedećem primjeru (linije 18 i 19).

```
11  max = niz[0];
12  for (i=0; i<BROJ_EL; i++) {
13    if (niz[i] % 2 == 0) {
14      max = niz[i];
15      break;
16    }
17  }
18  if (max % 2 != 0) {
19    printf("Niste unijeli niti jedan paran broj.");
20  } else {
21    for (i=0; i<BROJ_EL; i++) {
22      if (niz[i] % 2 == 0 && niz[i] > max)
23        max = niz[i];
24    }
25
26    printf("Najveci paran broj je: %d\n", max);
27  }
```

I sada ovo rješenje je potpuno tačno, ali bi moglo biti i kraće pri čemu ćemo koristiti samo jednu petlju. To možemo postići na sljedeći način: postavićemo `max` na prvi član niza. Ako je taj član niza paran, tražimo najveći član kao i ranije. Ako je neparan, zamijenićemo ga prvim parnim članom na koji naiđemo. Da bismo to uradili u jednoj petlji postavljamo uslov ovako:

```
11  max = niz[0];
12  for (i=0; i<BROJ_EL; i++)
13    if ( niz[i] % 2 == 0 && (max % 2 != 0 || niz[i] > max) )
14      max = niz[i];
15  if (max % 2 != 0)
16    printf("Niste unijeli niti jedan paran broj.\n");
17  else
18    printf("Najveci paran broj je: %d\n", max);
```

Dakle, `max` će postati `niz[i]` ako je: `niz[i]` paran broj **i** ako je ispunjen jedan od sljedećih uslova: `niz[i]` je veći od `max` ili ako je `max` neparan broj, što znači da je `niz[i]` prvi do sada pronađeni paran broj. Ako je i nakon petlje `max` neparno, znači da se u nizu ne nalazi niti jedan paran broj.

5.2.2 *Izbacivanje člana iz niza*

Sada ćemo se pozabaviti zadacima u kojima se traži da se iz niza izbaci element (elementi) koji zadovoljava(ju) određeni uslov. Šta uopšte znači izbaciti element iz niza?

Zadatak 7. *Napisati program koji omogućuje unos niza cijelih brojeva, a zatim izbacuje sve neparne elemente iz niza.*

Sljedeće rješenje bi naizgled zadovoljilo postavku zadatka:

```
for (i=0; i<n; i++) {
  if (niz[i] % 2 == 0) {
    printf("%d", niz[i]);
  }
}
```

ali ovdje ćemo ga smatrati netačnim jer ovim rješenjem element nije ustvari *izbačen* iz niza, on je samo *preskočen* prilikom ispisa. Recimo da se u nastavku programa tražilo da se izračuna srednja vrijednost članova niza ili pronađe maksimum/minimum itd. U svakoj sljedećoj petlji bismo morali dodavati uslov `if (niz[i]%2 == 0)` što bi u dovoljno velikom programu predstavljalo komplikaciju, onemogućilo bi nas da npr. koristimo neku gotovu funkciju za izračunavanje prosjeka ili traženje maksimuma. Osim toga, problem izbacivanja iz niza je vježba razumijevanja nizova, kao i sve ostalo što radimo u ovom poglavlju.

Dakle, ako u zadatku piše "izbaciti element iz niza", rješenje kao iznad ćemo smatrati *pogrešnim rješenjem*. Kako onda izbaciti element

1/15/01/98

iz niza? Potrebno je presložiti članove niza tako da niz više ne sadrži izbačeni element.

Recimo ako je niz glasio:

```
2 4 5 6 8
```

program treba dati rezultat

```
2 4 6 8
```

Najprije uočavamo da je broj elemenata niza koji je ranije bio 5 sada postao 4, odnosno veličina niza se smanjila za 1. U programskom jeziku C mi ne možemo stvarno promijeniti broj članova niza nakon što je niz deklarisan, tj. ne možemo promijeniti fizičku veličinu niza, ali možemo smanjiti neku vrijednost `n` koja nam predstavlja logičku veličinu.↑

↑ Poglavlje 5.1.5, str. 140.

Predstavićemo dva algoritma za izbacivanje elementa iz niza.

✓ Izbacivanje elemenata bez očuvanja redoslijeda

Izbacivanje elemenata bez očuvanja redoslijeda. Ako nam redoslijed elemenata u nizu nije bitan, možemo izbaciti element na sljedeći način:

1. na mjesto izbačenog elementa upisati posljednji;
2. smanjiti veličinu niza za jedan;
3. ponovo posmatrati prebačeni element (jer možda i njega treba izbaciti).

Kôd koji implementira ovaj algoritam bi glasio:

```
11  for (i=0; i<n; i++) {
12    if (niz[i] % 2 != 0) {
13      niz[i] = niz[n-1];
14      n--;
15      i--;
16    }
17  }
```

Trebamo iz niza izbaciti sve neparne brojeve kojih može biti i više, tako da petljom prolazimo kroz niz i tražimo elemente za izbacivanje. U liniji 12 nalazi se uslov izbacivanja koji u ovom zadatku glasi "da li je broj neparan". U liniji 13 smo prebacili posljedni element niza na `i`-to mjesto. Ako niz ima `n` elemenata, posljedni element se nalazi na indeksu `n-1`. U liniji 14 smo smanjili veličinu niza `n` za 1.

Konačno, kada prebacimo posljednji element trebamo i njega provjeriti. Npr. neka je niz glasio:

```
2 4 5 6 8 9
```

Petlja će redom prolaziti kroz elemente niza dok ne naiđe na broj 5 koji treba izbaciti (`i` je 2). Zatim će posljednji element 9 biti prepisan na mjesto izbačenog elementa 5:

```
2 4 9 6 8 9
```

1/15/01/98

Sada trebamo nekako obavijestiti program da nas ova devetka na posljednjem mjestu u nizu više ne interesuje, a to postižemo naredbom `n--` kojom smo smanjili veličinu niza za 1.

```
2 4 9 6 8
```

Kada ne bismo imali liniju 15, petlja bi se dalje nastavila uvećavanjem promjenljive `i` pa bi `i` bilo 3 i sada bismo provjeravali broj 6. Rezultat ne bi bio tačan, jer bi niz sadržavao neparan broj 9. Problem je u tome što je 9 prebačeno sa posljednjeg mjesta u nizu, ali i taj broj može biti neparan, tako da moramo vratiti brojač `i` kako bismo i njega provjerili (linija 15). Broj 9 ćemo izbaciti ponovo prebacivanjem posljednjeg elementa tako da će niz sada glasiti

```
2 4 8 6
```

Vidimo da su u odnosu na polazni niz brojevi 6 i 8 zamijenili mjesta, odnosno *redoslijed nije očuvan*.

✓ Izbacivanje elemenata uz očuvanje redoslijeda

Izbacivanje elemenata uz očuvanje redoslijeda. Nekada se dešava da elemente treba izbacivati tako da njihov redoslijed bude očuvan. U našem primjeru, niz prije izbacivanja koji je glasio

```
2 4 5 6 8 9
```

nakon izbacivanja treba glasiti

```
2 4 6 8
```

odnosno elementi koji nisu izbačeni ostaju istim redoslijedom kojim su bili. Kako ovo postižemo? Da bismo to shvatili zapišimo indekse članova niza prije izbacivanja broja 5:

```
niz[0]: 2 niz[1]: 4 niz[2]: 5 niz[3]: 6 niz[4]: 8
```

a nakon izbacivanja

```
niz[0]: 2 niz[1]: 4 niz[2]: 6 niz[3]: 8
```

Uočavamo da je broj 6 koji je ranije bio na indeksu 3 sada na indeksu 2, broj 8 koji je bio na indeksu 4 sada je na indeksu 3 i tako dalje do kraja niza. Algoritam izbacivanja dakle glasi:

1. sve elemente u nizu od izbačenog do kraja niza pomjeramo za jedno mjesto ulijevo (prema početku niza), tako da će susjedni element biti prepisan preko elementa koji treba izbaciti;
2. smanjiti veličinu za jedan;
3. ponovo posmatrati prebačeni element (jer možda i njega treba izbaciti).

17570198

Za pomjeranje svih elemenata niza od `i`-tog do kraja potrebna nam je for petlja. Implementacija ovog algoritma glasi:

```
11  for (i=0; i<n; i++) {
12    if (niz[i] % 2 != 0) {
13      for (j=i+1; j<n; j++) {
14        niz[j-1] = niz[j];
15      }
16      n--;
17      i--;
18    }
19  }
```

Pogledajmo na primjeru kako algoritam funkcioniše. Kada odredimo da element 5 treba izbaciti, počinje petlja u liniji 13 koja pomjera sve elemente za jedno mjesto ulijevo. Članovi niza će redom izgledati ovako:

```
2 4 6 6 8 9
2 4 6 8 8 9
2 4 6 8 9 9
```

U liniji 16 ćemo smanjiti veličinu niza za jedan tako da će konačno niz izgledati ovako:

```
2 4 6 8 9
```

Vidimo da je broj 5 izbačen iz niza na način da su ostali elementi sačuvali svoj redoslijed.

5.2.3 *Ubacivanje člana u niz*

Zadatak 8. *Napisati program koji omogućuje unos niza cijelih brojeva, a zatim u nizu pronalazi parove susjednih elemenata takve da je drugi član para tačno dvostruko veći od prvog. Ako takvi parovi postoje u nizu, između njih treba ubaciti novi član koji je jednak zaokruženoj srednjoj vrijednosti ta dva člana.*

Primjer: Ako je korisnik unio niz:

```
2 4 5 3 6 8 9
```

Nakon ubacivanja niz treba da glasi:

```
2 3 4 5 3 5 6 8 9
```

Između članova 2 i 4 ubačeno je 3, a između članova 3 i 6 ubačeno je 5 (što je 4,5 zaokruženo).

Ubacivanje člana u niz suštinski je slično izbacivanju: umjesto pomjeranja ulijevo, pomjerili bismo sve elemente za po jedno mjesto udesno, čime bi se na indeksu `i` ukazalo jedno slobodno mjesto. Naivna prepravka bi glasila ovako:

```
for (i=0; i<n; i++) {
  if (uslov) {
    for (j=i+1; j<n; j++)
      niz[j] = niz[j-1];
    n++;
    niz[i] = ubaceni;
  }
}
```

U liniji 4 smo samo zamijenili lijevu i desnu stranu dodjele, a u liniji 5 broj elemenata `n` se ne smanjuje nego se povećava. U liniji 6 nema više potrebe da ponovo razmatramo isti element, umjesto toga na indeks `i` upisujemo neku vrijednost `ubaceni`.

Već uočavate da je ovo rješenje ipak pogrešno, pa da vidimo zašto. Recimo da niz glasi: 2, 4, 5, 3, 6, 8, 9 te da prije broja 4 treba ubaciti broj 3. Kada bude `i=1` uslov će biti ispunjen pa će se izvršiti petlja u linijama 3 i 4. Najprije će biti `j=2` pa će se broj 4 prepisati preko broja 5. Zatim će biti `j=3`, vrijednost `niz[j-1]` je upravo postala 4 pa će se 4 prepisati i preko broja 3 itd. Vidimo da će svi članovi niza od indeksa `i` do kraja poprimiti vrijednost 4!

Rješenje ovog problema je da se petlja za pomjeranje udesno treba izvršavati unazad, od kraja prema indeksu `i`, kako se članovi niza ne bi prepisivali jedni preko drugih:

```
1  for (i=0; i<n; i++) {
2    if (uslov) {
3      for (j=n; j>i; j--)
4        niz[j] = niz[j-1];
5      n++;
6      niz[i] = ubaceni;
7    }
8  }
```

Radi elegantnijeg zaglavlja petlje u liniji 3, indeksi članova niza u liniji 4 su pomjereni za jedan.

Ostaje da napišemo uslov.

```
1  for (i=0; i<n-1; i++) {
2    if (niz[i] * 2 == niz[i+1]) {
3      ubaceni = round((niz[i]+niz[i+1])/2.);
4      for (j=n; j>i; j--)
5        niz[j] = niz[j-1];
6      n++;
7      niz[i] = ubaceni;
8    }
9  }
```

Petlja u liniji 1 sada ide do `n-1` zato da u uslovu ne bismo izašli iz opsega niza. U liniji 3 izračunavamo ubačeni član, pri čemu će se najprije izračunati tačna vrijednost razlike bez odsjecanja, što smo postigli tako što smo naveli nazivnik kao realnu vrijednost `2.`,↑ a zatim će se zaokružiti funkcijom `round`.↑ Ostatak programa možete napisati sami za vježbu.

↑ Poglavlje 1.3, str. 38.
↑ Poglavlje 3.1.8, str. 76.

Ovaj program ipak ima jedan mali nedostatak. Naime, ako su članovi niza bliski maksimalnoj dozvoljenoj vrijednosti, `niz[i]*2` bi moglo biti veće od te vrijednosti tako da bi došlo do prekoračenja.↑ Posljedica bi bila da bi se mogao pogrešno ustanoviti par. Recimo, ako je niz tipa `unsigned int`, a član niza ima vrijednost $2^{31} + 1$, nakon množenja sa 2 dobili bismo vrijednost 2! Ovaj problem bi se mogao zaobići tako što umjesto množenja koristimo dijeljenje, te bismo tre-

↑ Poglavlje 3.1.9, str. 78.

1775770198

bali prihvatati samo parne brojeve (kako se ne bi desilo da se kao par prepoznaju npr. 2 i 5):

```
2    if (niz[i+1] % 2 == 0 && niz[i] == niz[i+1] / 2) {
```

5.2.4 *Prebrojavanje članova niza i histogram*

Zadatak 9. *Korisnik unosi niz cijelih brojeva (najviše 100) koji se nalaze u intervalu [0,100]. Unos se završava kada se unese broj -1. Program treba ispisati članove niza koji se javljaju najveći i najmanji broj puta. Ako se više elemenata javlja isti (najveći ili najmanji) broj puta, potrebno je ispisati manji od njih.*

```
Unesite clanove niza (-1 za kraj): 3 15 15 8 3 17 1 17 -1
Najvise puta se javlja 3 a najmanje 1
```

Objašnjenje: Brojevi 3, 15 i 17 se javljaju po dvaput (od njih 3 je najmanji), a 1 i 8 po jednom (od njih 1 je najmanji).

Da bismo prebrojali koliko puta se u nizu javlja koji broj, potreban nam je po jedan brojač za svaku moguću vrijednost člana. Dato je da su članovi niza cijeli brojevi na intervalu [0,100], dakle možemo deklarisati niz od 101 brojača.

```
int brojaci[101];
```

Ovakav niz brojača se naziva *histogram niza*. Ako je član niza jednak nekom datom broju, povećavamo odgovarajući brojač za jedan. Naravno, ovo rješenje je moguće samo ako su članovi niza pozitivni cijeli brojevi na nekom relativno malom intervalu. Ako taj interval ne počinje od nule, možemo koristiti neki pomak npr. ako je dat interval [-50,50] uvećaćemo vrijednost člana niza za 50 kako bismo dobili indeks u histogramu.

Na osnovu ovoga neko bi mogao početi raditi ovako:

```
for (i=0; i<n; i++) {
  if (niz[i] == 0) brojaci[0]++;
  if (niz[i] == 1) brojaci[1]++;
  if (niz[i] == 2) brojaci[2]++;
  ...
}
```

Očito da ovako neće ići. Moramo proći kroz sve vrijednosti 0, 1, 2..., uporediti i-ti član niza sa svakom od tih vrijednosti i povećati odgovarajući brojač za jedan. A kao što znamo, za situacije gdje su nam potrebne sve vrijednosti na nekom intervalu koristimo for petlju:

```
for (i=0; i<n; i++) {
  for (j=0; j<=100; j++) {
    if (niz[i] == j) brojaci[j]++;
  }
}
```

1778198

Ovdje promjenljiva `j` poprima vrijednosti [0,100], `i`-ti član niza poredimo sa tom vrijednošću `j`, pa ako su jednaki povećavamo brojač na indeksu `j` za jedan. Ali može i bolje! Pogledajte ovo rješenje:

```
for (i=0; i<n; i++) {
  brojaci[niz[i]]++;
}
```

Neka recimo niz glasi:

```
3 15 15 8 3 17 1 17
```

Kada je `i=0` prvi član niza `niz[i]` je 3. Kada napišemo `brojaci[niz[i]]++` umjesto `niz[i]` možemo uvrstiti 3 pa će se u tom trenutku izvršiti operacija `brojaci[3]++` tj. brojač za broj 3 će se uvećati za jedan. U drugom prolazu `i=1`, `niz[i]=15`, `brojaci[niz[i]]=brojaci[15]` itd.

Sada ćemo dati rješenje do ovog mjesta, dakle sa određivanjem histograma:

```
1  #include <stdio.h>
2  int main() {
3    int niz[100], brojaci[101]={0};
4    int i, n, max, imax, min, imin;
5
6    printf("Unesite clanove niza (-1 za kraj)\n");
7    for (i=0; i<100; i++) {
8      scanf("%d", &niz[i]);
9      if (niz[i]==-1) break;
10     if (niz[i]<0 || niz[i]>100) {
11       printf("Broj nije na intervalu [0,100]. Ponovite unos");
12       i--;
13     }
14   }
15   n=i;
16
17   for (i=0; i<n; i++)
18     brojaci[niz[i]]++;
19   ...
```

U liniji 3 smo deklarisali niz `niz` sa 100 elemenata i niz `brojaci` sa 101 elementom, te smo sve članove ovog niza postavili na nulu.↑ Slijedi petlja za unos proizvoljnog broja članova niza (najviše 100) kao u zadatku 5 (poglavlje 5.1.5) s tim što se ovdje unos završava brojem -1 umjesto 0.[7] Zašto -1? Ako je dato da su članovi niza na intervalu [0,100] nula je validan član niza pa je ne možemo koristiti za kraj unosa. Ovaj prekid unosa dešava se u liniji 9, a u linijama 10-13 osiguravamo da su članovi niza na intervalu [0,100] i tražimo ponavljanje unosa ako nisu, što postižemo naredbom `i--`. Linija 15 je kao u zadatku 5, a linije 17-18 smo upravo objasnili.

↑ Poglavlje 5.1.2, str. 134.

[7] Obratite pažnju da ako je niz nekog od `unsigned` tipova, nećemo moći raditi na ovaj način. Naime ako u liniji 8 unesemo broj -1 dolazi do prekoračenja, pa će stvarna vrijednost člana niza biti pozitivan broj. Ovo možemo izbjeći tako što deklarišemo pomoćnu promjenljivu tipa `int`, pa nju unosimo u liniji 8, upoređujemo je sa -1 u liniji 9, provjerimo interval (onda nam ne bi trebalo ni `i--`), pa tek nakon toga je pridružimo članu niza.

Kako će glasiti niz `brojaci` po završetku ovog dijela programa? Neka je korisnik unio brojeve

```
3 15 15 8 3 17 1 17
```

Histogram ovog niza glasi:

```
0 1 0 2 0 0 0 0 1 0 0 0 0 0 0 2 0 2 0 0 0 ...
```

1/15/2019

(ako vam još uvijek nije jasno zašto, nadamo se da će postati u nastavku). Najveći član ovog niza je 2, a nalazi se na indeksima 3, 15, 17, što znači da se brojevi 3, 15 i 17 javljaju najviše puta u polaznom nizu. Dakle, sljedeći zadatak koji trebamo obaviti je naći *indeks u nizu najvećeg člana* (odnosi se na niz `brojaci`), a to smo objasnili u poglavlju 5.2.1:

```
20  imax=0;
21  for (i=0; i<=100; i++) {
22    if (brojaci[i] > brojaci[imax])
23      imax = i;
24  }
```

Na sličan način se pronalazi i *indeks najmanjeg člana*, pa to možemo obaviti u istoj petlji:

```
20  imax=imin=0;
21  for (i=0; i<=100; i++) {
22    if (brojaci[i] > brojaci[imax])
23      imax = i;
24    if (brojaci[i] < brojaci[imin])
25      imin = i;
26  }
```

U nekom drugom zadatku ovo bi bilo tačno rješenje, no histogram je specifičan, jer se u njemu javljaju nule za one članove kojih nema u polaznom nizu. Tako bi ovakav program ispisao da se u polaznom nizu najmanje puta javlja broj 0 (što je tačno jer se ne javlja nikako), ali to nije rješenje koje bi korisnik očekivao, pa ćemo malo zakomplikovati naše rješenje. Ne možemo uzeti 0 za početnu vrijednost `imin`, jer se broj 0 možda ne javlja nijednom, pa ćemo uzeti neku ilegalnu vrijednost koja nam označava da do sada još nije pronađen validan minimum:

```
20  imax = 0;
21  imin = -1;
22  for (i=0; i<=100; i++) {
23    if (brojaci[i] > brojaci[imax]) {
24      imax = i;
25    }
26    if (brojaci[i] > 0 && (imin == -1 || brojaci[i] < brojaci[imin]) {
27      imin = i;
28    }
29  }
```

Uslov u liniji 26 čitamo ovako: ako je broj u nizu brojača veći od nule (broj se javlja u polaznom nizu) i ako je manji od do sada pronađenog najmanjeg, ili do sada nije pronađen najmanji, uzimamo taj član niza kao najmanji. Redoslijed uslova je bitan: moramo prvo provjeriti da li je `imin==-1`, jer ako jeste, onda ne smijemo pristupiti članu niza `brojaci` na tom mjestu. Ovakav kod:

```
    if (brojaci[i] > 0 && (brojaci[i] < brojaci[imin] || imin == -1) {
```

ne bi bio ispravan jer bi u prvom prolazu kroz petlju pristupao članu niza `brojaci` na indeksu -1 što je izvan opsega niza. Kada zamijenimo

1/15/01/98

mjesta članovima uslova, ako je `imin==-1` sljedeći poduslov se neće ni evaluirati zbog logičkog operatora *ili* (`||`).

Ostaje još da ispišemo rezultat na ekran:

```
31  printf("Najvise puta se javlja %d, a najmanje %d\n", imax, imin);
32  return 0;
33 }
```

Zadatak 10. *Korisnik unosi niz realnih brojeva (najviše 100). Unos se završava kada se unese broj -1. Program treba ispisati članove niza koji se javljaju najveći i najmanji broj puta. Ako se više elemenata javlja isti (najveći ili najmanji) broj puta, potrebno je ispisati manji od njih.*

Ako su članovi niza realni brojevi ili ako je interval vrlo velik, ne možemo raditi kao iznad jer bi nam niz brojača trebao biti vrlo velik ili čak beskonačan. Možemo koristiti ovakav pristup:

```
17  for (i=0; i<n; i++) {
18    for (j=i; j<n; j++)
19      if (niz[i] == niz[j]) brojaci[i]++;
20  }
```

Ovo rješenje je nešto manje efikasno od prethodnog, jer imamo dvostruku petlju. Sada nam `i`-ti član niza brojača odgovara broju ponavljanja `i`-tog člana niza `niz`. Veličina niza `brojaci` treba minimalno biti kao veličina niza `niz`, a to je 100 elemenata.

5.2.5 *Sortiranje: Selection Sort*

Riječ **sortiranje** označava preslaganje nekog niza po veličini ili nekom drugom kriteriju. Npr. ako je prije sortiranja niz glasio:

```
4 -22 151 0 71 38 1
```

nakon što ga sortiramo od manjih ka većim brojevima niz treba glasiti:

```
-22 0 1 4 38 71 151
```

Postoji mnogo algoritama sortiranja. Neki algoritmi su vrlo efikasni, ali dosta kompleksni za razumijevanje i implementaciju. Osim toga, standardne biblioteke većine programskih jezika nude gotovu funkciju za sortiranje. U pravilu ne biste trebali implementirati sortiranje ručno. Ovdje ćemo raditi *selection sort* koji je dosta jednostavan, kao primjer algoritma za rad sa nizovima. Pojednostavljeno, algoritam selection sort glasi:

- Za svaki i-ti član niza:
  1. nađi najmanji član niza od i-tog mjesta do kraja,
  2. zamijeni taj član sa i-tim.

1757018 98

Recimo u primjeru iznad najprije će nam biti `i=0`, `niz[i]=4`. Najmanji član niza od `i`-tog mjesta do kraja je -22, pa ćemo zamijeniti ta dva člana. Nakon prve iteracije sortiranja niz glasi:

```
-22 4 151 0 71 38 1
```

U drugoj iteraciji je `i=1`, pa je ponovo `niz[i]=4`. Najmanji član nakon `i`-tog je 0. Nakon zamjene niz glasi:

```
-22 0 151 4 71 38 1
```

Nakon svake sljedeće iteracije niz treba glasiti:

```
-22 0 1 4 71 38 151
-22 0 1 4 71 38 151
-22 0 1 4 38 71 151
-22 0 1 4 38 71 151
```

Nakon posljednje iteracije niz je sortiran.

Pogledajmo implementaciju selection sort algoritma u C-u:

```
1  for (i=0; i<n; i++) {
2    min=i;
3    for (j=i+1; j<n; j++) {
4      if (niz[j] < niz[min])
5        min = j;
6    }
7    temp = niz[i];
8    niz[i] = niz[min];
9    niz[min] = temp;
10 }
```

Najprije imamo petlju koja prolazi kroz sve članove niza, odnosno uzima `i`-ti član. U linijama 2-6 nalazi se algoritam za pronalazak najmanjeg člana niza počevši od `i`-tog, konkretno tražimo *indeks u nizu* najmanjeg člana.

Promjenljiva `min` je ovdje ustvari ono naše `imin` o kojem smo pričali u poglavlju 5.2.1. Za početnu vrijednost uzimamo `i` – ako je `i`-ti član manji od svih članova poslije njega, on je već sortiran i ne trebamo uraditi ništa tako da će `min` ostati jednako `i`. Petlja u liniji 3 počinje od sljedećeg člana a to je `i+1`.

Ostaje još da objasnimo kod u linijama 7-9 a to je *zamjena dvije vrijednosti preko pomoćne*.

✓ Zamjena vrijednosti dvije promjenljive

Da bismo ovo objasnili uzmimo primjer koji ne koristi niz. Recimo da imamo promjenljive `a` i `b` čije vrijednosti trebamo zamijeniti. Npr. ako je bilo `a=3` i `b=5`, nakon zamjene treba biti `a=5` i `b=3`. Kako to uraditi? Uvešćemo privremenu promjenljivu `temp` (od eng. temporary).

```
temp = a;
a = b;
b = temp;
```

U prvoj liniji smo kopirali sadržaj promjenljive `a` u ovu pomoćnu promjenljivu `temp`. U drugoj smo kopirali `b` u `a`. `a` je sada prebrisana, ali se njen raniji sadržaj i dalje nalazi u `temp`. U trećoj liniji smo tu vrijednost kopirali u `b` čime su promjenljive zamijenjene. Nakon toga možemo slobodno zanemariti vrijednost promjenljive `temp`.

Postoje i drugi načini zamjene bez korištenja pomoćne promjenljive, ali ovi načini polaze od pretpostavke da su promjenljive brojevi i ne

1777098

rade npr. za stringove.

5.3 *Višedimenzionalni nizovi*

Nizovi sa kojima smo do sada radili imali su samo jednu dimenziju koju smo nazivali *dužina* tj. *broj elemenata* niza. No nizovi mogu imati i više dimenzija. Npr. dvodimenzionalni niz može imati dužinu i širinu, trodimenzionalni niz bi imao dužinu, širinu i visinu itd. Već kod četverodimenzionalnih nizova ova intuicija pada u vodu jer je naš svijet trodimenzionalan i teško nam je zamisliti četvrtu dimenziju osim na način da je poistovjetimo s vremenom. U programerskim problemima mogu se javiti i problemi sa pet ili više dimenzija. Recimo trodimenzionalan niz može biti *niz matrica* a ne nužno jedan 3D prostor.

Ovdje problem predstavlja i korištenje riječi *niz* kao prijevoda za engleski *array*. Riječ "array" u rječnicima ćete obično vidjeti prevedenu sa "polje", pa tako i većina domaćih autora za "array" u programiranju koristi termin "polje". No u programiranju javlja se i riječ *field* (koja označava jedan član *zapisa* ili *sloga*) pa se postavlja pitanje kako nju onda prevesti a da ne dođe do zabune? S druge strane, prije upotrebe u programiranju riječ *array* se u engleskom jeziku isključivo koristila za dvodimenzionalne objekte (matrice) pa se i tu pravi slična greška kao što i mi radimo kada koristimo termin *niz*.

Da bismo izbjegli zabunu 2D nizove ćemo ubuduće zvati **matrice**, a nizovi sa tri ili više dimenzija su svakako dosta rijetke zvjerke.

✓ Matrice kao 2D nizovi

Nizove sa ogromnim brojem dimenzija treba izbjegavati i iz drugih razloga. Recimo ako imamo niz sa 100 dimenzija pri čemu je svaka veličine 2 (minimalne koja ima smisla), koliko ustvari elemenata ima takav niz? Vidimo da je vrlo lako zauzeti svu dostupnu memoriju jer zauzeće memorije raste eksponencijalno sa brojem dimenzija. Često je bolje pokušati drugačije osmisliti problem tako da nam nije potrebno više od 3 dimenzije.

Interesantno je da sam C standard ne ograničava broj dimenzija, ali daje se preporuka da bi prevodioci trebali podržati "barem 256" dimenzija, pa je to još jedan razlog zašto treba izbjegavati veliki broj dimenzija.

Opšti oblik deklaracije niza sa proizvoljnim brojem dimenzija bi glasio:

```
tip naziv[d1][d2][d3]...;
```

17/57/01 98

Ovdje `tip` predstavlja tip elemenata niza, `naziv` je kao i ranije proizvoljan, a `d1`, `d2`... su dimenzije (širina, dužina, visina, vrijeme...). Za razliku od nekih drugih programskih jezika, u C-u se svaka dimenzija navodi u zasebnom paru uglastih zagrada. Recimo da želimo deklarisati 2D niz (matricu) cijelih brojeva dimenzija 3x5. Deklaracija bi

glasila:

```
int mat[3][5];
```

Dakle, najprije navodimo broj redova (visinu), a zatim broj kolona (širinu). Ta matrica bi mogla izgledati recimo ovako:

−18	3	21	7	0
16	−1	−212	88	1593
32	99	7	0	1

Ovim smo deklarisali 15 cjelobrojnih promjenljivih koje mogu imati različite vrijednosti, po želji. Kako onda pristupamo tim promjenljivim? Njihova imena su:

`mat[0][0]`	`mat[0][1]`	`mat[0][2]`	`mat[0][3]`	`mat[0][4]`
`mat[1][0]`	`mat[1][1]`	`mat[1][2]`	`mat[1][3]`	`mat[1][4]`
`mat[2][0]`	`mat[2][1]`	`mat[2][2]`	`mat[2][3]`	`mat[2][4]`

Kod 1D nizova, vrijednost njihove dimenzije je bila na intervalu $[0, N)$ gdje je N veličina koju smo definisali prilikom deklarisanja. Slično tome, kod matrice širina je u intervalu $[0, S)$ a visina u intervalu $[0, V)$. Generalno, elementima n-dimenzionalnog niza pristupamo tako što variramo neku i-tu dimenziju između 0 i veličine i-te dimenzije umanjene za 1.

Ovo variranje možemo postići n-terostrukom for petljom. U pravilu, za rad sa n-dimenzionalnim nizom potrebna nam je n-terostruka petlja. Npr. da bismo unijeli sve elemente matrice deklarisane iznad koristićemo dvostruku for petlju:

```
1  #include <stdio.h>
2  #define VISINA 3
3  #define SIRINA 5
4  int main() {
5    int i, j, mat[VISINA][SIRINA];
6    printf("Unesite matricu %dx%d:\n", VISINA, SIRINA);
7    for (i=0; i<VISINA; i++) {
8      for (j=0; j<SIRINA; j++) {
9        printf("Unesite clan [%d][%d]: ",i,j);
10       scanf("%d", &mat[i][j]);
11       if (mat[i][j] <= -10000 || mat[i][j] >= 10000) {
12         printf("Nije na intervalu\n");
13         j--;
14       }
15     }
16   }
17   ...
```

1757/0198

(Ovaj program će se nastaviti na sljedećim stranicama, pa ga završavamo sa tri tačke.)

Proanalizirajmo kod dat iznad. Najprije smo definisali dvije konstante koje prestavljaju visinu i širinu kako bismo lakše prilagodili program za matrice različitih dimenzija. U liniji 5 smo deklarisali dvije

cjelobrojne promjenljive `i` i `j` koje nam predstavljaju kontrolne promjenljive za for petlje, te matricu `mat` čije su dimenzije date konstantama. Linije 7 i 8 predstavljaju dvostruku for petlju o kojoj smo govorili. U liniji 10 unosi se element matrice na koordinatama `i` i `j` kao `mat[i][j]`. To je obična promjenljiva tipa `int`, pa je unosimo koristeći format `%d`.

Uslov u linijama 11-14 služi da osigura kako bi uneseni broj imao najviše četiri cifre (ne računajući eventualni predznak minus). Ako broj ima više od četiri cifre, ispisaće se poruka "Nije na intervalu" a zatim će se tražiti ponovni unos istog člana matrice. Ovo postižemo naredbom `j--` koja ponavlja unos na sličan način kao što smo vidjeli u poglavlju 5.2.2.↑ Obratite pažnju: da smo stavili `i--`, unos bi skočio na prethodni red matrice. Dakle, ponavljamo krajnje unutrašnju petlju.

↑ Str. 150.

Recimo da sada želimo ispisati lijepo matricu na ekran u formi tabele. Za ispis matrice ponovo će nam koristiti dvostruka petlja. Mogli bismo pokušati uraditi ovako:

```
18  for (i=0; i<VISINA; i++) {
19    for (j=0; j<SIRINA; j++) {
20      printf("%d", mat[i][j]);
21    }
22  }
```

Ovakav kod bi prouzročio ispis poput sljedećeg:

```
183217016-1-2128815933299701
```

Da bismo shvatili zašto, podsjetimo se primjera sa ispisom tablice množenja koji smo imali u poglavlju 4.1.7, zadatak 4.↑ Prvi problem je u tome što nismo ostavljali nikakav razmak između članova matrice. Da smo dodali jedan razmak u `printf` naredbi dobili bismo:

↑ Str. 103.

```
18 3 21 7 0 16 -1 -212 88 1593 32 99 7 0 1
```

jer nismo prelazili u novi red nakon svakog reda matrice. Potrebno je dodati ispis znaka `\n` i to u vanjsku for petlju jer unutrašnja petlja `j` ispisuje na ekranu jedan red, dok vanjska petlja `i` služi za prolazak kroz redove matrice. Dakle kod treba glasiti ovako:

```
18  for (i=0; i<VISINA; i++) {
19    for (j=0; j<SIRINA; j++) {
20      printf("%d ", mat[i][j]);
21    }
22    printf("\n");
23  }
```

Čime dobijamo:

```
18 3 21 7 0
16 -1 -212 88 1593
32 99 7 0 1
```

1 / 5 / 0 1 9 8

Ovo je već puno bolje. Da bismo dobili lijepo formirane kolone, podsjetimo se da su svi članovi matrice četverocifreni brojevi. To nam

ukazuje da možemo umjesto `%d` koristiti format `%5d` koji uvijek rezerviše 5 mjesta za ispis bez obzira koliko cifara ima broj (preostala mjesta lijevo od broja će biti popunjena razmacima). Kada stavimo format `%5d`, dobićemo izlaz:

```
   18    3   21    7    0
   16   -1 -212   88 1593
   32   99    7    0    1
```

koji izgleda onako kako želimo.

5.3.1 *Algoritmi za rad sa matricama*

U nastavku ćemo se potpuno fokusirati na matrice i dati neke primjere zadataka u kojima se koristi matrica.

Algoritmi primijenjeni nad višedimenzionalnim nizovima su u suštini vrlo slični onima koje smo koristili ranije, samo se trebamo sjetiti da koristimo n-terostruku petlju. Recimo, ako se traži suma svih članova matrice, napisaćemo:

```
18  suma = 0;
19  for (i=0; i<VISINA; i++) {
20    for (j=0; j<SIRINA; j++) {
21      suma += mat[i][j];
22    }
23  }
```

Da bismo dobili sada srednju vrijednost svih članova matrice, dijelimo ovu sumu sa brojem članova matrice, a taj broj je `VISINA*SIRINA`:

✓ Srednja vrijednost članova matrice

```
24  prosjek = (double) suma / (VISINA*SIRINA);
```

Obratite pažnju na zagrade zbog prioriteta operatora. Sjetimo se također da je matrica tipa `int`, pa je i njena suma prirodno tipa `int`, a `VISINA` i `SIRINA` su svakako cjelobrojne konstante (drugačije ne mogu ni biti jer ne možemo koristiti realne brojeve za dimenziju niza), pa moramo dodati kalup↑ (operator eksplicitne konverzije tipa) `(double)` kako ne bi došlo do zaokruživanja.

↑ Poglavlje 3.1.8, str. 76.

Ako želimo recimo srednju vrijednost svih članova matrice koji su parni brojevi, kao u zadatku 1 (poglavlje 5.1.3) pišemo:↑

↑ Str. 136.

```
18  suma = br_parnih = 0;
19  for (i=0; i<VISINA; i++) {
20    for (j=0; j<SIRINA; j++) {
21      if (mat[i][j] % 2 == 0) {
22        suma += mat[i][j];
23        br_parnih++;
24      }
25    }
26  }
27  prosjek = (double) suma / br_parnih;
```

17.09.98

Šta ako se traži prosjek svih članova čije su obje koordinate (`i` i `j`) parni brojevi? Onda bismo pisali:

```
18  suma = br_parnih = 0;
19  for (i=0; i<VISINA; i++) {
20    for (j=0; j<SIRINA; j++) {
21      if (i % 2 == 0 && j % 2 == 0) {
22        suma += mat[i][j];
23        br_parnih++;
24      }
25    }
26  }
27  prosjek = (double) suma / br_parnih;
```

Mada kraće rješenje bi bilo da u petljama koristimo `i+=2` i `j+=2`.

Kod kvadratne matrice, definišemo *glavnu dijagonalu* kao one elemente koji se nalaze na takvim koordinatama `i` i `j` da važi `i=j`. Kako ćemo izračunati sumu svih elemenata na glavnoj dijagonali? Mogli bismo pisati ovako:

✓ Glavna i sporedna dijagonala matrice

```
1  #include <stdio.h>
2  #define SIRINA 5
3  int main() {
4    int i, j, suma=0, mat[SIRINA][SIRINA];
5    for (i=0; i<SIRINA; i++)
6      for (j=0; j<SIRINA; j++)
7        scanf("%d", &mat[i][j]);
8
9    for (i=0; i<SIRINA; i++)
10     for (j=0; j<SIRINA; j++)
11       if (i==j)
12         suma += mat[i][j];
13
14   printf("Suma elemenata na glavnoj dijagonali je %d\n", suma);
15   return 0;
16 }
```

I ovaj program bi davao potpuno tačne rezultate, ali moglo bi se to i kraće zapisati. Ako znamo da je `i=j`, onda nam uošte nisu potrebne dvije petlje, nego možemo sumu izračunati u jednoj petlji, na sljedeći način:

```
9   for (i=0; i<SIRINA; i++)
10    suma += mat[i][i];
```

Kako ovaj program radi? U prvom prolasku kroz petlju `i` je 0, pa uzimamo element `mat[0][0]`. U drugom prolasku `i` je 1 pa na sumu dodajemo `mat[1][1]` itd. Kakav bi oblik imali elementi na sporednoj dijagonali, i da li možemo i to uraditi u jednoj petlji?

Zadatak 11. *Napisati program koji omogućuje korisniku da unese kvadratnu matricu, a zatim provjerava da li je unesena matrica gornja trougaona.*

1755701983

Matrica se definiše kao *gornja trougaona* ako su svi elementi ispod glavne dijagonale jednaki nuli.[8] Moguće je i da neki elementi iznad ili na dijagonali budu nula, to ne mijenja činjenicu da je matrica gornja trougaona. Drugim riječima, nul-matrica (matrica sastavljena od nula) je ujedno i gornja i donja trougaona matrica.

✓ Gornja i donja trougaona matrica

[8] Analogno, matrica je donja trougaona ako su svi elementi iznad glavne dijagonale jednaki nuli.

Da bismo lakše riješili zadatak, ilustrovaćemo gornju trougaonu matricu na slici 5.2.

	j=0	j=1	j=2	j=3	j=4
i=0	-18	3	21	7	0
i=1	0	-1	-212	88	1593
i=2	0	0	7	0	1
i=3	0	0	0	32	16
i=4	0	0	0	0	99

Slika 5.2: Gornja trougaona matrica

Vidimo da `i` i `j` čine koordinatni sistem čije je ishodište u gornjem lijevom uglu matrice, a vrijednosti koordinata rastu prema lijevo i dolje – baš kao i kod zadatka 7 u poglavlju 4.4.1. Možemo zaključiti kako su nam zadaci sa crtanjem dosta olakšali rad sa matricama i učinili ga lakše shvatljivim, mada su se tada činili poprilično nepotrebnim.

Dakle, sada je lako uočiti da je matrica gornja trougaona ako važi:

$$mat[i][j] = 0, \quad \forall i > j; 0 \leqslant i, j < SIRINA$$

Vidimo da se u formuli javlja znak $\forall$, pa zaključujemo da je ovo tip zadatka "za svaki",↑ odnosno trebamo provjeriti da li svi članovi matrice ispunjavaju uslov. Podsjetimo se, provjera se vrši tako što:

↑ Poglavlje 5.1.4, str. 137.

- pretpostavimo da je uslov ispunjen tako što ćemo postaviti logičku promjenljivu na 1;
- prođemo kroz sve članove i ako naiđemo na takav član da uslov nije ispunjen, postavimo promjenljivu na 0 i prekidamo petlju;
- ako uslov nikada nije bio ispunjen, logička promjenljiva će zadržati vrijednost 1.

Kôd rješenja bi glasio:

```
#include <stdio.h>
#define SIRINA 5
int main() {
  int i, j, suma=0, gornja_trougaona, mat[SIRINA][SIRINA];
  for (i=0; i<SIRINA; i++)
    for (j=0; j<SIRINA; j++)
      scanf("%d", &mat[i][j]);

  gornja_trougaona=1;
  for (i=0; i<SIRINA; i++)
    for (j=0; j<SIRINA; j++)
      if (i>j && mat[i][j] != 0)
        gornja_trougaona=0;

  if (gornja_trougaona)
    printf("Gornja trougaona\n");
```

```
17  else
18    printf("Nije gornja trougaona\n");
19  return 0;
20 }
```

Prije svega, u liniji 2 možemo uočiti da je definisana samo jedna konstanta jer je matrica kvadratna (uzeli smo da se ta konstanta zove `SIRINA` bez nekog posebnog razloga). Linije 5-7 služe za unos matrice. U liniji 9 postavili smo logičku promjenljivu na 1. Dvostruka petlja u linijama 10 i 11 prolazi kroz sve članove matrice. Uslovom u liniji 12 ograničili smo se na elemente ispod glavne dijagonale te provjeravamo da li su oni nula. Konačno u linijama 15-18 provjeravamo uslov i ispisujemo odgovarajuću poruku.

Uočavamo da ako je uslov ispunjen postavljamo logičku promjenljivu na nulu ali nismo prekinuli dvostruku petlju! Kako uopšte prekinuti dvostruku petlju? Možemo koristiti naredbu `goto` čija primjena nije preporučljiva kao što smo rekli ranije.↑ Možemo uvesti neku novu logičku promjenljivu `prekid` i dodati u uslove for petlje. U našem slučaju nije nam potrebna nova promjenljiva jer imamo `gornja_trougaona`, pa možemo pisati npr. ovako:

↑ Poglavlje 2.6, str. 61.

```
10  for (i=0; i<SIRINA && gornja_trougaona; i++)
11    for (j=0; j<SIRINA && gornja_trougaona; j++)
12      if (i>j && mat[i][j] != 0)
13        gornja_trougaona=0;
```

Možemo i postaviti promjenljive `i` i `j` na njihovu finalnu vrijednost:

```
10 for (i=0; i<SIRINA; i++)
11   for (j=0; j<SIRINA; j++)
12     if (i>j && mat[i][j]!=0) {
13       gornja_trougaona = 0;
14       i=j=SIRINA;
15     }
```

ali na ovaj način ne možemo nakon petlje znati da li je došlo do prekida ili nije preko vrijednosti promjenljivih `i` odnosno `j`. U svakom slučaju, takvo rješenje bi bilo poprilično "neobično" za nekoga ko nije iskusan C programer, pa ga je bolje izbjegavati.

Još jedna stvar koju možemo uočiti jeste da `j` petlja ne mora ići do širine matrice jer nas interesuju samo vrijednosti manje od `i`. Dakle ovaj fragment se može još jednostavnije napisati na sljedeći način:

```
10 for (i=0; i<SIRINA; i++)
11   for (j=0; j<i; j++)
12     if (mat[i][j]!=0)
13       gornja_trougaona = 0;
```

17570198

5.4 C++ vektori

Pored nizova, programski jezik C++ nudi više vrsta tzv. *kolekcija*, tj. tipova čija je svrha da sadrže veći broj elemenata nekog drugog

tipa, dakle koji su nalik na niz. Ovi tipovi se međusobno razlikuju po načinu rada, performansama raznih operacija itd. Detaljan opis svakog od njih zahtijevao bi poglavlje veliko najmanje kao ovo. Stoga ćemo se zadržati na tome da opišemo najjednostavniji takav tip: **vektor**.

5.4.1 *Rad sa vektorima*

Da biste deklarisali vektor, najprije je potrebno da uključite biblioteku `vector`. Na mjestu deklaracije navodite naziv tipa `std::vector`, zatim znak manje-od, tip članova vektora te veće-od, nakon čega slijedi ime vektora, te u *malim* zagradama broj članova vektora. Nakon toga vektor koristite na isti način kao niz.

Treba napomenuti da u ovoj deklaraciji oznake < i > nazivamo "šiljaste zagrade" (eng. angle brackets) i one služe da se označi tip članova vektora, a nemaju nikakve veze sa matematičkim operacijama manje-od i veće-od.

Primjer upotrebe možete vidjeti u sljedećem programu:

```
1  #include <iostream>
2  #include <vector>
3
4  int main() {
5    std::vector<int> vek(10);
6    std::cout << "Unesite deset clanova vektora: ";
7    for (int i=0; i<10; i++)
8      std::cin >> vek[i];
9    ...
10   return 0;
11 }
```

U liniji 5 deklarisan je vektor od 10 elemenata tipa `int` pod imenom `vek`. U linijama 7-8 vidimo da se sa ovim vektorom dalje radi kao sa nizom cijelih brojeva.

Za razliku od nizova, *moguće je* deklarisati vektor čija je veličina data promjenljivom. Pogledajte ovaj primjer:

```
#include <iostream>
#include <vector>

int main() {
  int n;
  std::cout << "Unesite broj clanova vektora: ";
  std::cin >> n;
  std::vector<int> vek(n);
  std::cout << "Unesite " << n << " clanova vektora: ";
  for (int i=0; i<n; i++)
    std::cin >> vek[i];
  ...
  return 0;
}
```

Prilikom deklarisanja vektora možete koristiti inicijalizaciju u vitičastim zagradama (kao i kod nizova), ili pak možete zadati element kojim

će biti popunjen čitav vektor. Recimo da želite da imate vektor od 100 slova 'a' mogli biste pisati `std::vector<char> slova(100, 'a');`.

Kod vektora je moguće koristiti i inicijalizacijske liste, koje se navode neposredno nakon deklaracije vektora, na sljedeći način:

```
std::vector<int> v{1,2,3,4,5};
```

5.4.2 *Promjena veličine vektora*

Veličina vektora se može mijenjati za vrijeme rada programa, i to na dva načina! Prvi način je korištenje funkcije `resize`, što se radi na sljedeći način:

```
12  std::cout << "Unesite novu velicinu vektora: ";
13  std::cin >> n;
14  vek.resize(n);
```

Ako je nova veličina vektora manja od prethodne, biće sačuvano prvih `n` elemenata a suvišni elementi će biti obrisani. Ako je veća, novi članovi vektora će biti popunjeni *default vrijednostima* za odgovarajući tip, npr. za numeričke tipove to je u pravilu nula. Dodajmo da postoji i varijanta funkcije `resize` sa dva parametra, pri čemu je drugi parametar vrijednost kojom će biti popunjeni novi članovi vektora.

Drugi način promjene veličine vektora je povećavanje za jedan dodavanjem novog elementa na kraj. To nam omogućuje da radimo sa vektorima čak i ako nam nije unaprijed poznata njihova veličina! Pogledajmo sljedeći primjer koji je dosta sličan ranijim primjerima sa nizovima proizvoljne veličine:

```
1  #include <iostream>
2  #include <vector>
3
4  int main() {
5    std::vector<int> vek;
6    std::cout << "Unesite prirodne brojeve (-1 za kraj): ";
7    do {
8      int broj;
9      std::cin >> broj;
10     if (broj != -1) vek.push_back(broj);
11   } while (broj != -1);
12   std::cout << "Unijeli ste " << vek.size() << " brojeva." << std::endl;
13   ...
14   return 0;
15 }
```

U liniji 5 deklarisan je vektor `vek` bez navođenja veličine, tj. njegova veličina će biti nula! Kako raditi sa takvim vektorom? To je vidljivo u liniji 10 gdje koristeći funkciju `push_back` dodajemo novi član u vektor. Nakon poziva ove funkcije veličina vektora će se uvećati za jedan, pa će nakon prvog unesenog broja biti jedan itd. Obratite pažnju da ne možemo direktno unositi `vek[i]` kao u ranijim primjerima jer bi tu `i` bilo veće od trenutne veličine vektora. Stoga najprije deklarišemo

1/11/2018

pomoćnu promjenljivu `broj` i nju unosimo (linije 8-9), a zatim takav broj dodajemo u vektor pod uslovom da je različit od -1.[9]

[9] Što znači da sam vektor može biti i tipa `unsigned int`.

Obratite pažnju i da mi ne moramo čuvati broj unesenih elemenata tj. broj članova vektora u nekoj posebnoj promjenljivoj. Umjesto toga možemo pozvati funkciju `size` što vidimo u liniji 12. Ovo nam omogućuje i da jednostavno obavimo neke operacije npr. sumu članova bismo odredili ovako:

```
14  int suma=0;
15  for (int i=0; i<vek.size(); i++)
16    suma += vek[i];
17  std::cout << "Suma je: " << suma << std::endl;
18  ...
```

Postoji i funkcija `pop_back` koja vraća posljednji element vektora, a zatim smanjuje veličinu vektora za 1:

```
std::cout << "Elementi ispisani unazad su: ";
for (int i=0; i<vek.size(); i++)
  std::cout << vek.pop_back();
```

Kod naveden iznad će ispisati samo otprilike pola članova vektora. Razlog za to je što se broj članova vektora mijenja u samoj petlji, pa npr. ako je vektor imao 5 elemenata, nakon prvog prolaza `i` će postati 1 a `vek.size()` će postati 4, nakon drugog će `i` biti 2 a `vek.size()` 3 itd. te se petlja završava nakon svega tri prolaza. Rješenje je da ili sačuvate vrijednost `vek.size()` u pomoćnu promjenljivu koju ćete koristiti kao uslov petlje, ili da koristite sljedeći oblik petlje:

```
18  std::cout << "Elementi ispisani unazad su: ";
19  while (vek.size() > 0)
20    std::cout << vek.pop_back();
```

Treći ispravan način bi bio da koristite metodu `vek.empty()` koja vraća logičku istinu ako je vektor prazan.

Pitaćete se sada, pa šta je sa pričom o ograničenoj količini memorije? Stvar je u tome da kada deklarišete `vector` koristi se dinamička alokacija memorije↑ kako bi se u memoriji stvorio niz, a taj niz je ustvari znatno veći nego što je dato prilikom deklarisanja. Stvarno zauzeće memorije vektora možete saznati funkcijom `capacity`.[10]

↑ Poglavlje C.2, str. 367.

[10] Način rada sa memorijom tipa `vector` nije definisan standardom, te se osjetno razlikuje ovisno od korištenog prevodioca.

Kada se korištenjem funkcija `resize` ili `push_back` premaši zauzeta količina memorije, biće kreiran *novi niz* sa veličinom većom od starog (npr. dvostruko), elementi će biti prebačeni iz starog u novi niz, te će stari niz biti obrisan. Ovo je dosta spora operacija, pa pri radu sa vektorom možete u nekim trenucima primijetiti neobjašnjivo usporavanje kod promjene veličine. Jedan način kako se to može izbjeći je funkcija `vek.reserve(n);` koja će unaprijed rezervisati memoriju za `n` članova vektora, pa sve dok vektor ima manje od `n` članova neće doći do realokacije.

1755098

5.4.3 *Osnovne operacije sa vektorima*

Za razliku od nizova vektori se mogu međusobno dodjeljivati i upoređivati. Pogledajte sljedeći isječak koda:

```
std::vector<double> original, kopija;
...
kopija = original;
```

Vektori moraju biti *istog tipa* (u ovom slučaju `double`), ali ne moraju biti iste veličine. Svi elementi koji su se ranije nalazili u vektoru `kopija` će biti obrisani, te će ovaj vektor postati iste veličine kao `original`. Možete i porediti dva vektora operatorom == recimo sljedeći kod bi trebao da provjeri da li su vektori `vek1` i `vek2` iste veličine i sadrže iste elemente istim redoslijedom:

```
bool isti_su = true;
if (vek1.size() == vek2.size()) {
  for (int i=0; i<vek1.size(); i++)
    if (vek1[i] != vek2[i])
      isti_su = false;
  }
} else
  isti_su = false;
if (isti_su) {
  ...
}
```

Ovo možemo kraće napisati kao samo:

```
if (vek1 == vek2) {
  ...
}
```

Interesantno da su dostupni i ostali operatori poređenja < > <= >= koji vrše *leksikografsko poređenje* članova vektora, odnosno sekvencijalno prolaze kroz oba vektora i zatim porede svaki član sa svakim, te se zaustavljaju kod prvog para članova za koje važi uslov poređenja.

Izuzev stvari koje smo istakli u ovom poglavlju, sa vektorima se radi na isti način kao i sa nizovima, tako da svi algoritmi za rad sa nizovima koje smo opisali ranije važe i za C++ `vector`.

5.4.4 *"Siguran" pristup članovima vektora*

Kao i kod nizova, kada pristupate članu vektora koristeći operator uglastih zagrada `vek[i]`, ako je `i` izvan opsega veličine vektora dolazi do *nedefinisanog ponašanja* odnosno može se desiti bilo šta. Zbog toga vektorski tip u C++u nudi i funkciju `at` koja predstavlja "siguran" pristup članovima vektora:

```
do {
  std::cout << "Unesite indeks u vektoru ili -1 za kraj: ";
  std::cin >> n;
  if (n != -1) std::cout << n << "-ti clan vektora je: " << vek.at(n);
} while (n != -1);
```

1715701198

Ako unesete broj manji od -1 ili veći od broja elemenata, program dat iznad će se *sigurno krahirati*. Ustvari, biće bačen *izuzetak* – posebna vrsta kontrolne strukture u programskom jeziku C++ koja nam omogućuje da upravljamo nepredviđenim situacijama. Izuzeci se mogu uhvatiti koristeći *try-catch* blok. Ako ovaj blok nije naveden, program će se "krahirati" odnosno prinudno će prekinuti s radom. Pomoću *try-catch* strukture možete postići da se desi nešto drugo npr. da se ispiše neka poruka ili da se zatraži da korisnik ponovo unese `n`. Upravljanje izuzecima izlazi van opsega ove knjige tako da ćemo se ovdje zaustaviti.

5.4.5 *Višedimenzionalni vektori*

Višedimenzionalni vektori kao takvi ne postoje, nego se realiziraju kao "vektor vektorâ", odnosno vektor čiji su članovi drugi vektori. Pogledajmo primjer matrice izvedene koristeći vektore:

```
1  #include <iostream>
2  #include <vector>
3
4  int main() {
5    int sirina, visina;
6    std::cout << "Unesite sirinu i visinu matrice: ";
7    std::cin >> sirina >> visina;
8    std::vector<std::vector<double>> matrica(visina, std::vector<double>(
       sirina));
9
10   std::cout << "Unesite clanove matrice: ";
11   for (int i=0; i<visina; i++) {
12     for (int j=0; j<sirina; j++) {
13       std::cin >> matrica[i][j];
14     }
15   }
16   ...
17   return 0;
18 }
```

Deklaracija matrice u liniji 8 može biti konfuzna pa ćemo je dodatno razjasniti. Kada deklarišemo vektor unutar znakova < i > navodimo tip svakog pojedinačnog člana vektora. U ovom slučaju, taj tip je `std::vector<double>` odnosno svaki član vektora je jedan vektor tipa `double`.

Nakon toga slijedi naziv vektora `matrica`, te u zagradi dva parametra. Kao što znamo, prvi parametar je veličina vektora, a to će u našem slučaju biti broj redova matrice tj. njena visina. Drugi parametar je vrijednost kojom će biti popunjen vektor. U ovom slučaju ta vrijednost će biti `std::vector<double>(sirina)` a to je vektor tipa `double` koji ima `sirina` elemenata.

Mada ovo može djelovati konfuzno, u linijama 11-15 vidimo da se sa ovom matricom radi kao i sa C matricama koje smo upoznali ranije.↑ To ujedno znači da možemo koristiti i sve ranije opisane algoritme za

↑ Poglavlje 5.3, str. 160.

1755/0198

rad sa matricama.

Pitanja za razumijevanje gradiva

1. Ako je veličina niza N, koji je indeks prvog člana u nizu, a koji posljednjeg?

2. Šta će se desiti ako izađemo iz opsega niza?

3. Na koje sve načine možemo inicijalizirati sve članove niza na nulu?

4. Koje vrste grananja poznajemo?

5. Kako glasi algoritam "za svako x"?

6. Da li je moguće deklarisati niz proizvoljne dužine u C-u i na koji način?

7. Koji su koraci algoritma za pronalazak najvećeg člana u nizu?

8. Na koji način izbacujemo element iz niza uz očuvanje redoslijeda?

9. Šta je to sortiranje i kako se ono u principu radi?

10. Šta su to matrice, kako se deklarišu i koriste?

11. Na koji način provjeravamo da li je matrica gornja trougaona?

12. Nabrojati razlike između C++ vektora i niza.

Zadaci za vježbu

Zadatak 12. *Napisati program koju učitava 5 elemenata dva niza realnih brojeva A i B, a zatim izračunava elemente trećeg niza C zadatog kao:* $C \equiv \{c_i \in \mathbb{R} | c_i = 2a_i + b_i, a_i \in A, b_i \in B\}$. *Drugim riječima, svaki element* c_i *skupa C jednak je sumi odgovarajućih elemenata* a_i *i* b_i *iz skupova A i B, pri čemu je* a_i *pomnožen sa dva. Na kraju je potrebno ispisati sve elemente nizova A, B i C.*

Zadatak 13. *Napišite program koji omogućuje korisniku da unese proizvoljan broj cijelih brojeva sa tastature, pri čemu broj -1 označava kraj unosa. Zatim se na ekranu ispisuje koliko unesenih brojeva je djeljivo sa 5, sa 7 i sa 11.*

1757019 8

Zadatak 14. *Prepravite prethodni program na sljedeći način: dozvoljeno je unositi brojeve između 0 i 100. Ukoliko se unese broj van ovog opsega, ponavlja se unos. Nakon što korisnik unese broj -1, treba za svaki broj napisati koliko puta se ponavlja. Brojeve koji se ne javljaju nijednom na ulazu ne treba ispisivati.*

Zadatak 15. *Napisati program koji najprije traži da se unesu dva niza pozitivnih cijelih brojeva A i B. Unos niza se prekida kada korisnik unese broj -1, a maksimalan broj elemenata u nizovima je 10. Nakon unosa, nizove A i B je potrebno spojiti u novi niz C. Konačno, tako formirani niz C treba ispisati na ekranu. Ovaj zadatak je vrlo jednostavno riješiti bez korištenja niza C, no za vježbu pokušajte ga uraditi tačno po koracima koji su dati iznad.*

Zadatak 16. *Napisati program koji učitava matricu cijelih brojeva dimenzija 10x10 te pronalazi najmanji element na glavnoj dijagonali.*

Zadatak 17. *Napravite program koji učitava realnu matricu dimenzija A×B te pronalazi sumu elemenata na rubu matrice. Korisnik najprije unosi dimenzije A i B (ne veće od 100) a zatim unosi elemente matrice. Rub matrice sačinjavaju elementi prvog i zadnjeg reda, te elementi prve i zadnje kolone. Pri tome pazite da se elementi u ćoškovima ne budu dvaput uračunati u sumu!*

Zadatak 18. *Napravite program učivata matricu cijelih brojeva dimenzija 10x10, a zatim pronalazi red sa najvećim i red sa najmanjim elementom. Ako se oba elementa nalaze u istom redu, program treba ispisati poruku "Najveci i najmanji element se nalaze u istom redu" i završiti s radom. U suprotnom, potrebno je zamijeniti mjesta ta dva reda u matrici i ispisati tako modifikovanu matricu na ekranu.*

17570198

1757018 8

6
Funkcije

U ovom poglavlju obradićemo funkcije u jeziku C koje uveliko olakšavaju čitanje, pisanje i održavanje kôda, te umanjuju mogućnost pravljenja grešaka. Pisanjem funkcija zaista olakšavamo sebi i drugima, a glavna svrha funkcija je ponovna iskoristivost: funkcije možemo pozivati neograničen broj puta sa različitim ulazima u cilju dobivanja različitih izlaza. Moguće je da funkcija izvršava neku akciju iako ne prihvata ulaze, ili čak ako nema izlaze (primjer bi bilo ispisivanje na ekran), i takva funkcija je korisna jer se ta radnja (npr. ispisivanje) može iskoristiti više puta. Najprije ćemo se upoznati sa deklaracijom funkcija prije nego pređemo na diskusiju o tome zašto su funkcije korisne, kada ih treba, a kada ne treba koristiti.

6.1 Uvod u funkcije u C-u

Iako funkcije u programiranju možemo porediti sa matematskim funkcijama jer imaju dosta sličnosti, ipak postoje neke fundamentalne razlike koje zbunjuju početnike u programiranju. Matematska funkcija je definisana kao relacija koja mapira elemente iz skupa A u skup B. Na sličan način možemo definisati i funkciju u programskom jeziku C (ili bilo kojem proceduralnom jeziku). Funkcije mapiraju određene ulaze na izlaze, pri čemu ovdje imamo veću slobodu, npr. C funkcija može da uopšte nema ulaza ili da nema izlaza (moguće je čak oboje: funkcija bez ulaza i bez izlaza), ili čak da modificira proslijeđene ulaze odnosno mijenja stanje.

✓ *Funkcija je logički grupisan skup naredbi koje zajedno vrše neku akciju. Tu akciju jednostavno možemo ponoviti proizvoljan broj puta ponovnim pozivanjem funkcije.*

✓ Funkcija (u programiranju)

1457019 8

Funkcije možemo posmatrati kao "crne kutije". One mogu imati neke ulaze, te ih transformiraju u određene izlaze. Šta se tačno dešava u "crnoj kutiji" nas nekada ne zanima. Npr. kada koristimo `printf` funkciju, mi ne znamo niti nas treba zanimati šta se tačno dešava i

kako ta funkcija ispisuje tekst na ekran, samo nas zanima kako koristiti tj. pozvati tu funkciju. Ovo se može ilustrirati kao na sl. 6.1.

Slika 6.1: Funkcije nekada zamišljamo kao crne kutije

Neku funkciju možemo pozivati više puta čime smanjujemo vrijeme pisanja i dužinu koda, jer jedan te isti kod (koji može biti dug i po nekoliko stotina linija) zamjenjujemo samo sa imenom funkcije. Ne samo to, funkcije možemo pozivati sa različitim parametrima (ulazima) da dobijemo različite izlaze, ili čak različito ponašanje. Npr. funkcija `printf` se različito ponaša ovisno da li je oznaka formatiranja `%d`, `%f` ili pak `%10.3f`.

6.1.1 *Sintaksa C funkcija*

Da bismo napisali funkciju, potrebno je odrediti nekoliko stvari poput povratnog tipa (tip izlaza funkcije), imena funkcije, broja i tipa parametara (ulazi funkcije), te tijelo funkcije odnosno skup naredbi koji izvršavaju tu akciju. Opšti oblik funkcije je:

```
POVRATNI_TIP ime_funkcije ([Parametri...]) {
  TIJELO FUNKCIJE...
}
```

Prije objašnjenja pojedinih dijelova, pređimo na primjer. Neka je potrebno napisati funkciju koja će računati treći stepen (kub) nekog broja x po izrazu $y = x^3$. Funkciji ćemo dati ime "kubiraj" vodeći se preporukama za davanje jasnih imena funkcijama. Čim neko pročita samo ime funkcije, trebalo bi da zna šta ta funkcija radi! Dalje, ulaz u funkciju će biti jedan broj, x, za koji ćemo izračunati kub (treći stepen). Taj ulaz nazivamo parametrom funkcije, a u ovom slučaju imamo samo jedan parametar (jedan ulaz). Još je potrebno odrediti tip izlaza funkcije (odnosno, zanima nas kojeg će tipa biti rezultat funkcije), te tip ulaza (odnosno, kojeg tipa će biti broj x). Neka to budu cijeli brojevi (odnosno tip `int`), mada se postupak ne bi puno razlikovao ni za druge tipove (npr. realan broj), jer je računanje u ovom slučaju identično.

Konačno, kada smo odredili sve komponente za pisanje funkcije, možemo napisati i njeno tijelo. Ono će se sastojati od višestrukog množenja broja x. Kompletna *definicija funkcije* bi glasila:

✓ Definicija funkcije

```
int kubiraj (int x) {
```

1733/0198

```
  int y = x * x * x;
  return y;
}
```

Ako uporedimo sa opštim oblikom od ranije, vidimo: povratni tip (u ovom slučaju `int`), jasno ime funkcije (`kubiraj`), te u malim zagradama spisak parametara i njihovih tipova. U našem slučaju je to jedan parametar imena `x` tipa `int`. Ukoliko imamo više parametara, njih razdvajamo znakom zarez, što ćemo vidjeti na sljedećem primjeru.

Tijelo funkcije se obavezno nalazi unutar vitičastih zagrada i ono predstavlja ono šta funkcija radi. Prvo smo deklarisali pomoćnu promjenljivu `y`, u nju smjestili rezultat stepenovanja promjenljive `x` (da, ovdje je `x` obična promjenljiva), te smo na kraju vrijednost promjenljive `y` "vratili" iz funkcije pomoću `return` naredbe. Svaka funkcija koja vraća neki rezultat **mora** imati u tijelu funkcije `return` naredbu; o ovome će biti govora kasnije. Funkcija se prekida nailaskom na `return` naredbu – naredbe koje navodimo nakon nje se neće izvršiti:

```
int kubiraj (int x) {
  int y = x * x * x;
  return y;
  printf ("Hello OR!"); /* nece se izvrsiti */
}
```

Znači, nailaskom na `return` naredbu se "izlazi" iz funkcije (i pri tome se "ponese" njen izlaz odnosno rezultat). Bolji prevodioci će vas upozoriti da imate kod koji će biti ignoriran kao u primjeru iznad. Kako iskoristiti ono što je funkcija vratila (izlaz), te kako proslijediti funkciji parametre (ulaze) naučiti ćemo malo kasnije.

Podsjetimo se da smo ranije naredbu `return` koristili za prekid čitavog programa, pošto se naš program sastojao samo od jedne funkcije `main`, pa je njen završetak bio ujedno i završetak programa. Šta ako se nalazimo u nekoj funkciji i želimo da prekinemo program, a da pri tome ne moramo razmišljati odakle je pozvana funkcija? U tom slučaju trebamo koristiti funkciju `exit` koja se nalazi u biblioteci `stdlib.h`. Funkcija ima jedan cjelobrojni parametar čije je značenje isto kao i kod naredbe `return` u `main` funkciji: 0 obično označava uredan završetak programa, a broj različit od nule predstavlja kôd greške čije značenje određuje programer.

✓ Funkcija `exit`

Ako unesete isječak koda dat iznad i probate ga pokrenuti kao program dobićete grešku koja bi mogla izgledati otprilike ovako:

```
(.text+0x20): undefined reference to 'main'
collect2: error: ld returned 1 exit status
```

Uzrok ove greške je činjenica da vaš program *ne sadrži funkciju* `main`. Kao što smo rekli na početku,↑ svaki C program mora sadržavati minimalno jednu funkciju, a to je funkcija `main`. Ova funkcija određuje ono što će se izvršiti kada pokrenemo program. Ako vaš program ne sadrži funkciju `main`, sistem ne zna šta da radi u trenutku pokretanja.

↑ Str. 22.

17/157/0198

Kompletan program koji možemo pokrenuti bi izgledao ovako:

```
#include <stdio.h>
int kubiraj (int x) {
  int y = x * x * x;
  return y;
}
int main() {
  int a = 10;
  int b = kubiraj (a);
  printf ("%d^3 = %d\n", b, a) /* 10^3 = 1000 */
  return 0;
}
```

Najprije uočavamo da se funkcija `kubiraj` nalazi izvan funkcije `main`, preciznije ona se nalazi prije funkcije `main`, a nakon preprocesorskih direktiva. Programski jezik C ne dozvoljava da se funkcija nalazi unutar druge funkcije (ugniježđene funkcije), mada će vam pojedini prevodioci to dozvoliti. Iako će taj program možda raditi, on nije usklađen sa standardom. Uobičajena praksa je da se preprocesorske direktive nalaze na samom vrhu koda, jer ono što je njima dato može biti korisno za sve funkcije. Funkcija se mora najaviti prije mjesta na kojem se poziva (još o tome kasnije).

Nakon naredbe `return` može se nalaziti bilo kakav izraz, to ne mora biti promjenljiva. Dakle, funkciju `kubiraj` možemo napisati bez korištenja pomoćne promjenljive:

```
int kubiraj (int x) {
  return x * x * x;
}
```

6.1.2 *Kako pisati funkcije?*

Prilikom rješavanja programskih zadataka koristeći funkcije, potrebno je da odgovorimo na tri pitanja, što vidimo i sa sl. 6.1:

1. *Šta su ulazi funkcije?* Koliko funkcija ima parametara i kojeg su tipa?
2. *Šta su izlazi funkcije?* Da li funkcija vraća neku vrijednost, kojeg je ona tipa?
3. *Kako od ulaza dobijamo izlaze*, tj. matematičke i druge formule pomoću kojih od ulaza možemo dobiti izlaz.

Zadatak 1. *Potrebno je napisati funkciju "maks" koja će vratiti vrijednost većeg od dva proslijeđena broja. Matematički, to možemo zapisati kao* $y = \max\{a, b\}$. *Npr. ako imamo* $a = 5$, $b = 3$, *funkcija treba da vrati* 5. *U slučaju da su brojevi a i b jednaki, treba biti vraćena vrijednost a ili b, svejedno je.*

175/0198

Slijedeći pitanja koja smo nabrojali ranije, moramo najprije odrediti parametre funkcije. Neka ovaj put želimo tražiti veći od dva *realna*

broja, pa ćemo koristiti tip `double`. Jasno je da su potrebna dva parametra tipa `double`, te da je povratni tip također `double`. Inače, u većini slučajeva u zadacima će biti zadano sa kojim tipovima funkcija treba da radi, pa će biti naglašeno npr. "funkcija koja vraća minimum dva cijela broja". Nekada se čak zadaju kompletne informacije o funkciji (povratni tip, ime, tipovi i broj parametara) što se naziva prototip funkcije, o čemu ćemo kasnije govoriti malo više.

Konačno, napišimo funkciju `maks`.

```
1 double maks (double x, double y) {
2   double rezultat;
3   if (x < y)
4     rezultat = y;
5   else
6     rezultat = x;
7   return rezultat;
8 }
```

U liniji 2 kreirana je privremena promjenljiva `rezultat` i ona je tipa `double`. U nju ćemo smjestiti vrijednost od promjenljive `x` ili `y`, zavisno koja bude veća. Na kraju u liniji 7 izlazimo iz funkcije, te vraćamo vrijednost veće promjenljive. Ovdje je važno da je promjenljiva `rezultat` tipa `double` zbog toga što je i povratni tip funkcije također `double`. Naravno, i promjenljive `x` i `y` su tipa `double`, a njihove vrijednosti smo smještali u `rezultat` promjenljivu, pa je i to jedan od razloga zbog kojeg je njen tip morao biti `double`.

Ova funkcija se mogla napisati i kraće, bez uvođenja pomoćne promjenljive. Moguće je direktno vratiti vrijednosti argumenata `x` ili `y`:

```
double maks (double x, double y) {
  if (x < y)
    return y;
  else
    return x;
}
```

Kao što možemo primijetiti, funkcija može sadržavati više `return` naredbi. Naravno, samo jedna od njih će se izvršiti, što će ujedno značiti i izlazak iz funkcije.

Vrijedno je još napomenuti da će neki prevodioci ovdje prikazati upozorenje tipa "došli smo do kraja funkcije koja vraća rezultat a nema `return` naredbe". Zaista, u velikom broju slučajeva prevodilac ne može znati koji će uslov biti zadovoljen, odnosno, da je moguće da niti jedan uslov ne bude zadovoljen. U primjeru iznad je očigledno da će se izvršiti jedna od dvije naredbe `return`, ali mogući su primjeri gdje to nije ni najmanje očigledno. U tome slučaju mi trebamo promijeniti strukturu grananja kôda u tijelu funkcije da zasigurno bude izvršena neka od više `return` naredbi. Pogledajmo sljedeću funkciju:

1/15/01/98

```
int neka_funkcija (int a, int b) {
  if (a * a < b)
    return 1;
```

```
  else if (a - b > 10)
    return 0;
  else if (a < b)
    return -1;
}
```

Šta je problem? Moguće je da se proslijede takve vrijednosti za `a` i `b` da *niti jedan uslov ne bude ispunjen*, npr. to će se desiti za `a = 5, b = 4`, zbog čega dobijamo upozorenje od prevodioca. Naravno, ukoliko budu proslijeđene vrijednosti zbog kojih se neće izvršiti niti jedna `return` naredba, tada će doći do nedefinisanog ponašanja i posljedice su nepredvidljive. Čak i u nekim slučajevima gdje je *sigurno* da će neki od uslova biti zadovoljen, vrlo vjerovatno će se desiti da prevodilac prikaže upozorenje o nepostojanju "sigurne" `return` naredbe. Pogledajmo.

```
double maks (double x, double y) {
  if (x < y)
    return y;
  else if (x >= y)
    return x;
}
```

Ovdje je *sigurno* da će jedan od dva navedena uslova biti zadovoljen jer uslovi "pokrivaju" sve slučajeve. Ipak, to prevodilac ne može znati jer ovi uslovi mogu biti prilično kompleksni (npr. `x >= y - 5 && y <= x * x -1`), te će biti prikazano upozorenje. Neki jezici (npr. C#, Java) uopšte ne dozvoljavaju ovakve funkcije gdje ne postoji `return` naredba za koju prevodilac zna da će se sigurno izvršiti te se ovi iznad programi u tim jezicima ne bi ni mogli prevesti!

Kako ćemo, onda, prepraviti program da prevodilac ne prikazuje upozorenja? Zasigurno postoji neki slučaj koji će biti zadovoljen ako svi prethodni slučajevi nisu zadovoljeni. Stoga, možemo funkciju `max` napisati na sljedeći način:

```
double maks (double x, double y) {
  if (x < y)
    return y;
  return x; /* sigurno je x maksimalan... */
}
```

Ovdje prevodilac "vidi" jednu `return` naredbu koja nije uslovljena i ne ovisi od ulaza u funkciju te neće prikazati upozorenje. Funkcija se mogla napisati još kraće uz korištenje kondicionalnog operatora:

```
double maks (double x, double y) {
  return x < y ? y : x;
}
```

1//D//0//1//98

Završimo ovo uvodno izlaganje sa još jednim primjerom. Ovaj put će se povratni tip razlikovati od tipa parametra funkcije. Neka je potrebno napisati funkciju *"veliko_slovo"* koja prihvata jedan znak kao parametar a vraća vrijednost 1 ili 0 u zavisnosti da li je proslijeđeni

znak veliko slovo.[1] Povratni tip funkcije će biti `int` (mogao je bilo koji cjelobrojni tip ali je konvencija koristiti tip `int` za logičke vrijednosti), te će broj različit od nule (u našem slučaju 1) predstavljati logičku vrijednost "tačno" (tj. znak je veliko slovo), a broj 0 će biti "netačno" (tj. znak nije veliko slovo).

[1] Ova funkcija postoji u biblioteci ctype.h pod imenom `isupper`.

```
int veliko_slovo (char znak) {
  if (znak >= 'A' && znak <= 'Z')
    return 1;
  return 0; /* mozda broj, znak, malo slovo... */
}
```

Tijelo funkcije bi trebalo biti jasno: ukoliko je znak "veći ili jednak" znaku "A" i "manji ili jednak" znaku "Z" tada proslijeđeni znak predstavlja veliko slovo te funkcija vraća logičku vrijednost "tačno" odnosno broj 1. U svim ostalim slučajevima (znak nije slovo uopšte, znak je malo slovo, broj...) vraćamo logičnu vrijednost "netačno" odnosno broj 0. Oko predstave slova i promjenljivih tipa `char` smo govorili u poglavlju 3.1.7, a biće još riječi i u poglavlju 8.

I ovu funkciju smo također mogli napisati kraće korištenjem kondicionalnog operatora:

```
int veliko_slovo (char znak) {
  return znak >= 'A' && znak <= 'Z' ? 1 : 0;
}
```

Možemo i još kraće, koristeći činjenicu da funkcija vraća logičku vrijednost. Naime, već znamo da se logički izrazi u C-u imaju vrijednost logičke neistine (0) ili istine (ne-0). Npr. izraz `a > b` će imati vrijednost ne-0 ako je `a` veće od `b`, a bit će 0 ako je `a` manje ili jednako `b`. Zahvaljujući tome, nama uopšte ne treba ispitivanje uslova (kondicionalni operator) u funkciji iznad. Možemo direktno "vratiti" rezultat evaluacije izraza:

```
int veliko_slovo (char znak) {
  return znak >= 'A' && znak <= 'Z';
}
```

Prikazani primjer demonstrira funkciju čiji povratni tip se razlikuje od tipa njenog ulaza odnosno tipa parametra.

6.1.3 *Korist od funkcija i potreba za njima*

Početnici u programiranju se često pitaju da li je baš nužno koristiti funkcije ako se sve može uraditi i bez njih? U nastavku ćemo pokušati dati odgovor na ovo pitanje.

Da li smo mi do sada (prije ovog poglavlja) koristili funkcije? Odgovor je potvrdan – već smo koristili funkcije koje su napisali drugi programeri odnosno timovi programera. To su funkcije poput `printf`, `scanf`, `pow`, `sqrt`... Već sada možemo pretpostaviti kakvo olakšanje imamo zahvaljujući funkcijama. Možemo obaviti neke zadatke za koje

17157198

nismo sami napisali kod, koje možda čak i ne razumijemo (npr. kako je napisana `printf` funkcija), i to sve na jednostavan način.

Da li smo ranije napisali neku funkciju, prije ovog poglavlja? Odgovor je i ovdje potvrdan – do sada smo napisali barem jednu funkciju. To je `main` funkcija! Ona je obavezna funkcija za sve C programe i to je startna tačka programa (tu počinje izvršavanje programa). Pored nje, možemo pisati razne funkcije poput iznad napisanih `kubiraj`, `max` i slično. Može se postaviti pitanje "može li se programirati bez funkcija?". Odgovor je da, teoretski. Sve što se isprogramira je moguće staviti u jednu funkciju (kao što je moguće sve petlje u programu zamijeniti samo jednom `while` petljom). A da li to treba tako raditi? Apsolutno ne. Izbjegavanje pisanja funkcija nema gotovo nikakvog smisla, osim kada se želi napisati neki jako ružan i nečitljiv kod i to kada se provode neka takmičenja sa tim kriterijima. Znači, pisanje koda pomoću funkcija je poželjno i jako se često koristi. Postoji nekoliko najvažnijih razloga zašto koristiti funkcije:

- *Bolja čitljivost* i razumljivost programa. Programski kod kojeg vi napišete će vrlo vjerovatno čitati neko drugi; on će možda biti dijeljen između više ljudi tj. unutar tima. Puno je lakše koristiti ili popravljati kod koji je podijeljen u logičke cjeline – funkcije.

- Omogućava nam *lakše pronalaženje grešaka* i ispravljanje koda. Kada imamo neki obiman i komplikovan program, vrlo je teško pronaći grešku u njemu. Ukoliko pak taj kod podijelimo u funkcije, možemo testirati ispravnost svake funkcije i utvrditi u kojoj funkciji se nalazi greška. Sve napisane funkcije testiramo neovisno od drugih funkcija i na taj način se pronalazi greška u manjem dijelu koda koji se nalazi u funkciji, bez potrebe gledanja ispravnosti ostalih funkcija.

- *DRY* (Don't Repeat Yourself – "nemojte se ponavljati") je uobičajena preporuka u programerskom svijetu. Ponavljanje koda na više mjesta u istom programu je loša praksa iz razloga što se na taj način nepotrebno produžuje program, što ga čini težim za razumijevanje i testiranje. Osim toga, ako se utvrdi da u tom kodu postoji greška, moramo je popraviti na svakom mjestu gdje je kod kopiran. Ovaj ponavljajući kod se može jednostavno smjestiti u funkciju i zatim pozivati funkciju.

- Olakšana je *ponovna upotreba koda*. Kada jednom napravimo funkciju koja rješava neki problem, npr. izračunava faktorijel broja, mi ovu funkciju možemo koristiti u svim programima gdje je potrebno izračunati faktorijel. Na taj način sebi olakšavamo posao. Također funkciju možemo dati i drugim članovima našeg tima. Vremenom na ovaj način ćete razviti svojevrsnu biblioteku gotovih funkcija koje

1157098

koristite u vašim programima, a kada uočite i popravite grešku u toj funkciji istovremeno ćete je ispraviti u svim vašim programima...

Od funkcija ne treba bježati, nego ih treba koristiti kada god se za njima ukaže potreba. Poziv funkcije može uzrokovati neznatno smanjenje performansi programa, ali se ovo u praksi ne primijeti.[2]

Kraće isječke koda ne bismo trebali odvajati u posebne funkcije iz razloga što to produžuje program i potencijalno smanjuje njegovu čitljivost. Npr. ako imamo jednostavan izraz tipa `kv = x * x;`, ne bismo trebali pisati `kvadriraj` funkciju:

```
int kvadriraj(int x) {
  return x * x;
}
...
int kv = kvadriraj (x);
```

U slučajevima kao iznad, nema baš pretjerane koristi pisati funkciju za kvadriranje. Ipak, ukoliko je potrebno isprogramirati računanje npr. sljedećeg izraza $15x - (x - y)^2 + y(3x + 11y^2 - 2xy)^2$, onda bi takva funkcija za kvadriranje mogla skratiti izraz i učiniti ga čitljivijim. Naravno, još bolje rješenje bi bilo smjestiti korištene izraze u pomoćne promjenljive i njih kvadrirati.

6.2 Funkcije u C-u

Do sada smo vidjeli primjere nekih funkcija, vidjeli smo neke greške koje možemo napraviti, te smo vidjeli i zašto koristiti funkcije, kada ih treba ili ne treba koristiti i sl. U nastavku ćemo se detaljnije posvetiti načinu pisanja funkcija i različitim pravilima koja se tiču pisanja funkcija.

6.2.1 Analiza dijelova funkcije

Prisjetimo se opšte sintakse funkcija u programskom jeziku C:

```
POVRATNI_TIP ime_funkcije ([Parametri...]) {
  TIJELO FUNKCIJE...
}
```

Razmotrimo sada svaki dio posebno:

Povratni tip funkcije: funkcije obično vraćaju neke vrijednosti kao svoje izlaze, te je potrebno specificirati kojeg tipa je izlaz odnosno kojeg tipa je ono što funkcija vraća. Kao povratni tip funkcije možemo navesti *bilo koji* tip koji postoji u jeziku C, a kasnije ćemo vidjeti da to mogu biti i pokazivači, kao i složeni tipovi podataka (strukture).↑ Iz funkcije se *ne može* vraćati niz, ali se u C++u može vraćati vektor. Ukoliko funkcija ne vraća ništa (nema izlaza) onda je njen povratni tip `void`, a takve funkcije nazivamo *procedure*.↑ Treba napomenuti i

[2] Prilikom poziva funkcije, parametri se postavljaju na *stek* (koji će biti objašnjen na str. 196), a zatim se obavlja skok na lokaciju funkcije. Ipak, kompajler može izbjeći ovo tako što će sadržinu funkcije prebaciti na mjesto njenog poziva, što možemo i eksplicitno zatražiti dodavanjem ključne riječi `inline` na početak prototipa funkcije.

↑ Poglavlje 9.3, str. 303.

↑ Poglavlje 6.2.5, str. 193.

1777017198

to da funkcija može imati najviše jednu povratnu vrijednost (ali ovo ograničenje možemo zaobići na načine koje ćemo obraditi kasnije).

Ime funkcije: za ime funkcije vrijede sva pravila kao i za imena promjenljivih↑ (može se samo sastojati od znakova engleskog alfabeta, brojeva te znaka donja crta, pri čemu mora početi slovom ili sa "_"), uz dodatno ograničenje da je *ime funkcije* ***jedinstveno*** *za cijeli program*. Ne možemo imati dvije funkcije istog imena! Ime funkcije treba biti opisno tako da se na osnovu samog imena funkcije može otprilike odrediti njena svrha, npr. `kvadriraj`, `ispisi`, `zamijeni`... Običaj je da je naziv funkcije u imperativu jer pozivom funkcije mi naređujemo računaru da obavi određeni zadatak.

↑ Poglavlje 1.2.3, str. 29.

Parametri: predstavljaju ulaze u funkciju. Svaki parametar se sastoji od tipa i imena, a parametara možemo imati proizvoljan broj.[3] Parametar funkcije može biti i druga funkcija (tačnije, pokazivač na funkciju, što je objašnjeno u dodatku↑). Funkcija ne može prihvatiti kao parametar niz, ali ipak postoji način kako poslati niz funkciji (o čemu ćemo govoriti u poglavlju 6.3). Konačno, funkcija može biti bez parametara, odnosno da nema ulaza.

[3] Standard jezika C propisuje minimalno 127 parametara, ali limit kod današnjih prevodilaca je dostupna memorija na steku (praktički neograničeno, možemo imati i po više desetina hiljada parametara).

↑ Poglavlje C.4, str. 403.

Tijelo funkcije: predstavlja skup naredbi koje definišu šta funkcija radi. Tijelo funkcije se obavezno nalazi unutar vitičastih zagrada `{}`, koje označavaju opseg funkcije. Ono može biti proizvoljne dužine, ali radi čitljivosti bolje je ne pretjerivati. Ako je tijelo funkcije predugo, poželjno je rastaviti ga na više funkcija. Svaka funkcija koja vraća vrijednost (povratni tip joj nije `void`) mora sadržavati `return` naredbu u svome tijelu kako bismo naveli izlaz funkcije. Ukoliko to nije slučaj, dolazimo do nedefinisanog ponašanja i posljedice su nepredvidljive.

Tip funkcije je *obavezno* navesti, mada starije verzije C standarda (C89 i C90) podržavaju da se tip funkcije izostavi, te se u tom slučaju podrazumijeva tip `int`. Ovo nije podržano ni u programskom jeziku C++.

Spomenuli smo da ime funkcije treba biti jedinstveno u programu, a to uključuje i funkcije koje se nalaze u standardnim C bibliotekama. Kompajler nam neće dozvoliti da napravimo vlastitu funkciju koja ima isti prototip kao i neka bibliotečna (npr. `double sqrt(double)`), čak i ako njenu biblioteku (`math.h`) nismo uključili. Ako se pak razlikuju parametri ili povratni tip dobićemo upozorenje.

Naredbu `return` je *obavezno* pisati u svakoj funkciji koja nije tipa `void`. U novijim verzijama standarda (C99 i dalje), kao i u jeziku C++, napravljen je izuzetak za `main` funkciju u kojoj ako izostavimo `return` naredbu, podrazumijeva se da je napisano `return 0`. Ipak, radi konzistentnosti mi ćemo uvijek navesti ovu naredbu i u `main` funkciji.

1/1/1/1/1/1/1/1/98

Još jedna bitna stvar u vezi vraćanja rezultata iz funkcije jeste da se iz funkcije može vratiti *samo jednu* vrijednost! Postoje indirektni načini da se ovo ograničenje zaobiđe (koristeći pokazivače ili strukturni tip podataka), ali za sada smatrajmo da je to nemoguće. Početnici često mogu napraviti ovakvu grešku:

```
int funkcija (int x) { /* funkcija vraca x^2 i x^3 */
  int x2 = x * x;
  int x3 = x2 * x;
  return x2, x3; /* Ovdje je greska! */
}
```

Ovaj program predstavlja validan C kod koji će se moći prevesti, nekada i bez upozorenja, ali neće raditi ono što se od njega očekuje. Naime, u naredbi `return` koristi se *operator zarez*↑ koji omogućava izvršavanje više naredbi u jednom izrazu, a kao rezultat vraća uvijek posljednju. Dakle ova funkcija će vraćati uvijek `x3` dok će `x2` biti jednostavno zanemaren. Npr. ako napišemo `broj=funkcija(3);` promjenljiva broj će dobiti vrijednost 27. Još gora situacija je što i ova konstrukcija radi:

↑ Poglavlje 3.2.4, str. 83.

```
int b1, b2;
b1, b2 = funkcija (5);
```

ali ovo će u `b2` dodijeliti 125 ($5^3 = 125$) dok će `b1` ostati neinicijalizirana. Drugim riječima, iznad napisani kôd je ekvivalentan sljedećem (koji je malo jasniji uz par zagrada):

```
int b1, b2;
b1, (b2 = funkcija (5));
```

6.2.2 *Pozivanje funkcija*

Napisanu funkciju potrebno je pozvati kako bismo imali koristi od nje. Potrebno joj je proslijediti ulaze (ako ih ima) te iskoristiti njene izlaze (ako ih ima). Poziv funkcije se sastoji od pisanja imena funkcije te, u malim zagradama, pisanja ulaza u funkciju. Pogledajmo primjer poziva npr. `kubiraj` funkcije:

```
1 int main() {
2   int x = 10;
3   int y = kubiraj (x);
4   printf ("%d^3 = %d", x, y) /* 10^3 = 1000 */
5   return 0;
6 }
```

Poziv funkcije se nalazi u 3. liniji. Unutar zagrada je navedena promjenljiva koju ćemo proslijediti kao ulaz funkciji. Rezultat funkcije možemo "uhvatiti" tako što tu funkciju iskoristimo u izrazu. U našem slučaju kreirana je promjenljiva `y` te je inicijalizirana rezultatom funkcije `kubiraj`.

Poziv funkcije se ne može koristiti sa lijeve strane dodjele, odnosno ne možemo funkciji "dodijeliti" promjenljivu, dok je obrnuto moguće

1757098

(promjenljivoj možemo dodijeliti rezultat funkcije). Zapravo, poziv funkcije sam po sebi predstavlja izraz. Pogledajmo konkretne primjere (pretpostavlja se da su u ovom programu napisane funkcije `kubiraj` i `max`):

```
int x = 10;
int y = 5 + kubiraj (3); /* 32 */
int z = kubiraj (5) + kubiraj (x); /* 1125 */
int m = max (x, y); /* m = 32 */
```

Također, rezultate funkcija možemo iskoristiti da budu ulazi drugih funkcija:

```
1 int x = 10;
2 printf ("x^3 = %d", kubiraj (x));
3 y = 17;
4 /* kubiraj veci od x i y */
5 int m = kubiraj (max (x, y)); /* m = 17^3 = 4913 */
6 /* izracunajmo x^9 kao (x^3)^3 */
7 int x9 = kubiraj (kubiraj (x)); /* 1 000 000 000 */
```

U liniji 2 smo vrijednost koju je vratila funkcija `kubiraj` direktno ispisali na ekran koristeći funkciju `printf`. U liniji 5 vrijednost koju je vratila funkcija `max` koristi se kao ulaz u funkciju `kubiraj`. Konačno, u liniji 7 smo najprije kubirali `x`, pa smo tako izračunatu vrijednost ponovo poslali funkciji `kubiraj`.

Kako je već rečeno, pozivi funkcija (odnosno rezultati funkcije) se ne mogu naći sa lijeve strane operatora dodjele. Ne možemo im nešto dodijeliti:

```
int x = 10, y = 5;
kubiraj (10) = 4; /* nema smisla */
kubiraj (10) = kubiraj (5); /* ni ovo */
/* nazalost, ne moze ni ovo, bar ne u C-u */
max (x, y) = 3;
/* u cilju da x ili y (zavisi koji je veci) postavimo na 3! */
```

Srećom, za posljednji poziv (odnosno da veću od proslijeđenih promjenljivih postavimo na neki broj) postoji rješenje koje će biti opisano u sljedećem poglavlju 7 (“Pokazivači”). Razlozi zašto ovo nije moguće su tehničke prirode: funkcija ne vraća “proslijeđenu” promjenljivu nego vraća njenu kopiju. U C++u je moguće funkciju deklarisati tako da vraća referencu, te je u tom slučaju navedeni izraz sasvim validan i radi na očekivan način.

Napomenimo još i da onaj ko poziva funkciju ima puno pravo da zanemari vrijednost koju funkcija vraća:

```
/* Sljedece naredbe ne rade nista */
max(3, 5);
int x = 15;
kubiraj(x);
```

Ustvari, u primjerima navedenim u ovoj knjizi mi smo to redovno radili prilikom pozivanja funkcija `printf` i `scanf` koje imaju povratnu vrijednost.↑

↑ Poglavlje B.1.4, str. 356.

Parametar ili argument

Na mnogim mjestima u literaturi možemo vidjeti upotrebu termina "parametar" i "argument" kada se govori o ulazima u funkciju. Negdje se to dvoje miješa ili čak smatra da je isto. Ipak, to nije isto. **Parametrom** se naziva promjenljiva (ili njeno ime) kada pišemo funkciju. Tada je parametar samo formalno ime za ulaze:

✓ Parametar funkcije

```
int kubiraj (int x) /* "x" je parametar ovdje */
  ...
```

S druge strane, **argument** funkcije je promjenljiva (ili izraz) koja se prosljeđuje funkciji prilikom njenog poziva. To sada nije samo formalno ime, nego prosljeđujemo stvarne promjenljive ili izraze:

✓ Argument funkcije

```
int z = kubiraj (y); /* "y" je argument ovdje */
int t = kubiraj (10); /* "10" je argument... */
int w = kubiraj (2 - x + t); /* "2 - x + t" je argument... */
```

Pojasnimo ovo još jednom u kontekstu pojma "ulaz funkcije". Funkcija prihvata ulaze (i naravno da se pod ulazima u funkciju ne smatraju standardni ulazi koje korisnik unosi sa tastature). Parametar je ime toga ulaza a argument je zapravo taj ulaz kojeg prosljeđujemo. Dodatna činjenica koja je zbunjujuća jeste što mnogi parametre nazivaju "formalnim argumentima", a drugi pak argumente nazivaju "stvarnim parametrima".

Također, može biti zbunjujuće što se parametar i argument nekada isto zovu:

```
int kubiraj (int x)
    ...
/* u mainu: */
int x = 10;
int y = kubiraj (x);
```

Ipak, promjenljiva `x` iz funkcije `main` i promjenljiva `x` iz funkcije `kubiraj` su *potpuno različite promjenljive*! One se nazivaju **lokalnim promjenljivim**↑ i vidljive su samo u funkcijama u kojima su deklarisane.

↑ Poglavlje 6.2.7, str. 196.

6.2.3 *Mjesto pisanja funkcija*

Sve funkcije napisane u C-u moraju biti *globalne*, što znači da ih ne možemo ugnijezditi u neku drugu funkciju ili strukturu. One moraju biti pisane na nivou datoteke. Dalje, sve funkcije koje se koriste moraju biti napisane *prije* mjesta korištenja ili barem moraju biti "najavljene" prije tog mjesta. Pojednostavljeno, ako funkcije koristimo (pozivamo) u `main` funkciji, one moraju biti napisane (ili najavljene) prije `main` funkcije. Također, standard jezika C (kao ni C++) ne dozvoljava da pišemo funkcije u funkcijama (odnosno funkcije unutar tijela drugih funkcija). Postoje neke nestandardne ekstenzije prevodilaca koje će dozvoliti pisanje funkcija u funkcijama, ali takav kôd, iako se uspješno prevodi i pokreće, *nije* po standardu te nije prenosiv.

1715701198

Pređimo na zadatak čije rješenje ćemo predstaviti u potpunosti, odnosno bit će prikazan kompletan program, sa funkcijom i njenim pozivom.

Zadatak 2. *Napisati funkciju "faktorijel" koja će računati faktorijel broja koji je proslijeđen kao parametar. Povratni tip funkcije treba da bude* `long long` *(razlog za ovo jeste što faktorijel funkcija brzo raste). Napisati i testni program u kojem se sa tastature unosi broj n (veći od 0) te se ispisuju faktorijeli svih brojeva od 1 do n.*

Prisjetimo se kako računamo faktorijel nekog broja. Po formuli, $n!$ se računa kao $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$. Prvo ćemo prikazati jednostavan program *bez funkcija* kao što bismo ga pisali prije ovog poglavlja (konkretno, opisano u poglavlju "4. Petlje").

```
1  #include <stdio.h>
2
3  int main() {
4    int i, n;
5    printf ("Unesite n: ");
6    scanf ("%d", &n);
7    long long fakt = 1;
8    for (i=1; i<=n; i++)
9      fakt *= i;
10   printf ("Faktorijel od %d je %lld\n", n, fakt);
11   return 0;
12 }
```

Kôd koji računa faktorijel nekog broja je petlja u `main` funkciji:

```
8    long long fakt = 1;
9    for (i=1; i<=n; i++)
10     fakt *= i;
```

Upravo će ovaj napisani kod biti tijelo naše `faktorijel` funkcije. Još je potrebno samo vratiti rezultat iz funkcije. Napišimo kompletno rješenje zadatka. Pri tome vodimo računa da je `faktorijel` funkcija napisana *iznad* `main` funkcije (u kojoj pozivamo `faktorijel` funkciju).

```
1  #include <stdio.h>
2
3  long long faktorijel (int n) {
4    long long fakt = 1, i = 0;
5    for (i = 1; i <= n; i++)
6      fakt *= i;
7    return fakt;
8  }
9
10 int main() {
11   int i, n;
12   printf ("Unesite n: ");
13   scanf ("%d", &n);
14   if (n < 1 || n > 20) {
15     printf ("Broj n nije ispravan!\n");
16     return 1;
17   }
18   for (i=1; i<=n; i++)
19     printf ("%d! = %lld\n", i, faktorijel (i));
```

```
20   return 0;
21 }
```

Tijelo funkcije (u linijama 4-7) je dovoljno jasno. Ono se sastoji od prethodno opisane petlje za računanje faktorijela. Ovdje smo ponovo morali deklarisati promjenljivu `i` jer su promjenljive u funkcijama lokalne (odnosno, nisu dostupne drugim funkcijama). Dodatno smo vratili tu vrijednost iz funkcije u liniji 7.

Pogledajmo deklaraciju funkcije u liniji 3. Povratni tip (kao i privremena promjenljiva `fakt`) su tipa **`long long`** što je traženo zadatkom.[4] Ime funkcije je `faktorijel`, što zadovoljava pravilo da ime mora biti opisno. Slijedi jedan jedini parametar `n`, i za njegov tip smo odabrali **`int`**, jer je on itekako dovoljan.

Unutar tijela funkcije smo deklarisali promjenljivu `i`, iskoristili petlju da izračunamo vrijednost faktorijela, te rezultat vratili iz funkcije pomoću **`return`** naredbe. Promjenljiva `fakt` također treba biti **`long long`** tipa jer ona akumulira vrijednosti prilikom množenja. Dodatno, povratna vrijednost funkcije je također bila tipa **`long long`**, pa neslaganje tipa promjenljive `fakt` (čiju vrijednost vraćamo iz funkcije) i povratnog tipa funkcije bi dovelo do gubitka podataka. Npr. da je povratna vrijednost funkcije bila ispravna (tj. **`long long`**), a da smo promjenljivu `fakt` deklarisali da bude tipa **`int`**, mi efektivno nismo ništa postigli jer će doći do prekoračenja opsega tipa (prilikom samog računanja, u petlji). Slična stvar bi se desila da smo uradili obrnuto, npr. da je povratni tip funkcije **`int`**, a promjenljiva `fakt` tipa **`long long`**. U tom slučaju tačno bismo izračunali faktorijel, ali prilikom vraćanja iz funkcije bismo trajno izgubili tu informaciju jer vršimo dodjelu većeg tipa u manji tip, i naravno da dolazi do gubitaka.

Funkcija `main` sadrži unos broja i provjeru ispravnosti istog, te petlju koja `n` puta ispisuje rezultat `faktorijel` funkcije. Rezultat funkcije `faktorijel` je direktno proslijeđen `printf` funkciji. Podsjetimo se da tip **`long long`** ispisujemo sa `%lld`. Imalo bi smisla staviti da je povratna vrijednost `faktorijel` funkcije tipa **`unsigned long long`** (jer je faktorijel uvijek pozitivan) te bismo taj tip ispisivali pomoću `%llu` formatnog stringa.↑

Fokusirajmo se sada na petlju koja ispisuje faktorijele brojeva od 1 do n. Vidimo da je to prilično jednostavno. Jedna petlja od 1 do n i pozivamo funkciju prosljeđujući joj kontrolnu promjenljivu `i` kao argument. Napišimo sada taj dio koda bez funkcija i vidjećemo da je komplikovanije i manje čitljivo.

```
for (i = 1; i <= n; i++) {
  int j; /* trebaju nam pomocne promjenljive */
  long long fakt = 1;
  for (j = 1; j <= i; j++)
    fakt *= j;
  printf ("%d! = %lld\n", i, fakt);
}
```

[4] Razlog za to je to što funkcija faktorijel veoma brzo raste (već za $n = 13$ imamo vrijednost $13! = 6227020800$), što ne može stati u standardni tip **`int`** na savremenim x86 računarima gdje je maksimalna vrijednost $2^{31} - 1 = 2147483647$. Tip **`long long`** će nam omogućiti da pohranimo broj $2^{63} - 1$ što je dovoljno za 20!.

↑ Poglavlje B.1.1, str. 353.

Za unutrašnju petlju smo morali uvesti novu kontrolnu promjenljivu `j`, jer je `i` bilo zauzeto, a gornja granica je sada `i`. Kod je manje čitljiv te skloniji greškama. Vidimo da je dekompozicija koda (rastavljanje koda na više logičkih cjelina) pomogla u čitljivosti (i održivosti) koda.

Izlaz programa izgleda ovako (za uneseni broj 10):

```
Unesite n: 10
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

6.2.4 *Prototip funkcije*

Da li je moguće pisati funkcije *nakon* `main` funkcije (odnosno, nakon mjesta poziva)? Odgovor je potvrdan, uz jedan uslov: funkciju prethodno moramo "najaviti" (jer prevodilac sekvencijalno procesira kod, liniju po liniju, te kada dođe do mjesta poziva funkcije on mora znati da ta funkcija postoji i eventualno gdje se nalazi). Pređimo prvo na primjer. Neka je potrebno napisati istu funkciju kao u prethodnom zadatku, ali ovaj put `faktorijel` funkciju treba napisati nakon `main` funkcije.

```c
#include <stdio.h>

/* NAJAVA funkcije (prototip) */
long long faktorijel (int n);

int main() {
  int i, n;
  printf ("Unesite n: ");
  scanf ("%d", &n);
  if (n < 1 || n > 15)
  {
    printf ("Broj n nije ispravan!\n");
    return 1;
  }
  for (i = 1; i <= n; i++)
    printf ("%d! = %lld\n", i, faktorijel (i));
  return 0;
}

/* nakon main funkcije */
long long faktorijel (int n) {
  long long fakt = 1, i = 0;
  for (i = 1; i <= n; i++)
    fakt *= i;
  return fakt;
}
```

Zanimljiva je 4. linija u kojoj smo *najavili* funkciju `faktorijel` koja je napisana kasnije. Ova najava funkcije se naziva **prototip funkcije**. To je deklaracija funkcije koja sadrži naziv funkcije, povratni tip funkcije te informacije o tipu i broju parametara. Ovo je potrebno uraditi zbog toga što prevodilac, kada obrađuje naš kod, to radi liniju po liniju. Kada na red dođe linija 16 (poziv `faktorijel`) funkcije), bez prototipa bi bila prijavljena greška da funkcija ne postoji. U slučaju da napišemo prototip funkcije, odnosno ukoliko najavimo funkciju (kao u primjeru iznad), prevodilac će prilikom obrade u tabeli funkcija sačuvati i informacije o `faktorijel` funkciji. Kada dođe do 16. linije, on neće prijaviti grešku, jer ima informaciju da je funkcija baš toga imena i tih parametara već *najavljena*. Prevodilac nastavlja dalje sa obradom, gdje će naići na implementaciju funkcije (i prototip i tijelo) i program će se, ako ne bude drugih grešaka, uspješno prevesti. U slučaju da postoji prototip funkcije, ali prevodilac kasnije ne pronađe implementaciju te funkcije, opet se prijavljuje greška i program se neće moći prevesti.

✓ *Prototip funkcije*

Kako se nazivi parametara *ne koriste* u prototipu, to ih možemo i izostaviti (što se većinom i radi), pa bi 4. linija glasila:

```
4  long long faktorijel (int);
```

Primijetimo da nakon prototipa slijedi znak tačka-zarez, dok kod implementacije funkcije to nije slučaj.

Prototip funkcije se *može navesti* unutar druge funkcije (za razliku od kompletne funkcije koja ne može biti uključena u drugu funkciju). Takav prototip se naziva **lokalni prototip** i on najavljuje postojanje funkcije samo unutar funkcije u kojoj je naveden. Lokalni prototip naveden u jednoj funkciji neće biti vidljiv u sljedećoj funkciji.

✓ *Lokalni prototip*

Napomenimo da prototip funkcije mora u potpunosti odgovarati zaglavlju funkcije kada istu pišemo (sa tijelom). Npr. za funkciju faktorijel iznad, nisu validni sljedeći prototipovi:

```
long long faktorijel (double); /* tip parametra */
long long Faktorijel (int); /* ime funkcije... */
long long faktorijel (int, int); /* broj parametara... */
```

Zadatak 3. *Napisati funkciju "euler" koja računa Eulerov broj e pomoću sljedeće sume*

$$e \approx \sum_{k=0}^{n} \frac{1}{k!}$$

Broj n je parametar funkcije i označava preciznost računanja. Prototip funkcije treba biti sljedeći:

```
double euler (int n);
```

a za računanje faktorijela iskoristiti funkciju `faktorijel` *iz prethodnog zadatka.*

1757019 8

U ovom rješenju ćemo funkciju `euler` napisati iznad `main` funkcije, a funkciju `faktorijel` ispod `main` funkcije. Navedenu sumu možemo

"razviti" da vidimo kako ide tok računanja:

$$e \approx \sum_{k=0}^{n} \frac{1}{k!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!} =$$
$$= 1 + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \cdots + \frac{1}{1 \cdot 2 \cdot 3 \cdot \ldots \cdot n}$$

Pređimo na rješenje.

```
#include <stdio.h>
/* prototip */
long long faktorijel (int);
double euler (int n) {
  int i;
  double suma = 0;
  for (i = 0; i <= n; i++)
    suma += 1. / faktorijel (i);
  return suma;
}
int main() {
  printf ("e = %.10f\n", euler (5));  /* n = 5 */
  printf ("e = %.10f\n", euler (10)); /* n = 10 */
  printf ("e = %.10f\n", euler (15)); /* n = 15 */
  return 0;
}
long long faktorijel (int n) {
  long long fakt = 1, i = 0;
  for (i = 1; i <= n; i++)
    fakt *= i;
  return fakt;
}
```

Izlaz programa bi mogao biti sljedeći:

```
e = 2.7166666667
e = 2.7182818011
e = 2.7182818285
```

Kako $n \to \infty$ to dobivamo veću preciznost broja e, ali već za $n > 20$ rezultati će postati netačni jer ćemo prekoračiti veličinu tipa `long long` prilikom računanja faktorijela. Funkcija `main` sadrži tri poziva `euler` funkcije sa različitim parametrima, a ispis je na 10 decimala kako bismo vidjeli preciznost posljednja dva poziva. Inače, vrijednost broja e je $e \approx 2.718281828459045$ što znači da za `n = 15` imamo prilično dobru aproksimaciju.

Ni funkciju `euler` ne treba posebno objašnjavati. Promjenljiva `suma` mora biti inicijalizirana na 0, a greška bi se mogla potkrasti unutar petlje:

```
    suma += 1. / faktorijel (i);
```

ukoliko ne vodimo računa o cjelobrojnom dijeljenju! Ovdje smo zbog toga umjesto "1" koristili "1." (što je ekvivalentno "1.0" a to je literal realnog tipa `double`)↑ jer bismo u suprotnom dijelili dva cijela broja, a rezultat toga dijeljenja je opet cijeli broj.

↑ Poglavlje 1.2.7, str. 35.

1755018198

Za vježbu probajte riješiti ovaj zadatak bez korištenja gotove funkcije `faktorijel`, a zatim bez korištenja funkcija uopšte, tako što ćete sav kod staviti u `main`. Uočićete da ima mnogo ponavljanja koda koji nije ni čitljiv. Znatno je lakše razumjeti prvo rješenje gdje smo ovaj kod razdvojili na više logičkih cjelina (funkcija). Ovako napisan program neće biti pogodan ni za održavanje niti za mijenjanje (promjenu specifikacija).

6.2.5 *Procedure (funkcije koje ne vraćaju ništa)*

Nekada nam je potrebna funkcija koja će obaviti određeni posao, odnosno izvršiti niz akcija. Tada nam rezultat te funkcije nije potreban, odnosno funkcija ne treba da vraća ništa. Takva funkcija se naziva **procedura** ili **potprogram**. Način da označimo da funkcija ne vraća ništa jeste da kao povratni tip funkcije stavimo tip `void`.

Zadatak 4. *Napisati program koji od korisnika traži da unese n brojeva (n se unosi sa tastature, mora biti u intervalu [1, 15]) i smješta ih u niz. Potrebno je da se taj niz ispiše u obrnutom poretku (korištenjem funkcije `ispisi_niz_unazad`). Za upit korisnika "Unesite i-ti element niza" koristiti proceduru tj. funkciju imena `ispisi` koja ne vraća ništa.*

Prototip `ispisi` funkcije bi mogao biti sljedeći:

```
void ispisi (int i);
```

Funkcija prihvata parametar koji označava redni broj elementa koji se unosi (kako bi mogla napraviti željeni ispis), ali ta funkcija ne vraća ništa odnosno povratni tip joj je `void`. Isto vrijedi i za `ispisi_niz_unazad` funkciju, ona neće vraćati ništa, ali će izvršiti određene akcije (ispisivanje niza). Prikažimo i kompletno rješenje:

```
1  #include <stdio.h>
2  void ispisi (int i) {
3    printf ("Unesite %i. element niza. Niz[%i] = ", i + 1, i);
4  }
5  void ispisi_niz_unazad (int niz[], int velicina) {
6    int i;
7    for (i = velicina - 1; i >= 0; --i)
8      printf ("%i, ", niz[i]);
9    printf("\n");
10 }
11 int main() {
12   int n = 0;
13   int niz [15];
14   printf ("Unesite broj n: ");
15   scanf ("%d", &n);
16   if (n < 1 || n > 15) {
17     printf ("Unos pogresan!\n");
18     return 0;
19   }
20   int i;
21   for (i = 0; i < n; i++) {
22     ispisi(i);  /* Poziv procedure */
```

```
23      scanf ("%i", &niz[i]);
24    }
25    /* Ispis niza u obrnutom poretku: */
26    printf ("Obrnut redoslijed:\n");
27    ispisi_niz_unazad (niz, n); /* poziv funkcije i prosljedjivanje niza */
28    return 0;
29  }
```

Ovdje moramo obratiti pažnju na funkciju `ispisi_niz_unazad` odnosno na njene parametre. Kada prosljeđujemo niz funkciji, mi uvijek moramo proslijediti i dodatnu informaciju o veličini tog niza. Prvi parametar je oznaka da je to niz (`int niz[]`), a drugi je njegova veličina (broj elemenata). Kod poziva funkcije (linija 26) proslijedili smo samo ime niza i to je ispravan način prosljeđivanja nizova funkcijama.↑

↑ Poglavlje 6.3, str. 206.

Ispis programa za unos tri broja može izgledati ovako:

```
Unesite broj n:  3
Unesite 1. element niza. Niz[0] =  7
Unesite 2. element niza. Niz[1] =  5
Unesite 3. element niza. Niz[2] =  10
Obrnut redoslijed:
10, 5, 7,
```

Jezik C (kao i jezik C++) ne razlikuje funkcije i procedure, odnosno procedurom smatra funkciju sa povratnim tipom `void`. To znači da mi slobodno možemo ignorirati rezultat neke funkcije (što često i radimo kada koristimo funkcije `scanf` i `printf`, iako i one vraćaju vrijednosti↑). Npr. sljedeće je potpuno validno (uz pretpostavku da postoji funkcija `kubiraj`):

↑ Poglavlje B.1.4, str. 356.

```
int main() {
  /* printf vraca broj ispisanih znakova */
  int x = printf ("Abc"); /* x = 3 */
  printf ("Def"); /* ignoriramo povratnu vrijednost */
  kubiraj (5); /* takodjer ignoriramo sta vraca kubiraj */
  ...
}
```

Jezici koji striktno razlikuju procedure i funkcije ne bi dozvolili da se ignorira povratna vrijednost funkcije (ili bismo tu funkciju morali označiti procedurom).

6.2.6 *Funkcije bez parametara*

Često naše funkcije nemaju ulaze (odnosno, nemaju parametara), pa ih kao takve treba i deklarisati. Po pravilu, izgled funkcije bez parametara bi izgledao ovako:

```
1  int unesi_broj () { /* zamjenska funkcija za unos broja */
2    int r = 0;
3    scanf ("%d", &r);
4    return r;
5  }
```

Sada nam je u nekoj mjeri olakšano unošenje cijelih brojeva:

```
printf ("Unesite broj: ");
int b1 = unesi_broj();
printf ("Unesite broj: ");
int b2 = unesi_broj();
```

Možemo primijetiti da naša `unesi_broj` funkcija ne traži nikakve ulaze, nego obavlja akciju unošenja broja. Ova funkcija bi bila još korisnija ako unosimo neke složenije tipove (npr. string ili Osobu; vidjeti poglavlje 9), pa bismo skratili naš program ako moramo često unositi promjenljive tih tipova. Uglavnom, kada pozivamo funkciju bez parametara potrebno je nakon imena funkcije navesti par malih zagrada `()`, kao u primjeru iznad `unesi_broj()`, a greška je napisati samo ime funkcije, jer to zapravo ne bi bio poziv funkcije nego pristup istoimenoj promjenljivoj.

Ipak, u jeziku C postoji čudna konvencija prilikom deklaracije funkcije bez parametara. Naime, ako napišemo samo prazne zagrade (kao u primjeru iznad, `int unesi_int()`), tada mi nismo napisali funkciju bez parametara, nego funkciju s proizvoljno mnogo parametara proizvoljnog tipa! Te parametre funkcija ne koristi, ali unatoč tome mi ih možemo proslijediti. Pogledajmo.

```
1 /* pretpostavka da imamo unesi_int funkciju
2 identicnu kao u prethodnom primjeru */
3 unesi_int (1, 2, 3, 4, 5); /* prosljedjujemo 5 int-ova */
4 int x = unesi_int (1.2, 1, 'A'); /* prosljedjujemo double, int i char */
5 unesi_int ("Neki string", 1, 12); /* cak i tekst moze */
```

Razlog za ovo jeste tehničke prirode (kada je izašla prva verzija jezika C, funkcije su se jedino tako mogle i deklarisati), ali ipak postoji mogućnost da ovo onemogućimo. Možemo deklarisati funkciju na način da eksplicitno kažemo da je ona funkcija bez parametara. To radimo tako što navedemo kao parametar tip `void`:

```
int unesi_int (void) /* funkcija sigurno ne prihvata parametre! */
```

Tek sada funkciju možemo pozvati samo na jedan način i to bez prosljeđivanja parametara:

```
int b1 = unesi_int(); /* OK */
int b2 = unesi_int (1); /* GRESKA */
int b3 = unesi_int ("tekst..."); /* GRESKA */
```

Upravo ovo je razlog zašto na internetu i u drugoj C literaturi možemo vidjeti da `main` funkciju deklarišu na ovaj način:

```
int main(void)
```

1/15/01198

Korisno je reći da je sada (u C99 i dalje) "zastarjela" deklaracija funkcija bez parametara sa praznim zagradama `()`, nego se *treba* napisati `void`. Zastarjeli (eng. deprecated) dijelovi jezika nekada u budućnosti mogu biti skroz izbačeni. Stoga, preporučujemo, prilikom deklaracije funkcije bez parametara da naznačimo da je funkcija bez parametara korištenjem `(void)`. Noviji standardi omogućavaju pisanje funkcija bez

parametara sa `()` isključivo da bi se stariji programi mogli prevesti, ali u novim programima (koje mi upravo pišemo) treba koristiti `(void)`.

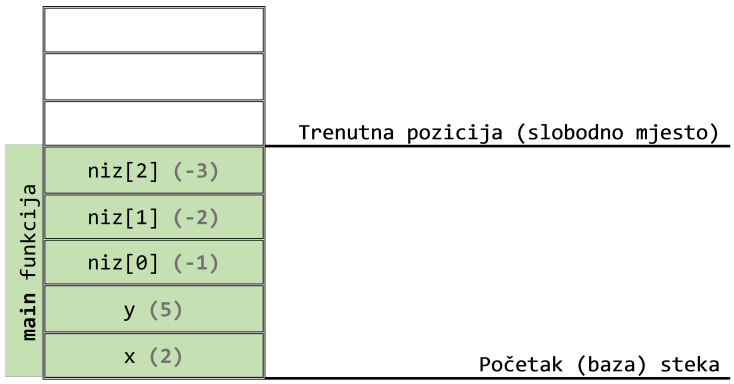
Napomenimo još da ovdje postoji *razlika* između C i C++ jezika. U C++-u kada napišemo `int unesi_broj()`, automatski se podrazumijeva da je to `int unesi_broj(void)`, pa to ručno ne moramo pisati. U C-u je to zapravo `int unesi_broj (*nepoznato*)`.

6.2.7 *Vidljivost promjenljivih deklarisanih unutar funkcije*

Promjenljive koje deklarišemo unutar funkcije nazivaju se **lokalne promjenljive**. One *nisu vidljive* u drugim funkcijama odnosno druge funkcije im *ne mogu pristupiti*. Tako, ako imamo promjenljivu `x` deklarisanu u dvije funkcije, to su *dvije različite* promjenljive koje nisu nikako povezane:

✓ Lokalne promjenljive

```
void fun_1 (void) {
  int x;
}
void fun_2 (void) {
  /* fun_2 ne moze pristupiti 'x' iz fun_1 */
  int x; /* ovo je razlicita 'x' */
}
```



Slika 6.2: Lokalne promjenljive iz `main` funkcije na steku

Kada deklarišemo promjenljivu unutar funkcije, ona se stavi na stek (engl. *stack*). Stek je dio memorije[5] koji je statički dodijeljen programu na kojem se smještaju, između ostalog, i lokalne promjenljive. Pojednostavljenu strukturu steka možemo pogledati na slici 6.2, gdje vidimo lokalne promjenljive deklarisane u `main` funkciji. Da, i promjenljive u `main` funkciji su lokalne, ne vide ih druge funkcije.

✓ Stek (stack)

[5] Standard C jezika ne propisuje način rada koji je ovdje dat, čak nije potrebno da stek uopšte postoji. Radi boljeg razumijevanja opisaćemo rad na PC arhitekturi.

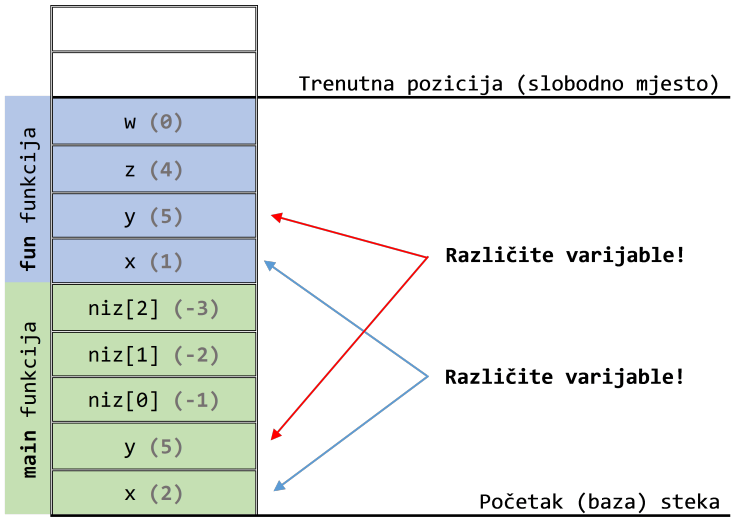
Razmotrimo sljedeći kod:

```
int main() {
  int x = 2, y = 5;
  int niz[] = {-1, -2, -3};
  ...
}
```

17370198

Stanje na steku izgleda otprilike kao na slici 6.2. Stek je LIFO (engl. *last in, first out* – zadnji ušao, prvi izašao) struktura, pa je poredak promjenljivih "obrnut". Naravno, postoje još neke informacije na steku, ali radi jednostavnosti smo se fokusirali samo na promjenljive (u zagradi sivom bojom su prikazane vrijednosti promjenljivih).

Sljedeća lokalna promjenljiva bi se smjestila na vrh steka (na trenutnu poziciju). Ukoliko sada funkcija `main` pozove neku drugu funkciju, npr. `fun` iz sljedećeg primjera, tada se na stek stavljaju i lokalne promjenljive iz `fun` funkcije (sl. 6.3).



Slika 6.3: Lokalne promjenljive iz `main` i `fun` funkcija

Ovdje postaje jasno da `main` funkcija *ne može* pristupiti promjenljivim kreiranim u `fun` funkciji niti obrnuto. Čak se promjenljive mogu zvati isto i biti istog tipa, ali su to dvije različite promjenljive. Izmjena jedne promjenljive (npr. `x` iz `fun` funkcije) nije povezana sa drugom promjenljivom (npr. `x` iz `main` funkcije).

```
int fun () { /* bez parametara */
  int x = 1, y = 5;
  int z = 4, w = 0;
  ...
}
int main() {
  int x = 2, y = 5;
  int niz[] = {-1, -2, -3};
  fun ();
}
```

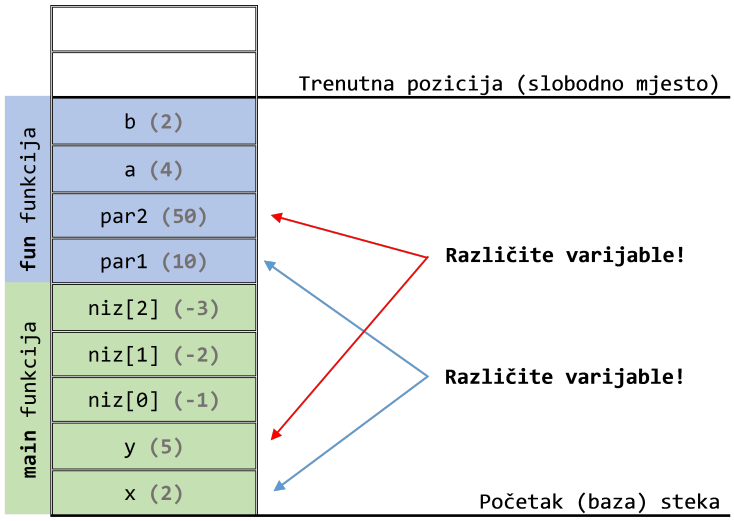
Ovdje funkcija `main` poziva `fun` funkciju, i kada je pozove, izvršavanje programa prelazi na `fun` funkciju, te se na steku kreiraju njene lokalne promjenljive. Već je rečeno da lokalne promjenljive `fun` funkcije nemaju veze sa lokalnim promjenljivim `main` funkcije iako se isto zovu i istog su tipa. Čak promjenljive `y` u obje funkcije imaju istu vrijednost (5). Opisano možemo vidjeti na slici 6.3.

17/10/2018

Parametri funkcija su također lokalne promjenljive te funkcije i vidljive su samo u toj funkciji. Ako sada promijenimo `fun` funkciju da prihvata dva parametra, `par1` i `par2`, kod će izgledati ovako:

```
int fun (int par1, int par2) {
  int a = 4, b = 2;
  ...
}
int main() {
  int x = 2, y = 5;
  int niz[] = {-1, -2, -3};
  fun (10, 50); /* par1 = 10, par2 = 50 */
}
```

Stanje na steku će izgledati približno kao na sl. 6.4.



Slika 6.4: I parametri `fun` funkcije su lokalne promjenljive

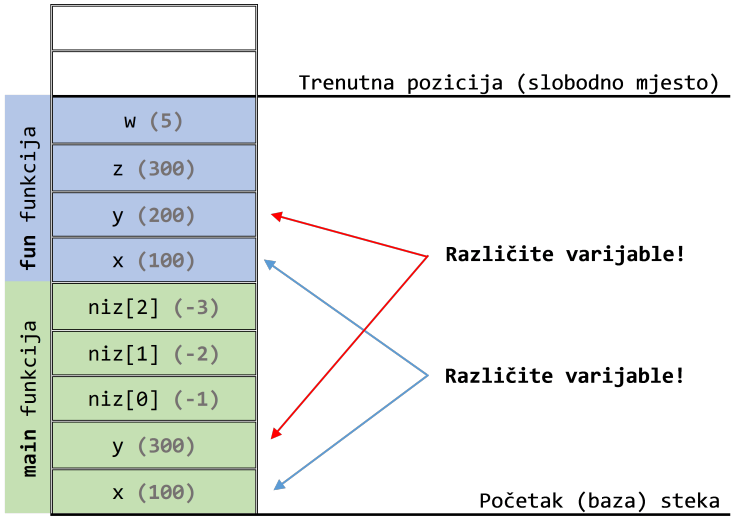
Napomenimo da stek u realnim implementacijama sadrži još neke dodatne informacije osim lokalnih promjenljivih (npr. sadrži i povratnu adresu iz funkcije, kako bi se izvršavanje moglo vratiti u funkciju koja je pozvala).[6]

Da stvar bude još više zbunjujuća, parametar funkcije također može imati isto ime kao i lokalna promjenljiva druge funkcije. Čak se ta promjenljiva može proslijediti kao argument funkcije. Pogledajmo sljedeći kod:

```
int fun (int x, int y, int z) {
  int w = 5;
}
int main() {
  int x = 100, y = 300;
  int niz[] = {-1, -2, -3};
    /* fun.x = x, fun.y = 200, fun.z = y */
  fun (x, 200, y);
  ...
}
```

[6] U nekim implementacijama niz na steku može biti u obrnutom redoslijedu, standard ne garantira redoslijed elemenata niza, pa čak ni promjenljivih na steku. Prikazane ilustracije su čisto radi shvaćanja pojma *lokalne promjenljive*.

Ovdje smo u liniji 8 kao prvi parametar `fun` funkcije (koji ima ime `x`) proslijedili lokalnu promjenljivu koja se također naziva `x`. Kao drugi parametar smo proslijedili vrijednost 200 (i tu vrijednost je poprimio parametar `y` iz `fun` funkcije). A kao treći parametar `fun` funkcije (sa nazivom `z`) proslijeđena je `y` promjenljiva iz `main` funkcije. Navedeno je prikazano na slici 6.5:



Slika 6.5: Lokalne promjenljive iz `main` i `fun` funkcija

Ovo sve dovodi do sljedećeg zaključka: funkcija ne može mijenjati vrijednosti promjenljivih drugih funkcija! Neka imamo npr. funkciju `povecaj` koja pokušava promijeniti vrijednost svog parametra i povećati ga za 1. Pogrešna implementacija bi izgledala ovako:

```
void povecaj (int x) {
  x = x + 1;
}
int main() {
  int broj = 5;
  povecaj (broj);
  printf ("Broj je %d\n", broj); /* ispisuje 5 */
  return 0;
}
```

Jasno je da promjenljiva `broj` i promjenljiva `x` iz `povecaj` funkcije nisu iste promjenljive i kada mijenjamo jednu promjenljivu (`x = x + 1`) to ne mijenja drugu promjenljivu (`broj` iz `main` funkcije). Ovakav poziv funkcije se naziva **poziv po vrijednosti**, jer se *vrijednost kopira*. Ovo može izgledati kao ograničenje, jer ne možemo napisati npr. funkciju `zamijeni` koja će izvršiti zamjenu dvije promjenljive. Ipak, postoji način da funkcija *indirektno* promijeni vrijednost neke promjenljive iz druge funkcije, i to se naziva **poziv po referenci**. To je moguće uz pomoć pokazivača o kojima govori sljedeće poglavlje (Poglavlje 7).

✓ Poziv po vrijednosti

Lokalne promjenljive nisu specifičnost dviju ili više funkcija. Moguće je deklarisati promjenljive u jednoj te istoj funkciji, ali da nisu

1/15/01/98

vidljive drugim promjenljivim iste funkcije (nakon što su prestale postojati). Promjenljive (osim globalnih promjenljivih) su vidljive u bloku u kojem su deklarisane a taj blok se naziva **vidljivost** (engl. *scope*). Susrećemo se i sa pojmovima "opseg" i "vidokrug". Unutar bloka koda u kojem je promjenljiva kreirana, ona je vidljiva naredbama. Nakon što se završi taj blok koda, promjenljiva "umire" odnosno prestaje postojati! Blok koda je obilježen vitičastim zagradama, pri čemu možemo imati više ugniježđenih blokova. Pogledajmo primjer:

✓ Vidljivost

```
int main() {
  int x = 5;
  { /* novi blok koda */
    int y = 3; /* 'y' je vidljiv samo u ovom bloku */
    /* ovdje je i 'x' vidljiv */
    y = x; /* y je sada 5 */

  } /* ovdje 'y' prestaje postojati :( */

  /* x = y; GRESKA: 'y' nije deklarisan */
  ...
}
```

U primjeru iznad promjenljiva `x` je lokalna za cijeli blok u kojem je deklarisana (a to je cijelo tijelo `main` funkcije). Promjenljiva `y` je lokalna samo za dio `main` funkcije, odnosno vidljiva je samo unutar onog bloka (par vitičastih zagrada) u kojem je i deklarisana. Nakon toga bloka, promjenljiva je sklonjena sa steka i prestaje postojati.

6.2.8 *Redoslijed određivanja vrijednosti parametara prilikom poziva*

Određeni broj stvari nisu definisane standardom jezika C, nego je prepušteno prevodiocu da se brine o njima. Na ova ponašanja se nikada ne smijemo osloniti, jer u suprotnom dobivamo pojavu **nedefinisanog ponašanja**. Na jednom prevodiocu na nekom operativnom sistemu naš program može raditi na jedan način, dok na drugom prevodiocu ili drugom operativnom sistemu (pa čak i drugoj verziji istog prevodioca) taj isti program možda ne bi nikako radio ili bi radio na drugi način.

✓ Nedefinisano ponašanje

Stoga, u našem programu se nikada ne smije desiti nedefinisano ponašanje, a jedan slučaj koji standardom nije definisan je upravo redoslijed određivanja vrijednosti parametara prilikom poziva funkcije. Npr. pogledajmo sljedeći primjer koda:

```
int x = 5, y = 3;
funkcija (++x, y, x++, x + y); /* sta ce biti proslijedjeno? */
```

1/15/0119 8

Ne možemo se osloniti da će prvo izvršiti `x++` pa onda `++x` ili obrnuto. Standard nam garantira da će se sve ove operacije zasigurno izvršiti prije nego se funkcija počne izvršavati, odnosno, imamo garanciju da će kao zadnji parametar funkcije biti proslijeđena vrijednost `x + y`, da

će biti izvršeno `++x` i `x++`, ali o redoslijedu izvršavanja odlučuje prevodilac (i taj redoslijed se može mijenjati od poziva do poziva).

Pogledajmo još očitiji primjer. Neka imamo funkcije koje računaju neki kompleksan izraz i njegovu vrijednost vraćaju kao rezultat. Te funkcije dodatno ispisuju na ekran neki tekst. Ukoliko te funkcije direktno proslijedimo kao argumente druge funkcije, mi nemamo garanciju koja funkcija će se prvo izvršiti.

```
int daj_x(void) {
  printf ("X");
  return izraz;
}
int daj_y(void) {
  printf ("Y");
  return drugi_izraz;
}
int racunaj (int x, int y) {...}
int main() {
  int i = racunaj (daj_x(), daj_y()); /* sta ce ispisati? */
}
```

Funkcija `daj_x` ispisuje "X", dok funkcija `daj_y` ispisuje "Y". Sigurno će se obje izvršiti prije nego se njihovi rezultati proslijede `Racunaj` funkciji, ali mi ne znamo (niti se ikada na to smijemo osloniti) šta će prvo biti ispisano, "X" ili "Y".

Pojava da funkcija pored vraćanja rezultata vrši još neku "vidljivu" akciju naziva se *sporedni efekat* (engl. *side effect*). Npr. naša `daj_x` funkcija vraća kao rezultat vrijednost nekog izraza (to je njena "glavna" akcija), ali dodatno ispisuje i slovo "X" što je njen sporedni efekat. Pojavu sporednog efekta imamo i u funkcijama koje smo već koristili, npr. `scanf` funkcija vraća broj uspješno učitanih promjenljivih, ali dodatno i vrši unos u proslijeđene promjenljive sa standardnog ulaza (tastature). Mi smo uglavnom ignorirali šta `scanf` funkcija vraća, osim kod provjere uspješnosti unosa. Slična stvar je i sa `printf` funkcijom – ona vraća broj ispisanih znakova ali kao sporedni efekat ima i ispisivanje na ekran. Da, sporedni efekat funkcije je ispisivanje na ekran, a njena glavna akcija (glavni efekat) je vraćanje broja znakova koje će ispisati.[7]

✓ Sporedni efekti funkcija

[7] Podsjetimo se sporednih efekata kod operatora o kojima smo govorili u poglavlju 3.2 (str. 79).

Rješenje predstavljenog problema je jednostavno: nemojmo koristiti sporedne efekte tamo gdje njihovo ponašanje nije definisano! Na primjer u pozivu koji je naveden iznad:

```
funkcija (++x, y, x++, x + y);
```

nedefinisano ponašanje (i probleme koje ono povlači) možemo fino izbjeći tako što sve izraze sa sporednim efektima (`x++` i `++x`) prebacimo van poziva funkcije:

```
++x;
funkcija (x, y, x, x + y);
x++;
```

1/15/01/98

Namjerno je spomenuto da ne koristimo sporedne efekte gdje njihovo ponašanje nije definisano. Taj slučaj je poziv iznad. Postoje slučajevi kada izrazi sa sporednim efektima ipak imaju sasvim definisano ponašanje, npr. `x = y++` ili `printf ("OR");` (ukoliko njen rezultat direktno ne prosljeđujemo drugoj funkciji).

Zanimljivo, izraz `i = i++;` (ili `i = ++i;`) također predstavlja nedefinisano ponašanje. Zašto? Zbog toga što ovdje vršimo dvije operacije, dodjela (`i = i`) i inkrementiranje promjenljive `i`. Problem je što izrazi `i++` i `++i` imaju sporedni efekat (kao što smo ranije objasnili), a vršimo dodjelu upravo toj istoj promjenljivoj! Problem nije kada radimo `x = y++`, ali jeste kada dodjeljujemo istoj promjenljivoj koju mijenjamo: `x = x++`.

✓ *Ako neka promjenljiva mijenja svoju vrijednost unutar nekog izraza, ona se u tom izrazu smije pojaviti samo na jednom mjestu.*

✓ Kako izbjeći nedefinisano ponašanje?

Funkcije koje imaju sporedne efekte se nazivaju "nečistim" funkcijama (engl. *impure functions*), jer, pored toga što vraćaju rezultat, utiču na neke druge funkcije ili izlaz. U primjeru ispod, `fun1` je "nečista" jer rezultat `fun2` funkcije zavisi od toga koliko puta smo pozvali `fun1` funkciju.

```
int x = 5;
int fun1 (void) {x--; return 10;} /* necista */
int fun2 (void) {return x;}
int kvadriraj (int x) {return x * x;} /* cista */
```

Da bi funkcija bila "čista" funkcija (engl. *pure*) (suprotno od "nečista"), ona mora zadovoljiti dva pravila, od kojih je prvo da nema sporednih efekata, a drugo da za iste ulaze uvijek daje iste izlaze.

U primjeru iznad smo koristili promjenljivu `x` koja je deklarisana van tijela bilo koje funkcije. Takva promjenljiva se naziva *globalnom* promjenljivom, a više o takvim vrstama promjenljivih biće rečeno u nastavku.

6.2.9 Globalne promjenljive

Gotovo svi dosadašnji primjeri deklaracija promjenljivih su bili primjeri *lokalnih* promjenljivih, odnosno promjenljivih koje su kreirane unutar funkcija. Programski jezik C nam omogućava da promjenljive deklarišemo van svih funkcija (na nivou datoteke) i te promjenljive nazivamo *globalnim*.

✓ *Globalnom promjenljivom nazivamo promjenljivu koja je deklarisana (kreirana) na nivou datoteke i kojoj sve funkcije u programu mogu pristupiti.*

✓ Globalna promjenljiva

17570198

```
1 int x = 1; /* globalna promjenljiva */
2 int ispisi(void) {
3   printf ("%d", x); /* koristi globalnu promjenljivu */
```

```
4  }
5  void povecaj(void) {x++;} /* isto, koristi globalnu promjenljivu */
```

Preporuka je da se globalne promjenljive koriste samo onda kada nemamo drugog rješenja za dati problem. Razlog za to je što te iste promjenljive može promijeniti neka druga funkcija. Promjenu može izvršiti čak i neka funkcija iz druge datoteke, jer su globalne promjenljive vidljive iz drugih datoteka (odnosno, više datoteka može dijeliti globalne promjenljive). Naravno, ovo možemo ograničiti, tj. učiniti da globalna promjenljiva bude vidljiva samo u datoteci gdje je deklarisana. Ovo će biti objašnjeno na kraju potpoglavlja o statičkim promjenljivim.↑

↑ Poglavlje 6.2.10, str. 204.

Ukoliko u funkciji imamo isto ime promjenljive kao ime globalne promjenljive, tada prioritet ima lokalna promjenljiva:

```
int x = 10;
int main() {
  int x = 4;
  printf ("%d\n", x); /* ispisuje 4! */
}
```

Ipak, postoji način kako da pristupimo globalnoj promjenljivoj `x` u `main` funkciji. Način je malo "ružan", uključuje korištenje nove ključne riječi, ali rješava problem:

```
1  int x = 10;
2  int main() {
3    int x = 4;
4
5    { /* lokalni blok */
6      extern int x;     /* pokazuje da je 'x' globalna */
7      printf ("%d\n", x); /* ispisuje 10 */
8    }
9
10   printf ("%d\n", x); /* ispisuje 4 */
11 }
```

Ipak, ovdje prikazan način je rogobatan i prilično nepotreban. Najjednostavnije je koristiti drugo ime unutar funkcije (npr. umjesto `x` smo u `main` funkciji mogli koristiti `x1` ili nešto slično). A ovo je još jedan razlog zašto je bolje izbjegavati globalne promjenljive kada god to možemo. Ako dvije funkcije trebaju međusobno komunicirati, neka komuniciraju preko parametara.

Još jedna bitna osobina globalnih promjenljivih jeste da su one automatski inicijalizirane na podrazumijevanu vrijednost. Npr. za tipove `int, double, float, ...` to je vrijednost 0.

```
int x; /* globalna, ima vrijednost 0 */
int main() {
  printf ("%d\n", x); /* garantirano ispisuje 0 */
}
```

1/11/2018

6.2.10 Statičke promjenljive

Već je spomenuto↑ da se lokalne promjenljive (deklarisane u funkciji) kreiraju tek kada se funkcija pozove, i da se po završetku funkcije one unište. Ponovni poziv te iste funkcije će ponovo kreirati lokalne promjenljive pa ih ponovo uništiti i sve tako. Zamislimo da je potrebno brojati koliko se neka funkcija puta pozvala.

↑ Poglavlje 6.2.7, str. 196.

Zadatak 5. *Napisati funkciju "kvadriraj" koja kvadrira proslijeđeni broj, ali dodatno ispisuje redni broj poziva*

Naravno, sljedeći pristup nema smisla jer se sve lokalne promjenljive uništavaju, i funkcija će *uvijek* ispisivati broj 1 kao redni broj poziva:

```
int kvadriraj (int x) {
  int brojac = 1;
  printf ("Poziv br. %d\n", brojac);
  brojac++;
  return x * x;
}
```

Jedno ispravno rješenje jeste korištenje globalnih promjenljivih, ali vidjećemo da to rješenje nije najbolje:

```
1  int brojac; /* globalna - automatski vrijednost 0 */
2  int kvadriraj (int x) {
3    brojac++;
4    printf ("Poziv br. %d\n", brojac);
5    return x * x;
6  }
7  int main() {
8    int x1 = kvadriraj (10);
9    int x2 = kvadriraj (-5);
10   printf ("%d, %d\n", x1, x2);
11   return 0;
12 }
```

Ispis programa je sljedeći:

```
Poziv br. 1
Poziv br. 2
100, 25
```

Prodiskutujmo zašto ovo nije najbolje rješenje. Prvo, ako pogledamo sekciju "Globalne promjenljive" (6.2.9), možemo zaključiti da globalne promjenljive treba izbjegavati iz više razloga, a jedan od njih je da više funkcija može dijeliti (i modificirati) tu promjenljivu. Druga stvar, šta ako trebamo imati 10 funkcija koje će također brojati vlastite pozive? Imali bismo 10 globalnih promjenljivih (vjerovatno sa imenima `brojac1`, `brojac2` itd. ili `brojac_ime_funkcije`), što dodatno "prlja" kod i isti je skloniji greškama. Neka se pojavi potreba za 100 funkcija s ovim ponašanjem, imat ćemo 100 globalnih promjenljivih, a vjerovatno nekada ćemo i ime pogriješiti.

1777098

Postavlja se pitanje: *može li se na neki način globalna promjenljiva ograničiti samo na tijelo funkcije*? Odgovor je potvrdan – možemo dobiti ponašanje globalne promjenljive (automatska inicijalizacija) tako da vrijednost te promjenljive ostaje prilikom višestrukih poziva funkcije. Te promjenljive se nazivaju statičke promjenljive.

✓ *Statičke promjenljive su globalne promjenljive čija je vidljivost ograničena na blok koda u kojem su kreirane.*

✓ Statička promjenljiva

Da bismo promjenljivu označili kao statičku, potrebno je dodati ključnu riječ `static` prilikom njene deklaracije.

```
static int brojac;
```

Napomenimo da je dozvoljeno i `int static brojac`, ali po standardu je preporučeno da `static` ključna riječ uvijek ide na početku deklaracije i dobit ćemo upozorenje od prevodioca ako to nije slučaj (jer je ovakva konstrukcija "zastarjela"). Kako je statička promjenljiva unutar funkcije zapravo globalna promjenljiva koja je vidljiva samo u toj funkciji, to znači da se i statičke promjenljive automatski inicijaliziraju na podrazumijevanu vrijednost, pa će tako promjenljiva `brojac` u primjeru iznad imati vrijednost 0.

Ako statičkoj promjenljivoj zadamo vrijednost prilikom deklarisanja, promjenljiva će dobiti tu vrijednost prilikom prvog poziva funkcije, ali se ta vrijednost neće promijeniti u narednim pozivima. Pogledajmo ispravno rješenje zadatka uz upotrebu statičkih promjenljivih:

```
int kvadriraj (int x) {
  static int brojac = 1;
  printf ("Poziv br. %d\n", brojac);
  brojac++;
  return x * x;
}
```

Kada se prvi put pozove funkcija, biće kreirana statička (dakle globalna) promjenljiva čija je vrijednost 1. Odmah nakon poziva `printf` brojač se uvećava na 2. Kada sljedeći put pozovemo funkciju, promjenljiva `brojac` će i dalje imati vrijednost 2, te će se na ekranu ispisati taj broj, a zatim će se promjenljiva povećati na 3 itd. Za razliku od toga, da smo napisali:

```
int kvadriraj (int x) {
  static int brojac;
  brojac = 1;
  printf ("Poziv br. %d\n", brojac);
  brojac++;
  return x * x;
}
```

promjenljiva `brojac` bi se svaki put vratila na početnu vrijednost 1.

Za razliku od globalnih promjenljivih, korištenjem statičke promjenljive možemo imati proizvoljno funkcija ovakvog ponašanja i svaka od njih može imati svoju promjenljivu istog imena (npr. `brojac`).

1/10/2018

Objasnimo još izjavu iz poglavlja o globalnim promjenljivim. Rečeno je da su sve globalne promjenljive vidljive i u drugim datotekama, ali da se to može ograničiti. Sada kad znamo za ključnu riječ `static`, to možemo objasniti. Vrijedi ista zakonitost kao za statičke promjenljive unutar funkcija. Ako globalnu promjenljivu (van svih funkcija) proglasimo `static` promjenljivom, onda je ona vidljiva samo u toj datoteci (nije joj moguće pristupiti iz drugih datoteka). Zbog toga smo u definiciji statičke promjenljive stavili da je njena vidljivost u bloku koda gdje je deklarisana. Pogledajmo to u kodu:

```
int x; /* x je globalna, vidi se u drugim datotekama */
static int y; /* y je globalna, ali vidljiva SAMO u ovoj datoteci */
void funkcija1 (void) {
  static int br; /* br je globalna, ali vidljiva samo u ovoj funkciji */
}
void funkcija2 (void) {
  static int br; /* br je globalna, ali vidljiva samo u ovoj funkciji */
}
```

Korisno je spomenuti da na ovaj način možemo i funkcije ograničiti na određenu datoteku. Podrazumijevano, sve funkcije koje napišemo (i sve one koje smo napisali u ovom poglavlju) su globalne funkcije! Mogu se koristiti u drugim datotekama (naravno, moraju se izvršiti neke akcije, poput uključivanja, najave – prototip i slično). Ako iz nekog razloga želimo ograničiti funkciju samo na datoteku u kojoj je pišemo ili ako u više datoteka želimo imati više funkcija istog imena, onda tu funkciju proglasimo statičkom. Npr. ovako bi izgledalo zaglavlje funkcije `kvadriraj` da je ona statička:

```
  static int kvadriraj (int x)
```

6.3 *Prosljeđivanje nizova funkcijama*

Do sada smo se upoznavali sa funkcijama koje prihvataju obične promjenljive kao parametre. Funkcijama je moguće proslijediti i nizove, ali su pravila nešto drugačija (što će biti objašnjeno nakon postavke zadatka).

Zadatak 6. *Napisati funkciju "suma_niza" koja računa sumu elemenata proslijeđenog niza realnih brojeva i vraća je kao rezultat.*

Ako analiziramo postavku zadatka možemo primijetiti da funkcija mora vraćati realan broj pošto ćemo računati sumu elemenata niza realnih brojeva. Također, zbog načina na koji se nizovi prosljeđuju funkcijama, mi nikada ne prosljeđujemo niz funkciji, nego prosljeđujemo informaciju na kojoj memorijskoj adresi taj niz počinje.[8]

Važno je reći da se korištenje proslijeđenog niza unutar funkcije *ne razlikuje* od standardnog korištenja niza, ali treba imati na umu dvije stvari: uvijek prosljeđujemo "originalni" niz (odnosno, svaka izmjena

[8] Drugim riječima, prosljeđuje se pokazivač. O pokazivačima će biti govora u sljedećem poglavlju (Poglavlje 7), a sada nećemo ulaziti u detalje.

1/15/2018

niza u funkciji će se odraziti i na originalni niz) i drugo, funkcija ne može znati broj elemenata niza nego joj to mi moramo posebno saopštiti.

Zahvaljujući iznad opisanim stvarima, možemo odrediti kako bi trebao da izgleda prototip `suma_niza` funkcije:

```
double suma_niza (double niz[], int vel);
```

Vidimo da niz kao parametar označavamo na jednostavan način: pored imena parametra stavimo par uglastih zagrada `double niz[]`, dok veličinu niza prosljeđujemo kao drugi parametar.

Prikažimo sada i kompletno rješenje uključujući i demonstraciju rada funkcije:

```
1  #include <stdio.h>
2  double suma_niza (double niz[], int vel) {
3    double s = 0;
4    int i;
5    for (i = 0; i < vel; i++)
6      s += niz[i];
7    return s;
8  }
9  int main() {
10   double niz1[] = {5.5, 3.1, 17, 23, 4};
11   double s = suma_niza (niz1, 5);
12   printf ("Suma prvog niza je %.2f\n", s);
13   double niz2[] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
14   /* direktno prosljedjujemo rezultat funkcije */
15   printf ("Suma drugog niza je %.2f\n", suma_niza (niz2, 7));
16   return 0;
17 }
```

Izlaz programa:

```
Suma prvog niza je 52.60
Suma drugog niza je 30.80
```

U rješenju iznad možemo vidjeti i način na koji prosljeđujemo niz funkciji. Potrebno je samo navesti ime niza kao prvi parametar funkcije (npr. `suma_niza (niz2, 7)`), a logičku veličinu niza kao drugi parametar. Podsjetimo se dva pojma iz poglavlja o nizovima: "logička" i "fizička" veličina niza. Logička veličina niza predstavlja broj elemenata niza kojeg koristimo u programu. Npr. deklarisali smo niz od 1000 elemenata, ali je korisnik unio samo 10 i to predstavlja logičku veličinu niza – tu veličinu prosljeđujemo i funkcijama, a naravno i kada vršimo neku akciju s nizom (npr. ispisujemo ga – ispisat ćemo samo unijete elemente, ostalih 990 je neinicijalizirano). Logička veličina niza se može mijenjati (npr. izbacimo neki element niza, njegova logička veličina je smanjena). S druge strane, fizička veličina niza je njegova veličina koju smo postavili prilikom kreiranja niza (u našem slučaju 1000).

Što se tiče funkcije `suma_niza`, u njoj možemo ignorirati činjenicu da smo proslijedili pokazivač a ne niz. Pristup elementima niza je bio na standardan način, kao da se radi o pravoj promjenljivoj. Samo treba

1/1/5/0/1/98

obratiti pažnju da svaka izmjena "niza" u funkciji će izmijeniti i originalni niz, što možemo vidjeti na jednom od primjera u nastavku.

Funkciju smo mogli napisati i kraće jer sumiranje elemenata ne ovisi od redoslijeda sumiranja, pa nam uopšte nije bila potrebna kontrolna promjenljiva `i`:

```
double suma_niza (double niz[], int vel) {
  double s = 0;
  while (vel != 0)
    s += niz[--vel];
  return s;
}
```

Demonstrirajmo primjerom da se izmjena elemenata niza u funkciji odražava na vrijednosti u main funkciji:

```
#include <stdio.h>
void promijeni (int niz[]) {
  niz[0] = 5;
}
int main() {
  int niz[2] = {1, 2};
  promijeni (niz);
  printf ("Prvi element: %d\n", niz[0]); /* ispis: 5 */
  return 0;
}
```

Zadatak 7. *Napisati funkciju "ispisi_niz" koja ispisuje niz cijelih brojeva uokviren uglastim zagradama. Npr. niz sastavljen od elemenata 1, 2, 3 i 4 funkcija treba da ispiše kao "[1, 2, 3, 4]". Primijetimo da iza posljednjeg elementa niza nema znaka zarez!*

Ispisujemo najprije $n-1$ elemenata niza (odnosno `vel - 1`) tako što nakon elementa ispišemo znak zarez sa razmakom. Nakon toga ćemo ispisati posljednji element niza (tj. element `niz[vel-1]`) bez zareza nakon njega. Treba obratiti pažnju da ne smijemo ispisivati posljednji element niza ukoliko niz ima 0 elemenata (`vel=0`), jer tada izraz `vel-1` ima vrijednost `-1` i greška nastaje kada pokušamo pristupiti tome elementu niza `niz[-1]`. Zbog toga imamo uslov koji provjerava da li je veličina niza pozitivna.

```
#include <stdio.h>
void ispisi_niz (int niz[], int vel) {
  int i;
  printf ("[");
  for (i = 0; i < vel - 1; i++)
    printf ("%d, ", niz[i]); /* n - 1 elemenata */
  if (vel > 0) /* posljednji element */
    printf ("%d", niz[vel - 1]);
  printf ("]");
}
int main() {
  int niz1[] = {1, 2, 3}, niz2[] = {4, 5, 6, 7, 8};
  ispisi_niz (niz1, 3);
  printf ("\n");
  ispisi_niz (niz2, 5);
```

```
16   return 0;
17 }
```

Izlaz programa je sljedeći:

```
[1, 2, 3]
[4, 5, 6, 7, 8]
```

Zadatak 8. *Potrebno je napisati funkciju "ispisi_matricu" koji će ispisati proslijeđeni 2D niz dimenzija n×5 na 4 mjesta širine. Znači, taj 2D niz (matrica) će sigurno imati 5 kolona, a broj redova nije poznat.*

Fokus kod ovog rješenja jeste na deklaraciji parametra funkcije, a ostalo bi trebalo biti jasno iz dosadašnjih poglavlja. Naime, unutar uglastih zagrada kod parametara funkcije možemo navesti broj, npr.

```
void ispisi_niz (int niz[100], int vel);
```

ali se taj broj ignorira jer smo rekli na početku ovog poglavlja da se ustvari umjesto niza prosljeđuje pokazivač na prvi element. Ipak, kod višedimenzionalnih nizova, broj elemenata svih dimenzija osim prve moramo navesti. I tu se prosljeđuje pokazivač, ali on je kompleksnijeg tipa i mora imati informaciju o svim dimenzijama niza osim prve (radi načina kako je višedimenzionalni niz predstavljen u memoriji). Zbog toga, kada funkciji prosljeđujemo npr. dvodimenzionalni niz, moramo navesti i broj kolona toga niza.

```
1  #include <stdio.h>
2
3  void ispisi_matricu (int mat[][5], int br_redova) {
4    int i, j;
5    for (i = 0; i < br_redova; i++) {
6      for (j = 0; j < 5; j++)
7        printf ("%4i", mat[i][j]);
8      printf ("\n");
9    }
10 }
11 int main() {
12   int matrica[2][5] = {{1, 2, 3, 4, 5}, {5, 4, 3, 2, 1}};
13   ispisi_matricu(matrica, 2);
14   return 0;
15 }
```

Naravno, moguć je i slučaj kada znamo maksimalan broj kolona (fizička veličina) ali broj kolona koje trenutno koristimo je manji (logička veličina niza).

Zadatak 9. *Napisati funkciju "zbir_matrica" koja računa zbir dvije matrice A i B (broj kolona neće biti veći od 100), te rezultat smješta u matricu C, koja se također prenosi kao parametar. Funkcija treba vratiti 0 u slučaju da matrice A i B nisu istih dimenzija, a 1 ukoliko je sve prošlo kako treba (sabiranje matrica koje su istih dimenzija)*

1/15/0198

Iz postavke zadatka trebamo zaključiti kakav će biti prototip funkcije `zbir_matrica`. Funkcija treba prihvatiti tri matrice (pri čemu znamo

da je broj kolona maksimalno 100), ali joj moramo proslijediti i informacije o veličinama tih matrica. Svaka matrica je opisana sa brojem redova m i brojem kolona n. Naravno, za matricu C te informacije ne moramo proslijediti jer ona treba biti istih dimenzija kao matrice A i B. Iz svega navedenog možemo napisati prototip funkcije:

```
int zbir_matrica (int A[][100], int mA, int nA,
                  int B[][100], int mB, int nB, int C[][100])
```

Prvu dimenziju matricâ (broj redova) možemo da ne navedemo, ali drugu moramo navesti zbog računanja indeksa (tehnički razlozi, jezici C i C++ višedimenzionalne nizove transformiraju u jednodimenzionalne po redovima što se može vidjeti u sljedećem zadatku). U nastavku slijedi djelimično rješenje (preskočeni su dijelovi za unos matrica A i B). Treba obratiti pažnju da je broj kolona kod deklaracije matrica identičan broju kolona u prototipu funkcije (100). Ukoliko ovo zanemarimo dobit ćemo upozorenje i vjerovatno pogrešne rezultate (dok u C++-u program se ne bi ni mogao prevesti).

```
1  #include <stdio.h>
2  int zbir_matrica (int A[][100], int mA, int nA,
3                    int B[][100], int mB, int nB, int C[][100]) {
4    if (mA != mB || nA != nB)
5      return 0;
6    int i, j;
7    for (i = 0; i < mA; i++)
8      for (j = 0; j < nA; j++)
9        C[i][j] = A[i][j] + B[i][j];
10   return 1;
11 }
12 int main() {
13   int A[20][100], B[20][100], C[20][100];
14   printf ("Unesite M i N za matricu A: ");
15   ...
16   zbir_matrica (A, m_a, n_a, B, m_b, n_b, C);
17   ...
18 }
```

Zadatak 10. *Napisati funkciju "matrica2niz" koja vrši prebacivanje elemenata matrice dimenzija Nx10 u jednodimenzionalni niz*

Cilj ovog zadatka jeste da sve elemente iz matrice prebacimo u niz, jer je nekada jednostavnije raditi sa nizovima (tražiti preklapanja i slično). Dodatno, u ovome zadatku ćemo uvidjeti zašto moramo navesti broj kolona matrice prilikom prosljeđivanja u funkcije, jer u C jeziku se upravo ovo automatski dešava. Sve matrice koje deklarišemo budu interno spašene kao nizovi, pri čemu se redovi nalaze jedan nakon drugog.

Naravno, kod ovakvog tipa zadataka gdje funkcija smješta neke podatke u niz, uvijek se pretpostavlja da taj niz može prihvatiti sve potrebne podatke. U našem slučaju, kada prosljeđujemo niz, moramo se osigurati da on ima elemenata kao i matrica (odnosno da ima do-

17570198

voljno prostora). Ukoliko to nije slučaj, dešava se nedefinisano ponašanje (najvjerovatnije izlazak izvan opsega niza). Prikažimo rješenje:

```
1  void matrica2niz (int mat[][10], int m, int n, int niz[]) {
2    int i, j;
3    for (i = 0; i < m; i++)
4      for (j = 0; j < n; j++)
5        niz[i * n + j] = mat[i][j]; /* ovdje koristimo 'n' */
6  }
7  int main() {
8    int mat[3][10] = {{1, 2, 3}, {5, 6, -1}};
9    int niz [30], i;
10   matrica2niz(mat, 2, 3, niz);
11   for (i = 0; i < 2 * 3; i++)
12     printf ("%d ", niz[i]);
13   printf("\n");
14   return 0;
15 }
```

Zanimljiva je 5. linija odnosno indeksiranje niza. Tu pristupamo elementu niza sa indeksom `i * n + j` gdje je `n` broj kolona u matrici. Isti je slučaj i kada prevodilac transformira matrice u jednodimenzionalne nizove, on mora znati broj kolona, tako da bi znao gdje da smješta elemente ostalih redova, i ovo je upravo razlog zašto moramo navesti drugu dimenziju (broj kolona) kada deklarišemo parametar tipa matrice.

6.4 *Rekurzivne funkcije*

Već smo vidjeli da funkcija može pozvati drugu funkciju (npr. naša `ispisi` funkcija poziva `printf` funkciju). Gotovo svi programski jezici omogućavaju da funkcija pozove samu sebe i tu funkciju nazivamo **rekurzivnom funkcijom**.

✓ *Rekurzivna funkcija je funkcija koja poziva samu sebe.* ✓ Rekurzivna funkcija

Također i u matematici poznat je pojam rekurzije, rekurzivnih pravila ili procedura koje se definišu svođenjem na jedan osnovni (bazni) slučaj. U prirodi postoje razne pojave koje se mogu opisati rekurzivnim funkcijama. Npr. za jednu klasu rekurzivnih funkcija koje se nazivaju *fraktali* otkriveno je da mnoge prirodne pojave imaju osobine srodne fraktalima (sl. 6.6). Upravo ćemo izlaganje početi funkcijom koja se matematički definiše rekurzivno, pa ćemo primijeniti i rekurzivne funkcije za rješavanje tog problema.

Zadatak 11. *Napisati funkciju koja izračunava faktorijel prirodnog broja.*

1/5/01/98

Faktorijel broja n prostim riječima je umnožak svih prirodnih brojeva od 1 do n uključujući i njega samog. No, u matematici faktorijel se obično definiše sljedećom rekurzivnom formulom:

Slika 6.6: Cvijet suncokreta je jedan primjer fraktalnog oblika koji se pojavljuje u prirodi

$$n! = \begin{cases} n(n-1)!, & n > 0 \\ 1, & n = 0 \end{cases}$$

Za ovu formulu kažemo da je rekurzivna zato što se znak ! nalazi i sa desne strane definicije.

Datu rekurzivnu definiciju možemo pretvoriti u C kod, pomoću rekurzivne funkcije. Napišimo funkciju `faktorijel` koja će slijediti napisanu formulu. Najprije, funkcija treba vratiti broj 1 za $n = 0$ (jer je $0! = 1$). Ako to nije slučaj (tj. $n > 0$) onda ponovo računamo $n(n-1)!$ odnosno pozivamo funkciju i njen rezultat množimo sa `n`:

```
1 long long faktorijel (int n) {
2   if (n == 0)
3     return 1;
4   return n * faktorijel (n - 1);
5 }
```

Naravno, za izračunavanje faktorijela mogli smo koristiti i `for` petlju kao u poglavlju 4,↑ ali je ovo "školski" primjer za objašnjavanje rekurzije jer je prilično jednostavan. Prikažimo pozive ove funkcije za $n = 5$:

↑ Str. 119.

```
faktorijel (5) =
   5 * faktorijel (4) =
   5 * (4 * faktorijel (3)) =
   5 * (4 * (3 * faktorijel (2))) =
   5 * (4 * (3 * (2 * faktorijel (1)))) =
   5 * (4 * (3 * (2 * (1 * faktorijel (0))))) = /* faktorijel(0) = 1 */
   5 * (4 * (3 * (2 * (1 * 1)))) =
   5 * (4 * (3 * (2 * 1))) =
   5 * (4 * (3 * 2)) =
   5 * (4 * 6) =
   5 * 24 =
   120
```

Krećemo sa izračunavanjem vrijednosti 5!. Pošto je $5 > 0$, izvršava se linija 4 u kojoj je potrebno izračunati izraz `n * faktorijel (n - 1)`.

Pošto se u tom izrazu nalazi poziv funkcije, najprije se poziva funkcija faktorijel sa vrijednošću 4. Dakle sada ponovo konstatujemo da je $4 > 0$, pa izračunavamo izraz `n * faktorijel (n - 1)` za $n = 4$. Na ovaj način dolazimo do $n = 0$ gdje je ispunjen uslov u liniji 2, te se vraća vrijednost 1. Ova vraćena vrijednost se uvrštava u izraz `n * faktorijel (n - 1)` za $n = 1$ što daje $1 \cdot 1 = 1$, pa se ponovo vraća 1. Ova jedinica se uvrštava u izraz za $n = 2$, pa imamo $2 \cdot 1 = 2$ i vraća se broj 2 itd. Na ovaj način na kraju dolazimo do vrijednosti 120.

Ono što je bitno za svaku rekurzivnu funkciju jeste da ona mora imati *uslov za prekid (terminiranje) rekurzije*. U našem primjeru iznad to je slučaj za $0! = 1$, odnosno linija 2. Ukoliko izostavimo terminirajući uslov, dolazi do beskonačne rekurzije koja se obično manifestuje kao krahiranje programa.

✓ Termininiranje rekurzije

Primjena rekurzije može izgledati nepotrebnom, jer se može dokazati da svako rješenje koje koristi rekurziju može biti napisano u nerekurzivnoj formi (odnosno bez rekurzije, korištenjem petlji, a vrijedi i obrnuto). Nekada je rekurzivno rješenje mnogo lakše napisati nego iterativno (petlje), pogotovo ako više puta trebamo riješiti jedan te isti problem. Drugim riječima, rekurzivne funkcije koristimo kada početni problem možemo razbiti na manji problem (ili više njih) koji se rješava na identičan način. Još jedan primjer toga je izračunavanje sume svih prirodnih brojeva između 1 i N. Iterativno rješenje nam je poznato:

```
1 unsigned int suma_brojeva (unsigned int n) {
2   unsigned int i, suma = 0;
3   for (i = 1; i <= n; i++)
4     suma += i;
5   return suma;
6 }
```

Ali radi razumijevanja rekurzije i strategije "podijeli pa vladaj", riješimo ovaj problem rekurzivno.

```
1 unsigned int suma_brojeva (unsigned int n) {
2   if (n == 0) return 0;
3   return n + suma_brojeva(n - 1);
4 }
```

Za rješavanje problema "suma brojeva do n" smo isti razbili na dva potproblema (čije rezultate saberemo): "vrijednost n" i "suma od n-1". U ovoj funkciji specijalan (terminirajući) slučaj je kada računamo sumu broja 0, i za nju znamo da je jednaka nuli.

Iz perspektive programa, funkcija će pozivati samu sebe dok ne dođe do slučaja kojeg "zna" riješiti. Npr. neka smo kao parametar proslijedili broj 5. Za $n = 5$ funkcija "ne zna" rješenje, pa će pokušati riješiti problem sa $n - 1$ (odnosno, pozvati samu sebe, ali ovaj put proslijediti broj 4). Opet, funkcija "ne zna" rješenje, pa smanjuje problem i prosljeđuje $n - 1 = 3$. I za $n = 3$ proces se nastavlja, funkcija smanjuje problem (strategija "podijeli pa vladaj") i prosljeđuje 2 itd.

1755/0198

Konačno za slučaj $n = 0$ funkcija "*zna*" rješenje (jer smo ga mi napisali, i ono je 0), pa će funkcija vratiti 0 (jer je sada "znala" kolika je suma brojeva od 0 do 0, što je očito 0).

Dalje se rekurzija "odmotava" unazad, jer je `suma_brojeva(0)` vratila 0, pa se to sabere sa brojem 1, a sve to se ponovo sabere sabere sa brojem 2 (prilikom pozivanja `suma_brojeva (2)`), i takav rezultat se propagira dalje do poziva `suma_brojeva (3)` i sve tako dok ne dođe do prvog poziva (`suma_brojeva (5)`).

Opisano možemo prikazati na sljedećem toku izvršavanja. Primijetimo da bismo izračunali sumu brojeva za $n = 5$, funkcija je sebe morala pozvati 5 puta (šesti put smo je mi pozvali, npr. iz `main` funkcije) da bi došla do broja 0, i tek onda je računanje "počelo".

```
suma_brojeva (5) =
   5 + suma_brojeva (5) =
   5 + (4 + suma_brojeva (3))) =
   5 + (4 + (3 + suma_brojeva (2)))) =
   5 + (4 + (3 + (2 + suma_brojeva (1)))) =
   5 + (4 + (3 + (2 + (1 + suma_brojeva (0)))) = /* trivijalan slucaj */
   5 + (4 + (3 + (2 + (1 + 0)))) =
   5 + (4 + (3 + (2 + 1))) =
   5 + (4 + (3 + 3)) =
   5 + (4 + 6) =
   5 + 10 =
   15
```

Zadatak 12. *Napisati funkciju ispisi_niz koja ispisuje proslijeđeni niz. Za rješavanje zadatka obavezno koristiti rekurziju!*

I ovo je tip zadatka kod kojeg je rješenje petljama jednostavnije, ali ovaj zadatak dobro ilustrira redoslijed izvršavanja rekurzije, odnosno da funkcija prvo pozove samu sebe određeni broj puta (sve do trivijalnog slučaja), pa da se onda vrši "vraćanje" u funkcije koje su je pozivale. Na ovom primjeru je to pogotovo očito jer se radi ispis. Prikažimo rješenje.

```
1  #include <stdio.h>
2  void ispisi_niz (int niz[], int n) {
3    if (n == 0)
4      return; /* nema sta ispisati, prazan niz */
5    ispisi_niz(niz, n - 1);
6    printf ("%d ", niz [n - 1]);
7  }
8  int main() {
9    int niz[] = {1, 5, 6, 0, 3};
10   ispisi_niz(niz, 5);
11   printf("\n");
12   return 0;
13 }
```

Ako pogledamo rješenje vidjet ćemo da će se ispis prvog elementa niza izvršiti *tek* nakon što se izvrši svako moguće rekurzivno pozivanje. Znači, funkcija će se prvo pozvati za `n == 5`, pa `n == 4` itd. i posljednju put za `n == 0`. Tada prestaju pozivanja i izvršava se `return;`

1/15/09/98

naredba. Izlazi se iz funkcije, ali se ulazi opet u funkciju koja ju je pozvala, samo ovaj put je argument `n == 1`, te se izvršava linija kôda nakon poziva funkcije (odnosno 7. linija, ispisivanje elementa niza), koja ispisuje `niz[n - 1]`, što je `niz[0]`. Iz funkcije se opet izlazi u istu funkciju sa `n == 2`, ispisuje se `niz[n - 1]` što je za $n = 2$ zapravo `niz[1]`, i proces se ponavlja. Razumijevanje ovakvog toka rekurzije je krucijalno za rješavanje problema pomoću rekurzije. Treba razumijeti da će prvi element niza biti ispisan tek nakon 6 uzastopnih poziva funkcije, te da će se rekurzija "odmotavati", ispisujući preostale elemente.

Prikažimo tok izvršavanja funkcije `ispisi_niz` za primjer iznad.

```
ispisi_niz (niz, 5)
   ispisi_niz (niz, 4)
      ispisi_niz (niz, 3)
         ispisi_niz (niz, 2)
            ispisi_niz (niz, 1)
               ispisi_niz (niz, 0) /* trivijalan slucaj */
                  return;         /* ne radi nista */
            ispisuje niz[0]: 1 (n = 1)
         ispisuje niz[1]: 5 (n = 2)
      ispisuje niz[2]: 6 (n = 3)
   ispisuje niz[3]: 0 (n = 4)
ispisuje niz[4]: 3 (n = 5)
```

Radi kompletnosti i dodatnog razumijevanja, pokušajmo riješiti sljedeći zadatak.

Zadatak 13. *Napisati funkciju `ispisi_niz_obrnuto` koja ispisuje niz u obrnutom redoslijedu (odnosno da posljednji element bude ispisan prvi, pretposljednji drugi itd.). Obavezno koristiti rekurziju.*

Na prvi pogled, ovaj zadatak može da izgleda preteško. Ako je onakav tok bio za ispisivanje niza u normalnom poretku, kako će tek biti u obrnutom. Ipak, rješenje je vrlo jednostavno, samo se na prvi pogled ne čini takvim (što je često s rekurzijom).

Ako pogledamo rješenje prethodnog zadatka, možemo vidjeti da smo prvo pozivali funkciju, pa onda ispisivali element niza. Rješenje ovog zadatka se ogleda u tome da samo te dvije linije *zamijenimo*! Umjesto

```
ispisi_niz(niz, n - 1);
printf ("%d ", niz [n - 1]);
```

dovoljno je napisati

```
printf ("%d ", niz [n - 1]);
ispisi_niz_obrnuto (niz, n - 1);
```

1/15/01/98

da bi niz bio ispisan unazad. Objasnimo sada šta smo uradili. U ovom rješenju prvo ispišemo element (konkretno $n - 1$-vi element koji je posljednji), pa pozovemo funkciju s parametrom `n-1`. Onda ispišemo taj element (što je $n - 2$-gi element, odnosno pretposljednji) i ponovo pozovemo funkciju sa `n-1` (u odnosu na originalno `n=5`, proslijedili smo

`n-2` što je 3), i sve tako dok ne proslijedimo 0 čime se završava lanac rekurzije.

Opisano možemo prikazati pomoću sljedećeg toka izvršavanja:

```
ispisi_niz_obrnuto (niz, 5)
    ispisuje niz[4]: 3 (n = 5)
    ispisi_niz_obrnuto (niz, 4)
      ispisuje niz[3]: 0 (n = 4)
      ispisi_niz_obrnuto (niz, 3)
            ispisuje niz[2]: 6 (n = 3)
            ispisi_niz_obrnuto (niz, 2)
              ispisuje niz[1]: 5 (n = 2)
              ispisi_niz_obrnuto (niz, 1)
                ispisuje niz[0]: 1 (n = 1)
                ispisi_niz_obrnuto (niz, 0)
                  return; /* ne radi nista */
```

Na kraju, radi kompletnosti, prikažimo kompletno rješenje (funkcija i glavni program) ovog zadatka:

```
#include <stdio.h>
void ispisi_niz_obrnuto (int niz[], int n) {
  if (n == 0)
    return;

  printf ("%d ", niz [n - 1]);
  ispisi_niz_obrnuto(niz, n - 1);
}
int main() {
  int niz[] = {1, 5, 6, 0, 3};
  ispisi_niz_obrnuto(niz, 5);
  printf("\n");
  return 0;
}
```

Ispis programa izgleda ovako:

```
3 0 6 5 1
```

U ovom poglavlju o rekurzivnim funkcijama korišteni primjeri uglavnom imaju iterativno rješenje koje je jednostavnije. Razlog za to je već naveden, da se na jednostavnim primjerima objasni rekurzija i njen tok. U nekim situacijama rekurzija se prirodno nameće (jer je i problem po definiciji rekurzivan), a nekada su iterativna rješenja znatno složenija od rekurzivnih. Pogledajmo primjer Pascalovog trougla dimenzije 5:

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 3 1
```

1757/0198

Vidimo da je svaki broj (osim ovih jedinica na ivicama trougla) zapravo jednak zbiru dva broja iznad njega. Pascalovi brojevi se označavaju sa

$P(m,n)$ gdje je m broj reda, a n broj "kolone" u Pascalovom trouglu. Npr. u 3. redu, 2. koloni (tj. $P(3,2)$) broj 2 je jednak zbiru jedinica iznad njega (odnosno $P(3,2) = P(2,1) + P(2,2)$). Slično vrijedi i za broj 3. u 4. redu, 2. kolona (tj. $P(4,2)$): $3 = 2 + 1$, ili broj 6 u posljednjem redu: $P(5,3) = P(4,2) + P(4,3) = 6 = 3 + 3$.

Ukoliko malo više obratimo pažnju, primijetit ćemo da se ovdje nameće rekurzivno rješenje! Da bismo izračunali $P(5,3)$ potrebno je izračunati $P(4,2)$ i $P(4,3)$, a da bismo izračunali $P(4,2)$ trebamo izračunati $P(3,1)$ i $P(3,2)$, a da bismo izračunali $P(3,1)$ trebamo izračunati... vidimo da je ovaj problem rekurzivan.

✓ Prekid (terminiranje) rekurzije

Inače, kada primijetimo da je problem rekurzivan (odnosno jednostavnije ga je riješiti rekurzijom), trebamo odrediti i *uslov prekida rekurzije*. Za Pascalov trougao to možda nije očito, ali ako malo pažljivije pogledamo, vidjećemo da se na ivicama trougla uvijek nalaze jedinice, a upravo su to prekidni slučajevi! Ukoliko smo na početku reda, odnosno kada je u pitanju prva "kolona" (tj. $n = 1$), u Pascalovom trouglu se treba nalaziti broj 1. Slično, ukoliko smo na kraju reda, i tada se treba nalaziti broj 1.

Još ostaje pitanje da programski odredimo šta znači "kraj reda". Primijetimo da je to za prvi red ujedno i prva "kolona". Za drugi red njegov kraj je druga "kolona", za treći red njegov kraj je treća "kolona" i slično za ostale redove, za i-ti red njegov kraj je i-ta kolona. Odnosno, kraj reda je zapravo slučaj $m = n$, i za taj slučaj (kao i za početak reda, $n = 1$), vrijednost je 1.

Napišimo sada u matematičkoj formi izraz za računanje $P(m,n)$:

$$P(m,n) = \begin{cases} 1, & \text{za } m = n \text{ ili } m = 1 \\ P(m-1,n-1) + P(m-1,n), & \text{inače} \end{cases}$$

Već slutimo kako bi izgledala implementacija ove funkcije u C-u. Nazovimo je "pascal"

```
1 int pascal (int m, int n) {
2   if (n == 1 || m == n)
3     return 1;
4   return pascal (m - 1, n - 1) + pascal (m - 1, n);
5 }
```

Rješenje je zaista kratko i očigledno, što nije tačno za iterativno rješenje kao što ćemo vidjeti u nastavku. Za trivijalne slučajeve smo vratili 1, a za ostale slučajeve smo vratili zbir brojeva u redu "iznad" trenutnog u Pascalovom trouglu (po formuli).

Dodatno, u funkciju je korisno dodati i provjeru valjanosti parametara, da se funkcija može "odbraniti" ako su isti neispravni. Npr. šta bi se desilo da neko proslijedi $n = 0$ ili $m = 0$? U tom slučaju rekurzija nikada ne bi stala (jer se nikada neće ispuniti uslovi za prekid), pa bi funkcija pozivala samu sebe u nedogled tj. sve dok se

1777198

stek ne popuni jer svaka funkcija ima svoje lokalne promjenljive koje zauzimaju memoriju.↑ Prosljeđivani argumenti bi bili $pascal(-1,-1)$, $pascal(-2,-2)$ itd. i očigledno da tome nema kraja. Slična stvar bi se desila u slučaju $n > m$, kada su oba broja pozitivna. Treba razmisliti *šta mi želimo* da bude rezultat funkcije ako proslijedimo nevalidne argumente? Da li da funkcija vrati -1 kao znak da nije uspješno? Da li vratiti trivijalan slučaj tj. 1? Zavisi od situacije ili postavke, a ovdje smo se odlučili na "ljepšu" varijantu, da funkcija vrati -1:

↑ Poglavlje 6.2.7, str. 196.

```
int pascal (int m, int n) {
  if (m < 1 || n < 1 || n > m)
    return -1;
  if (n == 1 || m == n)
    return 1;
  return pascal (m - 1, n - 1) + pascal (m - 1, n);
}
```

Prikažimo sada i iterativno rješenje postavljenog problema i uvjerimo se da je osjetno duže. Rješenje implementira binomni obrazac $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ (gdje n i k odgovaraju m i n brojevima kod `pascal` funkcije, samo su umanjeni za 1):

```
int binom (int n, int k) {
  if (n < 0 || k < 0 || k > n)
    return -1;
  int sum1 = 1, sum2 = 1, sum3 = 1, i;
  for (i = 1; i <= n; i++)
    sum1 *= i;
  for (i = 1; i <= k; i++)
    sum2 *= i;
  for (i = 1; i <= n - k; i++)
    sum3 *= i;
  return sum1 / (sum2 * sum3);
}
```

Naravno, rješenje bi bilo dosta kraće da smo koristili gotovu funkciju za računanje faktorijela:

```
  return faktorijel (n) / (faktorijel (k) * faktorijel (n - k))
```

ali poenta je bila na tome kako prirodnu definiciju problema, ako primijetimo da je rekurzivna, vrlo jednostavno možemo pretvoriti u rekurzivnu C funkciju. Iterativno rješenje ovdje može biti kraće ako uvedemo pomoćnu funkciju, ali uz uslov da smo našli "zaobilazni" način za računanje koeficijenata Pascalovog trougla. Odnosno, u ovom slučaju postoji binomna formula s kojom se ovaj problem može riješiti, ali u nekim drugim slučajevima (npr. problem Hanojskih kula), ne možemo naći alternativnu "formulu" koja bi olakšala iterativno rješavanje problema, nego zaista moramo dobro razmisliti kako rekurzivno rješenje prevesti u nerekurzivno (a to nekada uopšte nije jednostavan zadatak).

1777017198

Pitanja za razumijevanje gradiva

1. Šta su to funkcije u C-u? Zašto programiramo pomoću funkcija?
2. Šta označava greška "undefined reference to main"?
3. Koliko vrijednosti se može vratiti iz funkcije?
4. Šta je to prototip funkcije?
5. Kako nazivamo funkcije koje ne vraćaju ništa i čemu one služe?
6. Kako označavamo funkciju koja nema parametara?
7. Šta je to poziv po vrijednosti a šta poziv po referenci?
8. Šta su globalne a šta statičke promjenljive?
9. Na koji način se može proslijediti niz u funkciju?
10. Šta je to rekurzija?

Zadaci za vježbu

Zadatak 14. *Napisati funkciju* `int prost(int x)` *koja vraća 1 (logička istina) ako je broj x prost, a 0 (logička neistina) ako x nije prost. Zatim iskoristite ovu funkciju u programu koji ispisuje sve proste brojeve između 1 i 100 (svaki broj u zasebnom redu).*

Zadatak 15. *Napišite funkciju "obrnut" koja vraća primljeni pozitivan cijeli broj okrenut naopako. Npr. ako se funkcija pozove sa vrijednošću 12345 (dvanaest hiljada tristo četrdeset i pet), treba vratiti broj 54321 (pedeset četiri hiljade tristo dvadeset i jedan). Pri tome nije dozvoljeno koristiti nizove niti raditi bilo kakav ispis u funkciji! Napravite i kraći program koji testira tu funkciju.*

Zadatak 16. *Sinus broja x može se izračunati koristeći Taylorov razvoj:*

$$\sin(x) = \sum_{i=1}^{n} (-1)^{i-1} \frac{x^{2i-1}}{(2i-1)!} \tag{6.1}$$

17570198

Potrebno je napraviti funkciju "sinus" koja na ovaj način računa sinus. Pored broja x čiji se sinus računa, ova funkcija treba imati i parametar n koji određuje preciznost rješenja: veće n daje tačniju vrijednost sinusa, no obratite

pažnju da faktorijel koji se nalazi u nazivniku vrlo brzo raste tako da će kod dovoljno velikog n biti prekoračen opseg vrijednosti tipa `int`.

Pri rješavanju zadatka iskoristite funkciju `pow` *iz biblioteke* `math.h`*, te funkciju* `faktorijel` *koju smo napravili u poglavlju 6.2.3 (str. 187). Zatim napravite program koji poredi tačnost ove funkcije i ugrađene funkcije* `sin(x)` *(u biblioteci* `math.h`*) za neke vrijednosti* `x` *i* `n` *unesene sa tastature.*

Zadatak 17. *Napisati funkciju "fibonacci" koja prihvata broj n i vraća n-ti Fibonaccijev broj. Npr. za* $n = 15$ *funkcija treba vratiti broj 610.*

Zadatak 18. *Napisati funkciju* `double stepen(double b, int e)` *koja vraća potenciju broja* b^e *bez korištenja funkcije* `pow`*. Eksponent je cijeli broj što olakšava zadatak, ali može biti i negativan. Primijetite da ovakva funkcija neće imati probleme sa nepreciznošću o kojima smo govorili u poglavlju 3.1.6 (str. 72).*

17570198

7
Pokazivači (pointeri)

Pokazivači (engl. *pointers*) su promjenljive koje kao svoj sadržaj sadrže adrese drugih promjenljivih. Podsjetimo se šta su promjenljive, te kako da pristupimo njihovim adresama, kako bismo se mogli detaljnije upoznati sa pokazivačima.

U prvom poglavlju smo se upoznali sa promjenljivim,↑ te smo rekli da je promjenljiva samo ime lokacije u memoriji. Kada se deklariše promjenljiva, u memoriji se rezerviše prostor koji će čuvati podatke. Svaka "izmjena" promjenljive automatski znači da mijenjamo ono što se nalazi na memorijskoj lokaciji koja je dodijeljena promjenljivoj. Na primjer, ako napišemo sljedeću liniju:

↑ Poglavlje 1.2.1, str. 27.

```
int x;
```

mi smo u slobodnom dijelu memorije zauzeli određeni blok koji će moći da smjesti cjelobrojne vrijednosti (tipa `int`). Naredbom `x = 5;` mi smo na tu memorijsku lokaciju upisali vrijednost 5. Pomoću imena promjenljive mi pristupamo toj memorijskoj lokaciji, a naučit ćemo da to nije jedini način za pristup.

Memorija računara je organizovana u ćelije koje predstavljaju bajte (eng. byte). Svaka ćelija ima jedinstvenu adresu koju možemo predstaviti kao cijeli broj – slično kao što kuće u nekoj ulici imaju adrese označene brojevima redom od 1 dalje, tako i prvi bajt u memoriji dobija adresu nula, a svaki sljedeći dobija za jedan veću adresu.[1] Recimo da smo deklarisali promjenljivu `x` kao u primjeru iznad. Možemo pretpostaviti da se ona nalazi na fiktivnoj adresi 1000. Kada pristupamo toj promjenljivoj (npr. postavljamo joj vrijednost `x = 3`), procesor zna da se ona nalazi na adresi 1000 i tamo će upisati broj 3.

[1] Striktno govoreći, *pogrešno* je pretpostaviti da su memorijske adrese uvijek brojčane prirode, ali radi jednostavnosti ćemo upravo to uraditi kako bismo objasnili kako adresiranje funkcionira. Danas se na većini računara koristi *linearni adresni model* na kojem su adrese brojevi upravo kao u ovom primjeru. Više o tome u dodatku C.1.1 (str. 359).

Svaka promjenljiva će imati jedinstvenu adresu, a nju možemo saznati pomoću **adresnog operatora** `&`. Pogledajmo primjer:

✓ Adresni operator

```
1 #include <stdio.h>
2 int main() {
3   int x = 3;
4   printf("Vrijednost od x je %i: \n", x);
5   printf("Adresa od x je %x\n", &x);
```

1777017198

```
6   return 0;
7 }
```

U liniji 5 vidimo primjer korištenja adresnog operatora `&x` pomoću kojeg saznajemo adresu promjenljive. Adresni operator je unarni operator (primijenjuje se isključivo nad jednim operandom) tako da kod `&x` čitamo kao "adresa od `x`".

Izlaz programa iznad bi mogao izgledati ovako:

```
Vrijednost od x je 3:
Adresa od x je 51c65dec
```

Ispisana adresa je *nepredvidljiva* i vjerovatno će biti različita prilikom svakog pokretanja programa. Vidjeli smo da je adresa ispisana koristeći *heksadecimalni* brojevni sistem,↑ to je zbog formatnog stringa `%x`. Ispravan način ispisivanja adrese je pomoću `%p` formatnog stringa.

↑ Poglavlje 3.1.2, str. 68.

Vidjeli smo da možemo *ispisati* adresu neke promjenljive... da li tu adresu možemo *sačuvati* u neku drugu promjenljivu?

✓ ***Pokazivač*** *(eng. pointer) je promjenljiva u kojoj možemo sačuvati adresu neke druge promjenljive.*

✓ Pokazivač

Pokazivači su kao i ostale promjenljive, i one zauzimaju memoriju, te imaju svoj tip. Sintaksa za deklarisanje pokazivača je:

```
TIP* pokazivac;
```

Recimo, ako želimo deklarisati pokazivač koji će pokazivati na cjelobrojnu promjenljivu (tipa `int`) to možemo uraditi na sljedeći način

```
int* pokazivac;
```

Promjenljiva `pokazivac` nije tipa `int` nego je tipa *"pokazivač na* `int`*"* što se označava sa `int*`. Na sličan način možemo deklarisati pokazivače na promjenljive ostalih tipova:

```
float* pok_float; /* pokazivac na float */
char* pok_char;   /* pokazivac na char */
short* pok_short; /* pokazivac na short */
...
```

Naravno, kao i svaka druga promjenljiva, i pokazivač će biti neinicijaliziran↑ nakon deklaracije i ne smijemo ništa raditi sa njim dok mu ne dodijelimo vrijednost. Ovu dodjelu vrijednosti radimo na sljedeći način:

↑ Poglavlje 1.2.2, str. 29.

```
int broj = 10;
int* pokazivac;
pokazivac = &broj; /* adresa od 'broj' */
```

Promjenljivoj `pokazivac` pridružili smo vrijednost adrese promjenljive `broj`. Naravno, mogli smo kraće pisati `int* pokazivac = &broj;`.

Proširićemo ovaj primjer sa par linija kroz koje se vidi na koji način možemo očitati vrijednost pokazivača.

1/15/11/98

```
1  int broj;
2  broj = 5;
3  int *pokazivac;
4  pokazivac = &broj;
5  if (&broj == pokazivac) /* citanje vrijednosti */
6    printf ("Isti!\n");
7  else printf ("Razliciti!\n");
```

Na slikama ćemo pokazati šta se dešava nakon svake linije koda. Nakon prve linije, zauzima se prostor u memoriji i dodjeljuje mu se ime `broj` (sl. 7.1, zauzeta lokacija je osjenčena). Pretpostavimo da se taj prostor nalazi na adresi 0x1500 (to je adresa promjenljive `broj`). Podsjetimo se da je nakon deklaracije sadržaj promjenljive `broj` *nepoznat* jer ta promjenljiva nije inicijalizovana.

...	broj	...	...	...	...
?	?	?	?	?	?
...	0x1500	...	...	...	...

Slika 7.1: Memorijski model nakon deklarisanja promjenljive broj

Nakon toga, u 2. liniji promjenljivoj dodjeljujemo vrijednost 5 čime se postiže sljedeće (slika 7.2).

...	broj	...	...	...	...
?	5	?	?	?	?
...	0x1500	...	...	...	...

Slika 7.2: Memorijski model nakon dodjele

Linija 3 sadrži deklaraciju pokazivača, a njena ilustracija je zanimljiva jer pokazuje da su pokazivači obične promjenljive, da zauzimaju prostor, te da su nakon deklaracije neinizijalizovane. Na slici 7.3 se vidi da je i pokazivačka promjenljiva (imena `pokazivac`) zauzela memoriju i da se nalazi na fiktivnoj adresi 0x7532.

...	broj	...	...	pokazivac	...
?	5	?	?	?	?
...	0x1500	...	...	0x7532	...

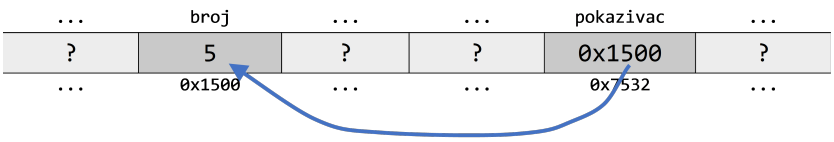
Slika 7.3: Zauzeta je još jedna lokacija: pokazivac

17570198

Konačno, linija 4

```
4  pokazivac = &broj;
```

vrši "povezivanje" pokazivača i promjenljive `broj`. To je prikazano na slici 7.4. Vidimo da je vrijednost promjenljive `pokazivac` broj 0x1500 što je adresa promjenljive `broj`. Dakle, pokazivač je promjenljiva koja



Slika 7.4: Konačno stanje nakon "povezivanja"

sadrži adresu neke druge promjenljive, odnosno kažemo da *pokazuje na tu drugu promjenljivu*.

U liniji 5 vrši se poređenje vrijednosti `&broj` i `pokazivac`, te ako su te dvije vrijednosti jednake na ekranu se treba ispisati tekst "Isti!" Možemo zaključiti da je adresa promjenljive `broj` 0x1500, a to je i vrijednost promjenljive `pokazivac`, te će izlaz programa biti:

```
Isti!
```

7.1 Rad sa pokazivačima

7.1.1 Stil deklaracije pokazivača

Već smo rekli da je i zvjezdica sastavni dio tipa, odnosno da je tip promjenljive `pokazivac` tipa `int*`. Ovaj tip se može pisati i sa razmakom, ovako: `int *`. Zapravo, postoji nekoliko različitih stilova deklarisanja pokazivača:

```
1 int *p;  /* standardni, C nacin */
2 int* p;  /* C++ nacin, implicira da je * dio tipa */
3 int * p; /* neki preferiraju ovako */
```

Prvi stil je onaj stari C stil korišten od autora jezika C, zvjezdica ide uz ime promjenljive; drugi stil jasno specificira šta je tip `p` promjenljive i često se naziva "C++ stil", dok se treći stil često susreće u raznim sistemskim bibliotekama (npr. *Windows API* - interfejs za programiranje aplikacija). Postoji i stil bez razmaka (jer jezik C ignorira razmake kada je to dozvoljeno) `int*p;` ali on je jako nečitljiv. Naravno, svi ovi stilovi su ispravni i ravnopravni (na kraju, ukusi su različiti) i programeri mogu pisati onako kako žele (ipak treba voditi računa o čitljivosti), ali postoje neke "kvake" kod ovih stilova koje se tiču vizualnog interpretiranja.

Razmotrimo sljedeću liniju:

```
int* p1, p2, p3; /* mozda problematicno! */
```

Postavlja se pitanje kojeg tipa su promjenljive `p1`, `p2` i `p3`. Neki će pomisliti da su sve tri promjenljive pokazivači, odnosno da su tipa `int*`, na osnovu analogije sa `int a, b, c;` gdje su sve promjenljive tipa `int`. Ipak, u liniji iznad, samo je prva promjenljiva (`p1`) pokazivačkog tipa, dok su preostale dvije obične cjelobrojne promjenljive. Linija iznad je ekvivalentna sa sljedeće dvije linije:

1/17/01/98

```
int* p1;    /* pokazivac */
int p2, p3; /* obicni intovi */
```

U toj situaciji, prvi stil (standardni C stil) je čitljiviji. U liniji ispod se jasno vidi da je samo `p1` tipa pokazivač (jer uz njega stoji znak zvjezdica), dok su promjenljive `p2` i `p3` obični cijeli brojevi. Neki preporučuju da je bolje čitati `*p` je tipa `int` odnosno "ono na šta `p` pokazuje je tipa `int`". Ovaj način čitanja ispravlja neke nejasnoće i unificira tipove i načine pisanja.

```
int *p1, p2, p3; /* samo p1 je pokazivac */
```

Ako ipak u jednoj liniji želimo deklarisati više pokazivača, onda je ispred svake pokazivačke promjenljive potrebno staviti zvjezdicu. I dalje je moguće miješanje pokazivača i običnih promjenljivih:

```
unsigned int *p1, *p2, *p3; /* sve pokazivaci */
char p4, *p5, *p6, p7, *p8; /* p5, p6 i p8 pokazivaci */
```

Koristeći `typedef`↑ možemo postići željeni efekat:

↑ Poglavlje 9.1, str. 299.

```
typedef int *pok_tip;
...
pok_tip p1, p2; /* i p1 i p2 su pokazivaci na int! */
```

Da zaključimo, stil deklaracije pokazivača je lični izbor programera (ili kompanije u kojoj radi), i ne treba pisati nečitljiv kôd. Treba izbjegavati pisanje mješovitih (i pokazivača i običnih) promjenljivih u jednoj liniji, nego ih treba rastaviti ili barem prigodnim komentarom naglasiti namjeru, sve to radi lakšeg održavanja kôda. Koji god stil da se odabere, važno je da programer bude konzistentan i da isti stil koristi sve vrijeme.

7.1.2 *Inicijalizacija pokazivača*

Već smo vidjeli da pokazivači pohranjuju adrese drugih promjenljivih koje dobivamo pomoću adresnog operatora. Važno je napomenuti da se adresni operator može primijeniti isključivo na promjenljive tj. ne može se primjenjivati na literale ili izraze. Nije validno sljedeće:

```
int broj;
int *pok1 = &(broj + 2); /* greska */
int *pok2 = &4; /* takodjer greska */
```

Iako će to kompajler dozvoliti, vjerovatno je greška inicijalizirati pokazivač brojem! C kompajler će nas upozoriti da razmotrimo šta radimo, dok će C++ kompajler prijaviti grešku!

```
int *pok = 1500; /* pazljivo sa ovim! */
pok = 200; /* i sa ovim! */
pok = broj; /* ista prica, ne smijemo! */
```

Ovo iznad je dozvoljeno na odgovornost programera, a kompajler nas upozori da provjerimo jesmo li sigurni da želimo upravo to. Recimo, adresa 200 vjerovatno ne pripada našem programu i mogu se desiti greške kada pokušamo pristupiti onome što se nalazi na toj adresi.[2]

Postoji jedan izuzetak kada pokazivaču smijemo dodijeliti broj (i kompajler nas neće upozoravati za to), a to je ukoliko želimo kreirati **nul-pokazivač**. To je pokazivač koji sadrži vrijednost 0 i *ne pokazuje ni na šta*. Ipak, umjesto pisanja broja 0 preporučljivo je da pišemo specijalnu makro konstantu `NULL`:

```
double *p = NULL;
```

Razlog korištenja nul-pokazivača je kako bismo osigurali da je pokazivačka promjenljiva inicijalizovana. Kada deklarišemo pokazivač on nije nul-pokazivač, njegova vrijednost je neka nepoznata adresa. Odnosno, možemo reći da takav pokazivač pokazuje negdje, ali ne znamo gdje. Postavljanjem pokazivača na nulu mi osiguravamo da pokazivač ne pokazuje nigdje tako da kasnije možemo provjeriti da li je to slučaj: `if (p == NULL) ...`

[2] C je programski jezik niskog nivoa, te se koristi za razvoj sistemskog softvera, drajvera i slično. Stoga su u C-u dozvoljene mnoge stvari koje su zabranjene u drugim jezicima, pa čak i u C++u. Tako je npr. dodjela numeričke vrijednosti pokazivaču zabranjena u C++u jer ona gotovo nikada nema smisla, ali u C-u je ipak možemo izvesti s ciljem demonstracije rada sa pokazivačima. U pravilu pokazivaču smijete dodijeliti isključivo adresu neke postojeće promjenljive.

✓ Nul-pokazivač

7.1.3 *Operator dereferenciranja*

Ako imamo pokazivač na neku promjenljivu, mi možemo koristeći pokazivač pristupiti vrijednosti te promjenljive odnosno "onome na šta pokazivač pokazuje" (eng. pointee). To se postiže pomoću **operatora dereferenciranja** gdje mi možemo ***indirektno*** pristupiti promjenljivoj i promijeniti/pročitati njenu vrijednost. Naravno, na osnovu do sada rečenog, jasno je da mi pristupamo memorijskoj lokaciji gdje se promjenljiva nalazi. Operator dereferenciranja ima simbol zvjezdice (`*`) i on je **unarni** operator (primjenjuje se samo nad jednim operandom).

Recimo da imamo kao ranije:

```
1 int broj = 3;
2 int *p = &broj; /* adresa od 'broj' */
3 *p = 10;
```

U liniji 3 iskoristili smo operator dereferenciranja da bismo pristupili onome na šta `p` pokazuje, a to je vrijednost na adresi koja se nalazi u tom pokazivaču. U našem primjeru imali smo da je promjenljiva `broj` dobila adresu 0x1500 te da je vrijednost promjenljive `p` 0x1500. Kada napišemo `*p` mi pristupamo vrijednosti na toj adresi, tako da operator dereferenciranja `*p` možemo čitati kao "vrijednost na adresi `p`".

Pošto je pokazivač deklarisan sa `int *p`, izraz `*p` je tipa `int` i može se nalaziti svugdje gdje i promjenljiva tipa `int`. Štaviše, `*p` je *l-vrijednost* – vrijednost koja se može nalaziti sa lijeve strane naredbe dodjele, tako da smo u liniji 3 dodijelili vrijednost 10. Drugim riječima, "vrijednost na adresi `p`" je postala 10, a pošto pokazivač `p` pokazuje na promjenljivu `broj`, ova promjenljiva će dobiti vrijednost 10! Pokažimo čitav

1777098

program:

```
1  #include <stdio.h>
2  int main() {
3    int broj = 3;
4    int *p = &broj;
5    *p = 10; /* dereferenciranje! */
6    printf ("broj: %d\n", broj); /* 10 */
7    printf ("*p: %d\n", *p); /* 10 */
8    /* sada mijenjamo promjenljivu 'broj'! */
9    broj = 40;
10   printf ("broj: %d\n", broj); /* 40 */
11   printf ("*p: %d\n", *p); /* 40 */
12   return 0;
13 }
```

Operator dereferenciranja je primijenjen u 5. liniji gdje smo promjenljivu `broj` *indirektno promijenili* (odnosno, pisali smo na memorijskoj adresi gdje se promjenljiva `broj` nalazila). To je dovelo do toga da u liniji 6 bude ispisano "broj: 10" iako je `broj` prvobitno inicijaliziran na 3. Dereferenciranje smo izvršili i u liniji 7 gdje smo *očitali* vrijednost onoga na šta pokazivač pokazuje. Naravno, promjenom promjenljive `broj` promijenio se i onaj dio memorije gdje se ona nalazi (sjetimo se da je promjenljiva samo ime za memorijsku lokaciju), pa će posljednji poziv `printf` funkcije u liniji 11 ispisati 40 iako nismo mijenjali pomoću pokazivača, odnosno, ispis programa će biti:

```
broj: 10
*p: 10
broj: 40
*p: 40
```

Dereferencirani pokazivač (`*p`) se može koristiti u bilo kojem kontekstu gdje se može koristiti i promjenljiva, pa su tako moguće i konstrukcije ispod. Zapravo, (dereferencirani) pokazivač se ponaša kao drugo ime za promjenljivu na koju pokazuje.

```
*p = 10 + broj;
int novi = *p + 13;
broj = *p + *p;
*p += 13;
novi = *p * *p;
```

Posebno je interesantna posljednja linija u kojoj se znak zvjezdica koristi sa *dva različita značenja*: dereferenciranje i množenje. Tačnije, znak zvjezdica u jeziku C ima tri uloge: kod deklaracije pokazivača (da označimo da je deklarisana promjenljiva pokazivač) i dva puta ulogu operatora, unarni operator (dereferenciranje) i binarni operator (množenje) koji se izvršava nad dva operanda.

✓ *Treba primijetiti da zvjezdica prilikom deklarisanja (kao npr.* `float *p;`*) ima potpuno drugačije značenje od dereferenciranja (kao npr.* `*p = 3.25;`*).*

✓ Znak zvjezdica `*`

Kao i kod deklarisanja pokazivača, prevodilac će zanemariti razmake između operatora dereferenciranja i pokazivača, pa je svejedno

17570198

da li pišemo `* p` ili `*p`. Tako smo kod izraza `*p * *p` dodali par razmaka kako bi bilo jasno da je u pitanju množenje dereferenciranih pokazivača `p`, ali potpuno isto značenje bi imao izraz `*p**p` ili `* p * * p` ili čak `* p** p`.

Moguće je pokazivač preusmjeriti, tako da on pokazuje na neku drugu adresu nego što je prvobitno pokazivao:

```
1 int broj1 = 3;
2 int broj2 = 5;
3 int *pok; /*  ni na sta */
4 pok = &broj1;
5 *pok = 10000; /* mijenjamo broj1 */
6 pok = &broj2;
7 *pok = -12; /* sad mijenjamo broj2 */
8 printf ("broj1: %d, broj2: %d\n", broj1, broj2);
```

U liniji 4 smo postavili da pokazivač `pok` pokazuje na promjenljivu `broj1`, a nakon linije 6 on će sada pokazivati na promjenljivu `broj2`, tako da će se na ekranu ispisati

```
broj1: 10000, broj2: -12
```

Ne smijemo dereferencirati neinicijalizirane pokazivače, kao ni pokazivače koji pokazuju na memoriju koja *ne pripada* našem programu. To dovodi do *nedefinisanog ponašanja* (engl. *undefined behavior*) i posljedice su nepredvidljive. Isto vrijedi i za dereferenciranje nul-pokazivača. U praksi, takvi programi će se najčešće krahirati, mada nije garantovano da će se to desiti.

```
int *pok1; /* neinicijaliziran! (npr. 0x123) */
/* mozda pisemo na nedozvoljenu lokaciju? */
*pok1 = 10; /* ned. ponasanje, PROBLEM! */
int *pok2 = NULL; /* nul-pokazivac */
*pok2 = 3; /* ned. ponasanje, vjerovatno KRAH! */
```

Mnogi će sada postaviti pitanje: kako onda dodijeliti vrijednost pokazivaču? Ranije smo vidjeli da direktno pridruživanje broja pokazivaču (bez operatora `*`) nema smisla, a sada vidimo da i kada koristimo taj operator dolazi do krahiranja. Odgovor je da pokazivač najprije mora pokazivati na neku promjenljivu da bismo zatim mogli pridružiti vrijednost toj promjenljivoj preko pokazivača:

```
int broj;
int *pok1 = &broj; /* inicijalizacija pokazivaca */
*pok1 = 10; /* ovo je sada ok */
```

U primjeru iznad (druga po redu linija) bitno je uočiti da je znak zvjezdice `*` dio deklaracije promjenljive `pok1` a ne operator dereferenciranja! Dakle, primjer iznad ne bi bio jednak kodu:

```
int *pok1;
*pok1 = &broj;
```

1/15/01198

koji je neispravan jer predstavlja dereferenciranje neinicijaliziranog pokazivača. Razlika je jasnija kada koristimo stil u kojem je oznaka tipa `int*` napisana spojeno:

```
int* pok1 = &broj; /* inicijalizacija pokazivaca */
```

7.2 *Pokazivači kao parametri funkcija*

Naravno, kao i svi ostali tipovi, i pokazivački tipovi mogu biti parametri funkcija što omogućava da funkcija direktno modificira promjenljive koje su deklarisane u drugim funkcijama (ovo je zapravo jedan od glavnih razloga zbog kojih su pokazivači i uvedeni u jezik C). Recimo da želimo napisati funkciju koja će promijeniti svoj parametar, npr. da postavi njegovu vrijednost na π. Naivni pokušaj kao što je u primjeru ispod neće raditi jer funkcija dobija *kopiju* promjenljive, modificiranjem kopije ne modificiramo i originalnu promjenljivu.↑

↑ Poglavlje 6.2.7, str. 196.

```
#include <stdio.h>
void postavi_na_pi (double n) {
  n = 3.1415925635; /* mijenja KOPIJU */
}
int main() {
  double broj = 0.5;
  postavi_na_pi (broj);
  printf ("Broj: %f\n", broj);
  return 0;
}
```

Program iznad će ispisati "Broj: 0.500000" što pokazuje da funkcija nije promijenila proslijeđenu promjenljivu. Poziv iznad se naziva *poziv po vrijednosti* ili *poziv po kopiji* jer prosljeđujemo kopiju promjenljive i funkcija ne može promijeniti original.

✓ Poziv po vrijednosti

Traženi efekat (da ipak promijenimo promjenljivu) postižemo tako što kao parametar funkcije navedemo *pokazivač* (tip `double *`), a prilikom poziva funkcije proslijedimo *adresu* promjenljive:

```
#include <stdio.h>
void postavi_na_pi (double *n) { /* pokazivac! */
  /* dereferenciramo pokazivac */
  *n = 3.1415925635;
}
int main() {
  double broj = 0.5;
  postavi_na_pi (&broj); /* adresa! */
  printf ("Broj: %f\n", broj); /* 3.141593 (zaokr.) */
  return 0;
}
```

Ovo se naziva *poziv po referenci*[3] jer se ne prosljeđuje promjenljiva nego referenca na promjenljivu (odnosno, njena adresa) i pomoću te reference je moguće mijenjati vrijednost promjenljive.

[3] Poziv po referenci ne treba miješati sa C++ referencama (koje u programskom jeziku C ne postoje).

Sad nam može biti jasnije zašto smo `scanf` funkciji morali prosljeđivati *adresu* promjenljive. Ova funkcija mijenja vrijednost promjenljive

1/15/01/98

tako što u nju upisuje ono što je korisnik unio putem tastature, pa joj je potrebna adresa promjenljive. Također nam može biti jasno zašto nam je dozvoljeno uzimati adresu samo promjenljivih, ali ne i izraza (tipa `&(broj + 4)`) jer, ako bismo napisali npr. `pok = &(broj + 4); *pok = 100` to bi bilo kao da pišemo `broj + 4 = 100` što nema smisla.

7.2.1 *Vraćanje više vrijednosti iz funkcije*

Kao što smo vidjeli, pomoću pokazivača funkcija može da promijeni vrijednost proslijeđenog argumenta. Na ovaj način se postiže "trik" pomoću kojeg iz funkcije "vraćamo" više vrijednosti. Znamo da na standardan način, pomoću `return` naredbe, to nije moguće jer se može vratiti samo jedna vrijednost.

> ***Zadatak 1.*** *Napišite funkciju "pretvori" koja kao parametar prihvata decimalnu vrijednost stepeni a vraća ekvivalentnu vrijednost pretvorenu u stepene, minute i sekunde. Na primjer, 112.53 stepeni nakon pretvaranja je 112° 31′ 48″.*

Ono što prvo trebamo zaključiti jeste da funkcija treba prihvatiti realan broj kao ulaz (stavit ćemo da je to tip `double`). Dalje, funkcija treba da ima 3 izlaza (stepene, minute i sekunde), a znamo da to ne možemo uraditi na uobičajen način. Umjesto toga napravićemo funkciju koja će ukupno imati 4 parametra, pri čemu će preostala tri biti pokazivači na izlazne vrijednosti, a sama funkcija neće vraćati ništa pomoću `return` naredbe (povratni tip će biti `void`).

```
1  #include <stdio.h>
2  void pretvori (const double dec, int *step, int *min, int *sek) {
3    /* stepeni su cijeli dio od dec */
4    *step = (int)dec;
5    /* decimalni dio pomnozen sa 60 daje minute */
6    double temp = (dec - *step) * 60.0;
7    /* uzimamo cijeli dio od temp */
8    *min = (int)temp;
9    /* sekunde su decimalni dio minuta pomnozen sa 60 */
10   *sek = (int)((temp - *min) * 60.0);
11 }
12 int main() {
13   double dec;
14   printf ("Unesite stepene: ");
15   scanf ("%lf", &dec);
16   int stepeni, minute, sekunde;
17   /* prosljedjujemo adrese */
18   pretvori(dec, &stepeni, &minute, &sekunde);
19   printf ("%.2f stepeni je %ddeg %dmin %dsec\n", dec, stepeni, minute,
        sekunde);
20   return 0;
21 }
```

Pokazivački parametri u funkciji su cjelobrojnog tipa (`int *`), jer su stepeni, minute i sekunde cijeli brojevi. Zapravo, za sekunde se mogao staviti realan broj pošto one također mogu biti decimalne, npr. 35° 11′

56.234", ali ćemo se zadovoljiti i zaokruženom vrijednosti.

Prvo analizirajmo `main` funkciju. Unosimo sa standardnog ulaza broj tipa **`double`** (ne zaboravimo da je formatni string `%lf` za unos promjenljivih tipa **`double`**, a `%f` za ispis istih). Nakon toga smo deklarisali 3 cjelobrojne promjenljive, te njihove adrese proslijedili `pretvori` funkciji koja će ih promijeniti. Na kraju smo ispisali tako modificirane vrijednosti.

Što se tiče funkcije `pretvori`, ona je prilično objašnjena komentarima. Napomenimo da korištenje kalupa za pretvaranje u cijele brojeve (npr. 8. linija) nije obavezno, jer se prilikom dodjele (**`int = double`**) automatski odsiječe decimalni dio. Ipak, preporučljivo je da napišemo kalup, jer se time jasnije vidi namjera šta se želi postići, a u nekim jezicima (tzv. strictly-typed jezici) je i obavezno. Također, primijetimo da je prvi parametar konstantan (odnosno tipa **`const double`**). Dobra praksa je *sve* promjenljive (pa i parametre) koje ne trebaju biti modificirane deklarišemo *konstantnim*. Objasnimo sada i algoritam korišten u rješavanju zadatka:

- neka je α decimalni dio stepeni (iz primjera: $\alpha = 112.53$)
- stepeni su: **cijeli dio** od α ($stepeni = 112$)
- necijeli dio je $\alpha - stepeni = 0.53$
- kada taj necijeli dio pomnožimo sa 60, $0.53 \cdot 60 = 31.8$ dobivamo minute
- minute su **cijeli dio** od toga odnosno $minute = 31$.
- necijeli broj od 31.8 je 0.8, kada njega pomnožimo sa 60, dobivamo 48 što predstavlja sekunde (ovdje su sekunde cijeli broj što je slučajnost, da to nije bio slučaj odnosno da smo dobili npr. 48.356, onda uzimamo cijeli broj od toga što je 48). Konačno, $sekunde = 48$.

Primjer izlaza programa:

```
Unesite stepene: 67.234
67.23 stepeni je 67deg 14min 2sec
```

Prikažimo još jedan primjer funkcije koja vraća više vrijednosti. Ovaj put funkcija neće samo dodjeljivati vrijednosti promjenljivim (pri čemu uopšte ne čita šta je prethodno bilo u promjenljivim) kao kod funkcija `postavi_na_pi` i `pretvori`, nego će i iskoristiti šta se u promjenljivim nalazilo prije poziva funkcije. Također, ovo će biti jako korisna funkcija i vrlo često se koristi.

Zadatak 2. *Napisati funkciju "zamijeni" koja mijenja vrijednosti proslijeđenih argumenata. Npr. ako imamo* `a = 10, b = 13` *nakon poziva funkcije* `zamijeni (a, b);` *vrijednosti trebaju biti* `a = 13, b = 10`*. Prototip*

1/15/0198

funkcije treba da bude

`void zamijeni (double *a, double *b)`

Naravno, jasno je da i ovdje moramo proslijediti adrese promjenljivih koje mijenjamo jer u suprotnom bismo mijenjali samo kopije. Vrlo često se može vidjeti da je zadan prototip funkcije u specifikacijama zadatka, razlog za to je da drugi ljudi mogu koristiti napisane funkcije.

```
1  #include <stdio.h>
2  void zamijeni (double *prvi, double *drugi) {
3    double temp = *prvi;
4    *prvi = *drugi;
5    *drugi = temp;
6  }
7
8  int main() {
9    /* necemo unositi brojeve... */
10   double b1 = 100.4, b2 = 40.77;
11   printf ("Prije poziva:\n b1 = %g \n b2 = %g\n\n",
12             b1, b2);
13   zamijeni (&b1, &b2);
14   printf ("Nakon poziva:\n b1 = %g \n b2 = %g\n",
15           b1, b2);
16   return 0;
17 }
```

Izlaz programa će izgledati ovako:

```
Prije poziva:
 b1 = 100.4
 b2 = 40.77

Nakon poziva:
 b1 = 40.77
 b2 = 100.4
```

Funkciju `main` nije potrebno pretjerano objašnjavati; ispisujemo vrijednosti prije poziva `zamijeni` funkcije, te onda pozivamo funkciju (koja mijenja njihove vrijednosti) te ponovo ispisujemo njihove vrijednosti da se "uvjerimo" da je zamjena bila uspješna. Kasnije ćemo objasniti zašto smo ovo *uvjerimo* stavili pod navodnike, jer ni kada program ispiše vrijednosti nakon "uspješne" zamjene, to ne znači da je naš program ispravan.

Funkcija `zamijeni` ima kratko tijelo: prvo u (običnu) promjenljivu smjestimo ono na šta pokazivač `prvi` pokazuje (u primjeru iznad u `temp` smo smjestili vrijednost 100.4). Nakon toga *prebrišemo* ono na šta pokazuje `prvi` i tu smještamo ono na šta pokazuje pokazivač `drugi` (40.77). Sada je stanje da i ono gdje pokazuje `prvi` i ono gdje pokazuje `drugi` ima istu vrijednost odnosno `*prvi` je jednak `*drugi`. Nakon toga, u `*drugi` stavimo ono što je u privremenoj promjenljivoj `temp`.

Objasnimo ovo ponovo pomoću ilustracija. Nakon što smo deklarisali promjenljive `b1` i `b2` u 12. liniji, imamo ovakvu reprezentaciju u memoriji:

1/15/01198

...	b1	...	b2	...	...	...	...	...	...	...
?	100.4	?	40.77	?	?	?	?	?	?	?
...	0x200	...	0x300	...	...	...	...	...	...	...

Slika 7.5: `b1` i `b2` se nalaze na fiktivnim adresama 0x200 i 0x300

Namjerno je predstavljeno da promjenljive `b1` i `b2` nisu u memoriji jedna iza druge jer to ne mora biti slučaj (iako je vjerovatno tako). Postoji još jedan razlog zašto promjenljive nisu jedna iza druge: ne znamo koliko jedna promjenljiva zauzima bajta memorije. Vjerovatno će `double` tip zauzeti 8 bajta, ali mi nemamo tu garanciju pa smo "razdvojili" promjenljive da uklonimo sve pretpostavke o arhitekturi, platformi i kompajleru. Također, adrese *namjerno* pišemo kao *heksadecimalne brojeve* iz nekoliko razloga: većina funkcija za ispis će adrese ispisati u heksadecimalnoj bazi jer je tako čitljivije, imamo manje cifri nego u binarnom zapisu, a lakša je konverzija u binarni sistem iz heksadecimalnog nego iz decimalnog; dalje, glavni razlog ovdje za heksadecimalnu predstavu je ipak da razlučimo šta su adrese a šta vrijednosti. Očito je da je 100.4 vrijednost, a adresa na kojoj se to nalazi je 0x200 pa je tako lakše "pratiti" ilustraciju.

Nakon što pozovemo funkciju `zamijeni (&b1, &b2)`, kreiraju se dvije ***lokalne promjenljive*** `prvi` i `drugi`. Ove promjenljive su pokazivačkog tipa i one su vidljive samo unutar funkcije.↑ Tada je memorijska predstava kao na slici 7.6 (lokalne promjenljive `zamijeni` funkcije su obojene narandžastom bojom):

↑ Poglavlje 6.2.7, str. 196.

...	b1	...	b2	...	prvi	...	drugi	...	...	...
?	100.4	?	40.77	?	0x200	?	0x300	?	?	?
...	0x200	...	0x300	...	0x700	...	0x800	...	...	...

Slika 7.6: `prvi` i `drugi` pokazuju na `b1` i `b2` respektivno

Kada smo proslijedili adrese, pokazivači `prvi` i `drugi` pokazuju na `b1` i `b2` respektivno (to možemo upratiti i gledajući vrijednosti tih pokazivača, 0x200 i 0x300). Također, pošto su i pokazivači promjenljive, i one su zauzele određenu memoriju na steku.

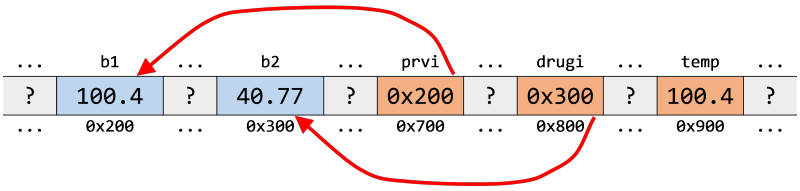
Nakon linije

```
3  double temp = *prvi;
```

kreirana je *obična* (ne pokazivačka) promjenljiva `temp` u koju smo smjestili ono na šta `prvi` pokazuje (a to je vrijednost 100.4 u primjeru iznad). Ova promjenljiva nam služi da sačuva vrijednost `b1`, jer ćemo tu vrijednost prebrisati u sljedećoj liniji. Pogledajmo šta se dešava na slici 7.7:

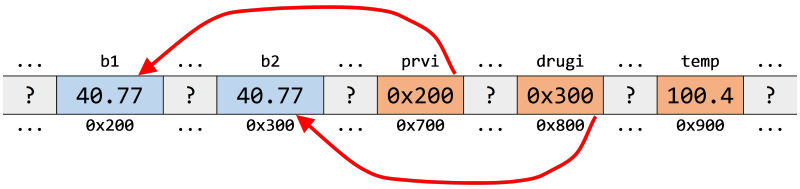
1/15/01 98

Na sljedećoj slici (sl. 7.8) ćemo vidjeti efekte 4. linije u kojoj imamo



Slika 7.7: Kreirana je promjenljiva `temp` na adresi 0x900 koja ima vrijednost `*prvi` odnosno `b1`, a to je 100.4

`*prvi = *drugi;`. Tada i `b1` i `b2` imaju *istu* vrijednost. Naravno, i `*prvi` i `*drugi` imaju istu vrijednost iako pokazuju na različite promjenljive odnosno vrijednosti samih pokazivača nisu jednake (`prvi != drugi` ali `*prvi == *drugi`).

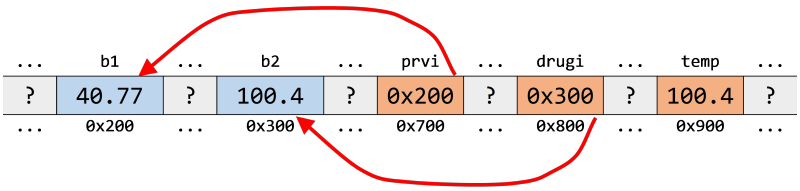


Slika 7.8: Zbog ovakvog stanja nam je potrebna `temp` promjenljiva koja je sačuvala prvobitnu vrijednost `b1` promjenljive!

Konačno, posljednja linija funkcije

```
5  *drugi = temp;
```

ima efekat da u promjenljivu `b2` (odnosno `*drugi`) pohranimo sačuvanu vrijednost iz `temp` promjenljive. Pogledajmo sliku 7.9 da shvatimo šta se tačno desilo.



Slika 7.9: Vraćena je vrijednost iz `temp` u `*drugi` (`b2`)

Nakon što se izvršavanje funkcije završilo, vrijednosti promjenljivih `b1` i `b2` su zamijenjene. Naravno, *uništene* su promjenljive `prvi`, `drugi` i `temp`! One više ne postoje (što ćemo moći jasno vidjeti na slici 7.10). Ovo će nam kasnije biti važno: lokalne promjenljive umiru (prestaju postojati) nakon završetka funkcije pa ne smijemo npr. vraćati njihove adrese u funkcijama koje vraćaju pokazivače, što ćemo obraditi kasnije.

17570198

7.2.2 *Pogrešne implementacije `zamijeni` funkcije*

Pošto se ovo poglavlje bavi pokazivačima koji su "osjetljivi" na raznorazne greške, a pogotovo što se takve greške *jako teško* detektiraju, potrebno je ukazati na probleme sa kojima se susrećemo u radu

...	b1	...	b2	...	...	...	...	...	...	...
?	40.77	?	100.4	?	?	?	?	?	?	?
...	0x200	...	0x300	...	...	...	...	...	...	...

Slika 7.10: Lokalne promjenljive su uništene a promjenljive b1 i b2 su zamijenile vrijednosti

sa pokazivačima. To je najbolje uraditi na konkretnim primjerima – *pogrešnim* implementiranjem funkcije `zamijeni`!

```
void zamijeni (double *prvi, double *drugi) {
  /* temp je sada pokazivac */
  double *temp = prvi;
  prvi = drugi;
  drugi = temp;
}
```

Ako iskoristimo istu `main` funkciju kao i u ispravnoj verziji programa, onda će ispis biti sljedeći:

```
Prije poziva:
a = 100.4
b = 40.77

Nakon poziva:
a = 100.4
b = 40.77
```

Vidimo da nešto ne štima. Vrijednosti promjenljivih *nisu* zamijenjene! Šta se zapravo desilo? Zamijenili smo samo pokazivače (njihove vrijednosti), ali *ne i ono na šta pokazivači pokazuju*! Dobili smo efekat da `prvi` pokazuje na `b2` a `drugi` da pokazuje na `b1`. Efektivno nismo ništa postigli, jer će svakako promjenljive `prvi` i `drugi` prestati postojati nakon što se funkcija završi. Ako pogledamo sliku 7.6, sa pogrešnom implementacijom `zamijeni` funkcije, `prvi` bi sadržavao adresu 0x300, a `drugi` adresu `0x200`.

Pređimo sada na drugu pogrešnu implementaciju koja se može prevesti. Ovdje je problem što je "programer" stavljao znak zvjezdica "dok ne proradi", ne vodeći računa o logici.

```
void zamijeni (double *prvi, double *drugi) {
  double temp = *prvi;
  prvi = drugi; /* dodjeljujemo pokazivace! */
  *drugi = temp;
}
```

Problem u kôdu iznad je taj što smo mi vršili dodjelu pokazivača (a ne onog na šta oni pokazuju), pa smo dobili efekat da oba pokazivača pokazuju na `b2`. Nakon što se završilo izvršavanje funkcije, obje promjenljive (`b1` i `b2`) imaju istu vrijednost i to 100.4! Posljednja linija je promijenila ono na šta `drugi` pokazuje i stavila mu vrijednost od `b1`, ali i `prvi` pokazuje isto gdje i `drugi` (oba pokazuju na `b2`)!

Izgleda da nikad kraja pogrešnim implementacijama, zar ne? Većina problema ima samo jedno ispravno rješenje, a sva ostala su pogrešna!

1/15/01/98

Ovdje ćemo pokazati još tri pogrešna rješenja, od čega su dva vezana isključivo za jezik C, odnosno u C++u se ne bi ni mogla prevesti. Također, kod svih dosadašnjih pogrešnih rješenja mogli smo vidjeti da nisu ispravna jednostavnim testiranjem funkcije. Kod sljedećeg rješenja *to nije slučaj*! Program "izgleda" kao da radi, čak i ispisuje ispravne rezultate ali to i dalje nije ispravan program!

Neki programeri ne vole odmah inicijalizirati promjenljive nego ih prvo deklarišu a nakon toga inicijaliziraju. To može rezultirati sljedećim pogrešnim kôdom:

```
void zamijeni (double *prvi, double *drugi) {
  double *temp;
  *temp = *prvi;
  *prvi = *drugi;
  *drugi = *temp;
}
```

Šta je problem iznad? Problem je što promjenljiva `temp` *nije inicijalizirana*, nego sadrži neko smeće, odnosno vrijednost koja se zadesila na tom mjestu u memoriji za vrijeme deklarisanja promjenljive. Rekli smo da ***nikad ne smijemo dereferencirati nevalidne pokazivače***, a nevalidni pokazivači su svi pokazivači koji nisu inicijalizirani adresom neke druge promjenljive, ili rezultatom neke funkcije koja vrši dinamičku alokaciju memorije, ili nisu nul-pokazivači. Podsjetimo se da je nedefinisano ponašanje ako dereferenciramo neinicijaliziran pokazivač, a to je upravo urađeno u liniji `*temp = *prvi`. Program može da izgleda kao da "radi", ali to ne mora biti slučaj. Npr. može se desiti da `temp` ima neku zatečenu vrijednost koja pripada memoriji gdje se izvršava operativni sistem ili čak može da ne sadrži broj nikako (`NaN`). U svakom slučaju nikad ne smijemo dozvoliti da imamo nedefinisano ponašanje u našem programu. Program iznad sa istom funkcijom `main` kao u ispravnoj verziji programa nekad krahira (testirano na serveru sa 64 bitnim kompajlerom) a nekad ispiše tačan rezultat!

Sljedeća dva rješenje su specifična za jezik C i cjelobrojne tipove (`char`, `short`, `int` i `long`) te njihove nepredznačne verzije (`unsigned`). U C++-u se neće ni prevesti a sa drugim tipovima (`double`, `float`, ...) neće se prevesti ni u C-u. Pogledajmo o čemu se radi:

```
#include <stdio.h>
void zamijeni (int *prvi, int *drugi) {
  int temp = prvi;
  *prvi = *drugi;
  *drugi = temp;
}
int main() {
  /* necemo unositi brojeve... */
  int b1 = 100, b2 = 40;
  printf ("Prije poziva:\n b1 = %d \n b2 = %d\n\n",
  b1, b2);
  zamijeni (&b1, &b2);
  printf ("Nakon poziva:\n b1 = %d \n b2 = %d\n",
  b1, b2);
```

```
  return 0;
}
```

Ovdje je greška u prvoj liniji funkcije! Izvršili smo nedozvoljenu dodjelu između različitih tipova! U promjenljivu tipa `int` smo stavili vrijednost pokazivača (tipa `int *`) što je, kao što smo već rekli, zabranjeno! Kompajler će nas upozoriti na ovu dodjelu, ali je činjenica da mnogi programeri ne obraćaju pažnju na kompajlerska upozorenja iako su jako korisna i većinom nam ukazuju na potencijalne greške. Rezultati dodjele iznad su nepredvidljivi (nedefinisano ponašanje), ali u većini slučajeva ispis bude sličan sljedećem:

```
Prije poziva:
b1 = 100
b2 = 40

Nakon poziva:
b1 = 40
b2 = 10485324
```

Posljednja vrijednost b2 ima vrijednost adrese od `b1` odnosno `&b1`. Druga pogrešna implementacija specifična za C je sljedeća:

```
void zamijeni (int *prvi, int *drugi) {
  int *temp = *prvi;
  *prvi = *drugi;
  *drugi = *temp;
}
```

Program iznad će vjerovatno krahirati u trećoj liniji funkcije koja glasi `*drugi = *temp`. Ustvari, sporna je `*temp` konstrukcija jer dereferenciramo pokazivač `temp`, a u njega smo smjestili vrijednost onoga na šta `prvi` pokazuje (tj. smjestili smo vrijednost promjenljive `b1` koja, u primjeru iznad, iznosi 100). Obratite pažnju da se u liniji `int *temp = *prvi` znak zvjezdice pojavljuje u različitim značenjima: najprije kao oznaka tipa promjenljive `temp`, a zatim kao operator dereferenciranja promjenljive `prvi`. Dakle, `temp` će biti pokazivač koji pokazuje na adresu datu u memorijskoj lokaciji na koju pokazuje `prvi` (što bi imalo smisla samo ako je prvi *dvostruki pokazivač*↑). Uz dosta sreće se može naštimati da i ovaj program "radi" (odnosno, ne krahira), ali ne vrijedi to ni pokušavati jer pošto `prvi` nije dvostruki pokazivač samim time imamo nedefinisano ponašanje.

↑ Poglavlje C.3, str. 381.

Prikazano je 5 *neispravnih* i jedna *ispravna* implementacija `zamijeni` funkcije. Što je zanimljivo, *sve implementacije se razlikuju* ***samo*** *u postavljanju* ***zvjezdice*** *na (pogrešna) mjesta*! Vrlo je važno da razumijemo značenje zvjezdice kada deklarišemo pokazivač te kada ga dereferenciramo. Funkcija `zamijeni` je trivijalna, a ipak smo na toliko mjesta mogli da pogriješimo, zamislimo šta bi se desilo sa nekom komplikovanijom funkcijom?

1/EE/01/98

7.3 Pokazivači i nizovi

Do prave izražajne moći pokazivači dolaze kada se primjenjuju nad nizovima. Pošto znamo da su nizovi u memoriji kontinualni (odnosno, svaki element niza je jedan pored drugog bez obzira na memorijski model), to znači da možemo pokazivač postaviti da pokazuje na jedan element niza i vrlo jednostavno pristupiti prethodnom ili sljedećem elementu tog istog niza.

Pređimo odmah na primjer:

```
int niz[] = {6, 12, 0, 1, -3};
int *pok = &niz[0];
```

U prvoj liniji je deklarisan niz od 5 elemenata (dimenziju nismo morali navoditi jer će to kompajler uraditi za nas↑). U drugoj liniji je deklarisan *običan* pokazivač na cijeli broj koji je inicijaliziran adresom prvog elementa niza (prisjetimo se da indeksiranje niza počinje od nule).

↑ Poglavlje 5.1.2, str. 134.

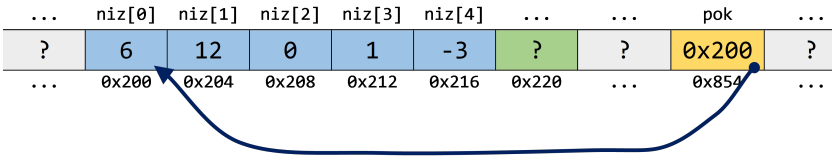
Za pokazivač `pok` kažemo da pokazuje na *prvi element niza* (sl. 7.11), a često se spominje i konfuzna fraza "pokazuje na niz" odnosno za `pok` se kaže "pokazivač na niz". Iako je ovo pogrešno jer `pok` ***nije*** *pokazivač na niz*[4] nego na element niza, ipak se ova terminologija vrlo često koristi kada se iz konteksta zna na šta se misli.

[4] Pokazivač na niz kao cjelinu je jedna specifična zvjerka. Više o njemu ćemo govoriti u dodatku C.3.3, str. 391.

✓ Pokazivač na niz

✓ *Pod "pokazivačem na niz" ćemo podrazumijevati na pokazivač na prvi element niza. Ukoliko se ipak misli na pokazivač na niz (kao cjelinu) onda će to biti posebno naglašeno.*

Prije nego pređemo na operacije koje možemo raditi sa ovakvim pokazivačem, podsjetimo kako ovo sve izgleda u memoriji. Pokazivač `pok` može pokazivati na bilo koji element niza, ne mora na prvi. Deklaracija (odnosno, inicijalizacija) je slična:



Slika 7.11: Promjenljiva `pok` pokazuje na prvi element niza

```
int *pok = &niz[2];
pok = &niz[4];
```

Kako se elementi niza ponašaju kao obične promjenljive (to one i jesu) tako i pokazivači na elemente niza slijede analogiju pokazivača na obične promjenljive, pa tako dereferenciranjem pokazivača dobivamo element niza na kojeg pokazuje:

```
int niz[] = {6, 12, 0, 1, -3};
int *pok = &niz[1];
*pok = 50; /* mijenja 2. element niza */
printf ("niz[1]: %i", niz[1]); /* ispisuje 50 */
```

1 / 3 / 7 / 0 1 98

7.3.1 Pokazivačka aritmetika

Ukoliko neki pokazivač pokazuje na element niza (a takav je naš pokazivač `pok`), onda nad njim možemo vršiti **pokazivačku aritmetiku** ili **aritmetiku nad pokazivačima**. Ona obuhvata neke aritmetičke operacije kojima možemo promijeniti na šta trenutni pokazivač pokazuje.

✓ *Pokazivačka aritmetika je skup pravila koja se mogu primijeniti isključivo nad pokazivačima na elemente niza s ciljem pristupa drugim elementima niza. Ona obuhvata ograničene operacije nad pokazivačima sa kojima pokazivače možemo "pomjerati" ostvarujući efekat pristupa elementima niza.*

✓ Pokazivačka aritmetika

Pokazivačka aritmetika obuhvata sabiranje i oduzimanje pokazivača sa cijelim brojem te oduzimanje dvaju pokazivača. Sve operacije koje imaju smisla nad pokazivačima na elemente niza date su u tablici 7.2.

Tablica 7.2: Skup pravila koja čine pokazivačku aritmetiku

#	Operacija	Opis i rezultat	Primjer
1	`pok + n`	Dodavanje *cijelog broja* pokazivaču ne mijenjajući ga. **Rezultat:** pokazivač na *n*-ti član niza *iza* `pok`	`int *p1 = pok + 3`
2	`pok - n`	Oduzimanje *cijelog broja* od pokazivača ne mijenjajući ga. **Rezultat:** pokazivač na *n*-ti član niza *ispred* `pok`	`int *p2 = pok - 5`
3	`pok += n`	Uvećavanje pokazivača za *cijeli broj*. **Rezultat:** pomjera `pok` za *n* članova niza iza trenutnog	`pok += 3`
4	`pok -= n`	Umanjivanje pokazivača za *cijeli broj*. **Rezultat:** pomjera `pok` za *n* članova niza ispred trenutnog	`pok -= 5`
5	`pok++` `++pok`	Prefiksno i postfiksno inkrementiranje pokazivača. **Rezultat:** pomjera `pok` na član niza iza trenutnog	`++pok`
6	`pok--` `--pok`	Prefiksno i postfiksno dekrementiranje pokazivača. **Rezultat:** pomjera `pok` na član niza ispred trenutnog	`pok--`
7	`p2 - p1`	Oduzimanje jednog pokazivača od drugog. **Rezultat:** cijeli broj koji označava broj članova niza između dva pokazivača	`int n;` `n = p2 - p1;`

1757018

Iz ovog slijedi da je dozvoljeno oduzimati pokazivače[5] a *nije dozvoljeno* sabirati ih. Isto tako nije dozvoljeno dijeljenje i množenje pokazivača. Razlog za to je što se ne zna šta bi logički trebao da predstavlja npr. količnik dva pokazivača (odnosno, dvije adrese). Ukoliko

[5] Radi potpunosti napomenimo da je razlika pokazivača tipa `ptrdiff_t` o kojem ćemo govoriti u dodatku C.1.3, str. 361.

pokušamo nešto od navedenog bit će prijavljena sintaksna greška od strane kompajlera. Ipak, pokazivačka aritmetika je definisana ako i samo ako pokazivači pokazuju na elemente istog niza (eventualno mogu pokazivati na jedan element iza niza što ćemo obrazložiti malo kasnije).

Za pokazivače koji ne pokazuju na elemente unutar jednog te istog niza (eventualno i na element iza niza) nije dozvoljeno koristiti pokazivačku aritmetiku. To su: pokazivači na obične promjenljive (koje nisu sastavni dio niza) ili dva pokazivača na elemente dva različita niza ili pokazivač ispred niza. Primjena pokazivačke aritmetike u ovim slučajevima rezultira *nedefinisanim ponašanjem* i posljedice su *nepredvidljive*!

```
int niz1[] = {1, 2, 3}, niz2[] = {4, 5, 6};
int broj;
int *pok1 = &broj;
int *pok2 = &niz1[2], *pok3 = &niz2[0];

/* SLJEDECE JE SINTAKSNO ISPRAVNO, ALI JE BESMISLENO: */
pok1 += 3; /* gdje sad pokazuje pok1? */
int n = pok2 - pok1; /* Nisu isti nizovi! */
pok3 -= 1; /* pok3 sad pokazuje ISPRED niza - ne valja! */
```

Ne smijemo dozvoliti da prilikom mijenjanja pokazivača taj pokazivač pokazuje *izvan* niza (kao `pok3` u primjeru iznad). Ovdje vrijedi jedna iznimka da je pokazivač ipak validan ako pokazuje na element direktno iza posljednjeg a to ćemo objasniti u paragrafu ispod. U nastavku ćemo također ukazati na ove greške kroz primjere.

```
int niz[] = {1, 2, 3, 4, 5};
int *p = &niz[1];
```

U primjeru iznad, validni pokazivači su: `p`, `p + 1`, `p + 2`, `p + 3`, `p - 1`. Također je validan pokazivač iza kraja niza `p + 4`, ali nisu validni ostali pokazivači, npr. `p - 2` ili `p + 5`.

7.3.2 *Sličnost između imena niza i pokazivača*

Kada u vašem programu navedete ime niza samo za sebe (bez uglastih zagrada), ono se automatski konvertuje u *pokazivač na prvi element niza*. Recimo, ako imamo neki niz deklarisan sa `int niz[10];`, tada se ime niza upotrijebljeno samo za sebe (`niz`) automatski pretvara u pokazivač na prvi element (`&niz[0]`). Pogledajmo primjer:

```
int niz[5] = {1, 2, 3, 4, 5};
int *pok;
pok = &niz[0];
/* isto kao i */
pok = niz;
```

Čak štaviše, sasvim su legalne i sljedeće konstrukcije:

```
int niz[4] = {1, 2, 3, 4};
*niz = 10; /* mijenja prvi element */
```

Vidjećemo kasnije da ovo nije "običan" pokazivač nego *konstantni* pokazivač,↑ odnosno adresa na koju pokazuje taj pokazivač ne može se mijenjati.[6]

↑ Poglavlje C.1.4, str. 362.

[6] Intuitivno je da ovaj pokazivač mora biti konstantan jer kompajler ne bi imao nikakvu informaciju da izbriše tu promjenljivu (odnosno oslobodi memoriju) ukoliko mi u programu pomjerimo taj pokazivač, jer ne bi znao gdje je početak tog niza.

Nad imenom niza kao pokazivačem možemo primjenjivati one elemente pokazivačke aritmetike koji ne pomjeraju pokazivač (ne mijenjaju adresu na koju pokazuje). Npr. ako želimo dobiti pokazivač na četvrti element niza (čiji je indeks 3), to možemo uraditi kao u sljedećem primjeru:

```
pok = niz + 3;
/* isto sto i ovo: */
pok = &niz[3];
```

Ako naziv niza možemo posmatrati kao pokazivač, logično je da je moguće i obrnuto: pokazivače, kao i nizove, možemo *indeksirati*. Ako pokazivač `pok` pokazuje na prvi element niza `niz`, tada je `pok[0]` isto što i `niz[0]`. Zapravo, indeksiranje niza je *identično* dereferenciranju pokazivača sabranim sa cijelim brojem, odnosno vrijedi:

```
pok[1] = 10;
/* isto sto i: */
*(pok + 1) = 10;
```

Iznad napisana dva načina pristupa elementima su *sinonimi*. Kako je sabiranje komutativno, vrijedi $pok + n$ je isto kao i $n + pok$. To dovodi do sljedeće zanimljive konstrukcije: `niz[3]` je isto što i `3[niz]`! Pokažimo na primjeru:

```
#include <stdio.h>
int main () {
  int niz[] = {1, 5, 7, 11};
  printf ("2[niz] = %d", 2[niz]);
  return 0;
}
```

Program iznad će ispisati "2[niz] = 7".

Ovo omogućava da relativno indeksiramo nizove. U primjeru ispod pokazivač `pok` pokazuje na 3. element niza (`niz[4]`), tada `pok[1]` pokazuje jedno mjesto naprijed u odnosu na `pok` a to je zapravo `niz[5]`.

```
int niz[] = {9, 3, 4, 1, 2, 8, 19, 0, -1};
int *pok = &niz[4];
pok[1] = 18; /* mijenja niz[5] sa 8 na 18 */
```

Kako je `pok[i]` *isto što* i `*(pok + i)`, a *znamo* da je moguće i oduzimati odnosno `*(pok - n)`, tako je moguće i *negativno indeksiranje*! Npr. iz primjera iznad, `pok` pokazuje na broj 2 (prije izmjene) odnosno na `niz[4]`. Tada je sasvim validno pristupiti *prethodnom* elementu niza sa `pok[-1]` i tada ćemo pristupiti zapravo 4. elementu niza (`niz[3]`). Nastavimo primjer od maloprije:

```
pok[-1]; /* isto sto i niz[3] */
pok[-4]; /* isto sto i  niz[0] */
```

1/5/0198

Naravno, i ovdje možemo izaći van granica niza i dovesti program u nedefinisano ponašanje.

Na pokazivač možemo dodati ili oduzeti cijeli broj n čime efektivno pokazujemo na n elemenata ispred ili iza pokazivača. Nakon dodavanja, dobijamo novi pokazivač koji pokazuje na novu adresu. Ovako možemo *simulirati* indeksiranje: `*(pok + n)` je isto što i `pok[n]`.

```
int *pok1 = niz;
int *pok2 = pok1 + 3;
*pok2 = 5;
/* isto kao i */
*(pok1 + 3) = 5;
```

Važno je primijetiti da su ovdje zagrade obavezne! Ukoliko ih ne stavimo imamo konstrukciju `*pok1 + 3 = 5` što nije legalno! Operator dereferenciranja (`*`) ima veći prioritet nego operator sabiranja (`+`) pa se izraz `*pok1 + 3` interpretira kao `(*pok1) + 3`. Ako ipak želimo povećati pokazivač, pa tako privremeni pokazivač dereferencirati onda moramo staviti zagrade, kao u posljednjoj liniji u kôdu iznad.

Slična priča vrijedi i za dodavanje/oduzimanje cijelog broja pokazivaču mijenjajući ga: `pok1 += 3`. Izraz `*pok1 += 3` ima značenje da ono na šta `pok1` pokazuje uvećamo za 3. A izraz `*(pok1 += 3)` ima značenje da izmijenimo `pok1` tako da pokazuje 3 mjesta iza, pa tako modificirani pokazivač dereferenciramo.

Do sada smo govorili da se ime niza upotrijebljeno samo za sebe automatski konvertuje u pokazivač na prvi element tog niza. Ipak postoje dva izuzetka kada to nije slučaj: kod primjene operatora `sizeof` i kod primjene adresnog operatora `&niz`.

Znamo da `sizeof` operator vraća veličinu promjenljive (ili tipa) u bajtima.↑ Shodno tome `sizeof (pokazivac)` će vjerovatno vratiti 4 ili 8 bajta (moguće su i druge vrijednosti, ovdje pojednostavljujemo), ali će `sizeof (niz)` ipak vratiti broj bajta koje je zauzeo čitav niz tj. svi njegovi elementi. Na taj način možemo saznati veličinu niza:

↑ Poglavlje 3.2.5, str. 83.

```
int niz[] = {1, 2, 5, 4};
const int vel1 = sizeof (niz) / sizeof (int);
/* isto je ispravno (i generalnije): */
const int vel2 = sizeof (niz) / sizeof (niz[0]);
```

Što se tiče izraza `&niz`, on nam daje ranije spomenuti "pokazivač na niz kao cjelinu".↑ Ništa od rečenog ne važi ako probamo poslati niz u funkciju, o čemu govoriti ubrzo.↑

↑ Poglavlje C.3.3, str. 391.
↑ Poglavlje 7.3.4, str. 245.

17570198

7.3.3 *Prefiksni i postfiksni operatori nad pokazivačima*

Kako već znamo, *nema* razlike pisali mi `++x` ili `x++` u kodu kada *ne koristimo* rezultat toga izraza. Npr. u sljedećem kodu linije 3 i 4 rade *istu stvar* bez obzira što u jednoj koristimo prefiksnu a u drugoj postfiksnu varijantu.

```
int *pok = niz;
...
pok++;
++pok;
...
```

Ali ima razlike ukoliko se rezultat ovih operatora ipak koristi, pa ima razlike između 1. i 2. linije u sljedećem kodu:

```
int *p1 = pok++;
int *p2 = ++pok;
```

Ukoliko iskombiniramo dereferenciranje i inkrementiranje onda dolazimo do izraza koje je malo teže protumačiti, npr.

```
int x = *pok++;
int y = ++*pok;
...
```

Da bismo jednostavno mogli tumačiti izraze, iznad kao i još komplikovanije izraze, potrebno je da znamo prioritete navedenih operatora, a u slučaju da operatori imaju iste prioritete, onda moramo naučiti i njihovu asocijativnost.↑ Zahvaljujući tim dvjema informacijama možemo jedinstveno odrediti redoslijed izvršavanja izraza poput onih u primjeru iznad.

↑ Poglavlje 3.2, str. 79.

Operator dereferenciranja ima *isti prioritet* kao i prefiksni operatori (`++` i `--`), ali ima manji prioritet od postfiksnih operatora (`++`, `--`). Zbog navedenog moramo pogledati asocijativnost navedenih operatora kako bismo odredili njihov redoslijed izvršavanja: *lijevo asocijativni* su postfiksni operatori `++` i `--`, dok su operator dereferenciranja i prefiksni `++` i `--` *desno asocijativni*.↑

↑ Poglavlje A, str. 351.

Kada znamo prioritet i asocijativnost možemo razumjeti redoslijed izvršavanja izraza koji uključuju kombinirane operatore dereferenciranja i inkrementiranja ili dekrementiranja. Kako se evaluiraju sljedeći izrazi? (Pretpostavka je da `p` pokazuje na element negdje u sredini niza od dovoljno (10ak) cjelobrojnih elemenata i da je `x` tipa `int`.)

(1) `x = *p++`
(2) `x = (*p)++`
(3) `x = *(p++)`
(4) `x = ++*p`
(5) `x = ++(*p)`
(6) `x = *++p`
(7) `x = *(++p)`

Isto bi vrijedilo (odnosno, istim redoslijedom bi se izrazi evaluirali) da smo umjesto inkrementiranja (`++`) koristili dekrementiranje (`--`). Objasnit ćemo svih 7 iznad napisanih slučajeva iako su samo 4 unikatna te imamo "duplikata" u smislu da neki imaju isto značenje (njih ćemo grupirati u srodne). Prvo, radi lakšeg praćenja, napravit ćemo primjer sa konkretnim podacima i ispisivati rezultate izraza. Također, pretpostavlja se da pokazivač `p` u svakom paragrafu pri objašnjavanju pokazuje na *isti* element niza (u primjeru ispod to je 3. element sa vrijednosti 7).

1779701198

```
int niz[] = {3, 6, 7, 11, 2, 4, 16};
int *p = niz + 2; /* isto kao: 'int *p = &niz[2]' */
int x;
```

Na osnovu promjenljivih deklarisanih u kôdu iznad objasnimo slučajeve numerirane iznad navedenog kôda.

`x = *p++` (1)

U `x` će smjestiti ono na šta `p` pokazuje te će ga nakon toga pomjeriti na sljedeći element.
Tumačenje: postfiksni operator ima veći prioritet od operatora dereferenciranja pa će se izraz iznad evaluirati kao `*(p++)`. Naravno, kako znamo, `p++` će uvećati `p` *tek nakon* što se *evaluacija izraza završi*, te će u `x` staviti ono na šta je `p` pokazivao prije izmjene.
Vrijednosti: `x = 7`, `p` pokazuje na 4. element niza (broj 11). Ovo je isti slučaj kao i (3) odnosno `*(pok++)`.

`x = (*p)++` (2)

U `x` će smjestiti ono na šta `p` pokazuje te će nakon toga uvećati ono na šta `p` pokazuje. Odnosno, izmijenit će element niza!
Tumačenje: zahvaljujući zagradama, prvo će se `p` dereferencirati odnosno `*p` će imati vrijednost 7 (rekli smo da u svakom novom objašnjenju p pokazuje na isti element na koji je pokazivao u kôdu iznad). Onda će se ta vrijednost smjestiti u `x` i inkrementirati nakon što se izvrši evaluacija cijelog izraza.
Vrijednosti: `x = 7`, `p` pokazuje i dalje na 3. element niza ali sada se tu nalazi broj 8! Naravno, pri objašnjavanju sljedećeg izraza nećemo uzimati u obzir ovu modifikaciju niza.

`x = ++*p` (4)

U `x` će smjestiti uvećanu vrijednost onoga na šta `p` pokazuje. Također, izmijenit će i niz.
Tumačenje: u izrazu iznad prefiksni operator `++` ima *isti* prioritet kao i operator dereferenciranja `*`. U tom slučaju gledamo asocijativnost: oba operatora su *desno* asocijativna, pa se onda izraz evaluira s desna na lijevo odnosno kao `x = ++(*p)`.
Vrijednosti: `x = 8`, `p` pokazuje i dalje na 3. element niza gdje je tu sada broj 8. Razlika ovog (`x = ++*p`) i prethodnog izraza (`x = (*p)++`) je samo u vrijednosti koja se smješta u `x`, pokazivač i dalje pokazuje na isti element i u nizu također budu iste vrijednosti. Ovaj slučaj je isti kao i slučaj (5) `x = ++(*p)`.

`x = *++p` (6)

Prvo će povećati pokazivač `p` da pokazuje na sljedeći element (odnosno na `niz[3]`), zatim će se dereferencirati taj element i dodijeliti u

1/5/70198

promjenljivu `x`.
Tumačenje: i ovdje imamo slučaj da prefiksni operator `++` ima *isti* prioritet kao i operator dereferenciranja `*`. Naravno, oba su desno asocijativna, pa izraz iznad evaluiramo sa desne strane i vidimo da će se prvo izvršiti pomjeranje pokazivača. Nakon toga se tako pomjereni pokazivač dereferencira i vrijednost se smjesti u promjenljivu `x`. Izraz se evaluirao kao `x = *(++p)`.
Vrijednosti: `x = 11`, `p` pokazuje na 4. element niza (a tu je broj 11). Jedina razlika ovog i prethodnog (`x = ++*p`) izraza je u redoslijedu operatora! Ovaj slučaj je isti kao i posljednji slučaj (7).

Radi kompletnosti i afirmacije gradiva, grupirati ćemo ove izraze stavljajući dodatne zagrade (koje je radi čitljivosti lijepo uvijek pisati). Može se kreirati još "varijacija" sa zagradama koje su čisto redundantne i možda čak zbunjujuće. Npr. možemo pisati `*(p)++` ili `(*p++)` ali to je isto kao i `*p++` a što je isto (ovdje su ipak zagrade korisne) kao i `*(p++)` – ne treba pretjerivati sa zagradama.

`x = *p++` *isto kao i* `x = *(p++)`

`x = (*p)++` *(nema varijacija)*

`x = ++*p` *isto kao i* `x = ++(*p)`

`x = *++p` *isto kao i* `x = *(++p)`

7.3.4 *Prosljeđivanje nizova funkcijama*

Upoznali smo se do sada sa prosljeđivanjem nizova funkcijama gdje su bile potrebne dvije informacije: "niz" i veličina niza.↑ Pisali smo nešto poput sljedećeg

↑ Poglavlje 6.3, str. 206.

```
void funkcija (int niz[], int velicina) {...}
```

Tada je napomenuto da se eventualne izmjene koje funkcija napravi nad proslijeđenim nizom odražavaju i na originalni niz. Sada možemo objasniti zašto je funkcija mogla da promijeni naš niz.

Naime, u prethodnom poglavlju smo objasnili da se ime niza konvertuje u pokazivač na prvi element niza. Kada smo pisali `f(niz, 10)` mi *nismo* prosljeđivali cijeli niz nego samo *pokazivač na njegov prvi element*. Također, ako je parametar funkcije `int niz[]`, to je potpuno ekvivalentno da smo napisali `int *niz`. U pitanju je samo "sintaksni šećer" tj. zagrade umjesto zvjezdice sugerišu da pokazivač koji funkcija prima pokazuje na član niza. Pogledajmo konkretan prototip:

```
void funkcija (int niz[], int velicina);
/* potpuno isto kao i */
void funkcija (int *niz, int velicina);
```

U prethodnom potpoglavlju↑ smo vidjeli da i pokazivače možemo

↑ Poglavlje 7.3.2, str. 240.

17/12/2019 98

indeksirati. Zbog toga smo u funkciji mogli pisati `niz[i]` jer je to identično kao `*(niz + i)`. A već je ranije spomenuto da se ime niza upotrijebljeno samo za sebe automatski pretvara u pokazivač na prvi element, pa je sljedeće dvoje *ekvivalentno*:

```
int niz[10] = {5, 2, ...};
...
/* poziv funkcije */
funkcija (niz, 10);
/* isto kao i */
funkcija (&niz[0], 10);
```

Vidjećemo kasnije da se funkcije mogu pisati tako da prihvataju dva pokazivača koja "omeđuju" niz odnosno skup informacija "$(pocetak, kraj)$" umjesto informacija "$(pocetak, velicina)$".

Zadatak 3. *Napisati funkciju "transformiraj" koja modificira proslijeđeni niz koristeći simulaciju indeksiranja. Elemente niza modificirati tako da svaki od njih postane količnik samog sebe sa aritmetičkom sredinom cijelog niza. Nije dozvoljeno indeksirati nizove pomoću uglastih zagrada. Elemente niza unositi sa tastature pri čemu se pretpostavlja da korisnik neće unijeti više od 100 elemenata.*

```
1  #include <stdio.h>
2  void transformiraj (double *niz, int vel) {
3    double suma = 0.;
4    int i;
5    for (i = 0; i < vel; ++i)
6      suma += *(niz + i);
7    suma /= vel;
8    for (i = 0; i < vel; ++i)
9      *(niz + i) /= suma;
10 }
11
12 int main() {
13   double niz[100];
14   int n, i;
15   printf ("Unesite n: ");
16   scanf ("%i", &n);
17   for (i = 0; i < n; ++i)
18     scanf ("%lf", niz + i); /* niz + i je adresa! */
19   transformiraj (niz, n);
20   /* ispis */
21   for (i = 0; i < n; ++i)
22     printf ("%.2f ", *(niz + i));
23   return 0;
24 }
```

U ovom zadatku je traženo da se za pristup članovima niza ne koristi indeksiranje (`niz[i]`) nego *simulacija indeksiranja*, a to je kôd `*(niz + i)` za koji znamo da mu je ekvivalentan.

Napravljena su dva prolaza kroz niz (računanje sume, linije 5-6 i modificranje elemenata, linije 8-9), te još jednom za ispis elemenata niza (linije 21-22). Zanimljiv je unos elemenata (18. linija): stavljeno je samo `niz + i` (što je opet pokazivač/adresa) umjesto ekvivalentnog `&niz[i]`. Naravno, ako baš želimo pisati sintaksu `*(niz + i)`

1777198

gdje dobivamo element (a ne adresu), onda bismo (za adresu) trebali pisati `&*(niz + i)`. Operatori referenciranja (`&`) i dereferenciranja (`*`) su međusobno suprotni i anuliraju se, tako `&*pokazivac` je efektivno samo `pokazivac`. Isto tako, `*&broj` je efektivno samo `broj`.

Napomenimo da ako želimo u funkciji saznati veličinu niza, ne možemo koristiti **`sizeof`** operator.↑ Već znamo da prosljeđivanjem niza funkciji mi zapravo prosljeđujemo pokazivač na njegov prvi element pa upotreba **`sizeof`** operatora će rezultirati veličinom pokazivača a ne cijelog niza:

↑ Poglavlje 3.2.5, str. 83.

```
void fun (int niz[]) {
  int vel = sizeof (niz) / sizeof (niz[0]);
  ...
}
```

`vel` u primjeru iznad bi moglo biti 1 ili 2, jer veličina tipa **`int`** na PC arhitekturi je 4 bajta, pokazivač je također 4 bajta ako je u pitanju 32-bitni sistem, a 8 bajta ako je u pitanju 64-bitni sistem.

Konačno, ponovićemo još jednom da se niz *ne može vratiti iz funkcije*.

7.3.5 *Rad sa nizovima koristeći pokazivačku aritmetiku*

Sve što smo do sada objasnili nam omogućuje da zadatke sa nizovima rješavamo tako što definišemo pokazivač na početak niza, a zatim njime "klizimo" do posljednjeg elementa povećavajući ga za jedan. Moguće je i obrnuto (od kraja ka početku). Ovakav pokazivač nazivamo **klizeći pokazivač**.

Zadatak 4. *Napisati funkciju "ispisi" koja ispisuje proslijeđeni niz cijelih brojeva (elementi trebaju biti razdvojeni razmakom) koristeći isključivo pokazivačku aritmetiku. Indeksiranje niza i trivijalna simulacija indeksiranja (`*(niz + i)`) nisu dozvoljeni.*

Podsjetimo se da pod *pokazivačkom aritmetikom* ne podrazumijevamo indeksiranje niza niti simuliranje indeksiranja, pa ukoliko se u zadatku kaže da program koristi pokazivačku aritmetiku, implicitno je zabranjeno koristiti sve varijacije indeksiranja. Kako, onda, ispisati niz? Koristeći postfiksne i prefiksne operatore inkrementiranja i dekrementiranja:

```
1  #include <stdio.h>
2  void ispisi (int *pok, int vel) {
3    int i;
4    for (i = 0; i < vel; i++) {
5      printf ("%d ", *pok);
6      pok++;
7    }
8  }
9  int main () {
10   int niz[] = {4, 3, 1, 8, 5};
11   const int vel = sizeof niz / sizeof niz[0];
```

```
12  ispisi (niz, vel);
13  /* Niz je ostao nepromijenjen, u sto se mozemo uvjeriti
14     tako sto ga ponovo ispisemo */
15  ispisi (niz, vel);
16  return 0;
17 }
```

Na početku rada funkcije `ispisi` pokazivač `pok` pokazuje na prvi element niza. Dakle, u liniji 5 ćemo ispisati taj prvi član a to je broj 4. Linija 6 pomjera pokazivač na sljedeći (drugi) član niza, pa će se u sljedećem prolazu kroz petlju ispisati broj 3. Petlja će se ponoviti `vel` puta a to je broj članova niza.

Primjećujete da smo u `main` funkciji (linija 11) odredili veličinu niza koristeći operator `sizeof`,↑ tako što smo najprije odredili veličinu u memoriji čitavog niza, a zatim je podijelili sa veličinom samo jednog (prvog) elementa.

↑ Poglavlje 3.2.5, str. 83.

Kako već znamo, postfiksni operator `++` ima veći prioritet od operatora dereferenciranja, a primjenjuje se tek nakon što se evaluacija izraza završi. Dakle, mogli smo skratiti iznad napisanu funkciju tako što ćemo u jednom izrazu automatski i dereferencirati i pomjerati pokazivač (izraz `*pok++`):

```
void ispisi (int *pok, int vel) {
  int i;
  for (i = 0; i < vel; ++i)
    printf ("%i ", *pok++);
}
```

Zadatak 5. *Prepravite prethodni zadatak sa funkcijom `ispisi` tako da ovaj put prilikom ispisa kao kontrolnu promjenljivu petlje koristite pokazivač.*

Funkcija `main` može biti ista kao i u rješenju prethodnog zadatka. Napišimo funkciju `ispisi`.

```
1 void ispisi (int *pok, int vel) {
2   int *iza_kraja = pok + vel;
3   /* isto kao i: */
4   /* int *iza_kraja = &pok[vel]; */
5   for (; pok < iza_kraja; ++pok)
6     printf ("%i ", *pok);
7 }
```

Potrebno je detaljnije analizirati iznad napisani kôd. Kreiran je pokazivač `iza_kraja` koji pokazuje tačno nakon posljednjeg elementa niza. Korištena je sintaksa `pok + vel` za koju znamo da proizvodi efekat kreiranja privremenog pokazivača koji pokazuje za `vel` elemenata iza u odnosu na `pok`. Ta sintaksa je ekvivalentna `&niz[vel]`. Ime pokazivača je samodeskriptivno i naglašeno je da pokazuje *iza kraja niza*. Znamo da indeksiranje počinje od 0 pa je prvi element `niz[0]` a posljednji `niz[vel - 1]`. Govorili smo da pokazivači *nisu validni* ako ne pokazuju na elemente niza, ali napravljen je jedan izuzetak u standardu jezika

1/15/2018

C gdje je pokazivač na element tačno iza kraja niza ipak validan.↑ U uslovu petlje je korišteno `pok < iza_kraja` a nikako nismo smjeli izjednačiti pokazivače odnosno pisati `pok <= iza_kraja` jer bismo u tom slučaju dereferencirali pokazivač iza kraja a to nipošto ne smijemo.

↑ Poglavlje C.1.2, str. 359.

Primijetite da u 5. liniji nismo pisali prvi dio zaglavlja petlje (inicijalizacioni dio) jer je promjenljiva `pok` već postavljena da pokazuje na pravo mjesto. Također smo mogli izostaviti i posljednji dio petlje (inkrement) na sljedeći način:

```
  for (; pok < iza_kraja; )
    printf ("%i ", *pok++);
```

ili (što je isto):

```
  while (pok < iza_kraja)
    printf ("%i ", *pok++);
```

Početnici često prave grešku poput sljedeće i onda program sadrži beskonačnu petlju (odnosno petlju bez prirodnog izlaza). Pored toga, takav program nakon određenog vremena uđe u nedefinisano stanje jer pokazivač pokazuje daleko iza kraja niza. Pogledajmo taj pogrešan primjer:

```
void ispisi (int *pok, int vel) {
  for (; pok < pok + vel; ++pok)
    printf ("%i ", *pok);
}
```

Ideju za kôdom kao iznad početnici dobiju razmišljanjem "ne treba mi dodatna promjenljiva `iza_kraja`" u cilju skraćivanja napisanog kôda. Problem iznad je u tome što uslov `pok < pok + vel` nikad nije zadovoljen. Ako pomjerimo `pok` da pokazuje na sljedeći element, automatski se pomjerila granica i na desnoj strani izraza jer se i u njoj koristi promjenljiva `pok`. Isti efekat bismo imali ako koristimo adresni operator i indeksiranje pokazivača:

```
void ispisi (int *pok, int vel) {
  for (; pok < &pok[vel]; ++pok) /* kobno! */
    printf ("%i ", *pok);
}
```

Ako već želimo pisati `pok + vel` i pri tome imati "potpunu" `for` petlju (sa svim dijelovima), onda ćemo uvesti novu pokazivačku promjenljivu kao u primjeru ispod.

```
void ispisi (int *pok, int vel) {
  int *i;
  for (i = pok; i < pok + vel; ++i)
  printf ("%i ", *i);
}
```

Možemo primijetiti jednu zanimljivu stvar u svim kôdovima iznad: **pokazivače možemo porediti relacionim operatorima**! Već smo demonstrirali poređenje pokazivača pomoću operatora jednakosti (`==`).

1/2/4/11/98

Slično vrijedi i za operator nejednakosti (`!=`). Ali sada vidimo da pokazivače možemo porediti relacijama "manje od" (`<`), "manje ili jednako od" (`<=`), "veće od" (`>`) te "veće ili jednako od" (`>=`). Ipak, postoje neka ograničenja za posljednja četiri relaciona operatora: smijemo porediti samo pokazivače koji pokazuju na elemente *istog niza* (eventualno na element tačno iza kraja niza). U svim ostalim slučajevima (npr. pokazivači pokazuju na elemente različitih nizova ili pokazuju na dio memorije koji nama pripada ili pokazuju ispred niza...) *nije definisano* poređenje takvih pokazivača i program koji to sadrži ima *nedefinisano ponašanje* te su posljedice nepredvidljive. Ovo je različito od prva dva operatora (`==` i `!=`) pomoću kojih možemo porediti bilo kakva dva (validna) pokazivača, čak iako ne pokazuju na isti niz.

Značenje relacije "veće" kod pokazivača je sljedeće: pokazivač `p1` na neki element niza je *veći* od drugog pokazivača `p2` na neki drugi element istog niza ako i samo ako `p1` pokazuje na element niza sa većim indeksom u odnosu na drugi element niza na kojeg pokazuje `p2`. Slično značenje ima i relacija "manji": pokazivač je *manji* od drugog pokazivača ako pokazuje na element sa manjim indeksom. Ovakve relacije su moguće zahvaljujući kontinualnoj reprezentaciji nizova u memoriji (svi elementi su jedan do drugog bez "razmaka" između).

Razlog za nedefinisano ponašanje kod poređenja tipa "veće-manje" ukoliko pokazivači ne pokazuju na isti niz je jednostavan: ne zna se kakav smisao bi trebao da ima npr. izraz `pok_n1 < pok_n2` (gdje pokazivači pokazuju na elemente različitih nizova `n1` i `n2`) jer ne znamo kojim redom se ti nizovi nalaze u memoriji tj. koji je ispred a koji iza. Shodno tome, ne smijemo ni nul-pokazivač tako porediti odnosno, izraz tipa `if (pok < NULL)` je besmislen čak i ako postoje arhitekture sa "negativnim" adresama.

Zadatak 6. *Napisati funkciju "ispisi_obrnuto" koja ispisuje niz u obrnutom redoslijedu (prvo posljednji element, pa pretposljednji itd.). Između elemenata treba biti znak razmak. Obavezno je korištenje pokazivačke aritmetike.*

Kako smo već rekli, ako se u postavci zadatka kaže da je obavezno koristiti pokazivačku aritmetiku, onda ne smijemo koristiti indeksiranje pokazivača niti simulaciju indeksiranja. Nismo bespomoćni, jer imamo operatore dekrementiranja (`--`) sa kojima se možemo "kretati" unazad.

Prvo ćemo prikazati *pogrešnu* implementaciju koja izgleda da radi. Zapravo, vjerovatno će gotovo uvijek ispisivati tačne rezultate.

```
void ispisi_obrnuto (int *pok, int vel) {
  int i;
   /* -1 da ne odemo iza kraja */
  int *kraj = pok + vel - 1;
  /* pokazivac 'kraj' pokazuje na posljedni el. */
  for (i = 0; i < vel; i++) {
```

```
    printf ("%i ", *kraj);
    kraj--;
  }
}
...
/* poziv je isti: */
ispisi_obrnuto (niz, vel);
```

Problem sa kôdom iznad je što nakon završetka petlje pokazivač `kraj` pokazuje na element *ispred* niza (odnosni *ispred* prvog elementa niza). Već smo rekli da takav pokazivač nije validan te program ima nedefinisano ponašanje, a problemi će gotovo sigurno nastati na memorijskim modelima koji nisu linearni (npr. kod globalnih nizova na MS DOS sistemima, ali postoji još dosta situacija sličnih ovoj). Ista priča vrijedi i da smo dekrementiranje radili odmah kada i dereferenciranje, odnosno da smo pisali `*kraj--` Prikažimo sada i *ispravno* rješenje koje ne rezultira nevalidnim pokazivačem.

```
1 void ispisi_obrnuto (int *pok, int vel) {
2   int i;
3   int *kraj = pok + vel; /* iza kraja */
4   for (i = 0; i < vel; ++i) {
5     /* prvo -- jer je pokazivao iza kraja! */
6     kraj--;
7     printf ("%i ", *kraj);
8   }
9 }
```

Iznad napisano rješenje je legalno i ne rezultira nedefinisanim ponašanjem. Opet smo iskoristili činjenicu da je pokazivač na element *iza kraja* niza savršeno *validan pokazivač*

Bitno je spomenuti i sljedeću stvar. Ukoliko povećamo pokazivač za 1 (npr. pomoću `p++` ili `p + 1`), to ne znači da smo brojčanu vrijednost adrese povećali za 1. Adresa će se povećati za onoliko bajta koliko zauzima tip na koji pokazivač pokazuje, pa ako je pokazivač deklarisan sa `int* p`, naredba `p++` povećava `p` za `sizeof(int)`, a na PC platformi je to uglavnom 4 bajta. To je još jedan od razloga zašto je bitno da pokazivač na element niza bude istog tipa kao i sam niz, mada će vam kompajler dozvoliti (uz upozorenje) da dodjeljujete pokazivače različitih tipova.

Prilikom pisanja vaših programa ne smijete nikada praviti pretpostavke o količini memorije koju određeni tip zauzima, jer takav program nije *portabilan*, odnosno neće raditi ispravno ako se kompajlira na različitom sistemu od onog za koji je pisan (npr. ako program pisan za PC pokušate pokrenuti na mobitelu sa ARM procesorom ili na mikrokontroleru).

Ipak, ako se pridržavate pravila da je pokazivač istog tipa kao i ono na šta pokazuje, možete zamisliti da svi tipovi podataka zauzimaju po jedan bajt, pa će i `p++` povećati adresu za 1. Drugim riječima, možete zamisliti da se pokazivači ponašaju kao cjelobrojne promjenljive.

17570198

Pređimo sad na zadatak sa početka priče o pokazivačima i nizovima,↑ samo ovaj put koristeći isključivo pokazivačku aritmetiku. Upoznat ćemo se sa početničkim greškama koje se ovdje mogu lako napraviti.

↑ Poglavlje 7.3.4, str. 245.

Zadatak 7. *Napisati funkciju "transformiraj" koja modificira proslijeđeni niz koristeći isključivo pokazivačku aritmetiku. Elemente niza modificirati tako da u svaki od njih postane količnik samog sebe sa aritmetičkom sredinom cijelog niza. Elemente niza unositi sa tastature pri čemu se pretpostavlja da korisnik neće unijeti više od 100 elemenata.*

```
1  #include <stdio.h>
2  void transformiraj (double *p, int vel) {
3    double suma = 0.;
4    int i;
5    /* ovdje MORAMO sacuvati kopiju od p */
6    /* jer cemo p pomjerati... */
7    double *kopija = p;
8    for (i = 0; i < vel; ++i)
9      suma += *p++;
10   suma /= vel;
11   p = kopija; /* vratimo kopiju... */
12   for (i = 0; i < vel; ++i)
13     *p++ /= suma;
14 }
15
16 int main() {
17   double niz[100];
18   int n, i;
19   printf ("Unesite n: ");
20   scanf ("%i", &n);
21   double *pok = niz;
22   for (i = 0; i < n; ++i)
23     scanf ("%lf", pok++); /* niz + i je adresa! */
24   transformiraj (niz, n);
25   /* ispis */
26   pok = niz;
27   for (i = 0; i < n; ++i)
28     printf ("%.2f ", *pok++);
29   return 0;
30 }
```

Rješenje zadatka ima dosta zanimljivih stvari i mjestâ gdje su moguće greške. Krenimo redom i pogledajmo prvo funkciju `transformiraj`. Važna je 7. linija u kojoj smo kreirali privremeni pokazivač (`kopija`) koji pokazuje na početak niza (odnosno prekopirali smo vrijednost iz promjenljive `p`). Ovo je urađeno zbog toga što smo u petlji ispod *pomjerali* pokazivač `p` i time smo *izgubili* informaciju o početku niza! Iskoristili smo kopiju i nakon prve petlje smo *vratili* tu informaciju u pokazivač `p`, te vršili transformaciju niza.

Napomenimo da smo se i na drugi način mogli vratiti na početak niza osim kreiranjem kopije. Naime, pošto u ovom slučaju tačno znamo za koliko smo mjesta pomjerili pokazivač, savršeno legalno smo mogli pisati i `p -= vel` nakon prve petlje. Također, mogli smo

1/15/70198

pisati nešto poput sljedećeg:

```
for (int i = 0; i < vel; ++i)
  p--;
```

U ovom slučaju `p` *ne bi* bio nevalidan pokazivač, jer ne bi ni pokazivao *ispred* niza. Zašto? Pa zbog toga što je petlja iznad pomjerala `p` sve do kraja niza, kada je izvršavanje petlje i završeno, ali ne zaboravimo da postfiksno inkrementiranje (`p++`) ima efekat tek nakon evaluiranja izraza pa je navedena petlja pomjerila pokazivač na jedno mjesto *iza kraja* niza. Stoga petlja za "vraćanje" tog pokazivača na početak može biti napisana upravo kako je napisana jer je `p` startao sa jednog mjesta iza kraja niza. Naravno, ova priča sa "jedno mjesto iza kraja niza" vrijedi jedino ukoliko je korisnik unio 100 elemenata i proslijedio cijeli niz funkciji! U slučaju da je korisnik unio npr. 3 elementa, funkciji je proslijeđeno `vel = 3`, pokazivač `p` neće pokazivati iza kraja niza nego će pokazivati na 4. element niza. Logički, to je iza kraja, ali memorijski naravno nije.

Objasnimo sada `main` funkciju. Unos pomoću `scanf` (linije 22-23) je klasičan, prosljeđuje se pokazivač (odnosno adresa) kojeg odmah i inkrementiramo da u sljedećem prolazu petlje unosimo sljedeći element niza. Kod ovakve petlje su početnici skloni pisati `niz++`, no podsjetimo se da je ime niza *konstantni pokazivač*↑ te se ne može mijenjati. Zbog toga smo u liniji 21 kreirali pomoćni pokazivač `pok` koji se koristi za unos, tako da u liniji 24 pokazivač `niz` i dalje pokazuje na početak i možemo ga proslijediti funkciji. U liniji 26 vraćamo `pok` na početak niza kako bismo niz ispisali, ponovo koristeći pokazivačku aritmetiku.

↑ Poglavlje 7.3.2, str. 240.

Zadatak 8. *Prepraviti prethodni zadatak sa* `transformiraj` *funkcijom tako da ona prihvata dva pokazivača koja omeđuju proslijeđeni niz. Prvi pokazivač pokazuje na početak niza dok drugi pokazivač pokazuje tačno iza kraja niza. Koristiti pokazivačku aritmetiku.*

Već smo naučili da pokazivače možemo porediti pomoću "<" operatora. Također smo naučili kako pokazivače efektivno možemo pomjerati. Stoga sljedeće rješenje treba biti prilično intuitivno.

```
1  void transformiraj (double *poc, double *iza_kraja) {
2    double suma = 0., *kopija = poc;
3    int br_el = 0;
4    while (poc < iza_kraja) {
5      suma += *poc++;
6      br_el++;
7    }
8    suma /= br_el;
9    while (kopija < iza_kraja)
10     *kopija++ /= suma;
11 }
12 int main() {
13   /* isto sve */
14   ...
15   /* prosljedjujemo 2 pokazivaca */
```

```
16  transformiraj (niz, niz + n);
17  ...
18 }
```

Ovdje smo uveli pomoćnu promjenljivu `br_el` koja je služila da saznamo broj elemenata proslijeđenog niza. Uslov za "kretanje" kroz niz je bilo `poc < iza_kraja` pri čemu ove pokazivače ne smijemo izjednačiti.

Obratimo pažnju da ako imamo pokazivač iza kraja, promjenljiva za veličinu niza (`br_el`) nam ustvari ni ne treba, jer je možemo saznati pokazivačkom aritmetikom tj. jednostavno napišemo `iza_kraja-poc`, pa bi linija 8 mogla glasiti i:

```
8  suma /= iza_kraja - poc;
```

a ostala pojavljivanja promjenljive `br_el` možemo ukinuti. Razlika dva pokazivača je cijeli broj, ali nije tipa `int` nego posebnog cjelobrojnog tipa `ptrdiff_t` o kojem ćemo više govoriti u dodatku C.1.3.↑ ↑ Str. 361.

7.4 *Pokazivači kao povratni tip funkcije*

Već je rečeno da kao povratni tip funkcije može biti bilo koji tip, pa tako i pokazivač. Sintaksa je jednostavna.

```
TIP *funkcija (...) { tijelo... }
/* validno je "priblizиti" zvjezdicu uz "TIP" */
TIP* funkcija (...) { tijelo... }
```

Kada je korisno da imamo funkciju koja vraća pokazivač? U većini slučajeva se to koristi da možemo imati ulančane pozive funkcija (ovo će biti ilustrirano kroz zadatak), a ima veliku primjenu i kada vraćamo pokazivače na (modificirane) parametre.

Počnimo sa jednostavnim primjerom. Neka trebamo napisati funkciju "veci" koja prihvata dva parametra i vraća veći od njih. Svrha ove funkcije je da ne bismo morali pisati uslov svaki put, poput sljedećeg kôda koji je prilično rogobatan i naporan za pisati ako nam slične konstrukcije trebaju više puta:

```
int a = 3, b = 1;
...
int max;
if (a > b)
  max = a;
else max = b;
```

Mogli smo iskoristiti i *kondicionalni* operator (također poznat i kao *ternarni* operator)↑ da dobijemo veću od dvije promjenljive. ↑ Poglavlje 3.2.3, str. 82.

```
int max = a > b ? a : b;
```

Traženu funkciju `veci` možemo napisati na sljedeći način:

```
int veci (int a, int b) {
  if (a > b)
    return a;
  return b;
```

1755198

```
}
...
int a = 3, b = 1;
...
int max = veci (a, b);
```

Funkciju `veci` smo smogli napisati još kraće!

```
int veci (int a, int b) {
  return a > b ? a : b;
}
```

Ukoliko sada trebamo promijeniti veću od dvije promjenljive dolazimo do problema! Funkcija `veci` će, kako znamo, vratiti *kopiju* promjenljive, tako da mijenjanjem `max` promjenljive ne bismo mijenjali niti `a` niti `b`. Sada bismo se opet morali vratiti rogobatnoj konstrukciji poput sljedeće, ako npr. želimo veću promjenljivu postaviti na 5:

```
int a = 3, b = 1;
if (a > b)
  a = 5;
else b = 5;
```

Opet pomoć možemo "zatražiti" od koncidionalnog operatora. Ali tu trebamo biti pažljivi jer prvo "intuitivno" rješenje neće raditi (barem ne u jeziku C, ali će raditi u jeziku C++):

```
(a > b ? a : b) = 5;
```

Kompajler će vjerovatno prijaviti grešku tipa "nije moguće dodjeljivati nečemu što nije l-vrijednost". Naime, u C-u, iznad napisani izraz se tumači tako da treći dio kondicionalnog operatora mora biti kondicionalni izraz, te nije moguće vršiti dodjelu. Ipak, iznad napisani izraz je savršeno legalan u C++-u zbog drugačijeg načina tumačenja kondicionalnog operatora.

Ako malo razmislimo, mogli bismo probati vršiti dodjelu unutar kondicionalnog operatora:

```
(a > b ? a = 5 : b = 5); /* greska u C-u, u C++u je opet OK */
```

Nažalost, ni ovo neće proći jer operator dodjele (`=`) ima *manji* prioritet nego kondicionalni operator. Konačno, možemo pisati sljedeće:

```
(a > b ? (a = 5) : (b = 5));
```

Sigurno znamo da će se izvršiti *samo jedan* dio kondicionalnog operatora (ili `a = 5` ili `b = 5`).

Sada, konačno, napišimo i funkciju `veci` koja će raditi kako treba. Ona će, naravno, vraćati pokazivač jer želimo modificirati proslijeđenu vrijednost:

```
1 #include <stdio.h>
2 int *veci (int * a, int * b) {
3   if (*a > *b)
4     return a;
5   return b;
6 }
```

1757098

```
int main() {
  int a = 3, b = 1;
  int *max = veci(&a, &b);
  *max = 5;
  /* ili cak direktno: */
  *veci (&a, &b) = 5;
  printf ("a: %i, b: %i", a, b);
  return 0;
}
```

Vidjeli smo da direktno možemo dereferencirati vraćeni pokazivač iz funkcije. Ispis programa će biti sljedeći.

```
a: 5, b: 1
```

7.4.1 *Viseći pokazivači*

Već znamo da sve lokalne promjenljive (koje nisu deklarisane kao `static`) u funkciji prestaju postojati kada se izvršavanje funkcije završi.↑ To uključuje i njene parametre jer su i oni lokalne promjenljive. Stoga nikad ne smijemo vratiti pokazivač na lokalnu promjenljivu.

↑ Poglavlje 6.2.7, str. 196.

> Nikada iz funkcije ne smijemo vratiti pokazivač na objekat koji prestaje postojati (npr. lokalna promjenljiva funkcije). To rezultira nedefinisanim ponašanjem i posljedice su nepredvidljive. Takav pokazivač (na promjenljivu koja je prestala postojati) se naziva **viseći pokazivač**.

```
int *funkcija () {
  int x=5; /* lokalna promjenljiva */
  /* vracamo pokazivac na promjenljivu koja umire! */
  return &x;
}
...
int *pok = funkcija();
*pok = 3; /* dereferenciranje viseceg pokazivaca - GRESKA */
/* isto i da smo direktno dereferencirali */
*funkcija() = 3; /* greska! */
```

Viseće pokazivače možemo dobiti na mnogo "podmukliji" način. Pogledajmo sljedeći primjer:

```
int *funkcija (int x) {
  /* i 'x' je lokalna promjenljiva! */
  return &x;
}
...
int a = 5;
int *pok = funkcija (a);
```

Prikažimo sada i *pogrešnu verziju* `veci` funkcije kakvu početnici pišu.

```
#include <stdio.h>
int *veci (int a, int b) {
  /* 'a' i 'b' su lokalne promjenljive */
  if (a > b)
```

1 // 5 // III 98

```
    return &a;
  return &b;
}
int main() {
  int a = 3, b = 1;
  *veci(a, b) = 5; /* greska */
  return 0;
}
```

Dobar kompajler će nas u slučajevima iznad upozoriti da vraćamo adresu lokalne promjenljive, ali postoje slučajevi koji "promaknu" i kompajleru pa je, naravno, odgovornost na programeru da vodi računa o ovome.

Sada prikažimo jednu "čudnu", ali ispravnu verziju `veci` funkcije:

```
#include <stdio.h>
int *veci (int *a, int *b) {
  int *prvi = a, *drugi = b;
  if (*a > *b)
    return prvi;
  return drugi;
}
int main() {
  int a = 3, b = 1;
  *veci(&a, &b) = 5; /* ok */
  return 0;
}
```

I ovdje vraćamo promjenljive koje će prestati postojati. Ali, ovaj kôd je ipak validan, jer nismo vraćali pokazivače na promjenljive koje će umrijeti (promjenljive `prvi` i `drugi` hoće umrijeti, ali neće ono na šta one pokazuju). Tačnije, mi ćemo vratiti kopiju pokazivača `prvi` ili `drugi` (zavisno šta je veće, `a` ili `b`), ali ti pokazivači pokazuju na promjenljive koje će postojati i nakon izvršavanja funkcije (jer su to promjenljive deklarisane u `main` funkciji).

Još jedan čest način kako neiskusni C programeri proizvode viseći pokazivač su pokušaji da se iz funkcije vrati niz. Naime, recimo da je programer htio da napravi funkciju koja prima cijeli broj a vraća niz njegovih cifara:

```
#include <stdio.h>
int* cifre (int broj) {
  int niz[10];
  ...
  return niz;
}
```

Prije svega, ako napišemo `return niz[10];` to bi nam vratilo 11-ti član niza (što je nedefinisano). Kôd oblika `return niz[];` se ne bi kompajlirao. Prazne uglaste zagrade dozvoljene su samo u prototipu funkcije (kao alternativni način označavanja pokazivača), te prilikom deklaracije niza, i to samo ako je navedena inicijalizacijska lista.↑ Također možemo zaključiti da se funkcija sa bilo kojim tipom povratne vrijednosti različitim od `int*` ne bi kompajlirala. Na taj način je pro-

↑ Poglavlje 5.1.2, str. 134.

1/11/2018

gramer došao do funkcije iznad koja se kompajlira i vrlo često čak i radi, ali ipak ona vraća viseći pokazivač.

Naime, već smo vidjeli↑ da sama riječ `niz` označava adresu prvog člana niza, tako da ona po tipu odgovara povratnom tipu funkcije `int*`. Problem je što je opseg (scope) niza funkcija, tako da kada se funkcija završi, `niz` predstaje da postoji. Dakle imamo istu situaciju kao i ranije, da vraćamo adresu promjenljive koja više ne postoji.

↑ Poglavlje 7.3.2, str. 240.

Zadatak 9. *Napisati funkciju "najveci" pomoću koje ćemo moći modificirati najveći element niza. Funkcija treba da nađe najveći element niza i vrati pokazivač na njega.*

Prvo ćemo prikazati *pogrešnu* verziju koja "liči" na ispravno rješenje.

```
int *najveci (int *niz, int vel) {
  int max = niz[0], i;
  for (i = 1; i < vel; ++i)
    if (max < niz[i])
      max = niz[i];
  return &max;
}
```

Ovdje ponovo vraćamo pokazivač na lokalnu promjenljivu, a to je promjenljiva `max` deklarisana u drugoj liniji funkcije. Kako onda *ispravno uraditi* zadatak?

To je moguće na dva načina. U prvoj varijanti nećemo pamtiti najveći element nego ćemo pamtiti *indeks* najvećeg elementa. Kasnije ćemo vratiti pokazivač na element sa indeksom koji odgovara najvećem elementu. To neće biti viseći pokazivač, jer ćemo vratiti pokazivač na element proslijeđenog niza.

```
1  #include <stdio.h>
2  int *najveci (int *niz, int vel) {
3    int i, max_i = 0;
4    for (i = 1; i < vel; ++i)
5      if (niz[max_i] < niz[i])
6        max_i = i;
7    return &niz[max_i];
8  }
9  int main() {
10   int niz[] = {1, 2, 3, 41, 5};
11   int *najv = najveci (niz, 5);
12   printf ("Najveci: %i\n", *najv);
13   *najv = 6;
14   printf ("4. el niza: %i\n", niz[3]);
15   /* ili direktno */
16   *najveci (niz, 5) = 400;
17   printf ("4. el niza: %i\n", niz[3]);
18   return 0;
19 }
```

Promjenljiva `max_i` sadrži *indeks* najvećeg elementa niza. Najprije postavljamo taj indeks na nulu (prvi element niza), a zatim određujemo indeks najvećeg. U liniji 7 iz funkcije se vraća adresa najvećeg člana

niza, pa pošto je niz deklarisan u mainu (linija 10) ovo neće biti viseći pokazivač. Program iznad će ispisati sljedeće.

```
Najveci: 41
4. el niza: 6
4. el niza: 400
```

Drugi način rješavanja je da pamtimo pokazivač na element. Ustvari, kompletan zadatak možemo riješiti koristeći pokazivačku aritmetiku, pa ćemo pronađeni pokazivač jednostavno vratiti iz funkcije.

Zadatak 10. *Prepraviti prethodni zadatak sa funkcijom* `najveci` *ovaj put uz ograničenje da se koristi pokazivačka aritmetika.*

```
1  int *najveci (int *niz, int vel) {
2    int *najveci = niz;
3    int *iza_kraja = niz + vel;
4    while (niz < iza_kraja) {
5      if (*najveci < *niz)
6        najveci = niz;
7      niz++;
8    }
9    return najveci;
10 }
```

Ako iskoristimo istu `main` funkciju kao i u prethodnom zadatku ispis će biti isti. Možemo primijetiti da smo imali jedno *viška* poređenje jer smo prvi element poredili sa samim sobom na početku izvršavanja petlje. Mogli smo i ovo zaobići da smo prije petlje napisali npr. `niz++;` ili čak postfiksno pomjeriti `niz` prilikom inicijalizacije `najveci` pokazivača:

```
/* inicijaliziraj i pomjeri */
int *najveci = niz++;
```

Analizirajmo funkciju: najprije postavimo pokazivač na prvi element niza, zatim iteriramo (krećemo se) kroz niz i poredimo ono na šta trenutni pokazivač pokazuje sa pokazivačem `najveci`, te ako je trenutni element veći, postavljamo pokazivač `najveci` da pokazuje na taj element. Ne zaboravimo inkrementirati pokazivač `niz`, jer bismo u suprotnom imali beskonačnu petlju!

Zadatak 11. *Napisati funkciju "zaokruzi" koja zaokružuje svaki element niza realnih elemenata na dvije decimale. Funkcija treba da vrati pokazivač na prvi element niza radi ulančanog povezivanja. Napisati i funkciju "ispisi" sa istim parametrima kao i "zaokruzi" koja će ispisivati elemente niza zaokruženo na 4 decimale. Koristiti pokazivačku aritmetiku. U* `main` *funkciji demonstrirati ulančani poziv funkcija.*

17557198

Podsjetimo se da realan broj možemo zaokružiti na dvije decimale tako što ga množimo sa 100, a zatim zaokružimo na uobičajn način, te konačno rezultat podijelimo sa 100.↑ Da još afirmiramo dereferenciranje pokazivača, nećemo koristiti funkciju `round`, nego ćemo uraditi konverziju (cast) u `int` nakon što prethodno saberemo vrijednost 0.5

↑ Poglavlje 3.1.8, str. 76.

(no obratite pažnju da je ovo tačno samo za pozitivne brojeve, kod negativnih bismo trebali oduzeti 0.5):

```
1  #include <stdio.h>
2  double *zaokruzi (double *niz, int vel) {
3    double *p = niz;
4    while (p < niz + vel) {
5      *p *= 100;
6      /* odsijecanje decimala: */
7      *p = (int)(*p + 0.5);
8      *p++ /=  100;
9    }
10   /* vratimo pokazivac na pocetak */
11   return niz;
12 }
13 void ispisi (double *p, int vel) {
14   while (vel--)
15     printf ("%.4f ", *p++);
16 }
17 int main() {
18   double niz[] = {12.348, 1.111, 1.67, 3};
19   const int vel = sizeof niz / sizeof niz[0];
20   /* ulancani poziv! */
21   ispisi (zaokruzi (niz, vel), vel);
22   printf("\n");
23   return 0;
24 }
```

Ne smijemo zaboraviti `p++` u posljednjem redu petlje. Traženi lančani poziv može se vidjeti u liniji 21. Namjerno je traženo zadatkom da se ispisuje na 4 decimale kako bismo vidjeli efekte zaokruživanja. Pogledajmo to na ispisu:

```
12.3500 1.1100 1.6700 3.0000
```

Pitanja za razumijevanje gradiva

1. Šta u C-u označava operator zvjezdica (*) a šta ampersand (&)?
2. Šta je deklarisano naredbom `int* a, b, c;`?
3. Šta je to nul-pokazivač i čemu služi?
4. Šta je to nedefinisano ponašanje?
5. Na koje načine možemo zaobići ograničenje C-a da se iz funkcije ne može vratiti više od jedne vrijednosti? Navedite sve što znate.
6. Koje aritmetičke operacije ima smisla vršiti nad pokazivačima?
7. Šta će se ispisati na ekranu ako imamo niz od 10 elemenata pod imenom `niz` i zatim napišemo `printf("%d", 2[niz]);`? Zašto?
8. Kada ima smisla vratiti pokazivač iz funkcije?

17570198

Zadaci za vježbu

Zadatak 12. *Napisati funkciju "zaokruzi" koja kao parametar prima niz proizvoljne dužine sastavljen od realnih brojeva dvostruke preciznosti, a zatim sve brojeve u nizu zaokružuje na jednu decimalu. Pri rješavanju zadatka obavezno koristiti pokazivačku aritmetiku, a ne indeksiranje niza (pristup elementima niza pomoću uglastih zagrada) ili njegovu simulaciju oblika* `*(niz+i)`*. Dakle, u ovom programu se trebate kretati kroz niz "klizećim pokazivačem" (poglavlje 7.3.5, str. 247).*

Zadatak 13. *Napisati funkciju "sortiraj" koja slaže po veličini niz realnih brojeva, pri čemu dimenzija niza nije unaprijed poznata nego predstavlja jedan od parametara funkcije. Pri rješavanju zadatka obavezno koristiti pokazivačku aritmetiku. Algoritam sortiranja dat je u poglavlju 5.2.5 (str. 158).*

Zadatak 14. *Napisati funkciju "drugi" koja u primljenom nizu cijelih brojeva pronalazi drugi po veličini i vraća pokazivač na taj član. U slučaju da niz ne sadrži drugi po veličini element funkcija treba vratiti nul-pokazivač.*

Zadatak 15. *Napisati funkciju "prekopiraj" koja treba da prekopira elemente niza na koje pokazuje pokazivac "izvor" u niz na kojeg pokazuje pokazivač "odrediste". Pretpostaviti da "odrediste" pokazuje na dovoljno velik niz koji može prihvatiti sve elemenata prvog niza; ovo ne treba provjeravati. Funkciju napisati korištenjem pokazivačke aritmetike.*

17570198

1757018 8

8
Stringovi (znakovni nizovi)

U programiranju često se javlja potreba da u memoriji računara predstavimo neki tekst, npr. ime i prezime ili tekst neke poruke. Zbog toga, većina programskih jezika posjeduje poseban tip podataka koji se naziva **string**. Riječ string na engleskom znači nit ili struna (npr. na gitari), ali u programiranju označava niz pojedinačnih znakova (karaktera) koji predstavlja tekstualni podatak i po određenim karakteristikama se razlikuje od običnog niza.

Kao što znamo, programski jezik C posjeduje tip `char`↑ pomoću kojeg se može predstaviti jedan znak (karakter), ali specifičan tip za string ne postoji. Umjesto toga koristi se jednostavno niz tipa `char`:

↑ Poglavlje 3.1.7, str. 74.

```
char ime[20], poruka[50];
```

Podsjetimo se i da u C-u postoji znakovni literal koji označavamo dvostrukim navodnicima.↑ Npr. u našem prvom C programu imali smo naredbu:

↑ Poglavlje 1.2.7, str. 35.

```
printf("Dobro dosli u svijet programiranja!\n");
```

U ovoj naredbi tekst `"Dobro dosli..."` je znakovni (string) literal i interno će biti tretiran kao konstantan niz tipa `char`.

Obratite pažnju da za nizove tipa `char` važi sve što i za druge nizove. Npr. u nekim programskim jezicima možemo međusobno dodjeljivati stringove ili koristiti operator + da spojimo dva stringa:

```
ime = poruka;
ime = ime + poruka;
```

U C-u naredbe poput ovih nisu ispravne, jer su u pitanju nizovi, a nizovi se ne mogu međusobno dodjeljivati i spajati.↑ Vjerovatno se pitate kako onda obaviti ove operacije? Pa na isti način kao i sa nizovima npr. cijelih brojeva. Preporučujemo da se sada vratite na poglavlje 5 i podsjetite.

↑ Poglavlje 5.1, str. 130.

1757011198

Kada smo radili sa nizovima cijelih brojeva proizvoljne dužine, koristili smo neku specijalnu vrijednost (npr. broj -1 je bio zgodan ako je u pitanju niz prirodnih brojeva) da označimo kraj niza. Stringovi su

u pravilu proizvoljne dužine. Npr. ako u našem programu omogućujemo korisniku da unese svoje ime, možemo reći da ime sigurno neće biti duže od 20 znakova i deklarisati ga sa `char ime[20];`. Međutim, ime korisnika može biti i "Una" i npr. "Abdulkerim". Slično je i kada se ime unosi iz datoteke.

Zbog toga je u ASCII tabeli↑ predviđen standardni znak koji označava kraj stringa, kako svaki programer ne bi izmišljao vlastitu oznaku. Taj znak se naziva *NUL karakter*, označava se sa \0 (backslash nula) a njegova ASCII vrijednost je nula. Npr. šta će se desiti ako u ranije deklarisanu promjenljivu `ime` unesemo "Ana"? Prvi član niza je slovo 'A', drugi je slovo 'n', treći slovo 'a', a četvrti član je NUL karakter odnosno \0.

↑ Poglavlje 3.1.7, str. 74.

Šta je sa ostalih 16 članova niza? Njihova vrijednost nije bitna jer će sve poslije \0 biti zanemareno, možemo pretpostaviti da su neinicijalizirani. Sve funkcije iz C standardne biblioteke će stati na znaku \0 i zanemariti sve nakon toga. Npr. ako napišemo

```
printf("ABC\0DEF");
```

na ekranu će biti ispisano samo `ABC`. Naredba `printf("\0");` neće ispisati ništa. Možemo reći da \0 označava logičku veličinu stringa kao u poglavlju 5.1.5.↑

↑ Str. 142.

Podsjetimo se još jednom da se jednostrukim navodnicima (znak apostrofa) označavaju literali tipa `char`, a dvostrukim string literali koji su interno posmatrani kao niz tipa `char`. Kodovi `'A'` i `"A"` označavaju različite stvari. `'A'` je literal tipa `char`, što je u programskom jeziku C cjelobrojna vrijednost i možete ga koristiti u računskim operacijama gdje će poprimiti ASCII vrijednost slova A (broj 65).↑

↑ Poglavlje 3.1.7, str. 74.

`"A"` je string literal koji će biti posmatran kao niz tipa `char` sa dva elementa: prvi je slovo 'A', a drugi \0. Ako ga pokušate posmatrati kao cijeli broj dobićete neku adresu na kojoj se nalazi taj niz. Interesantno je da ako napišete `'Zdravo'` ovo je očito pogrešno jer `char` može imati vrijednost samo jednog znaka, ali prevodilac će vam to dozvoliti uz upozorenje, a vrijednost izraza će biti posljednji znak (slovo 'o').

8.1 Osnovne operacije nad stringovima

Kao i kod nizova brojeva, string (niz tipa `char`) se može inicijalizirati prilikom deklaracije,↑ te se u tom slučaju ne mora navesti veličina. Pogledajmo sljedeće deklaracije:

↑ Poglavlje 5.1.2, str. 134.

```
1 char p[10] = {'a','b','c','\0'};
2 char pozdrav[10] = "Zdravo";
3 char ime[] = "Ahmed";
4 const char* prezime = "Peric";
```

U liniji 1 deklarisali smo niz tipa char i inicijalizirali ga na način kako smo to radili sa nizovima.↑. Inicijalizacija u liniji 2 je u suštini ek-

↑ Poglavlje 5.1.2, str. 134.

vivalentna, ali je kraća i preglednija. Prvih šest članova niza `pozdrav` inicijalizirano je slovima "Zdravo", a sedmi član će biti \0. U liniji 3 vidimo primjer deklaracije kod koje se dimenzija niza određuje na osnovu inicijalizacije. Šta mislite koliko članova će imati niz `ime`?

Napominjemo da je moguća i deklaracija koja se ne završava znakom \0, npr. sa `char p[3] = {'a','b','c'};`. Ovo je validan niz tipa `char`, ali *nije validan string*. Drugim riječima, dok god pristupamo pojedinim članovima niza sve je u redu, problem nastupa kada radimo sa stringom kao cjelinom.

Pogledajte i deklaraciju u liniji 4. Za razliku od prethodnih deklaracija u pitanju nije niz nego *pokazivač na string literal*. Literali se ne mogu mijenjati, pa bi svaki pokušaj da promijenimo string `prezime` mogao dovesti do krahiranja programa. Stoga ne bi bilo korektno da smo izostavili ključnu riječ `const`, mada bi se takav program mogao prevesti sa C prevodiocem (ali ne i sa C++ prevodiocem).

Deklaracija data ispod je pogrešna, jer se niz tipa `char` može inicijalizirati samo string literalom, a ne char literalom:

```
char ime[] = 'A';
```

Još jedna česta greška je obrnut slučaj:

```
char ime = "Ahmed";
```

Deklarisali smo promjenljivu tipa `char` (jedan znak) i inicijalizirali je stringom "Ahmed". Interesantno je da će se ovaj kod prevesti, a promjenljiva `ime` će za vrijednost imati neki čudan i neočekivan znak. Razlog je jednostavno shvatiti: string literal `"Ahmed"` biće tretiran kao niz tipa `char`, odnosno adresa prvog člana niza, a adresa je cijeli broj. Pošto je tip `char` također cjelobrojni tip, ovdje će jednostavno doći do odsjecanja na 8 bita↑ i ta vrijednost će se upisati u promjenljivu `ime`.

↑ Poglavlje 3.1.8, str. 76.

8.1.1 Ulaz i izlaz

Za ispis niza znakova na ekran možemo koristiti naredbu `printf` sa formatom `%s`, npr:

```
char ime[20], prezime[20], brindexa[10];
...
printf("%s %s (%s)", ime, prezime, brindexa);
```

Podsjetimo se da format `%c` služi za vrijednosti tipa `char` (jedan znak), dok `%s` služi za znakovne nizove. Brkanje ova dva formata je jedna od najčešćih grešaka kod početnika. Npr. takva zamjena se dešava u sljedećem kodu:

```
printf("%c", "Zdravo");
printf("%s", 'A');
```

Oba pogrešna koda će se prevesti bez problema. U prvoj liniji ćemo na ekranu dobiti neki čudan znak iz razloga koji smo objasnili maloprije,

1157011198

dok u sljedećoj naredbi program će na ekranu ispisati neko smeće i vjerovatno se krahirati. Naime, pošto format `%s` očekuje niz, `'A'` će biti protumačeno kao adresa prvog člana niza pa će program pokušati ispisati sadržaj memorije počevši od adrese 65, kojoj u pravilu ne smijemo pristupiti.

Što se tiče unosa znakova, primijetićete da naredba `scanf` također posjeduje format `%s`, ali njegova upotreba *nije preporučljiva* i u ovoj knjizi ćemo je smatrati greškom.

```
scanf("%s", ime);
```

Dva su osnovna razloga za to: ako korisnik unese više znakova nego što je deklarisano u nizu ime, doći će do prekoračenja dimenzije niza i izgledno je krahiranje programa. Vi ne možete spriječiti korisnika da unese više znakova od deklarisanog, a rekli smo da bismo trebali nastojati da se naši programi nikada ne krahiraju bez obzira šta korisnik unese. Drugi razlog je što na ovaj način nije moguće unijeti razmak. Ova ograničenja moguće je prevazići format opcijama koje su uvedene u C99 standardu.

Zbog nedostataka `scanf` C biblioteka nudi niz funkcija za unos stringova, no svaka od njih ima određene probleme:

- *gets* - ova funkcija je nesigurna i opasna, jer je moguće izazvati prekoračenje niza (kao i kod `scanf`), pa je u novijim verzijama standarda proglašena za *deprecated* (prevaziđenu).
- *fscanf* - funkcija za čitanje iz datoteke, moguće ju je primijeniti nad "datotekom" za standardni ulaz, što je nešto iznad nivoa znanja na kojem se trenutno nalazimo.
- *getchar* - čita samo jedan znak, moguće bi bilo koristiti je unutar petlje.
- *getline* - ova funkcija rješava sve postavljene probleme, ali uvedena je tek u ISO C11 standardu, zbog čega podrška za nju još uvijek nije univerzalna.

Zbog svega navedenog, odlučili smo se da napravimo svoju verziju funkcije `getline` pod nazivom "unesi", koristeći pozive funkcije `getchar` unutar petlje, što će ujedno poslužiti za bolje razumijevanje rada sa stringovima. Kôd funkcije dat je ispod. Naglašavamo da bi ovaj kôd mogao biti i znatno kraći, ali smo ga napisali na ovaj način radi lakšeg shvatanja početnicima.

1755798

```
1 void unesi(char niz[], int velicina) {
2   char znak = getchar();
3   if (znak == '\n') znak = getchar();
4   int i = 0;
5   while (i < velicina-1 && znak != '\n') {
6     niz[i] = znak;
```

```
7       i++;
8       znak = getchar();
9     }
10    niz[i] = '\0';
11  }
```

Funkcija ima dva parametra: niz koji treba unijeti i fizičku veličinu niza (dimenziju zadatu prilikom deklaracije),↑ na koji način ćemo osigurati da ta veličina nikada ne bude prekoračena. Osnovna petlja nalazi se u linijama 5-9. U svakom prolazu kroz petlju čita se jedan znak sa tastature koristeći funkciju `getchar` (linija 8). Petlja se prekida kada se dođe do veličine umanjene za jedan, jer moramo ostaviti mjesta za \0, ili kada se na ulazu nađe znak \n što znači da je pritisnuta tipka Enter (linija 5). Ako je uslov ispunjen, upisaćemo upravo uneseni znak u niz (linija 6) i povećati brojač za jedan. Ne smijemo zaboraviti da naš string korektno terminiramo znakom \0, što se radi u liniji 10.

↑ Poglavlje 5.1.5, str. 140.

Značenje linija 2 i 3 je nešto složenije. Naime, naredba `scanf` ostavlja na ulaznom toku znak koji ne odgovara zadatom formatu, što znači da ako npr. kod glasi:

```
scanf("%d", &n);
```

a vi ste unijeli broj 6 i pritisnuli tipku Enter, broj 6 će biti upisan u promjenljivu `n` a karakter \n će ostati na ulaznom toku. Ako bismo zatim pozvali našu funkciju `unesi` bez linije 3, petlja bi se odmah prekinula jer uslov `znak != '\n'` nije ispunjen i funkcija bi ostavila string praznim. Zbog toga smo linijom 3 omogućili da se znak \n koji se eventualno nalazi na samom početku ulaza preskoči kako bi programer mogao slobodno miješati pozive funkcija `scanf` i `unesi`.

Primjer poziva funkcije `unesi` može se vidjeti u sljedećem programu:

```
int main() {
  char ime[20], prezime[20];
  int brindeksa;
  printf ("Unesite vas broj indeksa: ");
  scanf ("%d", &brindeksa);
  printf ("Unesite vase ime: ");
  unesi(ime, 20);
  printf ("Unesite vase prezime: ");
  unesi(prezime, 20);
  ...
```

8.1.2 *Česte funkcije za rad sa stringovima i njihova implementacija*

Spomenuli smo na početku da se nizovi tipa `char` ne mogu međusobno dodjeljivati, niti se mogu spajati operatorom plus. Vjerovatno se pitate kako onda obaviti te zadatke u C-u. Odgovor je: kao i sa nizovima brojeva, trebamo koristiti petlju da znak po znak prepišemo iz jednog u drugi niz.

Srećom, standardna biblioteka `string.h` definiše niz korisnih funkcija koje obavljaju određene zadatke kako mi ne bismo morali. Neke

1757098

od njih su:

- *strlen* - vraća dužinu stringa - misli se na stvarnu dužinu a ne na deklarisanu, odnosno vraća broj znakova od početka do \0.
- *strcpy* - kopira string u drugi string brišući stari sadržaj; obratite pažnju da odredišni string treba biti deklarisan tako da može primiti kompletan izvorišni string, u suprotnom će doći do krahiranja.
- *strcmp* - leksikografsko poređenje dva stringa.
- *strcat* - spaja dva stringa tako što drugi dopisuje na kraj prvog; prvi string treba biti deklarisan kao dovoljno velik da primi sve dodatne karaktere.
- *strchr* - pronalazi karakter u stringu.
- *strstr* - pronalazi podstring u stringu.

Uobičajena praksa na kursevima C programiranja je da se prođe kroz implementacije upravo ovih bibliotečnih funkcija kako biste na tom primjeru vidjeli principe rada sa stringovima. Naravno, vi u vašim programima nećete morati ponovo implementirati već gotove funkcije (osim ako njihovo korištenje nije zabranjeno školskim ili fakultetskim zadatkom radi vježbe).

Zadatak 1. *Napišite funkciju koja vraća dužinu stringa (strlen).*

Krećemo sa prvim primjerom. Prototip funkcije glasi:

```
int duzina(char* s);
```

Naime rekli smo da se nizovi uvijek u funkcije šalju kao pokazivač na prvi element niza, a vraća se dužina. Samim time dužina niza nije parametar funkcije nego je to ono što funkcija vraća.[1]

Inače u funkcijama koje rade sa stringovima u pravilu se ne šalje broj elemenata jer on proizlazi iz pozicije znaka \0. Vaša funkcija može pozvati bibliotečku funkciju `strlen`, ali često je to suvišno kao što ćemo vidjeti kasnije.

Šta naša funkcija treba raditi? Trebamo proći kroz niz `s` i pronaći znak \0 u njemu. Dakle, mogli bismo funkciju uraditi ovako:

```
1 int duzina(char* s) {
2   int i, duz = 0;
3   for (i=0; ; i++) {
4     if (s[i] == '\0') break;
5     duz++;
6   }
7   return duz;
8 }
```

U liniji 3 vidimo klasičnu for petlju za prolazak kroz niz, s tim što nam broj elemenata niza nije poznat. Ono što nam jeste poznato je da

[1] Kada spominjemo bibliotečnu funkciju `strlen`, trebamo naglasiti da ona ustvari prima pokazivač tipa `const char*` (razlog za to ćemo objasniti već u sljedećem zadatku), a vraća `size_t`.

1/15/2018

string sigurno sadrži znak \0, pa će se `break` u liniji 4 sigurno desiti. Ovo je validna pretpostavka koju prave sve bibliotečne funkcije (`strlen` će se također krahirati ili vratiti besmislenu vrijednost ako je pozovete nad stringom koji ne sadrži \0), tako da se sve funkcije koje vi pravite moraju pobrinuti da stringovi uvijek sadrže \0. Mogli smo kraće pisati i ovako:

```
3    for (i=0; s[i] != '\0'; i++) {
```

Ovako napisana funkcija bila bi potpuno validna, odnosno radila bi ispravno, no moramo ovdje reći da *niko ne radi tako*. Programi koji rade sa stringovima praktično uvijek koriste pokazivačku aritmetiku↑ jer se time dobija koncizniji i pregledniji kod. Preporučujemo da vježbate koristiti pokazivačku aritmetiku u zadacima sa stringovima, jer će sva rješenja i primjeri koje nađete na internetu ili u literaturi također biti urađeni na taj način. Ista funkcija implementirana pokazivačkom aritmetikom glasila bi:

↑ Poglavlje 7.3.1, str. 239.

```
1  int duzina(char* s) {
2    int duz = 0;
3    while (*s != '\0') {
4      duz++;
5      s++;
6    }
7    return duz;
8  }
```

Obratite pažnju da nam promjenljiva `i` više ne treba. Za klizeći pokazivač↑ koristili smo `s` koji smo dobili kao parametar funkcije.

↑ Poglavlje 7.3.5, str. 247.

Linije 3 i 5 ove funkcije čine *standardnu petlju za rad sa stringovima* koja glasi:

```
while (*s != '\0') {
  ...
  s++;
}
```

Gotovo svaki program u kojem treba nešto uraditi sa stringom sadrži ovakvu petlju. Petlja while će se ponavljati sve dok pokazivač `s` ne bude pokazivao na znak \0, a naredbom `s++` pokazivač prelazi na sljedeći karakter. Tako karakter po karakter prolazimo kroz string i obavljamo neki zadatak koji se nalazi između ovih linija umjesto znaka tri tačke, pri čemu tekući karakter ima vrijednost `*s`. Bitno je da se naredba `s++` nalazi na kraju petlje kako ne bi bio preskočen početni karakter, niti ubrojan karakter \0.

Uslov u petlji se mogao kraće zapisati i ovako: `while (*s)`. Naime, kada pokazivač pokazuje na \0, vrijednost izraza `*s` će biti nula što odgovara logičkoj neistini te će se petlja prekinuti.

No, funkcija `duzina` se mogla napisati još kraće na sljedeći način:

```
1  int duzina(char* s) {
2    int duz = 0;
3    while (*s++) duz++;
```

1/15/2011 98

```
4     return duz;
5   }
```

Ovdje možemo vidjeti na djelu moć pokazivača za skraćivanje koda! Naime, kada napišemo kod `*s++` ovim smo mi uzeli tekuću vrijednost na koju pokazuje pokazivač s, a zatim pomjerili pokazivač na sljedeće mjesto (korištena je postfiksna varijanta operatora ++).↑ Ta tekuća vrijednost u ovom programu je neki znak, pa možemo uočiti da znak \0 ima ASCII vrijednost nule, što odgovara logičkoj neistini, a svi ostali znakovi imaju vrijednosti različite od nule, što odgovara logičkoj istini. Dakle, kada klizeći pokazivač naiđe na \0, vrijednost uslova će biti logička neistina i petlja će se prekinuti.

↑ Poglavlje 3.2.2, str. 81.

Ipak, trebate uočiti da je ovo skraćenje igrom slučaja pogodno za našu funkciju `duzina`, ali u većini drugih zadataka ga ne možemo koristiti. Naime, kada uslov petlje glasi `*s++`, u tijelu petlje pokazivač `s` već u prvom prolazu pokazuje na drugi znak po redu, pa ako se u zadatku traži npr. prebrojavanje samoglasnika, prvi znak neće biti ubrojan u samoglasnike. S druge strane, imaćemo jedan prolaz viška na kraju, pri kojem `s` pokazuje na \0. U ovom slučaju ta dva prolaza se međusobno ponište i vraćena vrijednost je tačna.

Zadatak 2. *Napisati funkciju "prebroji" koja prima string i jedan znak, a vraća broj koliko puta se dati znak pojavljuje u datom stringu (bibliotečna funkcija strchr).*

Prototip funkcije određujemo iz teksta zadatka (u prethodnom zadatku smo objasnili da veličina stringa nije bitna):

```
int prebroji(char* s, char znak);
```

Da bismo riješili ovaj zadatak, najprije trebamo uvrstiti *standardnu petlju za rad sa stringovima* koju smo vidjeli ranije.

```
1  int prebroji(char* s, char znak) {
2    int brojac = 0;
3    while (*s != '\0') {
4      ...
5      s++;
6    }
7    return brojac;
8  }
```

Ostaje da popunimo prazninu u liniji 4. Znamo da pokazivač `s` pokazuje na tekući znak u stringu, sada samo trebamo provjeriti da li je taj znak traženi i uvećati brojač ako jeste, što bi trebalo biti vrlo lako nakon što se podsjetimo primjera za rad sa nizovima koristeći pokazivačku aritmetiku.↑

↑ Poglavlje 7.3.1, str. 239.

```
1  int prebroji(char* s, char znak) {
2    int brojac = 0;
3    while (*s != '\0') {
4      if (*s == znak) brojac++;
5      s++;
```

1757019 8

```
  }
  return brojac;
}
```

Pogledajmo međutim jedan poziv ove funkcije:

```
char ime[] = "Aleksandar";
printf ("%d", prebroji(ime, 'a'));
```

Šta će ovaj program ispisati? Očigledno program broji samo mala slova 'a', nije ubrojano veliko slovo – što je prema postavci zadatka sasvim tačno (tako uostalom radi i bibliotečna funkcija `strchr`), ali po nekoj logici bi se moglo očekivati da se prebroje i velika slova.

Da bismo ispravili ovaj nedostatak, podsjetimo se ASCII tabele.↑ Sva velika slova nalaze se u jednom kontinualnom bloku između vrijednosti 65 i 90, i to abecednim redom, a mala također u bloku između 97 i 122 istim redom. Razlika između ova dva bloka je tačno 32 (što nije slučajno), pa tako npr. veliko slovo F ima kôd 70 a malo 102 (razlika: 32), veliko slovo K ima kôd 75 a malo 107 (razlika: 32) itd. Dakle da bismo od velikog slova dobili malo potrebno je da saberemo 32, a da bismo od malog dobili veliko trebamo oduzeti 32. Možemo recimo probati sva velika slova pretvoriti u mala prije poređenja:

↑ Poglavlje 3.1.7, str. 74.

```
int prebroji(char* s, char znak);
  int brojac = 0;
  while (*s != '\0') {
    *s += 32;
    if (*s == znak) brojac++;
    s++;
  }
  return brojac;
}
```

Ovdje imamo nekoliko problema. Prvo, ako tekući znak nije veliko slovo (npr. ako je malo), on će također biti uvećan za 32 i dobićemo nešto sasvim pogrešno. Ako vam se čini da to i nije prevelik problem, obratite pažnju u ASCII tabeli da npr. veliko slovo P ima ASCII vrijednost za tačno 32 veću od cifre 0, tako da u tom slučaju biste mogli greškom ubrojati nulu kao slovo P. Dakle trebamo popraviti ASCII vrijednost samo ako je tekući znak veliko slovo. Kako provjeriti da li je znak slovo? Jednostavno uočimo da se sva velika slova nalaze na spomenutom intervalu [65,90], dakle možemo pisati:

```
if (*s >= 65 && *s <= 90) *s += 32;
```

Ili, ako ne želite pamtiti ASCII vrijednosti, možete pisati ovako:

```
if (*s >= 'A' && *s <= 'Z') *s += 32;
```

jer se `char` literali mogu uvrstiti bilo gdje gdje bismo koristili ASCII vrijednost i obrnuto. Ustvari, ovakav kod je bolji jer je lakši za razumijevanje, nije potrebno konsultovati tablicu da vidimo šta predstavljaju "magične konstante" 65 i 90. Ali i 32 je magična konstanta, pa i nju možemo eliminisati sljedećim semantičkim kodom:

1715/0198

```
4  if (*s >= 'A' && *s <= 'Z') *s += 'a' - 'A';
```

odnosno `*s` uvećavamo za iznos razlike između ASCII kodova malog i velikog slova A.

Šta ako korisnik pozove funkciju tako da je promjenljiva `znak` veliko slovo? Trebali bismo isto rješenje primijeniti i na `znak` i to prije početka petlje u liniji 3:

```
3  if (znak >= 'A' && znak <= 'Z') znak += 'a' - 'A';
```

Program sada prividno radi ispravno, no probajmo nakon poziva funkcije prebroji ispisati ime:

```
char ime[] = "Aleksandar";
printf ("%d ", prebroji(ime, 'a'));
printf ("%s", ime);
```

Program će na ekranu ispisati 3 (što je tačno) a zatim `aleksandar`. Vidimo da je naša funkcija promijenila sadržaj niza `ime`, što je krajnje nepoželjno.

✓ *Funkcije ne bi trebale raditi stvari koje nisu očigledne iz njihove definicije. Pogotovo, funkcije koje primaju parametre preko pokazivača (dakle i nizove) ne bi smjele mijenjati ono na šta taj pokazivač pokazuje ako to nije njihova svrha, jer je to neočekivani sporedni efekat rada funkcije.*↑

✓ Sporedni efekti funkcija

↑ Str. 201.

Činjenicu da funkcija prima pokazivač ali ne mijenja ono na šta on pokazuje možemo posebno istaknuti tako što ćemo u prototipu umjesto tipa `char*` koristiti tip `const char*`. Kada ovako napravimo, prevodilac nam neće dozvoliti da promijenimo string u funkciji, što možemo zaobići deklaracijom pomoćnog pokazivača tipa `char*` (za što ćemo dobiti upozorenje). Štaviše, u programskom jeziku C++ ne možete npr. poslati literal u funkciju koja prima tip `char*`, dok nam mnogi C prevodioci čak neće ni dati upozorenje. Također i prototip funkcije `duzina` iz prethodnog zadatka treba prepraviti:

```
1  int duzina(const char* s);
```

Kako da ispravimo funkciju `prebroji`? Uvećanje vrijednosti znaka za 32 ne smijemo raditi nad pokazivačem `*s` pa ćemo uvesti neku pomoćnu promjenljivu `tmp` za te namjene. Time dobijamo finalnu verziju funkcije koja ne pravi razliku između velikih i malih slova, odnosno kaže se da je *case-insensitive*, dok je prethodna verzija pravila tu razliku, ili rekli bismo da je *case-sensitive*.

```
1  int prebroji(const char* s, char znak) {
2    int brojac = 0, tmp;
3    if (znak >= 'A' && znak <= 'Z')
4      znak += 'a' - 'A';
5    while (*s != '\0') {
6      tmp = *s;
7      if (tmp >= 'A' && tmp <= 'Z')
8        tmp += 'a' - 'A';
```

```
9       if (tmp == znak) brojac++;
10      s++;
11    }
12    return brojac;
13  }
```

Zadatak 3. *Napisati funkciju "uporedi" koja daje poredak dva stringa po leksikografskom poretku, te vraća vrijednost manju od nule ako je prvi string manji od drugog, veću od nule ako je prvi string veći od drugog, a o ako su jednaki (bibliotečna funkcija strcmp).*

Najprije podsjetimo se šta je to leksikografski poredak. To je poredak koji stringovi imaju nakon sortiranja, odnosno poredak u npr. telefonskom imeniku. Pogledajmo primjer sortiranog spiska riječi:

✓ Leksikografski poredak

```
aaa
aab
aac
aaca
aad
...
aaz
aba
abb
baa
baz
...
zaa
```

Kako se vrši leksikografsko poređenje? Najprije poredimo prvo slovo jednog stringa sa prvim slovom drugog, ako su jednaka prelazimo na drugo slovo prvog stringa i drugo slovo drugog stringa itd. Čim naiđemo na slova koja se razlikuju, znamo da je "manji" onaj string u kojem se na datoj poziciji nalazi slovo koje je "manje" po leksikografskom redoslijedu. Kao što smo rekli, slova su u ASCII tabeli složena po istom tom redoslijedu, pa je "manje" ono slovo čija je ASCII vrijednost manja. Ako su sva slova jednaka, "manji" je onaj string koji je kraći, a ako su jednake dužine onda su stringovi potpuno jednaki i funkcija treba vratiti 0.

Prototip funkcije sada ćemo definisati na sljedeći način:

```
int uporedi(const char* s1, const char* s2);
```

U prethodnom zadatku objasnili smo kako se pomoću tipa `const char*` osiguravamo od slučajne promjene stringa u funkciji koja je u pravilu nepoželjna.

Najprije trebamo uvrstiti standardnu petlju za rad sa stringovima, no u ovom slučaju imamo dva stringa pri čemu kroz oba prolazimo istovremeno. Dakle u petlji ćemo umjesto `s++` imati `s1++` i `s2++`, jer nakon što obradimo početna slova oba stringa trebamo preći na drugo slovo oba stringa. Šta će nam biti uslov za završetak petlje? Petlja bi se trebala prekinuti čim dođemo do kraja kraćeg stringa, jer ako nas-

17557098

tavimo sa upoređivanjem poslije tog kraja pristupaćemo nedefinisanim vrijednostima.

Prva verzija našeg programa dakle glasi:

```
1  int uporedi(const char* s1, const char* s2) {
2    while (*s1 != '\0' && *s2 != '\0') {
3      ...
4      s1++; s2++;
5    }
6    ...
7  }
```

Uslov u liniji 2 čitamo ovako: sve dok su oba pokazivača različita od nule, petlja se nastavlja, a kada jedan od njih postane nula - prekida se. Sljedeće pitanje je: šta raditi u tijelu petlje? Ovo smo objasnili kada smo tumačili leksikografski poredak, dakle ako je tekuće slovo prvog stringa "manje" od tekućeg slova drugog stringa, prvi string je "manji". Drugim riječima, odmah možemo izvršiti naredbu `return -1` jer nas naredni karakteri ne interesuju. Slično je i ako je slovo "veće", dok ako su slova jednaka ne radimo ništa, jer se petlja nastavlja na sljedeći par karaktera:

```
1  int uporedi(const char* s1, const char* s2) {
2    while (*s1 != '\0' && *s2 != '\0') {
3      if (*s1 < *s2) return -1;
4      if (*s1 > *s2) return 1;
5      s1++; s2++;
6    }
7    ...
8  }
```

Preostaje još da pokrijemo slučaj kada su stringovi jednaki do dužine kraćeg stringa. U tom slučaju kažemo da je "manji" onaj string koji je kraći. Kako ćemo to provjeriti? Recimo da je prvi string kraći od drugog. U tom slučaju `*s1` će po završetku ove petlje pokazivati na \0, a `*s2` neće, i obrnuto, ako je drugi string kraći onda će `*s2` pokazivati na \0 a `*s1` neće. Konačno, ako su jednake dužine, oba će pokazivati na \0. Mogli bismo svaki od ovih uslova raspisati u potpunosti, ali nakon pojednostavljenja uslova, kôd će izgledati ovako:

```
1  int uporedi(const char* s1, const char* s2) {
2    while (*s1 != '\0' && *s2 != '\0') {
3      if (*s1 < *s2) return -1;
4      if (*s1 > *s2) return 1;
5      s1++; s2++;
6    }
7    if (*s1 != '\0') return 1;
8    if (*s2 != '\0') return -1;
9    return 0;
10 }
```

Postoji još nekoliko načina kako se ova funkcija može kraće napisati. Za vježbu možete probati napraviti *case-insensitive* verziju ove funkcije.

Još da napomenemo da bibliotečnu funkciju `strcmp` nikada ne biste trebali pozivati ovako:

1 // D // UPI 98

```
if (strcmp(s1, s2) == -1)
  printf("Prvi je manji");
else ...
```

Naime, bibliotečna funkcija *ne vraća vrijednosti -1 ili +1*. Standard ne određuje koje vrijednosti funkcija treba vratiti, samo određuje da je to vrijednost manja odnosno veća od nule. Dakle, naša funkcija `uporedi` radi sasvim ispravno prema standardu, ali uobičajena implementacija funkcije `strcmp` je znatno brža od naše jer se ona obično piše na sljedeći način:

```
int strcmp(const char* s1, const char* s2) {
  while (*s1 != '\0' && *s1 == *s2) {
    s1++; s2++;
  }
  return *s1 - *s2;
}
```

Možete proanalizirati ovu funkciju da biste vidjeli da ona zaista vraća vrijednost 0 za jednake stringove, manju od 0 ako je prvi manji, a veću od 0 ako je prvi veći. Zbog ovoga je ispravno pisati:

```
if (strcmp(s1, s2) < 0)
  printf("Prvi je manji");
else ...
```

Zadatak 4. *Napisati funkciju "spoji" koja spaja dva stringa i vraća pokazivač na početak tako spojenog stringa (bibliotečna funkcija strcat).*

Sada imamo i prvu funkciju koja treba promijeniti jedan od stringova. Obratite pažnju da bibliotečna funkcija `strcat` funkcioniše tako što na string poslan kao prvi parametar nadovezuje drugi, što znači da se prvi string mijenja u funkciji a drugi ne. Pretpostavka je da pokazivač koji je prvi parametar pokazuje na niz čija je deklarisana veličina dovoljna da primi sve karaktere iz oba stringa. Funkcija to ne provjerava niti može provjeriti, pa programer koji je poziva mora paziti na takve stvari.[2]

[2] U standardnoj biblioteci postoji i funkcija `strncat` koja kao treći parametar ima broj znakova koji se kopiraju *n*. Postavljanjem n na fizičku veličinu niza osiguravamo se od prekoračenja.

Funkcija osim toga i vraća pokazivač na prvi parametar radi lakšeg *lančanog pozivanja*. Lančano pozivanje nam omogućuje da pišemo kraći kod. Npr. ako želimo da spojimo tri riječi u rečenicu mogli bismo pisati ovako:

```
printf("%s", spoji(rijec1, spoji(rijec2, rijec3)));
```

Prototip naše funkcije bi dakle trebao izgledati ovako:

```
char* spoji(char* s1, const char* s2);
```

Kako spojiti dva stringa u funkciji i vratiti rezultat? Neiskusni programeri često deklarišu pomoćne stringove u funkciji i zatim ih vraćaju:

```
char* spoji(char* s1, const char* s2) {
  char spojeno[1000];
  ...
```

1757019 8

```
  return spojeno;
}
```

Kao što znamo, *niz se ne može vratiti iz funkcije* i ovo bi bio viseći pokazivač.↑ Posebnu glavobolju zadaje činjenica da bi se takav program preveo i čak često davao tačan rezultat, ali bi povremeno neobjašnjivo krahirao iz razloga što je ustvari vraćen viseći pokazivač. Da bismo riješili ovaj zadatak nećemo kreirati pomoćne nizove nego ćemo jednostavno dodati sadržaj drugog niza na prvi te vratiti pokazivač na početak istog tog prvog niza.

↑ Poglavlje 7.4.1, str. 256.

Redoslijed koraka koje treba obaviti je sljedeći:

1. najprije trebamo postaviti pokazivač na kraj prvog stringa, kako bismo mogli dodavati nove karaktere;
2. zatim pomoću petlje kopiramo jedan po jedan karakter iz drugog stringa u prvi;
3. trebamo u pomoćnoj promjenljivoj sačuvati pokazivač na početak prvog stringa jer će nam trebati za kraj programa.

Da bismo pomjerili pokazivač `s1` do kraja stringa možemo pisati:

```
while (*s1 != '\0')
  s1++;
```

ili kraće:

```
while (*s1++);
```

Spominjali smo da znak \0 ima vrijednost nule što je logička neistina, pa pošto u istoj naredbi vršimo i pomjeranje pokazivača, tijelo petlje je prazno (nakon zagrade se nalazi samo tačka-zarez). U ovom slučaju ne radimo ništa sa sadržajem stringa, pa nam nije bitno to što smo preskočili prvo slovo. Ipak, obratite pažnju da nakon prve petlje pokazivač `s1` pokazuje na sam znak \0, a nakon druge pokazuje iza njega, tako da u takvim slučajevima morate paziti da slučajno ne dereferencirate taj pokazivač jer bi moglo doći do krahiranja. U našem programu treba nam baš pokazivač na znak \0, jer planiramo preko njega pisati sadržaj drugog stringa (ako to ne uradimo nećemo ništa napraviti jer \0 označava kraj stringa), tako da bismo nakon druge petlje trebali dodati još jedno `s1--;`.

Sljedeći zadatak je da kopiramo sve znakove iz stringa `s2` u string `s1`, odnosno, da se preciznije izrazimo, sve znakove od lokacije na koju pokazuje `s2` do najbližeg znaka \0 kopiramo sekvencijalno na lokaciju na koju pokazuje `s1`. Ovo je ujedno zadatak koji obavlja bibliotečna funkcija `strcpy`. Kod koji obavlja taj zadatak bi glasio:

```
while (*s2 != '\0') {
  *s1 = *s2;
  s1++;
  s2++;
}
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Naredba `*s1 = *s2` vrši dodjelu vrijednosti na koju pokazuje `s2` na lokaciju na koju pokazuje `s1`, dakle trenutni karakter će biti kopiran. Nakon toga pomjeramo oba pokazivača, što se nastavlja sve dok ne dođemo do kraja stringa `s2`. Zapamtite i da svaki string mora biti korektno *nul-terminiran* (mora se završavati znakom \0), pa nakon ove petlje treba dodati i `*s1 = '\0'`.

No ovo se moglo kraće zapisati i na sljedeći način:

```
while (*s1++ = *s2++);
```

Da bismo razumjeli ovaj ultra-kompresovani kod, najprije uočavamo da se ovdje pojavljuje jedan znak jednakosti a ne dva. Dakle ovo je naredba dodjele koja će ono na šta pokazuje `s2` pridružiti lokaciji na koju pokazuje `s1`, pa će se tek nakon te dodjele izvršiti pomjeranje oba pokazivača za po jedno mjesto, jer je u pitanju postfiksni operator ++. Zapamtimo i to da naredba dodjele ima vrijednost dodijeljenog, pa će u trenutku kada se iz `s2` prepiše u `s1` znak \0, ta vrijednost biti nula odnosno neistina, te će se petlja prekinuti. Pošto smo sav posao stavili u uslov petlje, tijelo je prazno.

Kompletan kod funkcije `spoji` (koristeći kraće varijante petlji) glasi:

```
1 char* spoji(char* s1, const char* s2) {
2   char* ret = s1;
3   while (*s1) s1++;
4   while (*s1++ = *s2++);
5   return ret;
6 }
```

Zadatak 5. *Napisati funkciju "podstring" koja prima dva stringa s1 i s2, te provjerava da li je s2 podstring stringa s1. Ako jeste, funkcija treba vratiti pokazivač na prvi znak u stringu s1 koji predstavlja početak podstringa, u suprotnom treba vratiti nul-pokazivač (bibliotečna funkcija strstr).*

Prototip funkcije dakle glasi:

```
const char* podstring(const char* s1, const char* s2);
```

Da bismo shvatili kako se ova funkcija koristi i šta radi, pogledajmo sljedeći kod:

```
char s1[] = "Ovo je neki primjer";
char s2[] = "primjer";
char* s3 = podstring(s1,s2)
```

Nakon ovog bloka pokazivač `s3` treba pokazivati na slovo `'p'` u stringu `s1`, dakle na lokaciju u prvom primljenom stringu na kojoj počinje podstring.

Šta znači kada kažemo da funkcija treba vratiti *nul-pokazivač*?↑ Često nam je potrebno da imamo neki pokazivač koji je korektno inicijaliziran, ali ne pokazuje nigdje posebno. Ako kasnije želimo da provjerimo da li pokazivač ima tu početnu vrijednost, to bi sa neinicijaliziranim

↑ Poglavlje 7.1.2, str. 225.

1757018 98

pokazivačem bilo nemoguće, jer on pri svakom pokretanju programa pokazuje negdje drugo. Tipičan primjer ovakvog slučaja je pokazivač koji pokazuje na prvi član nekog niza koji ispunjava određeni uslov – šta ako nijedan član niza ne ispunjava uslov?

Uobičajena praksa u programiranju je da se takvom pokazivaču pridruži adresa nula, jer je praktično garantovano da se na toj adresi neće nalaziti nešto na šta biste željeli pokazivati. U programskom jeziku C definisana je konstanta `NULL` koja se može koristiti da deklarišemo nul-pokazivač. Ranije je ista konstanta postojala i u programskom jeziku C++, no od standarda C++11 preferira se korištenje konstante `nullptr`.

Osim toga, korisna osobina adrese nula je da ona ima logičku vrijednost neistine. U sljedećem programu možemo vidjeti kako se poziva funkcija `podstring` pri čemu koristimo spomenutu osobinu nul-pokazivača:

```
1  #include <stdio.h>
2  int main() {
3    int s1[100], s2[100];
4    printf ("Unesite string: ");
5    unesi (s1, 100);
6    printf ("Unesite podstring: ");
7    unesi (s2, 100);
8    if (podstring(s1, s2))
9      printf ("Podstring se nalazi u stringu.\n");
10   else
11     printf ("Podstring se ne nalazi u stringu.\n");
12   return 0;
13 }
```

U liniji 8 biće pozvana funkcija `podstring`, pa ako ta funkcija vrati nul-pokazivač, on ima logičku vrijednost neistine pa će se ispisati poruka "...ne nalazi". U suprotnom biće vraćena neka adresa koja je po definiciji veća od nule, što ima vrijednost logičke istine.

Zadatak određivanja podstringa je u suštini jednak pronalasku podniza u nizu brojeva. Algoritam pronalaska podniza bi glasio ovako: za svaki član niza 1 provjeravamo da li se na tom mjestu nalazi podniz 2. Ovu provjeru vršimo tako što poredimo jedan po jedan član niza 2 sa odgovarajućim članom niza 1, što se prekida čim nađemo razliku ili dođemo do kraja niza 1. Ako smo došli do kraja niza 2 znači da je podniz pronađen. Daćemo rješenje zadatka:

```
const char* podstring(const char* s1, const char* s2) {
  while (*s1 != '\0') {
    const char* p = s1;
    const char* q = s2;
    while (*p++ == *q++)
      if (*q == '\0') return s1;
    s1++;
  }
  return NULL;
}
```

Koristimo standardnu petlju za prolazak kroz string `s1`. Kod svakog znaka kreiramo dva pomoćna pokazivača `p` i `q` iz razloga što ne smijemo izgubiti niti trenutnu lokaciju u stringu `s1` (jer je to ono što vraćamo ako je pronađen podstring) niti u stringu `s2` (u suprotnom se nećemo moći vratiti na tu lokaciju ako provjera ne uspije).

Izrazito kompresovana while petlja u liniji 5 podsjeća na ranije obrađenu funkciju `strcpy`, ali se ovdje koriste dva znaka jednakosti za poređenje umjesto jednog za dodjelu. Dakle ovdje se porede vrijednosti na koje pokazuju pokazivači `p` i `q`, te ako su one jednake uvećavaju se oba pokazivača za jedno mjesto i petlja se nastavlja, u suprotnom petlja se prekida. Unutar petlje moramo provjeriti da li smo došli do kraja stringa `s2` (pokazivač `q`) kako ne bismo prošli iza znaka \0 (linija 6). Ako jesmo, podstring je pronađen i možemo vratiti lokaciju njegovog početka, a to je upravo pokazivač `s1`. Ako podstring nije pronađen, `s1` prelazi na sljedeći karakter stringa (linija 7) tako da će `p` krenuti od te lokacije a `q` opet ispočetka.

Konačno, obratite pažnju i na liniju 9 u kojoj se vraća vrijednost `NULL` što predstavlja ranije spomenuti nul-pokazivač.

8.1.3 Rješavanje zadataka sa stringovima

Pri rješavanju zadataka u kojima se traži određena operacija nad stringom, trebate uvijek imati na umu da string nije nikakvo čudovište iz bajki, to je tek običan niz tipa `char` koji se završava znakom \0. Dakle, možemo primijeniti znanja stečena u poglavlju 5 o radu s nizovima. Pri tome, pokazivačku aritmetiku koristimo iz razloga elegancije i čitljivosti kôda, ali vi uvijek možete koristiti klasičan pristup sa for petljom.

Zadatak 6. *Napisati funkciju "max_slovo" koja prima string, a pronalazi i vraća slovo koje se najviše puta ponavlja u tom stringu. Zanemariti razliku između velikih i malih slova.*

Uočavamo da prebrojavanje koliko slovo se koliko puta javlja u stringu predstavlja tek poseban slučaj histograma↑ koji nam je poznat.

↑ Poglavlje 5.2.4, str. 155.

U engleskom alfabetu ima 26 slova, pa ćemo deklarisati niz od isto toliko brojača (`brojac`) koji nam označava koliko puta se dato slovo pojavljuje u stringu. Sada petljom prolazimo kroz string. Ako je tekući znak `*s` veliko slovo npr. 'E', trebamo povećati odgovarajući brojač za 1, a to će biti `brojac[4]` jer je 'E' peto slovo po redu. Uopšte, indeks brojača dobijamo tako što od trenutnog slova oduzimamo ASCII vrijednost slova 'A' a to je 65 (što ne moramo pamtiti jer u kod uvršavamo `char` literal 'A'). Isto tako važi i za mala slova.

Preostaje da pronađemo indeks najvećeg člana u nizu brojača (str. 148), te da vratimo njemu odgovarajuće slovo, a to je ništa drugo nego `imax+'A'`.

1/15/2018

```
char max_slovo(const char* s) {
  int brojac[26] = {0};
  while (*s != '\0') {
    if (*s > 'A' && *s < 'Z')
      brojac[ *s - 'A' ]++;
    if (*s > 'a' && *s < 'a')
      brojac[ *s - 'a' ]++;
    s++;
  }
  int i, imax = 0;
  for (i = 1; i < 26; i++)
    if (brojac[i] > brojac[imax])
      imax = i;
  return imax + 'A';
}
```

8.2 Neki algoritmi za rad sa stringovima

8.2.1 Pretvaranje broja u string

Česti zadaci za rad sa stringovima podrazumijevaju mogućnost međusobne konverzije stringova i brojeva (najčešće cijelih), što nije trivijalno, jer za to moramo poznavati ne samo rad sa stringovima i ASCII tabelu, nego i rastavljanje broja na cifre.↑ Pogledajmo jedan primjer zadatka.

↑ Poglavlje 4.4.4, str. 124.

Zadatak 7. *Napisati funkciju "trazi_broj" koja provjerava da li se u primljenom stringu pojavljuje zadana cjelobrojna vrijednost.*

Primjer upotrebe date funkcije može se vidjeti u kodu ispod:

```
char s[] = "Broj 123 je veci od 12 ili 3";
int broj = 123;
if (trazi_broj(s, broj))
  printf("Nalazi se.\n");
else
  printf("Ne nalazi se.\n");
```

Dakle funkcija prima string i neki cijeli broj (`int`) te vraća logičku vrijednost 0 ili 1. Neki bi se zaletjeli da napišu ovakvo trivijalno pogrešno rješenje:

```
int trazi_broj(const char* s, int broj) {
  while (*s != '\0') {
    if (*s == broj) return 1;
    s++;
  }
  return 0;
}
```

Problem je što bi ovakva funkcija `trazi_broj` bila ustvari identična funkciji `strchr` koju smo radili na str. 270. Ona bi dakle pronašla u stringu karakter čija je ASCII vrijednost `broj`.

Nama treba funkcija koja pronalazi broj kao sadržaj stringa. Dakle, da bismo našli broj 123 u stringu, potrebno je pronaći karaktere '1', '2' i '3', odnosno njihove ASCII vrijednosti, a to su 49, 50 i 51.

1770198

Jedan pristup rješavanju ovog zadatka je da napišemo kod koji pretvara broj u string, a zatim pozovemo bibliotečnu funkciju `strstr` (ranije napisana funkcija `podstring`). Prije svega, broj tipa `int` može sadržavati najviše deset cifara plus predznak, pa ćemo deklarisati string veličine fiksno 12 elemenata. Da bismo upisali broj u string najprije nam treba petlja koja rastavlja broj na cifre,↑ pri čemu svaku cifru treba pretvoriti u odgovarajući karakter. Naime, ako biste u string samo stavili cifru 5 na sljedeći način:

↑ Poglavlje 4.4.4, str. 124.

```
char sbroj[12];
sbroj[0] = 5; sbroj[1] = '\0';
```

na odgovarajućem mjestu bi se nalazio karakter čija je ASCII vrijednost 5, a to nije cifra 5. Cifra 5 ima ASCII vrijednost 53. Generalno uočavamo da se ASCII vrijednost cifre može dobiti tako što se na njenu vrijednost sabere broj 48, a to je ASCII vrijednost cifre nula – koju možemo predstaviti sa `'0'`.

Petlja za rastavljanje broja na cifre glasi:

```
while (broj != 0) {
  cifra = broj % 10;
  broj /= 10;
}
```

Sada ovu petlju možemo adaptirati tako da se cifre stavljaju u string, a ne smijemo zaboraviti da ga završimo znakom \0.

```
char sbroj[12];
char* sb = sbroj;
while (broj != 0) {
  cifra = broj % 10;
  *sb = cifra + '0';
  sb++;
  broj /= 10;
}
*sb = '\0';
```

Ovo je na prvi pogled ispravno rješenje, ali ako pokrenemo ovakav program, uočićemo da ćemo dobiti broj okrenut naopako. Naime, petlja za sjeckanje broja daje nam cifre s desna na lijevo, dok rad sa pokazivačem prolazi kroz string s lijeva na desno. Potrebno je da najprije postavimo pokazivač na lokaciju posljednje cifre.

```
char sbroj[12];
char* sb = sbroj+11;
int tmp = broj;
*sb-- = '\0';

while (broj != 0) {
  *sb-- = broj % 10 + '0';
  broj /= 10;
}
```

U ovom programu smo postavili pokazivač `sb` na posljednji član stringa `sbroj` i popunjavali prema lijevo. Ako broj ima manje cifara od maksimalnog mogućeg, imaćemo viška praznih mjesta na početku

stringa `sbroj`, ali to nam neće predstavljati problem jer ćemo funkciji `podstring` poslati pokazivač `sb` na njegovoj trenutnoj poziciji.

Ovaj program sada radi ispravno za pozitivne brojeve, no ostaje da ga adaptiramo tako da radi za negativne brojeve i nulu. U slučaju negativnog broja moramo uzimati apsolutnu vrijednost cifre, upisaćemo znak `'-'` na početak stringa, a podršku za nulu postići ćemo zamjenom while petlje sa do-while. Konačno ostaje da se pozove funkcija `podstring`. Kompletna funkcija `trazi_broj`:

```
int trazi_broj(const char* s, int broj) {
  char sbroj[12];
  char* sb = sbroj+11;
  int tmp = broj; /* Zbog predznaka */

  do {
    *sb-- = abs(broj % 10) + '0';
    broj /= 10;
  } while (broj != 0);

  /* Predznak */
  if (tmp < 0) *sb-- = '-';

  if (podstring(s, sb)) return 1;
  return 0;
}
```

No čak ni ovo rješenje nije bez problema. Naša funkcija `trazi_broj` će pronalaziti i broj kao dio nekog drugog broja, npr. ako string glasi `"Primjer -41572 aa"`, u njemu ćemo pronaći broj 15. Rješavanje ovog problema bi tražilo da dodamo posebne uslove da li se ispred i nakon pronađenog podstringa nalaze cifre.

8.2.2 *Ekstrakcija broja iz stringa*

Funkcija `trazi_broj` je postala prilično komplikovana i duga, pa ćemo umjesto toga probati riješiti zadatak na drugi način: pronaći nešto što liči na broj u stringu te napraviti konverziju u tip `int`, a zatim možemo takvu vrijednost uporediti sa primljenom. Okvirni algoritam bi bio sljedeći: kada u stringu naiđemo na znak cifre, konvertujemo sve cifre od tog mjesta do posljednje cifre u `int`, a zatim takav `int` poredimo sa primljenim.

Najprije, ako imamo standardnu petlju za prolazak kroz string, kako znamo da je trenutni karakter cifra? U ASCII tabeli uočavamo da se cifre nalaze u bloku na vrijednostima [48,57] pa možemo pisati:[3]

```
if (*s >= 48 && *s <= 57) ...
```

odnosno, da bismo izbjegli magične konstante, koristimo `char` literale:

```
if (*s >= '0' && *s <= '9') ...
```

Kako pretvoriti niz cifara u broj? Procedura je ustvari obrnuta od rastavljanja broja na cifre koju smo radili ranije. Najprije od cifre treba

[3] Ili možemo koristiti funkciju `isdigit` iz biblioteke `ctype.h` koja vraća broj različit od 0 ako je znak cifra, a broj 0 ako nije.

1 7 5 7 0 1 9 8

oduzeti ASCII vrijednost nule tj. `'0'` kako bismo dobili numeričku vrijednost cifre. Zatim tu cifru sabiramo na broj koji se u svakom prolazu kroz petlju množi sa deset. Petlja koja transformiše niz cifara u stringu u broj bi glasila:

```
tmp = 0;
while (*s >= '0' && *s <= '9') {
  tmp = tmp * 10 + *s - '0';
  s++;
}
```

U prvoj liniji koda postavljamo promjenljivu `tmp` na nulu jer će nula pomnožena sa 10 opet dati nulu. U svakom sljedećem prolazu na `tmp` se množenjem sa 10 dodaje još jedna nula s desne strane, a zatim se tu upisuje cifra. Ostaje još da uporedimo `tmp` sa primljenom promjenljivom `broj`:

```
1  int trazi_broj(const char* s, int broj) {
2    int tmp;
3    while (*s != '\0') {
4      if (*s >= '0' && *s <='9') {
5        tmp = 0;
6        while (*s >= '0' && *s <= '9') {
7          tmp = tmp * 10 + *s - '0';
8          s++;
9        }
10       if (tmp == broj) return 1;
11     }
12     s++;
13   }
14   return 0;
15 }
```

Ovakva funkcija će raditi sasvim ispravno pod pretpostavkom da je broj pozitivan. Ako broj može biti i negativan, moramo dodati još neke uslove, što se može vidjeti u kodu koji se nalazi ispod ovog pasusa. Najprije moramo provjeriti da li se u stringu nalazi znak minus, te postaviti neku promjenljivu `predznak` na vrijednost -1 (linija 4), u suprotnom njena vrijednost je 1 (linija 14). Prilikom poređenja trebamo množiti `tmp` sa ovim predznakom (linija 11). Obratite pažnju i na liniju 12 (vraćanje pokazivača `s`) kojom rješavamo problem sa brojem koji se nalazi na samom kraju stringa, kao i na situaciju da npr. u stringu `"tekst-123b456"` promjenljiva predznak ne bi bila vraćena na 1.

```
1  int trazi_broj(const char* s, int broj) {
2    int tmp, predznak = 1;
3    while (*s != '\0') {
4      if (*s == '-') predznak = -1;
5      else if (*s >= '0' && *s <='9') {
6        tmp = 0;
7        while (*s >= '0' && *s <= '9') {
8          tmp = tmp * 10 + *s - '0';
9          s++;
10       }
11       if (predznak * tmp == broj) return 1;
12       s--;
```

```
13      }
14      else predznak = 1;
15      s++;
16    }
17    return 0;
18  }
```

8.2.3 Rad sa rečenicama i riječima

Zadatak 8. *Napisati funkciju "nadji_rijec" koja prima dva stringa s1 i s2, pri čemu s1 predstavlja neku rečenicu, a s2 jednu riječ. Funkcija treba vratiti logičku istinu (1) ako se riječ nalazi u rečenici, a neistinu ako se ne nalazi. Riječ se definiše kao niz karaktera omeđenih sa jednim ili više znakova razmak, početkom ili krajem stringa.*

Na prvi pogled čini se da je to funkcija `podstring` koju smo ranije već napravili, no postoji bitna razlika: ovdje se traži samo kompletna riječ, a ne dio riječi. Riječ je ovdje definisana kao niz znakova omeđenih razmacima, početkom ili krajem stringa. Za rješenje ovog zadatka krenućemo od funkcije `podstring` (str. 278) i probati modificirati kôd koji smo dali tamo. Vidimo da u liniji 6 te funkcije trebamo nekako dodati provjeru da li pronađeni podstring predstavlja jednu kompletnu riječ. Ovdje se možemo poslužiti pokazivačima `s1` (koji pokazuje na početak pronađene riječi) i `p` (koji pokazuje iza njenog kraja) da provjerimo da li se ispred ili iza nalaze razmaci. Naivan uslov bi glasio:

```
const char* nadji_rijec(const char* s1, const char* s2) {
  while (*s1 != '\0') {
    const char* p = s1;
    const char* q = s2;
    while (*p++ == *q++)
      if (*q == '\0' && *(s1-1) == ' '
        && *p == ' ') return s1;
    s1++;
  }
  return NULL;
}
```

Ovaj kod ima dva problema: prvo, ako je pronađena riječ prva riječ u stringu, `s1` pokazuje na početak stringa što znači da će pristup lokaciji `*(s1-1)` dovesti do krahiranja. Osim toga, ako je pronađena riječ posljednja, `p` će pokazivati na \0. Sada uočavate značaj napomene da je riječ pored razmaka omeđena i početkom odnosno krajem stringa.

Ispravno rješenje je dato ispod. Da bismo riješili prvi problem, uvodimo pomoćni pokazivač `poc` koji pokazuje na početak stringa (linija 2), te provjeravamo da li mu je `s1` jednako (linija 8), dok za drugi jednostavno dodajemo još jedan uslov (linija 9):

17570198

```
1  const char* nadji_rijec(const char* s1, const char* s2) {
2    const char* poc = s1;
```

```
3     while (*s1 != '\0') {
4       const char* p = s1;
5       const char* q = s2;
6       while (*p++ == *q++)
7         if (*q == '\0'
8            && (s1 == poc || *(s1-1) == ' ')
9            && (*p == ' ' || *p == '\0')) return s1;
10      s1++;
11    }
12    return NULL;
13  }
```

Uočavamo u liniji 8 da sada neće doći do krahiranja ako `s1` pokazuje na početak stringa, jer ako je ispunjen uslov `s1 == poc`, uslov `*(s1-1) == ' '` se neće ni provjeravati zbog kratkospojne evaluacije.↑

↑ Poglavlje 2.4.2, str. 53.

Alternativan pristup rješavanju ovog zadatka je da na početak glavne petlje dodamo jednu pomoćnu petlju koja preskače sve razmake i na taj način osigurava da `p` uvijek pokazuje na početak riječi:

```
4   while (*s1 == ' ') s1++;
```

8.2.4 *Izbacivanje podstringa*

U programima se često traži da se iz stringa izbaci neki dio ili npr. riječ. Ovo je u suštini izbacivanje podniza iz niza, što je razrada problema izbacivanja člana iz niza uz očuvanje redoslijeda.↑ Podsjetimo se ovog algoritma:

↑ Poglavlje 5.2.2, str. 150.

```
1  for (i=0; i<n; i++) {
2    if (uslov) {
3      for (j=i; j<n-1; j++) {
4        niz[j] = niz[j+1];
5      }
6      n--;
7      i--;
8    }
9  }
```

Kako bi glasio kod koji iz niza izbacuje podniz od recimo 5 elemenata? Možemo uočiti tri mjesta u kodu iznad gdje se broj 1 treba zamijeniti brojem 5.

```
1  for (i=0; i<n; i++) {
2    if (uslov) {
3      for (j=i; j<n-5; j++) {
4        niz[j] = niz[j+5];
5      }
6      n-=5;
7      i--;
8    }
9  }
```

Pogledajmo sada ekvivalentan kod koji iz stringa izbacuje blok od 5 znakova, koristeći pokazivačku aritmetiku:

```
1  char* p = s1;
2  while (*p != '\0') {
```

1/5/0198

```
3     if (uslov) {
4       char *q = p, *r = p+5;
5       while (*q++ = *r++);
6     } else p++;
7   }
```

Umjesto for petlje koristimo standardnu petlju za prolazak kroz string (linije 2 i 6). `else` u liniji 6 igra istu ulogu kao `i--` u varijanti koja koristi indeksiranje niza (linija 7 u pretprošlom isječku koda). U liniji 5 možemo prepoznati našu skraćenu implementaciju funkcije `strcpy` (str. 276). Kopiranje se vrši sa lokacije `p+5` (za 5 mjesta pomaknuto od trenutne) na trenutnu lokaciju `p` čime je ustvari blok niza koji se nalazi između te dvije lokacije izbačen.

Iskoristićemo sljedeći zadatak da demonstriramo ove principe i proširimo znanje rada sa riječima.

Zadatak 9. *Napisati funkciju "izbaci_rijec" koja prima string koji predstavlja rečenicu i neki prirodan broj n, a zatim iz primljenog stringa izbacuje n-tu po redu riječ. Riječ se definiše kao niz karaktera omeđenih sa jednim ili više znakova razmak, početkom ili krajem stringa.*

Definicija riječi je kao i ranije, pa ćemo okvirni algoritam za rješavanje ovog zadatka opisati na sljedeći način:

1. pronalazimo n-tu riječ u stringu i postavljamo pokazivač `p` na njeno prvo slovo;
2. pokazivač `q` postavljamo na kraj n-te riječi – ustvari, postavićemo ga na početak (n+1)-ve riječi jer ćemo na taj način izbaciti i razmake koji se nalaze između te dvije riječi, tako da nakon izbacivanja riječi nećemo imati višak razmaka;
3. izbacujemo sve između pokazivača `p` i `q`.

Najprije, kako ćemo pronaći n-tu riječ? Jedan naivan pristup bi bio da brojimo razmake npr. ovako:

```
int broj_rijeci=0;
while (*s != '\0') {
  if (*s == ' ')
    broj_rijeci++;
  if (broj_rijeci == n) {
    ...
  }
  s++;
}
```

1/15/01198

Problem se dešava kada između dvije riječi imamo dva (ili više) razmaka, što će se brojati kao dvije (ili više) riječi. Dakle, potrebno je da pored tekućeg znaka provjerimo i da prethodni znak nije razmak. Ako ste pažljivo analizirali ranije zadatke, primijetićete da i ovdje moramo praviti poseban slučaj za prvu riječ u rečenici da se ne bi desilo da pristupamo znaku prije prvog. Nešto elegantniji je suprotan pristup: ako

tekući znak nije razmak a prethodni je bio razmak ili početak stringa, povećavamo broj riječi za jedan.

Prostudirajte sljedeće rješenje:

```
int razmak=1, broj_rijeci=0;
while (*s != '\0') {
  if (*s == ' ')
    razmak=1;
  else if (razmak == 1) {
    razmak=0;
    broj_rijeci++;
  }
  s++;
}
```

Pored promjenljive `broj_rijeci`, sada imamo i promjenljivu `razmak`. Ovo je logička promjenljiva čije je značenje: "da li je prethodni znak bio razmak?" U linijama 3 i 4 postavljamo ovu promjenljivu na istinu ako je trenutni znak razmak. Linija 5 počinje sa `else` što znači da se kôd u linijama 6 i 7 izvršava ako trenutni znak nije razmak a prethodni je bio. Obratite pažnju i na to da smo u liniji 1 postavili početnu vrijednost promjenljive `razmak` na 1, što znači da će se ovaj uslov ispuniti i ako prvi karakter u stringu nije razmak. Konačno, u liniji 6 smo postavili logičku promjenljivu na neistinu da se uslov ne bi ponovo izvršio za drugo, treće itd. slovo riječi.

Prelazimo na sljedeći korak u rješenju zadatka. Potrebni su nam pokazivači na početak n-te i (n+1)-ve riječi. Mada, ako malo razmislimo, drugi pokazivač nam i nije potreban – kada `s` bude pokazivalo na početak (n+1)-ve riječi trebamo prosto izbaciti sve između pokazivača `pocetak` i `s` i završiti funkciju (linije 11-14). U liniji 12 smo mogli koristiti i funkciju `strcpy`, ali to bi čak bilo duže od našeg skraćenog while-a.

```
void izbaci_rijec(char* s, int n) {
  int razmak=1, broj_rijeci=0;
  char* pocetak;
  while (*s != '\0') {
    if (*s == ' ')
      razmak=1;
    else if (razmak == 1) {
      razmak=0;
      broj_rijeci++;
      if (broj_rijeci==n) pocetak=s;
      if (broj_rijeci==n+1) {
        while (*pocetak++ = *s++);
        return;
      }
    }
    s++;
  }
}
```

Ova funkcija je sada 90% tačna, ali podsjetimo se da u programima koje radimo uvijek moramo razmišljati o rubnim situacijama. Šta će se

desiti kada iz stringa izbacujemo: prvu riječ, posljednju riječ, riječ prije prve (npr. nultu), riječ nakon posljednje?

Kada programer u funkciju pošalje vrijednost $n = 1$, biće izbačena *prva* riječ u stringu (što odgovara našem govornom jeziku a ne C konvenciji da se prvi član neke kolekcije označava indeksom nula). Našim razmatranjem promjenljive `razmak` postigli smo da će izbacivanje prve riječi ispravno raditi čak i kada se prije nje ne nalaze razmaci. Za slučaj $n = 0$ moramo dodati uslov za koji će se funkcija prekinuti bez izmjene stringa, jer bismo u suprotnom imali rad sa neinicijaliziranim pokazivačem `pocetak`. U slučajevima $n < 0$ i $n > br$ (broj riječi) uočavamo da uslov u liniji 11 neće nikada biti ispunjen i string će ostati nepromijenjen.

Ostaje pitanje izbacivanja posljednje riječi. Problem je što ni ovdje uslov u liniji 11 neće nikada biti ispunjen! Naime, taj uslov je ispunjen kada naiđemo na početak (n+1)-ve riječi, a to se neće nikada desiti ako je n posljednja riječ u rečenici. Za ovaj slučaj moramo dodati poseban uslov nakon glavne (vanjske) while petlje, tj. na sami kraj funkcije, koji bi glasio ovako:

```
18    if (broj_rijeci == n) *pocetak = '\0';
```

Podsjetimo se da C ignoriše sve nakon znaka \0 tako da je dovoljno postaviti tekući karakter na \0 da "obrišemo" sve što se nalazi nakon tog mjesta. Ako se u rečenici ukupno nalazi n riječi, `broj_rijeci` će imati vrijednost n nakon petlje, a to je slučaj koji tražimo.

Razmislimo sada o jednoj varijanti ovog zadatka u kojoj ćemo promijeniti definiciju riječi. Mnogi su nepažljivi prilikom korištenja znakova interpunkcije pa im se desi da ne ubace razmak nakon znaka. Npr. ako u rečenici imamo "...ovako,ali..." prethodni program će ovo posmatrati kao jednu riječ. Čest primjer je string "auto-stop" – da li je ovo jedna riječ ili dvije?

Zadatak 10. *Napisati funkciju "izbaci_rijec2" koja prima string koji predstavlja rečenicu i neki prirodan broj n, a zatim iz primljenog stringa izbacuje n-tu po redu riječ. Riječ je neprekinuti niz velikih ili malih slova.*

U odnosu na prethodni zadatak promijenila se posljednja rečenica tj. definicija riječi; ova funkcija treba u primjeru "auto-stop" izbrojati dvije riječi a ne jednu. Znači da ranije dat kod funkcije `izbaci_rijec` trebamo prepraviti na način da se kao graničnik između riječi, umjesto znaka razmaka `' '` može javiti bilo koji znak koji nije veliko ili malo slovo.

Uočavamo da se znak `' '` odnosno razmak pojavljuje na samo jednom mjestu, a to je linija 5. Uslov da je neki znak veliko ili malo slovo bi glasio:[4]

```
if (*s >= 'A' && *s <= 'Z' || *s >= 'a' && *s <= 'z')
```

[4] Isti zadatak obavlja funkcija `isalpha` iz biblioteke `ctype.h`.

17570198

pa ostaje da ovako dat uslov negiramo. Osim ove linije ćemo prepraviti još samo naziv logičke promjenljive:

```
void izbaci_rijec2(char* s, int n) {
  int neslovo=1, broj_rijeci=0;
  char* pocetak;
  while (*s != '\0') {
    if ( !(*s >= 'A' && *s <= 'Z' || *s >= 'a' && *s <= 'z') )
      neslovo=1;
    else if (neslovo == 1) {
      neslovo=0;
      broj_rijeci++;
      if (broj_rijeci==n) pocetak=s;
      if (broj_rijeci==n+1) {
        while (*pocetak++ = *s++);
        return;
      }
    }
    s++;
  }
  if (broj_rijeci == n) *pocetak = '\0';
}
```

Kvalitetno i dobro napisano rješenje programskog zadatka možemo prepoznati po tome što kasnije promjene postavke zahtijevaju izmjene samo u malom broju linija koda.

8.2.5 *Ubacivanje podstringa*

Još jedna varijanta zadataka sa stringovima je i ubacivanje nekog podstringa npr. ubacivanje riječi u rečenicu. Kao i u prethodnom poglavlju, uočavamo da je ovo varijanta problema *ubacivanja elementa u niz* uz očuvanje redoslijeda.

Posmatrajmo opšti oblik koda za *ubacivanje* elementa u niz dat u poglavlju 5.2.3:

```
for (i=0; i<n; i++) {
  if (uslov) {
    for (j=n; j>i; j--)
      niz[j] = niz[j-1];
    n++;
    niz[i] = ubaceni;
  }
}
```

Sada ovo rješenje trebamo adaptirati za stringove tj. za pokazivačku aritmetiku. Okvirni algoritam za ubacivanje podstringa u string bi glasio:

- postaviti pokazivač `p` na lokaciju gdje treba izvršiti ubacivanje;
- pokazivač `q` postaviti na kraj stringa;
- koristeći pokazivač `q`, pomjeriti sve znakove u stringu za n mjesta udesno (n je dužina podstringa), tako što će `q` klizati unaprijed prema `p`;

17570198

- ovim je stvoren prostor za upis podstringa.

Ovaj algoritam ćemo demonstrirati na sljedećem zadatku.

Zadatak 11. *Napisati funkciju "zamijeni1" koja u primljenom stringu zamjenjuje svaku cifru 1 sa tekstom "jedan". Pretpostaviti da je primljeni string dovoljno velik da primi sve dodatne karaktere.*

Primjer poziva funkcije dat je ispod:

```
char tekst[100] = "Ovo je 1 primjer 1131";
zamijeni1(tekst);
printf("%s", tekst);
```

Ovaj kod bi trebao na ekranu ispisati:

```
Ovo je jedan primjer jedanjedan3jedan
```

Obratite pažnju na drugu rečenicu zadatka. Kao što znamo, funkcija ne može znati kako ste vi deklarisali vaš string, pa se mora dopisati ovakva pretpostavka. Vidimo u primjeru da je `tekst` deklarisan sa veličinom 100, što je više nego dovoljno.

Pošto je string `"jedan"` duži od stringa `"1"`, ovo možemo posmatrati kao slučaj ubacivanja podstringa dužine 4 znaka (slovo `'j'` će se prepisati preko znaka `'1'`, tako da ga ne računamo). Prema prethodnom algoritmu, pišemo kod koji u stringu pronalazi karakter `'1'` (linija 3), postavlja pokazivač `kraj` na završetak stringa (linije 4 i 5), pomjera sve za četiri mjesta udesno (linije 6-9), te na tako dobijenu lokaciju upisuje string `"jedan"` (linije 10 i 11).

```
1  void zamijeni1(char* s) {
2    while (*s != '\0') {
3      if (*s == '1') {
4        char* kraj = s;
5        while (*kraj != '\0') kraj++;
6        while (kraj > s) {
7          *(kraj+4) = *kraj;
8          kraj--;
9        }
10       *s++ = 'j'; *s++ = 'e'; *s++ = 'd';
11       *s++ = 'a'; *s = 'n';
12     }
13     s++;
14   }
15 }
```

Po kriteriju koji smo dali na kraju prethodnog zadatka, ovo rješenje je vrlo loše, jer bi promjena postavke (npr. zamjena broja dva) zahtijevala promjene u čak 4 linije koda. Možemo rješenje učiniti kvalitetnijim tako što ćemo napraviti pomoćnu funkciju koja u datom stringu zamjenjuje bilo koji podstring nekim drugim podstringom, a zatim ćemo tu funkciju samo pozvati iz naše funkcije `zamijeni1`. Nazovimo tu funkciju `zamijeni_string`. Inače u vašim rješenjima je dozvoljeno, čak i poželjno, da koristite pomoćne funkcije radi bolje čitljivosti.

1777098

```
1  void zamijeni_string(char* s, const char* pretraga, const char* zamjena) {
2    int pomak = strlen(zamjena) - strlen(pretraga);
3    char* pozicija = strstr(s, pretraga);
4    while (pozicija) {
5      char* kraj = pozicija;
6      while (*kraj != '\0') kraj++;
7      while (kraj > pozicija) {
8        *(kraj+pomak) = *kraj;
9        kraj--;
10     }
11     const char* tmp = zamjena;
12     while (*tmp != '\0') *pozicija++ = *tmp++;
13     pozicija = strstr(s, pretraga);
14   }
15 }
16 void zamijeni1(char* s) {
17   zamijeni_string(s, "1", "jedan");
18 }
```

U liniji 2 odredili smo za koliko slova udesno trebamo sve pomaći, tj. koliko mjesta u stringu trebamo osloboditi, i to smo nazvali `pomak`. Petlja je sada potpuno drugačija, jer koristimo gotovu funkciju `strstr` čiji princip rada je opisan na stranici 278. Kada ova funkcija vrati nul-pokazivač, uslov u petlji će biti neistina i petlja će se završiti, u međuvremenu će pozicija biti početak podstringa `pretraga`. U liniji 11 nismo koristili gotovu funkciju `strcpy` da bismo string `zamjena` prekopirali na `poziciju`, jer bi ta funkcija kopirala i znak \0 što nam ne odgovara, nego smo koristili petlju slično kao u zadatku 4. Konačno, u liniji 14 provjeravamo da li se podstring nalazi na još nekom mjestu.

Međutim, šta ukoliko je string `pretraga` duži od stringa `zamjena`? U našem zadatku ovo se neće nikada desiti, tako da prethodno rješenje možemo smatrati za dovoljno dobro, no ne bi bilo u redu da se ovdje zaustavimo kada smo već dali solidno rješenje zamjene stringa. Kada je pretraga duža od zamjene, tu više nemamo ubacivanje u string, nego izbacivanje iz stringa, tako da je algoritam totalno drugačiji. Radi preglednosti funkciju `zamijeni_string` ćemo razdvojiti na dvije funkcije, `zamijeni_string_ubaci` koja vrši ubacivanje (poziva se kada je `pomak` pozitivan), a `zamijeni_string_izbaci` vrši izbacivanje (`pomak` je negativan). Da ne bismo ponovo računali pomak, proslijedićemo ga iz funkcije `zamijeni_string` u pomoćne funkcije.

```
1  void zamijeni_string_ubaci(char* s, const char* pretraga, const char*
     zamjena, int pomak) {
2    char* pozicija = strstr(s, pretraga);
3    while (pozicija) {
4      char* kraj = pozicija;
5      while (*kraj != '\0') kraj++;
6      while (kraj > pozicija) {
7        *(kraj+pomak) = *kraj;
8        kraj--;
9      }
10     while (*zamjena != '\0') *pozicija++ = *zamjena++;
11     pozicija = strstr(pozicija+pomak, pretraga);
12   }
```

```
13  }
14  void zamijeni_string_izbaci(char* s, const char* pretraga, const char*
       zamjena, int pomak) {
15    char* pozicija = strstr(s, pretraga);
16    while (pozicija) {
17      char* tmp = pozicija;
18      while (*tmp != '\0') {
19        *tmp = *(tmp+pomak);
20        tmp++;
21      }
22      while (*zamjena != '\0') *pozicija++ = *zamjena++;
23      pozicija = strstr(s, pretraga);
24    }
25  }
26  void zamijeni_string(char* s, const char* pretraga, const char* zamjena) {
27    int pomak = strlen(zamjena) - strlen(pretraga);
28    if (pomak > 0)
29      zamijeni_string_ubaci(s, pretraga, zamjena, pomak);
30    else
31      zamijeni_string_izbaci(s, pretraga, zamjena, -pomak);
32  }
33  void zamijeni1(char* s) {
34    zamijeni_string(s, "1", "jedan");
35  }
```

8.3 C++ *stringovi*

Kao što smo rekli, programski jezik C ustvari ne posjeduje tip za stringove, nego se koriste nizovi tipa `char`. Stoga je u programskom jeziku C++ uveden stvarni tip `string` koji rješava neke od nedostataka pri radu sa C stringovima. Ipak, osnovni algoritmi za rad sa stringovima koji su predstavljeni u ovom poglavlju većinom se mogu primijeniti i na C++ string.

8.3.1 *Rad sa C++ stringovima*

Kao što je C string ustvari niz tipa `char`, tako i C++ `string` je ustvari vrlo sličan vektoru tipa `char`,↑ a najčešće se tako i realizira, pa vam preporučujemo da se podsjetite kako se radi sa vektorima. Za stringove su nam dostupne funkcije `size`, `resize`, `push_back` i `at` poznate iz vektora, a tu je u funkcija `length` koja je sinonim za `size`.

↑ Poglavlje 5.4, str. 166.

Da biste mogli koristiti tip `string` u vašem C++ programu, najprije morate uključiti biblioteku koja se također zove `string`. Kao i kod vektora, kada deklarišete string on je prazan tj. veličine je nula. Ne samo da je moguće dodijeliti jedan string drugom, nego je C++ stringu moguće dodijeliti C string (niz tipa `char`) kao i string literale. Pogledajmo primjer:

```
1  #include <iostream>
2  #include <string>
3
4  int main() {
```

1755/1198

```
5         std::string ime = "Zoran", prezime;
6         char p[10] = "Pasic";
7         prezime = p;
8         std::cout << ime << " " << prezime << std::endl;
9         std::cout << ime.length() << " " << prezime.length() << std::endl;
10        return 0;
11 }
```

U liniji 5 deklarisana su dva stringa `ime` i `prezime`. Stringu `ime` smo odmah dodijelili tekst "Zoran" tj. pridružili smo mu znakovni (string) literal, dok je `prezime` ostao prazan. Da smo u tom trenutku na ekranu ispisali `prezime.length()` dobili bismo nulu. U liniji 7 možemo vidjeti da se C++ stringu može pridružiti C string običnom naredbom dodjele (znakom =), dok obrnuto nije moguće.

Kada od C++ stringa želite dobiti C string, možete koristiti funkciju `c_str()`, te tako dobiveni string možete kopirati sa `strcpy`:

```
1  #include <iostream>
2  #include <string>
3  #include <cstring>
4
5  int main() {
6          std::string ime = "Zoran", prezime = "Pasic";
7          char p[10];
8          std::strcpy(p, prezime.c_str());
9          ...
10         return 0;
11 }
```

U liniji 3 smo pored biblioteke `string` morali uključiti i biblioteku `cstring`,[5] jer se u njoj nalazi funkcija `strcpy`. Pozivom `prezime.c_str()` dobili smo pokazivač tipa `const char*` koji možemo iskoristiti samo kako bismo taj string ispisali ili prekopirali u neki C string. Naravno, nije nam garantovano da `prezime` nema više od 9 znakova, pa možemo iskoristiti funkciju `strncpy` da se zaštitimo od grešaka.

[5] Ova biblioteka se u C-u zove `string.h` – pogledajte napomenu o imenovanju biblioteka datu u uvodu u knjigu

Kao i kod vektora, moguća je deklaracija stringa sa specificiranom veličinom u malim zagradama, ali isključivo pod uslovom da se navede i drugi parametar, a to je vrijednost znaka kojim će string biti popunjen, npr. ovako: `std::string s(50, ' ');`.

8.3.2 *Unos stringova u C++u*

Za unos podataka u C++u možemo koristiti `scanf`, ali preporučljivo je koristiti objekat ulaznog toka `cin`.↑ Ipak, kod unosa tekstualnih podataka pomoću ovog objekta postoji nekoliko specifičnosti na koje treba skrenuti pažnju.

↑ Poglavlje 1.2.9, str. 37.

Prije svega, objekat `cin` zna raditi sa vrijednostima tipa `char` i tipa `string`, pri čemu se unos u string prekida sa znakom razmaka, tj. moguće je unijeti samo jednu riječ. Ako to nije ono što želimo, koristi se funkcija `getline` koja (kao što njeno ime kaže) omogućuje unos čitave linije teksta odjednom, a taj unos se prekida tipkom Enter. Sintaksa

1/1/5/7/0/1/9/8

korištenja ove funkcije glasi: `std::getline(std::cin, s);` pri čemu je `s` neki string koji unosimo, a `std::cin` se navodi doslovce, tj. u pitanju je poznati objekat.

```
std::string rijec, recenica;
std::cout << "Unesite recenicu: ";
std::getline(std::cin, recenica);
std::cout << "Unesite rijec: ";
std::cin >> rijec;
```

Funkcija `getline` se može koristiti i za unos C stringova, ali je ovdje sintaksa nešto drugačija.

```
char recenica[50];
std::cin.getline(recenica, 50);
```

Kao i kod funkcije `unesi`,↑ moramo kao drugi parametar dati fizičku veličinu niza kako je on deklarisan, kako ne bi došlo do prekoračenja veličine. Ujedno, ova funkcija `cin.getline` radi na isti način kao i naša funkcija `unesi` pa je možemo koristiti umjesto nje.

↑ Poglavlje 8.1.1, str. 265.

8.3.3 *Povećavanje stringova*

Budući da je string interno vektor `char`-ova, to se veličina stringa može dinamički povećavati pozivom funkcije `push_back` kao i sa vektorima:

```
std::string s;
...
for (int i=0; i<n; i++)
  s.push_back('a');
```

ali postoji i lakši način. Nad stringom definisan je operator + koji ima značenje spajanja stringova. Dostupan je naravno i += kombinovani operator. Ovaj operator radi slično kao i naša funkcija `strcat`↑ s tim što tu ne moramo voditi računa o veličini stringova jer će rezultat spajanja biti dovoljno velik da primi sve znakove iz oba stringa. Osim toga možemo spojiti dva stringa da dobijemo treći. Pored spajanja dva C++ stringa operator + možemo primijeniti da na C++ string saberemo C string ili string literal. Pogledajmo neke primjere:

↑ Str. 275.

```
1  #include <iostream>
2  #include <string>
3
4  int main() {
5    std::string ime = "Zoran", prezime = "Pasic";
6    std::string puno_ime = ime + prezime;
7    std::cout << "Puno ime: " << puno_ime << ", duzina: " << puno_ime.length()
       << std::endl;
8    std::string rezultat;
9    ...
10   for (int i=0; i<vel; i++)
11     rezultat += "Ok ";
12   ...
13   return 0;
14 }
```

U liniji 6 možemo vidjeti spajanje dva stringa operatorom +. Izvorni stringovi `ime` i `prezime` ostaju nepromijenjeni, a novi string `puno_ime` će imati 10 znakova što ćemo saznati u liniji 7. U liniji 11 vidimo kako se kombinovanim operatorom += string može povećavati proizvoljan broj puta literalom.

8.3.4 *Operacije nad stringovima*

I druge često korištene funkcije se kod C++ stringova postižu koristeći operatore. Kao i kod vektora, stringove možemo porediti koristeći operator ==. Kao i ranije, ovdje se može pojaviti i C string. Sljedeći program daje primjer unosa vektora stringova, pri čemu nisu dozvoljeni duplikati:

```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4
5  int main() {
6    std::vector<std::string> vektor;
7
8    std::cout << "Unesite niz rijeci ('KRAJ' za zavrsetak): ";
9    while (true) {
10     std::string rijec;
11     std::cin >> rijec;
12     if (rijec == "KRAJ") break;
13
14     bool pronadjena = false;
15     for (int i=0; i<vektor.size(); i++)
16       if (vektor[i] == rijec)
17         pronadjena = true;
18     if (pronadjena)
19       std::cout << "Vec ste unijeli rijec " << rijec << std::endl;
20     else
21       vektor.push_back(rijec);
22   }
23   ...
```

U liniji 12 vidimo da se C++ string može porediti sa literalom, a u liniji 16 string se poredi sa članom vektora (koji je string).

Pored toga dostupni su i operatori < i > koji daju *leksikografski poredak*↑ dvije riječi. Ova tri operatora zajedno eliminišu potrebu za korištenjem funkcije `strcmp`. Pogledajmo primjer sortiranja↑ vektora stringova koji je unesen u prethodnom programu:

↑ Str. 273.

↑ Poglavlje 5.2.5, str. 158.

```
24   for (int i=0; i<vektor.size(); i++) {
25     int max = i;
26     for (int j=i+1; j<vektor.size(); j++)
27       if (vektor[j] > vektor[max])
28         max = j;
29
30     std::string tmp = vektor[i];
31     vektor[i] = vektor[max];
32     vektor[max] = tmp;
33   }
```

Pitanja za razumijevanje gradiva

1. Kako se označava string literal?

2. Navedite primjer validnog niza tipa `char` koji nije validan string.

3. Zašto je potrebno koristiti ključnu riječ `const` u deklaracijama oblika `const char* ime = "Alen";`?

4. Na koji način možemo sigurno unijeti string pomoću tastature?

5. Zašto u radu sa stringovima koristimo pokazivačku aritmetiku?

6. Na koji način se u funkcijama postiže da se ne razlikuju velika i mala slova?

7. Na koji način pretvaramo cijeli broj (vrijednost tipa `int`) u string, a na koji način string u `int`?

8. Na koji način string razdvajamo na riječi?

9. Koje osobine C++ stringova nam olakšavaju rad u odnosu na C stringove (nizove tipa `char`?

Zadaci za vježbu

Zadatak 12. *Napisati funkciju "samoglasnici" koja će prebrojati broj samoglasnika u primljenom stringu.*

Zadatak 13. *Napišite funkciju koja u primljenom stringu sva mala slova pretvara u velika. Funkcija treba da vraća pokazivač na početak istog stringa koji je proslijeđen funkciji kao parametar. Ovo omogućuje da se funkcija poziva lančano, npr. možete pisati:*

```
char s[] = "Ovo je primjer.";
printf("%s", velika(s));
```

Zadatak 14. *Napišite funkciju koja string zamjenjuje prvom riječi u stringu. Npr. ako je dat string "Danas je lijep dan.", funkcija ga treba pretvoriti u "Danas". Funkcija vraća broj riječi u polaznom stringu (prije odsijecanja ostalih riječi), dakle u navedenom primjeru funkcija treba vratiti broj 4.*

Zadatak 15. *Modificirajte funkciju "uporedi" koju smo dali na str. 273, tako da ima još jedan parametar tipa `int`. Ako taj parametar ima vrijednost*

1757018

različitu od nule, funkcija prilikom poređenja dva stringa treba zanemariti razliku između velikih i malih slova.

Zadatak 16. *Napišite funkciju pod nazivom rot13 koja vrši šifrovanje i ujedno dešifrovanje primljenog stringa po ROT13 algoritmu.*

Pod pojmom "ROT13 algoritam" misli se na to da se svako slovo u primljenom tekstu zamjenjuje slovom koje je udaljeno za 13 mjesta u engleskoj abecedi. Npr. slovo A se zamjenjuje slovom N, slovo B slovom O itd. Ako se pri brojanju mjesta dođe do slova Z, brojanje se nastavlja od početka, pa se npr. slovo M zamjenjuje slovom Z, a slovo N se zamjenjuje slovom A jer smo krenuli od početka abecede, slovo O se zamjenjuje slovom B itd. Iz ovoga vidimo da ista funkcija radi i šifrovanje i dešifrovanje teksta pošto engleska abeceda ima tačno 26 slova.

Pazite da se u funkciji velika slova zamjenjuju velikim a mala malim, dok ostale znakove (razmake, brojeve itd.) ostavljate nepromijenjenim. Radi lakšeg lančanog pozivanja, funkcija treba da vraća pokazivač na prvo slovo primljenog stringa.

Zadatak 17. *Napisati funkciju koja u datom stringu zamjenjuje sve "whitespace" znakove jednim razmakom. Dakle, novi redovi (\n) i tabovi (\t) trebaju biti zamijenjeni razmacima. Također, ukoliko se javlja više razmaka zaredom, treba ih zamijeniti jednim razmakom. Whitespace znakove na početku ili kraju stringa treba izbaciti. Ovo uključuje i kombinacije razmak-tab ili novi red-razmak itd. Funkcija treba vratiti pokazivač na prvi znak primljenog stringa radi lakšeg lančanog pozivanja. Napravite i kraći program koji demonstrira korištenje ove funkcije.*

Zadatak 18. *Napišite funkciju pod imenom "ukloni_komentare" koja iz datog stringa, koji predstavlja C kod, izbacuje sve komentare. Funkcija treba da podržava komentare u C stilu i u C++ stilu. Pretpostavite da će komentari u C stilu biti uvijek propisno zatvoreni. Napravite i kraći program koji demonstrira korištenje ove funkcije.*

17570198

1757018 8

9
Vlastiti tipovi podataka

U dosadašnjem dijelu knjige upoznali smo se sa različitim tipovima podataka koje podržava programski jezik C: `int`, `float`, `char`, različite vrste nizova, pokazivači... No ovi tipovi ne mogu pokriti sve moguće slučajeve koji se pojavljuju u programiranju. Ponekad se desi da nam je potreban neki tip koji nije podržan u programskom jeziku C.

Neke od mogućnosti koje programski jezik C nudi za definisanje vlastitih tipova podataka su: *typedef, enum, struct i union*.

9.1 *Ključna riječ* `typedef`

Jednu vrstu vlastitog tipa podataka u programskom jeziku C definišemo pomoću ključne riječi `typedef`. Ova ključna riječ se koristi kako bismo nekom postojećem tipu podataka dali drugačije ime (alias) radi bolje čitljivosti koda, te promovisanja semantike tipa. Recimo pogledajte sljedeći primjer:

```
typedef int visina;
typedef float cijena;
visina h;
cijena km;
```

Tip `visina` je ustvari drugo ime (alias) za tip `int`, pa kada napišemo `visina h;` promjenljiva `h` je ustvari tipa `int`, s njom možete raditi sve što i sa promjenljivom tipa `int`. Međutim onaj ko čita program može odmah uočiti da je `h` visina a ne recimo temperatura, pa samim time može zaključiti npr. da 1000 kilometara ili negativna vrijednost nisu logične vrijednosti ali su ipak dozvoljene (u suprotnom smo mogli izjednačiti visinu sa `unsigned int`).

Sintaksa naredbe `typedef` grubo rečeno je sljedeća:

- ključna riječ `typedef`,
- neka validna definicija tipa u programskom jeziku C,
- naziv za vlastiti tip.

17570198

Često ćete pročitati i da se `typedef` koristi tako što deklarišete "promjenljivu" određenog tipa, zatim ispred svega dodate `typedef`, i nakon toga "promjenljiva" je ustvari naziv novog tipa.

`typedef` se često koristi i kako bi se kod učinio lakšim za čitanje. Npr. pogledajte ovaj primjer:

```
typedef int (*red_matrice)[10];
void dijagonala(red_matrice r, int visina) {
  ...
}
```

Kada pošaljemo matricu u funkciju mi ustvari dobijamo pokazivač na red matrice,↑ a sintaksa za deklarisanje ovakvog pokazivača je poprilično nezgrapna, što je potrebno da bi se isti razlikovao od niza pokazivača. Kôd u kojem se javlja ovakav pokazivač brzo postaje vrlo neugodan za čitanje. Zbog toga smo u primjeru iznad definisali vlastiti tip pod nazivom `red_matrice`.

↑ Poglavlje C.3.3, str. 391.

Ovdje možete uočiti i spomenutu osobinu da se `typedef` koristi kao deklaracija promjenljive. Kôd je na prvi pogled zbunjujući jer naziv tipa nije posljednja stvar u redu, ali vrlo lako možete zaključiti logiku: najprije napišemo deklaraciju jednog pokazivača na red matrice, zatim ispred dodamo ključnu riječ `typedef`.

Možete zaključiti da sa `typedef` ne možemo uraditi ništa suštinski novo u odnosu na ono što smo do sada radili, to je samo drugi način zapisivanja imena tipova. `typedef` se koristi i npr. kada želimo uvesti jedinstveno ime za veličinu koja ovisi o platformi, pa će ta veličina npr. na 32-bitnom sistemu imati veličinu 32 bita, a na 64-bitnom sistemu 64 bita.

9.2 *Pobrojani tip (`enum`)*

Promjenljive koje imaju **pobrojani tip podataka** mogu poprimiti isključivo jednu od vrijednosti koje su navedene u definiciji tipa. Npr. rekli smo da u programskom jeziku C ne postoji logički (booleov) tip podataka, a to je tip za koji promjenljive mogu imati samo jednu od dvije vrijednosti: istina (true) ili neistina (false). Ovo bismo mogli riješiti tako što kreiramo novi *pobrojani tip* pod nazivom `boolean` pri čemu definišemo da su vrijednosti koje podržava taj tip `true` i `false`.

Pobrojani tip u programskom jeziku C kreira se koristeći ključnu riječ `enum`. Najprije se navodi `enum`, zatim naziv tipa, te se u vitičastim zagradama navode vrijednosti koje taj pobrojani tip može poprimiti. Nakon što ovako opišemo neki pobrojani tip možemo deklarisati promjenljive tog tipa tako što navedemo ključnu riječ `enum` (nije potrebno u C++u), naziv tipa, te zatim nazive jedne ili više promjenljivih eventualno uz inicijalizaciju. Konkretno, na sljedeći način smo deklarisali promjenljivu `t` opisanog tipa.

1/15/01/98

```
enum boolean { false, true }
enum boolean t;
```

Pogledajmo još neke primjere:

```
1 enum Boja { Crvena, Plava, Bijela, Crna, Smedja };
2 enum Boja b = Crvena;
3
4 enum boolean { false, true } t;
5
6 enum DanUSedmici { Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota,
     Nedjelja };
```

U liniji 1 opisan je novi pobrojani tip `Boja`, a promjenljive tog tipa mogu poprimiti vrijednosti navedene u zagradi. Naravno mi znamo da ima više boja osim ovih, ali ako ih budemo željeli koristiti moramo ih dodati u definiciju pobrojanog tipa. U liniji 2 deklarisana je promjenljiva pobrojanog tipa `Boja` pod nazivom `b` te joj je odmah pridružena vrijednost `Crvena`.

Obratite pažnju da se u liniji 1 nakon zatvorene vitičaste zagrade nalazi znak tačka-zarez. Ovo je još jedan od izuzetaka od pravila koje smo dali u prvom poglavlju: da se kod otvorene ili zatvorene vitičaste zagrade (ili tamo gdje su one izostavljene) ne piše znak tačka-zarez. Razlog zašto je ovdje to obavezno jeste što nam ključna riječ `enum` omogućuje da odmah nakon kreiranja novog tipa deklarišemo i promjenljive tog tipa! Primjer za to možete vidjeti u liniji 4: definisan je tip `boolean` o kojem smo govorili ranije,[1] ali je odmah zatim i deklarisana promjenljiva `t` tog tipa.

Konačno u liniji 6 možete vidjeti još jedan koristan pobrojani tip koji ćete često koristiti u vašim programima. Rekli smo da je uobičajena praksa u programskom jeziku C da se promjenljive i funkcije imenuju isključivo malim slovima,↑ a konstante isključivo velikim, pri čemu su riječi razdvojene znakom donje crte. No, za vlastite tipove podataka koristićemo tzv. CamelCase notaciju kod koje je prvo slovo svake riječi veliko, a ostala mala, a riječi nisu razdvojene. Primjer ovoga je naziv tipa `DanUSedmici`.

[1] Ključne riječi `bool` i `Bool` su rezervisane u programskom jeziku C iako se ne koriste u samom jeziku, ali se koriste u standardnoj biblioteci. S druge strane, tip `bool` je u C++u dio jezgre programskog jezika.

↑ Poglavlje 1.2.3, str. 29.

Interesantna osobina pobrojanog tipa u programskom jeziku C jeste što on također predstavlja cjelobrojni tip, tj. interno se realizira kao tip `int`. Kada napišete kôd kao što je ovaj:

```
enum Boja { Crvena, Plava, Bijela, Crna, Smedja };
```

vrijednostima *Crvena, Plava...* biće pridruženi redom brojevi 0, 1, 2... tako da, kada u programu gdje je definisana boja na opisani način napišete `Smedja` to je potpuno isto kao da ste napisali broj 4:

```
if (Bijela+Bijela == Smedja) printf("2+2=4\n");
```

Ovo možete i promijeniti! Ako prilikom definicije vrijednosti koje spadaju u pobrojani tip navedete znak jednakosti a zatim neki cjelo-

1/15/01/98

brojni literal, ta vrijednost će biti izjednačena sa navedenim cijelim brojem a ne onim koji bi to trebao biti po redu, npr.:

```
enum boolean { true=1, false=0 };
```

Ova osobina nam omogućuje i da vršimo aritmetičke operacije nad pobrojanim vrijednostima. Npr. sljedeći kod je sasvim legalan:

```
1 enum DanUSedmici { Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota,
     Nedjelja };
2 enum DanUSedmici dan = Subota;
3 dan++; /* Nedjelja */
```

U liniji 2 promjenljiva `dan` dobila je vrijednost `Subota`, a to je cijeli broj 5. U liniji 3 povećaćemo `dan` za jedan, čime on ima vrijednost 6. To je validna vrijednost i odgovara danu `Nedjelja`. Dakle, na ovaj način možemo napisati petlju koja prolazi kroz sve dane u sedmici:

```
for (dan=Ponedjeljak; dan<=Nedjelja; dan++)
```

Obratite pažnju da se kod aritmetičkih operacija nad pobrojanim tipom ne vrši kontrola ispravnosti. To znači da ako nakon prethodnog koda izvršimo još jednom `dan++`, promjenljiva će sada imati vrijednost 7 koja nije navedena u definiciji tipa, dakle to je ilegalna vrijednost koja ne odgovara niti jednom danu u sedmici.

Napomenimo da u programskom jeziku C++ operacije poput `dan++` ili `dan=5` nisu dozvoljene kako se ne bi dolazilo u ovakve situacije.

Promjenljive pobrojanog tipa se kod ulaza i izlaza ponašaju kao cijeli brojevi npr.

```
1 enum DanUSedmici dan = Subota;
2 printf("%d", dan); /* 5 */
3 scanf("%d", &dan); /* Unosimo broj */
```

Kada u liniji 2 probamo ispisati promjenljivu `dan`, na ekranu ćemo dobiti broj 5, a ne riječ "Subota". Slično je i za unos, jedini način da unesemo promjenljivu pobrojanog tipa je kao broj. Kako da ovaj problem riješimo? Na žalost, jedino rješenje koje nam programski jezik C nudi jeste switch-case:

```
switch (dan) {
  case 0:
    printf("Ponedjeljak");
    break;
  case 1:
    printf("Utorak");
    break;
  case 2:
    printf("Srijeda");
    break;
  ...
  default:
    printf("Ilegalan dan!");
}
```

Možemo i nešto bolje od ovoga. Pošto je pobrojani tip cijeli broj, možemo deklarisati niz sa imenima dana te koristiti pobrojani tip kao indeks u nizu:

```
1 enum DanUSedmici { Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota,
     Nedjelja };
2 char imena_dana[7][12] = { "Ponedjeljak", "Utorak", "Srijeda", "Cetvrtak",
      "Petak", "Subota", "Nedjelja" };
3 ...
4 enum DanUSedmici dan = Subota;
5 printf("%s", imena_dana[dan]);
```

Najprije obratimo pažnju na deklaraciju niza `imena_dana` u liniji 2. U pitanju je niz stringova, odnosno matrica tipa `char`↑ pri čemu je svaki red matrice dužine 12 znakova, što je dovoljno za najduže ime dana (Ponedjeljak). Ukupno matrica ima 7 redova. Ovaj niz smo inicijalizirali sa 7 stringova, zbog čega smo u inicijalizaciji koristili dvostruke navodnike. U liniji 5 ispisali smo na ekranu ime dana sadržanog u promjenljivoj `dan` pobrojanog tipa `DanUSedmici`. Znamo da ova promjenljiva ima vrijednost cijelog broja iz [0,6], a ako to nije sigurno možemo pisati:

↑ Poglavlje C.3.2, str. 386.

```
if (dan < Ponedjeljak || dan > Nedjelja)
  printf("Ilegalan dan!");
else
  printf("%s", imena_dana[dan]);
```

Vidimo u liniji 5 da je korišten format `%s` jer na ekranu ispisujemo string, te da je promjenljiva `dan` korištena kao indeks u nizu `imena_dana`, dakle dobili smo odgovarajući red matrice što je string sa nazivom dana "Subota" kojeg ispisujemo na ekranu.

9.3 *Strukturni tip podataka (`struct`)*

Do sada smo se upoznali sa ključnim riječima `typedef` i `enum` koje nam ustvari ne nude ništa suštinski novo, nego samo da damo preglednija imena postojećim tipovima podataka i vrijednostima. Za razliku od njih, *strukturni tip podataka*, koji se kraće naziva i **struktura**, omogućuje nam da definišemo novi tip podataka koji se sastoji od više elemenata nekog od poznatih tipova.

Npr. u mnogim programima potrebno je raditi sa datumima. Primijetili smo da, za razliku od nekih drugih programskih jezika, C ne posjeduje tip podataka *Date*, pa smo umjesto toga koristili tri promjenljive tipa `int`. Npr. ako smo koristili u programu tri datuma, moglo se desiti da imamo promjenljive koje se zovu *dan_rodjenja, mjesec_rodjenja, godina_rodjenja, dan_upisa, mjesec_upisa...* što je prilično nepregledno. Umjesto toga, možemo definisati novi strukturni tip pod nazivom "Datum", čiji su elementi cjelobrojne promjenljive "dan", "mjesec" i "godina".

1777198

9.3.1 *Definicija strukturnog tipa i deklaracija promjenljivih*

Definicija strukture započinje ključnom riječi `struct`, nakon koje slijedi *naziv strukture* koji se još naziva i *tag*. U vitičastim zagradama navodimo *članove ili elemente strukture* koristeći istu sintaksu kao da deklarišemo promjenljive. Nakon zatvorene vitičaste zagrade nalazi se znak tačka-zarez, kao i kod `enum`. Ovo bi bila naša struktura `Datum`:

```
struct Datum {
  int dan;
  int mjesec;
  int godina;
};
```

Elementi strukture su tri cjelobrojne promjenljive koje smo nazvali dan, mjesec i godina. Mogli smo koristiti i kraću deklaraciju:

```
struct Datum { int dan, mjesec, godina; };
```

Obratite pažnju da se nakon deklarisanja svakog elementa (ili grupe elemenata istog tipa) nalazi znak tačka-zarez, te da se znak nalazi i nakon zatvorene vitičaste zagrade. Pravila za imenovanje struktura i članova struktura su kao i za ostale C identifikatore:↑ naziv se može sastojati od velikih slova, malih slova, cifara i znaka donja crtica (undescore), pri čemu ne može počinjati cifrom. Ipak, mi ćemo kod imenovanja struktura koristiti CamelCase notaciju kod koje svaka riječ počinje velikim i nastavlja se malim slovima, a riječi međusobno nisu razdvojene, dok članove strukture imenujemo malim slovima kao i promjenljive.

↑ Poglavlje 1.2.3, str. 29.

Da bismo shvatili šta se tačno dešava u primjeru, odgovorimo na pitanje: koliko memorije je zauzeto ovom naredbom?

Odgovor glasi: nimalo, jer nije ništa deklarisano. Kada smo naveli definiciju tipa `Datum`, mi smo u biti samo dali opis od čega se jedan datum sastoji, ali još uvijek nismo deklarisali niti jednu promjenljivu tog tipa. Da bismo deklarisali tu promjenljivu, navodimo ponovo ključnu riječ `struct` (nije potrebno u C++u), naziv tipa, te zatim naziv jedne ili više promjenljivih tog tipa, npr:

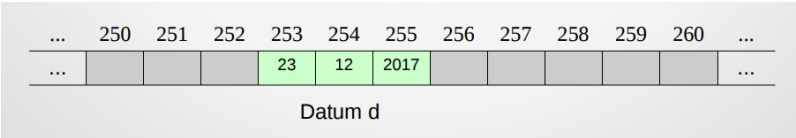
```
struct Datum { int dan, mjesec, godina; };
struct Datum d; // u C++ moze samo: Datum d
```

Promjenljiva `d` sastoji se od tri cjelobrojna člana: `dan`, `mjesec` i `godina`. Pošto je svaki od tih članova tipa `int`, što znači da na PC platformi tipično zauzima 4 bajta, za promjenljivu `d` možemo reći da zauzima 12 bajta memorije. No memorijsko zauzeće strukturne promjenljive nije uvijek jednako zbiru veličina članova strukture.[2] Naime, na PC arhitekturi podaci u memoriji moraju biti *poravnati* (eng. aligned) na veličinu *riječi* (eng. word), koja na PC arhitekturi iznosi 4 bajta. Tako npr. strukturni tip koji se sastoji od promjenljivih tipa `float` i `short` neće zauzimati 6 nego 8 bajta, dok recimo niz od 10 elemenata tipa `char` kao član strukture će zauzeti 12 bajta.

[2] Na poravnanje utiče čak i redoslijed kojim su navedeni članovi strukture u njenoj definiciji. Npr. ako u strukturi imate elemente tipa `char`, `int` i `short`, biće zauzeto 12 bajta jer će svaki od njih biti zasebno poravnat, ali ako promijenite redoslijed u `int`, `short`, `char` struktura će zauzimati 8 bajta.

17157019

Kako to ustvari izgleda u memoriji? Na slici 9.1 možemo vidjeti tri cjelobrojne vrijednosti koje se nalaze uzastopno u memoriji, a sve zajedno čine promjenljivu `d`.



Slika 9.1: Izgled promjenljive `struct Datum d` u memoriji. Ovdje "adrese" predstavljaju apstraktne memorijske lokacije pri čemu svaka promjenljiva zauzima jednu lokaciju.

Kada jednom definišemo strukturu, možemo deklarisati koliko želimo promjenljivih tog tipa. Zbog toga se strukture u pravilu definišu u globalnom opsegu da bi se mogle koristiti u svim funkcijama u programu. Ako bismo posebno definisali strukturu u svakoj funkciji, moglo bi se desiti da se te definicije suptilno razlikuju, i generalno ne želimo da tjeramo programere da provjeravaju da li je definicija na svim mjestima jednaka. Pored toga, funkcije ne mogu primati niti vraćati promjenljive strukturnog tipa ako one nisu definisane u globalnom opsegu. Kada strukturu definišete na samom početku programa (ispod `#include` direktiva) time obavještavate programera da je definicija jedinstvena za program i da je može slobodno koristiti bilo gdje.

Kao i kod pobrojanog tipa, uočavamo da se nakon zatvorene vitičaste zagrade nalazi znak tačka-zarez, što je izuzetak od pravila datog na samom početku knjige. Razlog je to što ključna riječ `struct` omogućuje da odmah nakon definicije strukture deklarišemo i neke promjenljive. Sljedeća linija koda sadrži definiciju strukture `Datum`, koju možete koristiti u svim narednim linijama koda, te odmah deklariše i promjenljivu `d` koja je tipa `Datum`.

```
struct Datum { int dan, mjesec, godina; } d;
```

Česta greška prilikom definisanja struktura je da zaboravimo ovaj posljednji tačka-zarez:

```
struct Datum {
  int dan;
  int mjesec;
  int godina;
}
int x;
```

Za ovaj kod prevodilac će nam prijaviti grešku da riječ `int` nije validan naziv za promjenljivu jer je u pitanju rezervisana riječ. Naime, pošto nakon definicije strukture `Datum` nema tačka-zareza, očekuje se da je sljedeća riječ naziv promjenljive tog tipa, a rekli smo da za programski jezik C nema velike razlike između novih redova i razmaka.

1/5/0198

9.3.2 *Anonimne strukture*

Jedna interesantna osobina je da, ako prilikom definisanja strukture odmah navedemo i promjenljivu tog tipa, ne moramo davati naziv

strukturi. Ovakva struktura se naziva *anonimna struktura*. Npr. možemo pisati:

```
struct { int dan, mjesec, godina; } d;
```

Ovim smo definisali anonimnu strukturu koja se sastoji od tri cjelobrojne promjenljive dan, mjesec i godina, te smo deklarisali promjenljivu tog tipa pod nazivom `d`. Ako bismo željeli kasnije deklarisati strukturnu promjenljivu sa istim elementima, morali bismo ponovo pisati kompletnu definiciju strukture:

```
struct { int dan, mjesec, godina; } d2;
```

Zbog ovoga se anonimne strukture koriste samo kada želimo opisati neki strukturni tip koji se nikada više neće koristiti osim na tom mjestu.

9.3.3 *Imenovanje strukturnog tipa pomoću* `typedef`

U literaturi i na Internetu ćete često vidjeti da se strukturni tipovi imenuju koristeći ključnu riječ **`typedef`** koju smo opisali u poglavlju 9.1. Čak je na mnogim mjestima to i jedini način imenovanja struktura, pa bi oni koji su naučili raditi na taj način mogli zaključiti da je opis dat u ovoj knjizi zbunjujući. Imenovanje strukturnih tipova pomoću **`typedef`** omogućuje nam da u kasnijim deklaracijama promjenljivih ne moramo svaki put ponavljati ključnu riječ **`struct`**.

Rekli smo da se **`typedef`** koristi tako što najprije deklarišemo "promjenljivu" željenog tipa, a zatim ispred dodajemo ključnu riječ **`typedef`**, čime "promjenljiva" postaje ustvari naziv tipa kojeg možemo koristiti u programu. Kod strukturnog tipa "promjenljiva" je ustvari anonimna struktura a naziv "promjenljive" je naziv tipa. Pogledajmo deklaraciju strukturnog tipa `Datum` pomoću **`typedef`**:

```
1 typedef struct {
2   int dan, mjesec, godina;
3 } Datum;
4 Datum d;
```

Najprije je deklarisana anonimna struktura (kao u primjerima iznad) pod imenom `Datum`. Dakle, deklarisana je promjenljiva koja se zove `Datum` a koja je tipa anonimne strukture koja se sastoji od dana, mjeseca i godine. Zatim smo ispred ove deklaracije dodali ključnu riječ **`typedef`** čime je promjenljiva `Datum` ustvari postala novi tip pod imenom `Datum`. Sada se u liniji 4 deklariše promjenljiva `d` tog novog tipa. Obratite pažnju da kod ovakve deklaracije ne moramo ponovo navoditi ključnu riječ **`struct`**.

17570198

9.3.4 *Rad sa strukturnim promjenljivim*

Nakon što smo deklarisali promjenljivu `d` tipa `Datum`, šta možemo raditi s njom? Direktno ne možemo ništa osim međusobno dodjelji-

vati promjenljive, jer je to tip podataka koji nije kao takav poznat programskom jeziku C ni njegovoj standardnoj biblioteci. Međutim, možemo pristupati članovima strukture koji su ustvari obične promjenljive nekog od poznatih tipova, u našem slučaju tipa `int`. Za pristup članovima strukture koristi se *operator tačka*. Način korištenja operatora biće jasniji na primjeru:

```
1 struct Datum {
2   int dan, mjesec, godina;
3 };
4 struct Datum d;
5 d.dan = 21;
6 scanf("%d", &d.mjesec);
7 d.godina = d.dan * 100 - 83;
```

Promjenljiva `d` je strukturnog tipa `Datum`. Pomoću operatora tačka pristupamo članovima strukture navedenim u definiciji strukturnog tipa (linija 2).[3] Dakle, kada smo u liniji 5 napisali `d.dan` dobili smo običnu promjenljivu tipa `int`. Isto važi i za `d.mjesec` i `d.godina` u linijama 6 i 7. Pošto su ovo obične promjenljive tipa `int`, možemo im pridruživati vrijednost (linija 5), unositi ih koristeći `scanf` (linija 6), ili ih koristiti u računskim operacijama (linija 7).

[3] Obratite pažnju na prioritet i asocijativnost operatora tačka date u tabeli u prilogu A (str. 351). Operator tačka ima najviši prioritet, te je jednak prioritetu operatora [] i () (pristup članu niza i poziv funkcije), a svi ovi operatori su lijevo asocijativni.

Koju vrijednost imaju dan, mjesec i godina nakon linije 4 tj. nakon što smo tek deklarisali promjenljivu strukturnog tipa? Kao i sa običnim promjenljivim, članovi strukture *nisu inicijalizirani* dok to izričito ne uradimo kao u liniji 5. Možemo istovremeno inicijalizirati sve članove strukture prilikom deklaracije promjenljive strukturnog tipa na sličan način kao što smo to ranije radili sa nizovima:↑

↑ Poglavlje 5.1.2, str. 134.

```
struct Datum d = {21, 12, 2017};
```

Kao što smo rekli, standardna C biblioteka ne poznaje naš tip `Datum`, tako da ni funkcije `printf` i `scanf` ne znaju direktno raditi sa ovim tipom podataka. Jedino rješenje je da radimo sa članovima strukture pojedinačno:

```
printf("Unesite datum: ");
scanf("%d%d%d", &d.dan, &d.mjesec, &d.godina);
printf("Unijeli ste datum: %d. %d. %d\n", d.dan, d.mjesec, d.godina);
```

Kod složenijih struktura preporučujemo da odmah napravite funkcije za unos i ispis na ekran promjenljivih ovog tipa, što će bitno skratiti vaš program.

9.3.5 *Strukture i funkcije*

17/17/01/98

Pa kad smo već tu, pogledajmo kako bi izgledale te funkcije. Prije svega, napominjemo da se promjenljive strukturnog tipa mogu *slati u funkcije i vraćati iz funkcija* podjednako kao i promjenljive bilo kojeg drugog tipa. Sjetite se da se iz funkcije može vratiti samo jednu vrijednost.↑ Strukture nam omogućuju da zaobiđemo to ograničenje:

↑ Poglavlje 6.2.1, str. 183.

jednostavno definišemo strukturu koja sadrži sve što želimo da vratimo iz funkcije i zatim vratimo promjenljivu tog strukturnog tipa.

```
1  #include <stdio.h>
2  struct Datum {
3    int dan,mjesec,godina;
4  };
5  struct Datum unesi_datum(void) {
6    struct Datum d;
7    printf("Unesite datum: ");
8    scanf("%d%d%d", &d.dan, &d.mjesec, &d.godina);
9    return d;
10 }
11 void ispisi_datum(struct Datum d) {
12   printf("%d. %d. %d", d.dan, d.mjesec, d.godina);
13 }
14 int main() {
15   struct Datum d1 = unesi_datum();
16   struct Datum d2 = unesi_datum();
17   struct Datum d3;
18   /* Racunamo d3... */
19   ispisi_datum(d3);
20   printf("\n");
21   return 0;
22 }
```

Najprije uočavamo da je strukturni tip `Datum` definisan u globalnom opsegu (iznad funkcija), kako bi isti bio dostupan iz svih funkcija. Funkcija `unesi_datum` (linije 5-10) služi za unos sa tastature jedne promjenljive tipa `Datum`. Ona nam skraćuje kod na način da ne moramo svaki put pisati `printf("Unesite datum: ");` te komplikovani `scanf` koji smo vidjeli na prethodnoj stranici. Zbog toga ova funkcija nema parametara (ne prima ništa), a vraća jednu promjenljivu strukturnog tipa.

Svaki put kada navodimo strukturni tip pišemo i ključnu riječ `struct` (što se može izbjeći primjenom `typedef`), dakle funkcija vraća vrijednost tipa `struct Datum`. U funkciji je najprije deklarisana pomoćna promjenljiva `d` (linija 6), elementi strukture su uneseni kao pojedinačne vrijednosti tipa `int` (linija 8), te je ova promjenljiva jednostavno vraćena iz funkcije naredbom `return` (linija 9).

Poziv funkcije možemo vidjeti u linijama 15 i 16 gdje je funkcija pozvana odmah prilikom deklarisanja promjenljivih `d1` i `d2`, te su ove promjenljive inicijalizirane onim što je funkcija vratila.

Program smo također pojednostavili funkcijom `ispisi_datum`, jer ne moramo svaki put kod ispisa datuma pisati kompleksan `printf` koji vidimo u liniji 12. Ova funkcija prima kao parametar datum koji treba ispisati na ekran, a ne vraća ništa jer je sav posao koji ona obavlja ispis na ekran (funkcija je tipa `void`). Primjer poziva te funkcije vidimo u liniji 19.

Zadatak 1. *Napisati program koji omogućuje unos dva datuma, a zatim na ekranu ispisuje broj dana koliko je proteklo između ta dva datuma.*

Već imamo program koji služi za unos dva datuma. Sada trebamo

1757019 8

izračunati traženi broj dana. Računanje razlike između dva datuma je dosta kompleksno jer svaki mjesec ima različit broj dana, a moramo uzeti u obzir i prestupne godine. Prilično jednostavno rješenje je napraviti funkciju koja izračunava broj dana između primljenog datuma i nekog referentnog datuma u prošlosti, a zatim ta dva broja oduzmemo. Kako ne bismo rukovali sa negativnim razlikama, možemo kao referentni datum uzeti 1. 1. 1. godine (mada Gregorijanski kalendar tada nije važio). Pokazuje se da brojevi sa kojim rukujemo neće biti preveliki.

Dakle, napravićemo funkciju `broj_dana` koja prima jedan datum, a vraća broj dana između 1. 1. 1. i tog datuma:

```
int broj_dana(struct Datum d);
```

Najprije ćemo izračunati broj dana koji odgovara godinama, zatim mjesecima, te ćemo na kraju sabrati dane. Godina je prestupna ako je djeljiva sa 4, a nije djeljiva sa 100, osim godina koje su djeljive sa 400. Napravićemo funkciju koja vraća 1 ako je godina prestupna, a 0 ako nije:

```
int prestupna(int godina) {
  if (godina%4 == 0 && godina%100 != 0 || godina%400 == 0)
    return 1;
  return 0;
}
```

Sada ćemo petljom na ukupan broj dana sabrati po 365 dana za svaku godinu, s tim što ćemo za prestupne sabrati još 1:

```
int ukupno=0, i;
for (i=1; i<d.godina; i++)
  ukupno += 365 + prestupna(i);
```

Slijede mjeseci. Za broj dana u mjesecu ne postoji neka jednostavna formula, pa možemo napraviti niz `bdum` (skraćeno od "broj dana u mjesecu") koji za svaki indeks `i` daje broj dana u i-tom mjesecu. Pošto indeksi u nizovima u C-u kreću od nule, deklarisaćemo niz od 13 elemenata pri čemu je prvi element nula.

```
int bdum[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Ali ako je godina prestupna, u februaru treba biti 29 dana.

```
if (prestupna(d.godina)) bdum[2]++;
```

Sada trebamo na ukupan broj dana sabrati dane za mjesece od januara do trenutnog mjeseca, dodati dane datuma i možemo vratiti tako izračunat ukupan broj dana.

```
for (i=1; i<d.mjesec; i++)
  ukupno += bdum[i];
ukupno += d.dan;
return ukupno;
```

Time je završena funkcija `broj_dana`. Ostaje da je pozovemo iz maina koji sada izgleda ovako:

```
int main() {
  struct Datum d1 = unesi_datum();
  struct Datum d2 = unesi_datum();
  int razlika = broj_dana(d2) - broj_dana(d1);
  printf("Izmedju ovih datuma je proteklo %d dana.\n", razlika);
  return 0;
}
```

Ovaj program će davati negativn broj dana ako je drugi datum prije prvog, što možemo riješiti koristeći funkciju `abs` da odredimo apsolutnu vrijednost razlike. Za dva susjedna datuma ćemo dobiti rezultat 1. Postavka zadatka nije precizno definisala da li "broj dana između dva datuma" uključuje spomenuta dva datuma ili ne, tako da smo usvojili uobičajenu konvenciju da je uključen jedan od dva datuma. Npr. ako računamo broj dana između 10. januara i 12. januara uobičajen odgovor je "dva dana" pri čemu smo uračunali 12. a nismo uračunali 10.

9.3.6 Strukture i nizovi, podstrukture

Članovi strukture mogu biti promjenljive svih poznatih tipova, pa tako i nizovi. Pogledajmo sljedeći primjer:

```
1 struct Student {
2   char ime[20];
3   char prezime[20];
4   int broj_indeksa;
5 } s;
```

Definisana je struktura `Student`, te je odmah deklarisana i promjenljiva `s` tog tipa. Elementi strukture su ime, prezime i broj indeksa. Kao što znamo, string u C-u se realizuje kao niz tipa `char`, pa je element strukture `ime` niz tipa `char` od 20 elemenata.

Kada iskoristimo *operator tačka*, dobijamo član strukture koji je kao obična promjenljiva onog tipa koji je naveden u definiciji. Dakle, kada napišemo `s.ime`, u ovom slučaju će to biti običan niz tipa `char` s kojim radimo kao i sa bilo kojim nizom tog tipa. Sljedeća naredba će na ekran ispisati ime studenta `s`:

```
printf("%s", s.ime);
```

Ako želimo da pristupimo članu niza, npr. prvom slovu imena, koristimo operator uglastih zagrada nakon imena niza, a kako je `s.ime` u cjelosti ime niza, zagrade se stavljaju nakon toga:

```
printf("%c", s.ime[0]); /* Prvo slovo imena */
```

U poznate tipove spadaju i neki drugi strukturni tipovi, pod uslovom da su definisani iznad tekuće strukture. Drugim riječima, član strukture može biti druga struktura. Takav slučaj možemo vidjeti na sljedećem primjeru:

17577/0198

```
1 struct Datum { int dan,mjesec,godina; };
2 struct Student {
3   char ime[20];
4   char prezime[20];
5   int broj_indeksa;
6   struct Datum datum_rodjenja;
7 } s;
```

U liniji 1 definisana je struktura `Datum`, a zatim slijedi definicija strukture `Student`. Jedan od članova ove strukture je i `datum_rodjenja` koji je tipa `struct Datum` (linija 6). Konačno, deklarisan je jedan `Student` pod nazivom `s` (linija 7).

Kada u našem programu napišemo `s.datum_rodjenja`, ovaj izraz se ponaša kao promjenljiva tipa `struct Datum`. Šta možemo raditi sa promjenljivom tog tipa? Ništa direktno (osim dodjele), ali možemo pristupiti njenim članovima koristeći operator tačka. Ovaj operator se navodi desno od naziva strukturne promjenljive, nakon čega navodimo član strukture `Datum`. Dakle, možemo pisati `s.datum_rodjenja.dan` čime pristupamo elementu `dan` strukturne promjenljive `datum_rodjenja` koji je obična promjenljiva tipa `int`:

```
printf("Unesite datum rodjenja studenta: ");
scanf("%d", &s.datum_rodjenja.dan);
```

Moguće je da se struktura nalazi unutar druge strukture, koja se nalazi unutar treće strukture i tako do u beskonačnost. Jednostavno koristimo operator tačka kako bismo se "spustili jedan nivo" u ovoj hijerarhiji struktura. Jedna česta greška je da se kod pristupa članu strukture navodi tip člana – nije ispravno pisati `s.datum_rodjenja.int` ili `s.Datum.dan` u ovom primjeru, jer su `int` i `Datum` tipovi.

Interesantna osobina strukturnih promjenljivih je da ih možemo sigurno međusobno dodjeljivati, za razliku od recimo nizova, i svi članovi strukture će se korektno kopirati – pa čak i nizovi! Npr. znamo da ako imamo dva stringa koji se zovu `ime1` i `ime2`, bilo bi neispravno pisati `ime1=ime2`. Također, ako imamo dva studenta `s1` i `s2`, nije ispravno pisati `s1.ime=s2.ime` iz istog razloga. Ali sasvim je ispravno pisati `s1=s2` čime će se kopirati između ostalog i ime.[4] No, promjenljive strukturnog tipa se ipak ne mogu međusobno upoređivati koristeći operator ==.

[4] Ako je član strukture pokazivač, kôd tipa `s1=s2` će kopirati adresu na koju pokazivač pokazuje, što u vašem konkretnom programu može biti "plitka kopija".

Zadatak 2. *Pomoću struktura definisati osnovne geometrijske oblike u Dekartovom dvodimenzionalnom prostoru. Zatim napraviti program koji omogućuje osnovne operacije nad ovim oblicima.*

1/15/0198

Ovaj program može biti proizvoljno obiman, tako da ćemo ovdje samo dati neke osnovne principe rješavanja. Prije svega potrebno je da definišemo tačku u prostoru, a zatim preko tačke možemo definisati ostale oblike. Tačka je opisana preko koordinata x i y:

```
struct Tacka {
  double x, y;
};
```

Za koordinate naravno koristimo realne brojeve. Jedna od korisnih funkcija koja će nam zatrebati je udaljenost između tačaka. Funkcija prima dvije tačke, a vraća realan broj koji predstavlja udaljenost.

```
double udaljenost(struct Tacka t1, struct Tacka t2) {
  return sqrt( (t1.x-t2.x)*(t1.x-t2.x) + (t1.y-t2.y)*(t1.y-t2.y) );
}
```

Za funkciju `sqrt` treba nam biblioteka `math.h`.

Sada možemo definisati kružnicu preko tačke koja predstavlja centar kružnice i poluprečnika.

```
struct Kruznica {
  struct Tacka centar;
  double poluprecnik;
};
```

Kvadrat možemo definisati preko jedne tačke (npr. donjeg lijevog ugla) i dužine stranice. Ovako definisan kvadrat ima sve stranice paralelne osama koordinatnog sistema.

```
struct Kvadrat {
  struct Tacka dolje_lijevo;
  double stranica;
};
```

Pravougaonik možemo definisati na više načina. Jedan je da zadamo dvije dijagonalne tačke (npr. donji lijevi ugao i gornji desni), a drugi je preko jedne tačke i dvije dužine stranica. Ovo je opet pravougaonik čije su stranice paralelne osama. Pravougaonik i Kvadrat čije stranice nisu paralelne osama bi bilo nešto teže zadati. Jedan način je da u strukturu dodamo i ugao rotacije oblika (u radijanima) u odnosu na polazno stanje.

Pogledajmo i neke funkcije. Recimo možemo napraviti funkciju koja prima kružnicu i tačku, te provjerava da li se tačka nalazi unutar kružnice:

```
int unutar_kruznice(struct Kruznica k, struct Tacka t) {
  if (udaljenost(t, k.centar) < k.poluprecnik) return 1;
  return 0;
}
```

Ako bismo željeli provjeriti da li se tačka nalazi *na kružnici*, morali bismo uzeti u obzir nepreciznost realnih brojeva↑ jer funkcija `sqrt` može vratiti brojeve sa dosta velikim brojem decimala.

↑ Poglavlje 3.1.6, str. 72.

```
int na_kruznici(struct Kruznica k, struct Tacka t) {
  if (udaljenost(t, k.centar) - k.poluprecnik < EPSILON) return 1;
  return 0;
}
```

Provjera da li se dvije kružnice preklapaju bi izgledala ovako:

```
int kruznice_preklapanje(struct Kruznica k1, struct Kruznica k2) {
  if (udaljenost(k1.centar, k2.centar) < k1.poluprecnik + k2.poluprecnik)
   return 1;
  return 0;
}
```

9.3.7 *Nizovi struktura*

Vratimo se na nizove. Možemo deklarisati niz vrijednosti bilo kojeg poznatog tipa, pa tako i niz promjenljivih strukturnog tipa. Za primjer posmatraćemo sljedeću deklaraciju:

```
1 struct Datum { int dan,mjesec,godina; };
2 struct Student {
3   char ime[20];
4   char prezime[20];
5   int broj_indeksa;
6   struct Datum datum_rodjenja;
7 };
8 struct Student studenti[100];
```

U liniji 8 deklarisan je niz od 100 vrijednosti čije je ime `studenti`, a tip svakog člana niza je `struct Student`. Kako radimo sa ovakvim nizom? Kao i kod svakog niza, najprije trebamo pristupiti nekom članu niza koristeći uglaste zagrade. Recimo, prvi član ovog niza bi bio `studenti[0]`. Ovo je sada promjenljiva tipa `struct Student` sa kojom radimo kao i sa bilo kojom promjenljivom tog tipa. Recimo, ako bismo željeli na ekran ispisati ime studenta koji je prvi član niza, mogli bismo pisati:

```
printf("%s", studenti[0].ime);
```

Obratite pažnju da je kompletan kod `studenti[0]` jedna promjenljiva tipa `struct Student`, pa na tu promjenljivu primjenjujemo operator tačka koji se navodi desno od nje (nakon uglastih zagrada). Zatim desno od tačke navodimo član strukture `ime`. Sada je `studenti[0].ime` promjenljiva tipa string (niz tipa `char`), pa je možemo ispisati koristeći format `%s`. Ako bismo željeli ispisati prvo slovo imena, pisali bismo `student[0].ime[0]`. Ako bismo željeli pomoću tastature unijeti mjesec rođenja sedmog studenta, kod bi glasio:

```
scanf("%d", &studenti[6].datum_rodjenja.mjesec);
```

Zadatak 3. *Napisati funkciju "stariji_od" koja prima niz studenata i neki datum, te na ekran ispisuje imena i prezimena studenata (u zasebnim redovima) koji su rođeni prije primljenog datuma a broj indeksa im je paran broj.*

17570198

Najprije ćemo napraviti jednu pomoćnu funkciju koja poredi dva datuma, te vraća 1 ako je prvi datum prije drugog, a 0 ako nije.

```
1 int datum_prije(struct Datum d1, struct Datum d2) {
2   if (d1.godina < d2.godina) return 1;
```

```
3     if (d1.godina == d2.godina) {
4       if (d1.mjesec < d2.mjesec) return 1;
5       if (d1.mjesec == d2.mjesec && d1.dan < d2.dan) return 1;
6     }
7     return 0;
8   }
```

Sada nam je lako napisati funkciju `stariji_od`:

```
1  void stariji_od(struct Student* studenti, int vel, struct Datum d) {
2    int i;
3    for (i=0; i<vel; i++) {
4      if (datum_prije(studenti[i].datum_rodjenja, d) &&
5        studenti[i].broj_indeksa % 2 == 0)
6          printf("%s %s\n", studenti[i].ime, studenti[i].prezime);
7    }
8  }
```

Funkcija je tipa `void` jer ne vraća ništa, posao koji obavlja je ispis imena i prezimena na ekran. Funkcija prima niz studenata, što znači da su joj parametri pokazivač na prvi član niza i broj članova niza, a treći parametar je datum `d` sa kojim upoređujemo datume rođenja studenata.

U funkciji ćemo napraviti petlju koja prolazi kroz sve članove niza, i-ti član niza je `studenti[i]`, a tipa je `struct Student`. Sada možemo pozvati funkciju `datum_prije` da provjerimo da li je datum rođenja prije datog. Kao prvi parametar funkcije poslaćemo datum rođenja i-tog studenta: `studenti[i].datum_rodjenja`.

Funkcija vraća logičku vrijednost tako da možemo odmah preći i na drugi uslov koji treba provjeriti, a to je da li je broj indeksa paran. Broj indeksa i-tog studenta je `studenti[i].broj_indeksa` pa uzimamo modulo ove vrijednosti sa 2. Konačno, u liniji 6 ispisujemo ime i prezime studenta ako je uslov ispunjen.

9.3.8 Pokazivači na strukture

Možemo deklarisati pokazivač pomoću kojeg pokazujemo na vrijednost bilo kojeg od poznatih tipova. Tako bi deklaracija pokazivača na vrijednost tipa `struct Student` glasila:

```
struct Student* p;
```

Znak zvjezdica se navodi nakon tipa a prije naziva promjenljive.

Podsjetimo se da ime niza predstavlja adresu prvog člana niza, pa tako ovaj pokazivač p sada može pokazivati na prvi član ranije deklarisanog niza studenti:

```
struct Student* p = studenti;
```

1755/0198

Da bismo pristupili nekom članu strukture preko pokazivača, najprije moramo dereferencirati pokazivač. Podsjetimo se da se to radi primjenom operatora dereferenciranja (zvjezdica): `*p`. Ako bismo pisali recimo `*p.broj_indeksa` ne bismo dobili željeni rezultat jer operator tačka ima veći prioritet od operatora zvjezdica, pa bi ovo bilo isto što

i `*(p.broj_indeksa)`. Dakle prevodilac će `p` posmatrati kao neku strukturnu promjenljivu čiji je član pokazivač `broj_indeksa`, a zatim će pokušati da dereferencira ovaj pokazivač. Umjesto toga trebamo pisati ovako:

```
(*p).datum_rodjenja.dan = 23;
```

Pošto je pokazivač na strukturu vrlo česta pojava u programiranju, osmišljen je poseban *operator strelica* koji istovremeno dereferencira pokazivač i pristupa članu strukture, a piše se kao znak minus nakon kojeg slijedi znak veće-od. Ovaj operator ujedno ima vrlo visok prioritet, kao i tačka. Tako je prethodni kod ekvivalentan sa sljedećim:

```
p->datum_rodjenja.dan = 23;
```

Obratite pažnju da strelicu stavljamo nakon pokazivača `p`, ali nakon člana strukture `p->datum_rodjenja` se nalazi tačka jer to nije pokazivač nego obična strukturna promjenljiva.

Zadatak 4. *Ranije napisanu funkciju "stariji_od" prepisati koristeći pokazivačku aritmetiku.*

```
1 void stariji_od(struct Student* studenti, int vel, struct Datum d) {
2   struct Student* p = studenti;
3   while (p < studenti+vel) {
4     if (datum_prije(p->datum_rodjenja, d) && p->broj_indeksa % 2 == 0)
5       printf("%s %s\n", p->ime, p->prezime);
6     p++;
7   }
8 }
```

Pokazivač na strukturu može i sam biti član strukture, pa se na taj način realizuju dinamičke strukture kao što su liste i stabla.

9.3.9 *Bitpolja*

Strukture nam daju još jednu interesantnu mogućnost, a to je da definišemo cjelobrojne tipove promjenljivih čija je veličina (u bitima) proizvoljna. Jedini uslov je da takva promjenljiva bude član strukture. Recimo, na PC platformi tip `int` se sastoji od 32 bita (4 bajta), `short` ima 16 bita, a `char` 8 bita. Ali šta ako želimo cjelobrojnu promjenljivu od recimo 12 bita? To možemo postići sljedećom definicijom:

```
struct NekiTip {
  int x:12;
};
```

Broj iza dvotačke označava broj bita uključujući predznak koje će imati cjelobrojni član strukture `x`. Ako ne želimo predznak, možemo pisati `unsigned int`.

Ova funkcionalnost se naziva **bitpolja** (eng. bitfields).

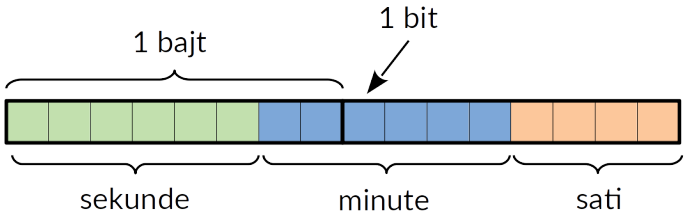
Jedna od primjena bitpolja je da kreiramo strukture koje zauzimaju vrlo malu količinu memorije, a sadrže veći broj elemenata koji su "up-

1/15/01/98

akovani" (eng. packed) u manji prostor. Pogledajmo definiciju strukture `Vrijeme`:

```
struct Vrijeme {
  unsigned int sati:4, minute:6, sekunde:6;
};
```

Struktura `Vrijeme` se sastoji od tri cjelobrojne vrijednosti: sati, minuta i sekundi. Da smo koristili obične vrijednosti tipa `int`, svaka promjenljiva tipa `Vrijeme` zauzimala bi 12 bajta odnosno 96 bita. Koristeći bitpolja čini se da smo postigli da promjenljiva zauzima svega 4+6+6=16 bita odnosno dva bajta. Naime, znamo da su sati u opsegu 0-11 što znači da su nam dovoljna 4 bita (vrijednosti 0-15), dok za minute i sekunde koristimo po 6 bita (opseg 0-63) (sl. 9.2).



Slika 9.2: Organizacija strukture Vrijeme u memoriji

No, ipak to neće biti slučaj. Pored ranije spominjanog poravnanja,↑ za bitpolja važi da strukturni tip ne može zauzimati manje memorije od osnovnog tipa, a to je u ovom slučaju `unsigned int` koji zauzima 4 bajta, pa trebamo prepisati strukturu tako da se koristi tip `unsigned char`:

↑ Str. 304.

```
struct Vrijeme {
  unsigned char sati:4, minute:6, sekunde:6;
};
```

Uočavamo da je ipak potrebno sate predstavljati u opsegu 0-23, pa ćemo promijeniti definiciju strukture tako da koristimo 5 bita za sate. Da li sada jedna promjenljiva tipa Vrijeme zauzima 17 bita? Ipak ne, takva promjenljiva bi zauzimala 24 bita jer nije moguće direktno raditi sa memorijskim blokovima manjim od jednog bajta tj. osam bita. Drugim riječima, memorijski zahtjevi jedne strukturne promjenljive će biti zaokruženi na cijeli broj bajta. Neko bi mogao uočiti da isto zauzeće od 3 bajta (24 bita) postižemo i bez bit-polja, koristeći tri obične vrijednosti tipa `unsigned char`.

Obratite pažnju da su operacije nad bitpoljima u pravilu sporije od operacija nad običnim strukturnim promjenljivim jer ih procesor računara direktno ne podržava, pa će prevodilac generisati mašinski kod koji svaki put kada radite sa ovakvom promjenljivom vrši matematičke operacije nad bitima (shift, and, or) kako bi bio postignut očekivani rezultat.

1715701198

9.4 *Unija (`union`)*

Da bismo zaokružili priču o vlastitim tipovima podataka, upoznaćemo se sa još jednom mogućnošću C-a koja se rijetko koristi, a prividno je vrlo slična strukturama. U pitanju je **unija**.

Deklaracija unije je vrlo slična deklaraciji strukture, osim što se koristi ključna riječ `union`:

```
union Unija {
  int a, b, c;
} u;
u.a = 15;
u.b = 32;
printf("%d", u.a);
```

Na prvi pogled unija se ponaša kao i struktura uz jednu vrlo bitnu razliku: svi članovi unije zauzimaju istu memorijsku lokaciju! To znači da će prethodni program ispisati broj 32, jer članovi strukture `u.a` i `u.b` zauzimaju iste memorijske lokacije tako da kada promijenimo jedan, promijenili smo i drugi.

U prethodnom primjeru promjenljiva `u` zauzima 4 bajta (kao i jedna promjenljiva tipa `int`) jer je zauzeta samo jedna memorijska lokacija a ne tri.

Članovi unije ne moraju biti istog tipa. Ako tipovi članova unije imaju različitu veličinu, sama unija će imati veličinu najvećeg tipa. Npr. u sljedećem primjeru:

```
union Unija {
  char x;
  double y;
  float z;
} u;
```

najveći korišteni tip je double (8 bajta), pa će promjenljiva `u` zauzimati 8 bajta.

Unija se koristi kada želimo da iste podatke posmatramo na više različitih načina. Pogledajmo ovaj primjer:

```
union Unija {
  int i;
  float f;
  unsigned char c[4];
} u;
u.f = 0.1;
printf("%d", u.i);
```

Sva tri člana unije imaju istu dužinu (4 bajta), te zauzimaju istu memorijsku lokaciju. Kada pristupimo članu `u.f` i dodijelimo mu vrijednost 0.1, mi ovu memorijsku lokaciju posmatramo kao realnu promjenljivu kodiranu IEEE-754 standardom. S druge strane, kada joj pristupimo preko člana `u.i` mi isti binarni zapis tumačimo kao predznačnu cijelu vrijednost. Ovu vrijednost možemo dodatno razložiti kao četiri osmobitna cijela broja (4 člana niza tipa `unsigned char`).

17457018 98

Unije olakšavaju direktnu manipulaciju nad manjim jedinicama podataka (npr. biti) od kojih se neki veći podatak sastoji, što je čest slučaj kod razvoja ugradbenih sistema, mrežnih protokola i slično. Pored toga, unije se koriste i za uštedu prostora kada se neki članovi strukture ne koriste uvijek. Npr. u strukturi `Uposlenik` želimo da držimo iznos plate ako je radnik trenutno zaposlen, a datum penzionisanja ako je u penziji. Mogli bismo pisati ovako:

```
struct Uposlenik {
  char ime[20], prezime[20];
  int u_penziji;
  union {
    float plata;
    struct Datum datum_penzionisanja;
  } d;
};
```

Na ovaj način smo uštedili 4 bajta što na velikim nizovima može biti značajno.

Pitanja za razumijevanje gradiva

1. Čemu služi ključna riječ `typedef`?

2. Šta će na ekranu ispisati sljedeći kod?

```
enum Boja { Crvena, Plava, Bijela, Crna, Smedja };
printf("%d", Crna);
```

3. Na koji način možemo na ekranu ispisati ime boje `b`, ako je pobrojani tip `Boja` definisan kao u prethodnom primjeru?

4. Koliko memorije zauzima strukturna promjenljiva `s` data na sljedeći način:

```
struct Datum { int dan, mjesec, godina; }
struct Student {
  char ime[20], prezime[20];
  int broj_indexa;
  struct Datum datum_rodjenja;
} s;
```

5. Koje su prednosti imenovanja strukturnog tipa pomoću ključne riječi `typedef`?

6. Na koje sve načine možemo vratiti više od jedne vrijednosti (npr. tri cijela broja) iz funkcije?

7. Kojom naredbom unosimo godinu rođenja studenta `s` iz primjera iznad?

17570198

8. Šta se u programskom jeziku C označava znakovima minus i veće-od: `->` ?

9. Šta su to bitpolja i zašto se koriste?

10. Šta je to unija?

Zadaci za vježbu

Zadatak 5. *U primjeru sa geometrijskim oblicima datom na str. 311 dodati funkcije za računanje obima i površine kružnice.*

Zadatak 6. *U istom primjeru dodati novu strukturu* `struct Pravougaonik` *koja opisuje pravougaonik čije su stranice paralelne koordinatnim osama. Zatim dodati funkcije za obim i površinu pravougaonika, te funkciju koja provjerava da li se tačka nalazi u pravougaoniku.*

Zadatak 7. *Data je struktura* `struct Vrijeme` *koja sadrži tri cjelobrojna člana:* `sati`, `minute` *i* `sekunde`*. Napišite funkciju "unos_vremena" koja omogućuje unos vrijednosti tipa* `struct Vrijeme`*, a zatim funkciju "proteklo" koja određuje koliko sekundi je proteklo između dva vremena koja prima kao parametre v1 i v2. Obratite pažnju da ako je v1 ispred ili iza v2 funkcija treba u oba slučaja vratiti pozitivan broj sekundi. U mainu treba omogućiti unos dva vremena, a zatim ispisati koliko sekundi je proteklo između njih koristeći navedene dvije funkcije.*

Zadatak 8. *Prepravite prethodni program tako da određuje koliko sati, minuta i sekundi je proteklo između dva vremena. Funkcija proteklo treba vraćati* `struct Vrijeme`*.*

Zadatak 9. *Data je struktura podataka* `struct Osoba`*. Napravite program koji sadrži niz od 100 osoba te omogućuje unos osobe ili ispis do sada unesenih osoba.*

```
struct Osoba {
  char ime[15];
  char prezime[20];
  int telefon;
}
```

17570198

1757018 8

10
Rad sa datotekama

U narednom poglavlju upoznaćemo se sa funkcionalnostima programskog jezika C koje nam omogućuju rad sa datotekama. No prije toga, potrebno je da objasnimo šta su to uopšte datoteke, čemu služe i kako se radi s njima.

10.1 O datotekama

Memoriju računara razvrstavamo u *internu i eksternu memoriju*. Interna memorija računara je znatno brža od eksterne, ali joj je potrebno neprekidno napajanje električnom energijom, u suprotnom se njen sadržaj briše. Za trajnu pohranu podataka koristimo uređaje eksterne memorije kao što su hard disk, SSD, USB, optički uređaji (DVD) i drugi, koji su osim toga u pravilu za nekoliko redova veličine jeftiniji od interne memorije.

Kada u našem programu deklarišemo promjenljive (nizove, strukture itd.), one se smještaju u internu memoriju računara (RAM). Po završetku rada programa sve promjenljive se brišu, uključujući i globalne. Naravno naš program bi mogao uvijek biti pokrenut, ali šta ako je potrebno uraditi restart računara ili ako nestane električnog napajanja (potroši se baterija)? Moramo imati neki način da trajno zapišemo podatke kako oni ne bi bili zaboravljeni između dva pokretanja. U te svrhe koriste se **datoteke** ili **fajlovi** (eng. *file* što u prevodu znači spis, dosje).

10.1.1 Datotečni sistem

17570198

Računar može imati jednu ili više jedinica eksterne memorije. Npr. u vašem laptopu možda imate više hard diskova, ali čak i ako imate samo jedan disk on može biti podijeljen na logičke jedinice koje se nazivaju *particije*, a koje će operativni sistem vidjeti kao zasebne diskove.

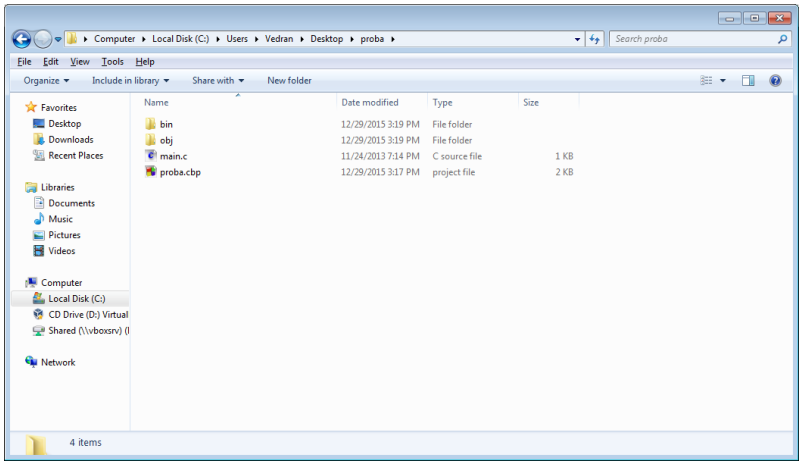
Operativni sistem Windows označava jedinice eksterne memorije

slovima engleske abecede, i to abecednim redom, pri čemu se po defaultu počinje od slova C,[1] a nakon slova se stavlja znak dvotačka. Tako će prva tj. primarna jedinica eksterne memorije (prvi hard disk ili particija na koju je instaliran sam operativni sistem) biti označena sa C:, sljedeća će biti označena sa D:, pa E: i tako dalje. Na operativnom sistemu Linux ove slovne oznake ne postoje, nego su različite jedinice "montirane" pod određenim lokacijama (što ćemo objasniti kasnije).

[1] Slova A i B su i danas po defaultu rezervisana za zaboravljene floppy diskove, mada postoji način da ih iskoristimo.

Na jednoj jedinici nalazi se veći broj datoteka, koje su organizovane u *direktorije* odnosno *foldere* (eng. *folder* u prevodu znači fascikla). Naime, kako je rasla veličina diskova, dešavalo se da se na jednom disku nalazi više stotina datoteka, što je postajalo nepregledno. Direktoriji (folderi) nam omogućuju da datoteke organizujemo u pregledne kategorije, baš kao što fascikle i registratori pomažu kod organizacije velikog broja papira.

Sama jedinica predstavlja početni ili *korijenski direktorij* (eng. root). Unutar svakog direktorija mogu se nalaziti datoteke i drugi direktoriji, tako da ovaj sistem direktorija čini *stablo direktorija*.



Slika 10.1: Windows Explorer – standardni upravitelj datotekama na operativnom sistemu Windows

Upravitelj datotekama (eng. file manager) je program pomoću kojeg se možete kretati kroz ovo stablo i pristupiti svakom direktoriju i datotekama koje se u njemu nalaze. Na operativnom sistemu Windows standardni upravitelj je Windows Explorer kojeg možete vidjeti na slici 10.1. Prikazan je direktorij *proba* unutar kojeg se nalaze direktoriji *bin* i *obj* te datoteke *main.c* i *proba.cbp*. S lijeve strane možemo vidjeti nekoliko različitih pogleda na stablo koji nam olakšavaju obavljanje različitih zadataka sa računarom.

17570198

10.1.2 Putanja

U dva različita direktorija mogu se nalaziti dvije datoteke pod istim imenom. Npr. ako imate direktorij pod imenom *Prvi* i direktorij *Drugi*,

u oba se može nalaziti datoteka koja se zove *napomena.txt* i to su dvije različite datoteke. Dakle, potrebno je da na neki način program koji radi s datotekom razlikuje ove dvije datoteke.

✓ *Putanja ili put (eng. path) je string koji daje ne samo naziv datoteke nego i njenu lokaciju u stablu direktorija.*

✓ Putanja

Nazivi direktorija i datoteka međusobno se razdvajaju *separatorom*. Na operativnom sistemu Windows separator je backslash, a na Linuxu slash (kosa crtica). Tako bismo u našem primjeru imali datoteke *Prvi\napomena.txt* i *Drugi\napomena.txt* kao dvije različite datoteke.

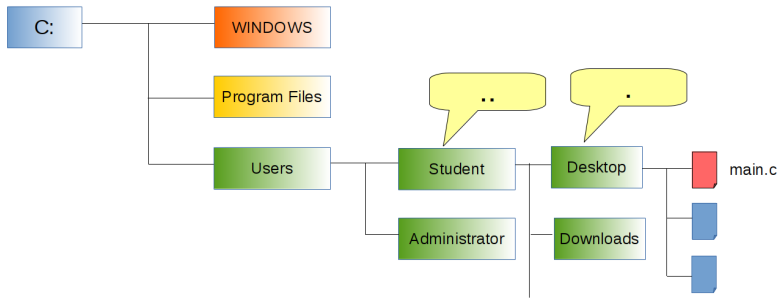
Ako navedete samo ime datoteke *napomena.txt* bez putanje, kojoj datoteci ćete ustvari pristupiti? To ovisi od toga koji direktorij je trenutno označen kao *tekući direktorij*. Npr. ako je tekući direktorij *Prvi*, pristupićete datoteci *Prvi\napomena.txt*. U svakom trenutku neki direktorij se za datu aplikaciju smatra za tekući. Postoje naredbe pomoću kojih mijenjate tekući direktorij i na taj način se "krećete" kroz stablo direktorija.

✓ *Apsolutna putanja je vrsta putanje koja kreće od samog korijenskog direktorija, a na Windowsu uključuje i slovnu oznaku jedinice, tako da ona jednoznačno imenuje konkretnu datoteku na sistemu. Primjer apsolutne putanje bi bio: "C:\Program Files\CodeBlocks\bin\gcc.exe".*

✓ Apsolutna putanja

Kada koristite apsolutnu putanju, nije bitno koji direktorij je tekući. Apsolutna putanja na Linuxu počinje znakom slash a na Windowsu slovnom oznakom jedinice nakon koje slijedi backslash.

Postoji također i *relativna putanja*. Primjer relativne putanje bi bio *"Drugi\napomena.txt"*, jer ova putanja ne počinje znakom slash, niti slovnom oznakom jedinice, a sadrži separator (backslash). Kod relativne putanje smatra se da se prvi direktorij naveden u putanji nalazi ispod tekućeg. Npr. ako je tekući direktorij *C:\Users\Student*, a relativna putanja je *Drugi\napomena.txt*, pristupićete datoteci čija je apsolutna putanja *C:\Users\Student\Drugi\napomena.txt*.



Slika 10.2: Značenje specijalnih oznaka "." i ".."

1.5.0.98

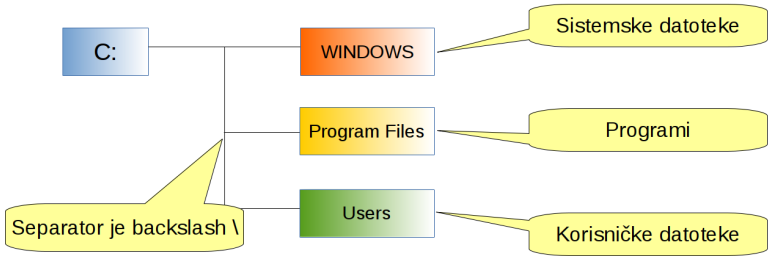
Kod relativnih putanja postoje dva specijalna, rezervisana naziva za direktorije, a to su "." (jedna tačka) i ".." (dvije tačke – ali ne

dvotačka!) Direktorij "." označava tekući direktorij, a ".." označava direktorij koji je u hijerarhiji iznad tekućeg, tj. njegov roditelj. Npr. ako je tekući direktorij *C:\Users\Student\Desktop*, putanja ".." bi označavala direktorij *C:\Users\Student* (slika 10.2). Koristeći ovu oznaku možete praviti relativne putanje kao što je *..\..\dir\file.txt*. Kojoj apsolutnoj putanji bi ona odgovarala?

10.1.3 Standardna hijerarhija i prava pristupa

Za svaki operativni sistem definisani su određeni obavezni (sistemski) direktoriji i datoteke koji moraju postojati na sistemu, a pored toga vi možete kreirati i vlastite, ali sistemske direktorije i datoteke ne biste smjeli modificirati ni brisati.

Standardna hijerarhija operativnog sistema Windows skicirana je na slici 10.3. Windows je u pravilu instaliran na disku C:. Ispod ovog korijenskog direktorija nalaze se standardni direktoriji "WINDOWS", "Program Files" i "Users". *C:\WINDOWS* sadrži datoteke koje pripadaju samom operativnom sistemu i ne biste ih trebali mijenjati. Ispod *C:\Program Files* instaliraju se programi, a *C:\Users* je prostor koji je dat na raspolaganje korisnicima. Unutar *C:\Users* pronaći ćete po jedan direktorij za svakog korisnika registrovanog na sistemu. Unutar vašeg korisničkog direktorija nalazi se vaš desktop, dokumenti, konfiguracija programa koje koristite itd.

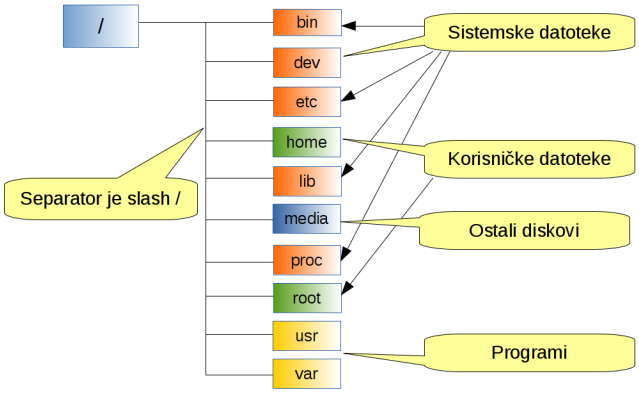


Slika 10.3: Standardna hijerarhija direktorija operativnog sistema Windows

Za svaku datoteku ili direktorij definisana su *prava pristupa*. Običan korisnik ne može mijenjati sistemske datoteke ali ih većinom može čitati, dok administrator sistema u pravilu može sve. Tako i za bilo koju proizvoljnu datoteku ili direktorij možete definisati ko sve ih smije mijenjati, ko ih može samo čitati, a ko ne može ni to. U pravilu predviđeno je da korisnici drže sve svoje datoteke ispod svog korisničkog direktorija *C:\Users\korisnik*, ali na Windowsu ovo pravilo nije strogo provedeno.

17570198

Na operativnom sistemu Linux stvari stoje nešto drugačije (slika 10.4). Prije svega, ne postoje slovne oznake jedinica. Umjesto toga, svakoj jedinici možemo pridružiti određeni direktorij na sistemu ("montirati" je na toj lokaciji). Sve datoteke i direktoriji koji se nalaze ispod te



Slika 10.4: Standardna hijerarhija direktorija operativnog sistema Linux

lokacije nalaze se fizički na odgovarajućoj jedinici. Npr. možemo reći da se korijenski sistem nalazi na prvom disku, a direktorij */home* i sve ispod njega na drugom.

Direktoriji */bin* i */lib* sadrže programske komponente operativnog sistema, */etc* sadrži konfiguraciju, a */dev* i */proc* sadrže specijalne datoteke koje odgovaraju uređajima na sistemu i sistemskim procesima. Administrator sistema naziva se *root* te je njemu pridružen istoimeni direktorij, dok ostali korisnici imaju korisničke direktorije ispod */home*. Ako koristite povremeno neke druge jedinice eksterne memorije (USB ili optičke diskove), oni će se nalaziti ispod */media*.

Za razliku od Windowsa, na Linuxu su striktno primijenjena prava pristupa, tako da običan korisnik ne može pisati datoteke skoro nigdje osim u svom direktoriju */home/korisnik*.

10.2 *Rad sa datotekama u C-u*

Nakon što smo ukratko naučili šta su datoteke i direktoriji, te kako je organizovana njihova hijerarhija na operativnim sistemima Windows i Linux, preći ćemo na osnovne funkcionalnosti koje nam u tom smislu nudi programski jezik C.

10.2.1 *Otvaranje i zatvaranje datoteke*

Prije početka rada sa datotekom ona se mora *otvoriti*, što se u C-u postiže koristeći funkciju `fopen`. Također po završetku rada datoteku treba *zatvoriti*, što obavlja funkcija `fclose`. Sve funkcije za rad sa datotekama nalaze se u biblioteci `stdio.h` koju smo i ranije koristili.

Otvaranje datoteke postiže nekoliko ciljeva. Najprije, na ovaj način provjeravamo da li imamo pravo čitati iz datoteke ili pisati u nju, da li ona postoji, da li je putanja koju smo zadali validna itd. Zatim, operativni sistem se obavještava da neki program radi sa datotekom. Na

1/10/2018

pojedinim operativnim sistemima nije dozvoljeno da više programa istovremeno piše u istu datoteku. Kod uređaja koji se mogu dodavati i uklanjati sa sistema dok sistem radi (kao što je USB uređaj), sistem pokušava da spriječi gubitak podataka tako što ne dozvoljava da se uređaj ukloni dok svi programi ne završe s radom, o čemu nas obavještava čuvena poruka "Sada je sigurno ukloniti uređaj" (eng. "safe to remove").

Prototip funkcije fopen glasi:

```
FILE* fopen(const char* putanja, const char* mod);
```

Prvi parametar je string koji sadrži putanju do datoteke, a pošto je korištena ključna riječ `const` možemo prosljeđivati i literale. Kod navođenja putanje kao literala na operativnom sistemu Windows moramo obratiti pažnju da znak backslash predstavlja "escape znak" tj. početak escape sekvence. Npr. ako biste u kodu naveli

```
fopen("C:\novi folder\nastavnici...", "r");
```

svaki par znakova `\n` bi bio protumačen kao oznaka za novi red. Podsjetimo se da kada želimo u znakovnom literalu imati znak backslash navodimo dva backslasha,↑ tako da u putanjama navedenim u formi literala moramo udvostručiti svaki backslash:

↑ Poglavlje 1.2.4, str. 33.

```
FILE* ulaz = fopen("C:\\Users\\Student\\Desktop\\proba.txt", "r");
```

Međutim, ako putanja dolazi sa ulaza tj. nije literal, onda ovo udvostručenje nije potrebno! Npr. ako program glasi

```
char ime_datoteke[1000];
printf("Unesite ime datoteke: ");
unesi(ime_datoteke, 1000);
FILE* ulaz = fopen(ime_datoteke, "r");
```

Kada korisnik bude unosio naziv datoteke, kuca samo jedan backslash:

```
Unesite ime datoteke: C:\Users\Student\Desktop\proba.txt
```

Drugi parametar funkcije `fopen` je *mod* (režim, način) rada sa datotekom. Programski jezik C poznaje sljedeće modove:

- "r" – *read-only*, režim samo za čitanje; datoteka mora postojati;
- "w" – *write-only*, režim samo za pisanje; ako datoteka ne postoji biće kreirana nova, a ako postoji biće ispražnjena;
- "a" – *append* (dopisivanje), režim samo za pisanje; ako datoteka ne postoji biće kreirana nova, a ako postoji novi sadržaj će biti dopisan na kraj datoteke;
- "r+" – *read-write*, režim za čitanje i pisanje; datoteka mora postojati, kursor za čitanje i pisanje je postavljen na početak;

1/5/0198

- "w+" – *write-update*, režim za čitanje i pisanje; ako datoteka ne postoji biće kreirana nova, a ako postoji biće ispražnjena;
- "a+" – *append-update*, režim za čitanje i pisanje; ako datoteka ne postoji biće kreirana nova, kursor za čitanje i pisanje je postavljen na kraj datoteke;

U režimu samo za čitanje naredbe koje vrše pisanje će proizvesti grešku, slično u režimima samo za pisanje naredbe koje čitaju iz datoteke daju grešku. Vidimo da se tri režima za čitanje i pisanje koji se završavaju znakom plusa razlikuju samo po poziciji kursora i ponašanju u situaciji kada datoteka ne postoji.

✓ *Kursor je jedan zamišljeni pokazivač koji određuje na koju lokaciju u datoteci se odnosi sljedeća operacija čitanja odnosno pisanja.*

✓ Kursor

Svaki put kada pročitate neki znak ili niz znakova iz datoteke, kursor se pomjeri iza kraja pročitanog bloka. Kada pročitate posljednji znak iz datoteke, kursor će pokazivati "iza kraja", odnosno pokazivaće na jednu specifičnu lokaciju sa koje ne smijete čitati (doći će do greške), a koja se naziva "kraj datoteke", ili na engleskom *end-of-file* (skraćeno *EOF*).

Razmotrimo i šta nam funkcija `fopen` vraća. U pitanju je pokazivač na tip `FILE` koji je specifičan bibliotečni tip sa kojim se do sada nismo susretali. Ako proučimo dokumentaciju, saznaćemo da je u pitanju struktura[2] koja sadrži nekoliko informacija o datoteci, ali njen sadržaj ovisi od operativnog sistema sa kojim radimo. Naglašavamo da ovaj pokazivač nije isto što i kursor.

[2] Ne moramo pisati ključnu riječ `struct` pošto je iskorištena varijanta deklaracije sa `typedef` (poglavlje 9.3.3, str. 306).

✓ *Pokazivač na datoteku (eng. file pointer tj. `FILE*`) je podatak koji prosljeđujemo svim funkcijama koje rade sa datotekama, pomoću njega označavamo na koju od datoteka se naredba odnosi.*

✓ Pokazivač na datoteku

Naime, naš program može istovremeno raditi sa više datoteka, npr. možemo imati program koji iz jedne datoteke čita podatke te ih istovremeno u drugu piše čime vršimo kopiranje datoteke. Svaku od tih datoteka trebamo zasebno otvoriti funkcijom `fopen`, te ono što nam ova funkcija vrati pohraniti u zasebne promjenljive tipa `FILE*`, kao u sljedećem primjeru:

```
1  #include <stdio.h>
2  int main() {
3    FILE* ulaz = fopen("prva.txt", "r");
4    FILE* izlaz = fopen("druga.txt", "w");
5    /* Kopiramo sve sa ulaza na izlaz... */
6    fclose(ulaz);
7    fclose(izlaz);
8    return 0;
9  }
```

U linijama 3 i 4 smo otvorili datoteke `prva.txt` i `druga.txt`. Pokazivač na datoteku `prva.txt` se zove `ulaz`, a na `druga.txt` se zove `izlaz`. Zatim slijedi kopiranje (što za sada ne znamo kako se radi), te na kraju te datoteke moramo i zatvoriti. Funkcija `fclose` je također jedna od funkcija koje rade s datotekama, te ona također prima pokazivač na datoteku.

Ovaj program ipak nije potpuno ispravan. Naime, *svaki put kada otvaramo datoteku moramo provjeriti da li je otvaranje uspjelo*. U slučaju da otvaranje datoteke ne uspije, funkcija `fopen` će vratiti nul-pokazivač, što znači da će svaki naredni pokušaj pristupa toj datoteci dovesti do krahiranja programa. Neiskusni programeri mogu misliti da je u određenim situacijama otvaranje datoteke uvijek uspješno, ali takve situacije praktično ne postoje. Može doći do iznenadnog kvara uređaja, naziv datoteke ili putanja mogu biti neispravni, itd.

✓ Uvijek obavezno provjeriti povratnu vrijednost `fopen`

Primjer programa koji korektno provjerava da li su datoteke uspješno otvorene dat je ispod:

```
1  #include <stdio.h>
2  int main() {
3    FILE* ulaz = fopen("prva.txt", "r");
4    if (ulaz == NULL) {
5      printf("Otvaranje datoteke prva.txt nije uspjelo\n");
6      return 1;
7    }
8    FILE* izlaz = fopen("druga.txt", "w");
9    if (izlaz == NULL) {
10     fclose(ulaz);
11     printf("Otvaranje datoteke druga.txt nije uspjelo\n");
12     return 2;
13   }
14   /* Kopiramo sve sa ulaza na izlaz... */
15   fclose(ulaz);
16   fclose(izlaz);
17   return 0;
18 }
```

U liniji 4 provjeravamo da li je datoteka uspješno otvorena tako što upoređujemo pokazivač sa NULL. Ako otvaranje nije uspješno, možemo odmah prekinuti program, jer u ovom konkretnom programu nastavak rada nema smisla, mada u nekom drugom programu bismo mogli uraditi drugačije. Uočite da je u liniji 6 vraćena vrijednost različita od nule, što u pravilu znači da je došlo do greške. Koju vrijednost iz intervala [1,255] ćete izabrati nije toliko važno pod uslovom da onaj ko želi da tu vrijednost iskoristi zna šta ona znači.[3]

[3] Ustvari, ISO C90 standard definiše konstante `EXIT_SUCCESS` i `EXIT_FAILURE` i samo za te dvije konstante je garantovano da imaju značenja uspješnog i neuspješnog završetka. No tretman izlazne vrijednosti je svakako ovisan od sistema.

Uočavamo da u linijama 5 i 6 nema poziva `fclose` funkcije. Ako datoteka nije uspješno otvorena, nema smisla da je zatvaramo. Štaviše, ako bismo pozvali `fclose`, došlo bi do krahiranja programa, jer je `ulaz` nul-pokazivač.

U liniji 10 zatvaramo datoteku `ulaz`, jer ako smo došli do ove tačke u programu, ona je uspješno otvorena. Kada se program prekine sve otvorene datoteke će biti zatvorene, tako da striktno govoreći ova na-

1755019

redba nije potrebna. No, trebamo se vježbati da pratimo koje smo sve resurse zauzeli prilikom rada programa i oslobađamo ih kada nam oni više ne trebaju. Otvorena datoteka je zauzet resurs i trebali bismo je korektno zatvoriti prije poziva naredbe `return` koja će završiti program (linija 12). Osim toga, ako datoteku u koju smo upisali neke podatke ne zatvorimo funkcijom `fclose`, zna se desiti da podaci ne budu uspješno zapisani.[4] Još jednom napominjemo da na ovom mjestu datoteka `izlaz` *nije* uspješno otvorena.

[4] Razlog za to je što operativni sistem ima pravo da podatke ne zapiše odmah, nego ih drži u međuspremniku (bufferu) radi boljih performansi. Ovaj buffer će biti zapisan na disk prilikom zatvaranja, a možemo ga i ručno zapisati pozivom funkcije `fflush()`. Međutim, ako se program prekine, npr. usljed krahiranja, i buffer biva izgubljen.

10.2.2 *Rad sa karakterima*

Funkcije za rad sa tekstualnim datotekama možemo podijeliti u dvije grupe: funkcije koje rade na nivou karaktera (znaka) i funkcije koje rade sa formatiranim podacima. U prvu grupu spadaju `fgetc`, `fputc`, `fgets` i `fputs`, a u drugu `fscanf` i `fprintf`.

✓ `fgetc`

Funkcija `fgetc` čita jedan znak iz datoteke i vraća njegovu ASCII vrijednost, te ujedno pomjera kursor za jedan znak udesno (prema kraju datoteke). Primjer poziva funkcije je:

```
int c = fgetc(ulaz);
```

Funkcija ima jedan parametar, a to je pokazivač na datoteku koji smo dobili pozivom funkcije `fopen`. Uočavamo da funkcija vraća vrijednost tipa `int`, a ne tipa `char`. Naime, pored dozvoljenih 256 vrijednosti karaktera, funkcija može vratiti specijalnu vrijednost EOF koja označava da se nalazimo na kraju datoteke i da ne možemo dalje čitati iz nje. EOF po standardu mora biti negativan cijeli broj (najčešće -1), tako da nam treba tip `int`.

Ako je datoteka otvorena u nekom od režima samo za pisanje ("w" ili "a"), prilikom poziva `fgetc` doći će do greške.

Osnovna petlja za rad sa datotekom koristeći funkciju `fgetc` izgleda ovako:

```
while ((c = fgetc(ulaz)) != EOF)
  ...
}
```

Kôd koji radi nešto korisno sa očitanom vrijednošću `c` možemo staviti umjesto oznake tri tačke. U zaglavlju `stdio.h` je definisana konstanta `EOF` koju koristimo za uslov petlje. Obratite pažnju da u primjeru iznad ne postoji višak zagrada: operator dodjele `=` ima niži prioritet od operatora poređenja `!=`, pa bi kod

```
while (c = fgetc(ulaz) != EOF)
```

promjenljivoj `c` pridružio rezultat poređenja povratne vrijednosti funkcije `fgetc` sa `EOF`, a to je logička vrijednost 0 ili 1. Ono što želimo je da se najprije promjenljivoj `c` pridruži povratna vrijednost funkcije, pa zatim ako je ta vrijednost jednaka `EOF` da se petlja završi.

1/4D/0198

Umjesto poređenja očitane vrijednosti sa EOF, možemo koristiti funkciju `feof`. Ova funkcija prima pokazivač na datoteku a vraća logičku vrijednost 0 ako nismo na kraju datoteke a 1 ako jesmo. Dakle, petlja identična gornjoj koja koristi `feof` bi glasila: ✓ `feof`

```
do {
  c = fgetc(ulaz);
  ...
} while (!feof(ulaz));
```

Zadatak 1. *Napisati program koji u tekstualnoj datoteci ulaz.txt prebrojava samoglasnike.*

Rješenje zadatka je dato ispod.

```
1  #include <stdio.h>
2  int main() {
3    int c, samoglasnika=0;
4
5    FILE* ulaz = fopen("ulaz.txt", "r");
6    if (ulaz == NULL) {
7      printf("Otvaranje datoteke prva.txt nije uspjelo\n");
8      return 1;
9    }
10
11   while ((c = fgetc(ulaz)) != EOF) {
12     if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' ||
13       c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U')
14       samoglasnika++;
15   }
16
17   printf("Ukupno %d samoglasnika.\n", samoglasnika);
18
19   fclose(ulaz);
20   return 0;
21 }
```

Potrebno je najprije otvoriti datoteku `ulaz.txt` u modu za čitanje `"r"` (linija 5), provjeriti da li je uspješno otvorena (linije 6-10), te uvrstiti petlju za čitanje znak po znak iz datoteke (linije 11, 12, 16), a na kraju zatvoriti datoteku (linija 20). Sve ovo su fragmenti koda koji su nam već poznati. Dodaćemo promjenljivu `samoglasnika` koja predstavlja brojač znakova u datoteci koji su samoglasnici. Ostaje da napišemo uslov da li je znak samoglasnik. Na žalost, ne postoji neka formula kojom to možemo odrediti iz ASCII vrijednosti, tako da nam ne preostaje ništa drugo nego da navedemo sve samoglasnike u uslovu (linije 13-15).

Funkcija `fputc` zapisuje jedan karakter u datoteku. Prototip funkcije glasi: ✓ `fputc`

```
int fputc(char znak, FILE* datoteka);
```

Prvi parametar je ASCII vrijednost znaka koji želimo zapisati u datoteku, a drugi parametar je pokazivač na datoteku. Funkcija vraća EOF u slučaju greške, a ako je znak uspješno zapisan vraća isti taj

17570198

znak. Ako je datoteka otvorena u režimu samo za čitanje ("r"), doći će do greške.

Obratite pažnju da se vrijednost EOF ne treba zapisivati u datoteku jer se ona ustvari ne nalazi u datoteci, to je signal koji dobijamo od funkcije `fgetc` da smo došli do kraja. Ako zapišete tu vrijednost, kada kasnije otvorite datoteku u npr. programu Notepad vidjećete suvišan karakter sa ASCII kodom 127 ili 255 na kraju datoteke. Naime, EOF ima vrijednost -1, a `fputc` prima tip `char`, pa će ovdje doći do prekoračenja.↑

↑ Poglavlje 3.1.9, str. 78.

Sada imamo dovoljno informacija da napišemo nedostajući dio našeg programa za kopiranje datoteka:

```
while ((c = fgetc(ulaz)) != EOF)
  fputc(c, izlaz);
```

Kada `c` bude jednako `EOF`, tijelo petlje se neće izvršiti i taj znak neće biti upisan u datoteku.

✓ `ferror`

Pomoću funkcije `ferror` možemo saznati da li je prethodna operacija nad datotekom dovela do greške. Ako ne prekinemo program (funkciju, petlju) nakon greške, sljedeća operacija nad datotekom se jednostavno neće izvršiti, ili će doći do krahiranja. Funkcija `ferror` prima pokazivač na datoteku, a vraća logičku vrijednost 0 (nije bilo greške) ili 1 (došlo je do greške). Primjer poziva ove funkcije dat je ispod.

```
FILE* ulaz = fopen("nesto.txt", "r");
fputc('a', ulaz); /* Greska */
if (ferror(ulaz))
  printf("Ne moze se pisati u nesto.txt.\n");
```

Sada možemo napisati kompletan program koji kopira datoteku znak po znak:

```
1  #include <stdio.h>
2  int main() {
3    int c;
4    FILE* ulaz = fopen("prva.txt", "r");
5    if (ulaz == NULL) {
6      printf("Otvaranje datoteke prva.txt nije uspjelo\n");
7      return 1;
8    }
9    FILE* izlaz = fopen("druga.txt", "w");
10   if (izlaz == NULL) {
11     fclose(ulaz);
12     printf("Otvaranje datoteke druga.txt nije uspjelo\n");
13     return 2;
14   }
15   while ((c = fgetc(ulaz)) != EOF) {
16     fputc (c, izlaz);
17     if (ferror(ulaz) || ferror(izlaz)) {
18       printf("Greska.\n");
19       break;
20     }
21   }
22   printf("Kopiranje zavrseno.\n");
23   fclose (ulaz);
```

```
24    fclose (izlaz);
25    return 0;
26  }
```

Umjesto da čitamo znak po znak iz datoteke, možemo odjednom očitati čitav string. Za to koristimo funkciju `fgets`. Prototip funkcije glasi:

✓ `fgets`

```
char* fgets(char* string, int br, FILE* datoteka);
```

Prvi parametar ove funkcije je pokazivač tipa `char` koji pokazuje na neku lokaciju gdje možemo upisati sadržaj očitan iz datoteke. Npr. to može biti ime niza (stringa). Pošto je u pravilu veličina niza ograničena, drugi parametar ograničava broj znakova koji će biti učitani iz datoteke. Biće učitano najviše br-1 znakova iz datoteke (pri čemu će biti ostavljeno i jedno mjesto za znak \0), a može biti učitano i manje znakova ako se dođe do oznake za novi red ili kraja datoteke. Treći parametar je uobičajeni pokazivač na datoteku.

Primjećujemo da je ova funkcija vrlo slična našoj funkciji `unesi`,↑ s tim što čita znakove iz datoteke umjesto sa standardnog ulaza.

↑ Poglavlje 8.1.1, str. 265.

Funkcija će vratiti pokazivač na učitani string (isti pokazivač koji je primljen) ako je čitanje uspjelo, što olakšava njeno lančano pozivanje, a ako čitanje nije uspjelo vratiće nul-pokazivač.

Slično tome, funkcija `fputs` zapisuje string u datoteku.

✓ `fputs`

```
void fputs(char* string, FILE* datoteka);
```

Prvi parametar je string (adresa niza tipa `char`) koji treba zapisati, a drugi pokazivač na datoteku u koju zapisujemo.

Da bismo demonstrirali primjenu ovih funkcija, iskoristićemo ih da napravimo drugu varijantu programa koji kopira datoteku:

```
1  #include <stdio.h>
2  int main() {
3    char s[100];
4    FILE* ulaz = fopen("prva.txt", "r");
5    if (ulaz == NULL) {
6      printf("Otvaranje datoteke prva.txt nije uspjelo\n");
7      return 1;
8    }
9    FILE* izlaz = fopen("druga.txt", "w");
10   if (izlaz == NULL) {
11     fclose(ulaz);
12     printf("Otvaranje datoteke druga.txt nije uspjelo\n");
13     return 2;
14   }
15   while(fgets(s, 100, ulaz))
16     fputs(s, izlaz);
17   fclose (ulaz);
18   fclose (izlaz);
19   return 0;
20 }
```

U liniji 3 deklarisan je string veličine 100 znakova koji predstavlja buffer, odnosno privremeni smještajni prostor za čitanje iz datoteke.

Veličina ovog buffera nije bitna, odnosno program će raditi podjednako dobro sa bufferom bilo koje veličine, ali veći buffer će logično dovesti do bržeg rada i većeg zauzeća memorije.

Linije 15 i 16 sadrže petlju koja vrši kopiranje. Uslov while petlje je poziv funkcije `fgets` koja će vratiti nul-pokazivač (tj. 0) kada dođe do kraja datoteke, a ujedno će napuniti buffer. Vidimo da će u svakom prolazu kroz petlju buffer `s` biti napunjen sa najviše 100 znakova (ustvari 99 znakova plus \0), a zatim će biti zapisan u izlaznu datoteku.

10.2.3 *Rad sa formatiranim datotekama*

Funkcije koje smo do sada predstavili su praktične za rad sa datotekama koje sadrže samo string(ove). Međutim, šta ako datoteka sadrži npr. neke numeričke vrijednosti? Morali bismo čitati cifre jednu po jednu kao ASCII vrijednost, a zatim ih pretvarati u cijeli broj na način kako smo to radili sa stringovima,↑ s tim što bismo morali vjerovatno podržati i realne brojeve, što postaje poprilično komplicirano.

↑ Poglavlje 8.2.2, str. 282.

Dodatni problem je što format datoteke odnosno njena struktura nije precizno definisana. Na koji način ćemo znati da se jedan podatak (u okviru datog reda) završava a drugi počinje? Npr. mogli bismo za razdvajanje podataka unutar jednog reda koristiti zarez, čime dobijamo CSV format (eng. comma-separated values – vrijednosti razdvojene zarezom). Ali šta ako baš želimo da predstavimo znak zarez u okviru nekog podatka? Ovaj problem se javlja bez obzira koji znak koristimo za razdvajanje.

Formatirane datoteke su tekstualne datoteke kod kojih je precizno definisan izgled svakog reda. Jedan red datoteke predstavlja *slog*. Za svaki slog definiše se od kojih podataka se sastoji, koliko prostora zauzima koji podatak odnosno čime su razdvojeni. Ovo će postati malo jasnije kada pogledamo neke primjere.

Funkcija `fscanf` ponaša se slično kao funkcija `scanf`, s tim što se podaci čitaju iz datoteke umjesto sa standardnog ulaza. Prvi parametar funkcije je pokazivač na datoteku, zatim slijedi format string koji izgleda isto kao i kod `scanf`, te nakon toga adrese promjenljivih u koje upisujemo očitane vrijednosti. Primjer poziva funkcije fscanf:

✓ `fscanf`

```
fscanf(ulaz, "%d %d %s\n", &sirina, &duzina, ime);
```

Promjenljiva `ulaz` je pokazivač na datoteku iz koje čitamo podatke, a dalje slijede parametri kao i kod *scanf*.

Nakon poziva funkcije `fscanf`, kursor će se nalaziti iza posljednjeg pročitanog podatka. Kao i `scanf`, `fscanf` vraća *broj uspješno očitanih vrijednosti (promjenljivih)*, što se može iskoristiti da se detektuju kako greške, tako i kraj datoteke. U slučaju da je `fscanf` odmah naišla na kraj datoteke, ona vraća konstantu `EOF` koja je po standardu definisana kao negativna vrijednost (u pravilu -1). Pogledajte sljedeću petlju:

1770198

```
while(fscanf(ulaz, "%d %d %s\n", &sirina, &duzina, ime) == 3)
  printf("Ucitano je: %d,%d,%s\n", sirina, duzina, ime);
```

Ova petlja će čitati iz datoteke podatke red po red, pri čemu je svaki red oblikovan na sljedeći način: najprije se nalaze dva cijela broja razdvojena znakom razmaka, a zatim string (koji ne smije sadržavati razmake). Primjer izgleda ulazne datoteke:

```
15 40 Sarajevo
20 9 Tuzla
```

Ove podatke ćemo učitati u promjenljive `sirina`, `duzina` i `ime`. Zatim će ti podaci biti ispisani na ekran pozivom funkcije `printf`. Obratite pažnju da, za razliku od `scanf`, u funkciji `fscanf` moramo eksplicitno zatražiti da se učita znak \n iz datoteke, u suprotnom čitanje se neće nastaviti dok se taj znak ne učita i petlja će se prekinuti nakon prvog reda.

U slučaju uspješnog čitanja reda funkcija će vratiti broj 3. Ako dođemo do kraja datoteke ili do reda koji nije formatiran kako je opisano, `fscanf` će vratiti neki broj manji od 3, te će se petlja prekinuti.

Vidjeli smo da `fscanf` može čitati stringove, ali pod uslovom da oni ne sadrže razmake. Kako raditi sa datotekama gdje string može sadržavati razmake, kao što je slučaj sa već spomenutim CSV formatom? Recimo da datoteka izgleda ovako:

```
500000,Sarajevo,13.7
100000,Mostar,17.4
200000,Banja Luka,15.5
```

Prvi podatak u svakom redu je broj stanovnika, drugi je ime grada a treći prosječna temperatura u gradu. Ako bismo podatke čitali petljom sa `fscanf` kao ranije

```
while(fscanf(ulaz, "%d,%s,%f\n", &br_stanovnika, ime, &temperatura) == 3)
  printf("Prosjecna temperatura u %s (%d) je %g\n", ime, br_stanovnika,
   temperatura);
```

na ekranu bismo dobili podatke samo za Mostar i Sarajevo. Kod čitanja podataka za Banja Luku, funkcija `fscanf` bi vratila 2 (uspješno očitan samo broj stanovnika i ime), te bi promjenljiva `ime` sadržavala vrijednost `Banja`. Razlog je to što se `fscanf` kod čitanja stringova sa formatom `%s` uvijek zaustavlja na znaku razmaka (kao i `scanf`).

Jedno rješenje bi moglo biti korištenje funkcije `fgets` koja nema ovaj nedostatak. No, `fgets` će uvijek pročitati onaj broj znakova koji mu damo kao parametar, bez obzira koji su to znakovi, pa ako stavimo npr. da naziv grada ima 20 znakova, prvi grad će nam se zvati `Sarajevo,13.7` (naredba `fgets` se zaustavlja na prelasku u novi red). Znači nakon čitanja podataka iz datoteke korištenjem `fgets`, morali bismo ručno raščlaniti zareze i brojeve.

Ispravno rješenje ovog problema je čitati znakovne podatke iz datoteke koristeći funkciju `fgetc` sve do znaka zareza ili kraja datoteke.

1/15/0198

Za čitanje numeričkih podataka ćemo i dalje koristiti `fscanf` da ne bismo ručno morali konvertovati brojeve u `int` odnosno `double` tip.

```
1  while(fscanf(ulaz, "%d,", &br_stanovnika) == 1) {
2    i=-1;
3    do {
4      if (i<19) i++;
5      ime[i] = fgetc(ulaz);
6    } while(ime[i] != ',' && ime[i] != EOF);
7    ime[i] = '\0';
8    if (fscanf(ulaz, "%f\n", &temperatura) == 0) break;
9    printf("Prosjecna temperatura u %s (%d) je %g\n", ime, br_stanovnika,
      temperatura);
10 }
```

U liniji 1 očitali smo broj stanovnika i prvi znak zareza, tako da nam kursor pokazuje na prvo slovo imena grada. `i` će nam poslužiti kao indeks u stringu `ime`. Postavljamo ga na -1 jer će nam to pojednostaviti uslov do-while petlje. U liniji 4 smo se pobrinuli da `i` ne izađe izvan deklarisane veličine stringa (20 znakova). Naime, ako se u datoteci bude nalazilo ime grada duže od 19 znakova, preostali karakteri do znaka zarez moraju biti učitani kako bismo ispravno pozicionirali kursor. Uslovom u liniji 4 smo postigli da se ti znakovi čitaju jedan preko drugog u posljednje mjesto u stringu. Petlja se prekida kada dođemo do znaka zarez ili kraja datoteke (linija 6), a zatim terminiramo string znakom \0. Konačno, u liniji 8 smo učitali i temperaturu kao i znak `\n`, pa ako smo u međuvremenu došli do kraja datoteke, prekidamo petlju.

Funkcija `fprintf` upisuje podatke u datoteku. I ova funkcija ima isti oblik kao `printf`, s tim što ispisuje podatke u izlaznu datoteku umjesto na ekran. Njen prvi parametar je pokazivač na izlaznu datoteku, a zatim slijede parametri kao i kod `printf`. Obje spomenute funkcije vraćaju *broj uspješno ispisanih karaktera* što je znatno teže iskoristiti u programu. ✓ `fprintf`

Pogledajmo varijantu petlje koja čita podatke iz ulazne datoteke, a zapisuje u izlaznu datoteku:

```
1  while (fscanf(ulaz, "%d %d %s\n", &sirina, &duzina, ime) == 3) {
2    fprintf(izlaz, "%d %s\n", sirina*duzina, ime);
3    if (ferror(izlaz)) {
4      printf("Greska\n");
5      break;
6    }
7  }
```

Ako su podaci uspješno učitani, `fscanf` će vratiti 3 i prelazimo na tijelo petlje. U liniji 2 upisuje se površina tijela (širina puta dužina), a provjeru ispravnosti pisanja vršimo pomoću funkcije `ferror` (linija 3 i dalje).

Ako bi ulazna datoteka izgledala ovako:

```
15 40 Sarajevo 20 9 Tuzla
```

1/15/0198

bili bi uspješno učitani samo podaci za Sarajevo, pošto `fscanf` nije naišla na znak \n. Ako bi glasila

```
Sarajevo 15 40
```

`fscanf` će odmah vratiti 0 jer prvi podatak nije cijeli broj i izlazna datoteka će ostati prazna.

Pogledajmo i primjer programa koji iz datoteke učitava niz realnih brojeva:

```
1  #include <stdio.h>
2  int main() {
3    double niz[100];
4    int i, vel;
5
6    FILE* ulaz = fopen("proba.txt", "r");
7    if (!ulaz) {
8      printf("Greska pri otvaranju.\n");
9      return 1;
10   }
11
12   i=0;
13   while (i<100 && fscanf(ulaz, "%lf", &niz[i]) == 1)
14     i++;
15   printf("Ucitano %d elemenata.\n", i);
16   fclose(ulaz);
17   /* ... */
18   return 0;
19 }
```

Pošto niz ima 100 elemenata (linija 3), ne smijemo pročitati više od 100 brojeva iz datoteke, jer bi se program mogao krahirati. Brojač `i` ujedno predstavlja indeks u nizu, tako da u uslov petlje možemo dodati da taj brojač mora biti manji od 100: `i<100` (linija 13). Ovaj uslov se mora nalaziti prije poziva funkcije `fscanf` zbog kratkospojne evaluacije, jer kada je `i` jednako 100, ne smijemo čitati i-ti element.

Obratite pažnju da se u formatu nalazi samo `"%f"`, što znači da su brojevi razdvojeni razmacima (znak razmaka se ne mora navesti u formatu). Pošto se u svakom prolazu kroz petlju čita samo jedna vrijednost (jedan član niza), funkcija `fscanf` vraća 1. Ako se u datoteci nalazi manje od 100 brojeva, `fscanf` će nakon čitanja posljednjeg broja vratiti nulu ili `EOF`, tako da nam `i` u tom slučaju predstavlja broj uspješno učitanih članova niza.

U slučaju da je zadatkom specificirano da su brojevi u datoteci razdvojeni znakom zareza, mogli smo pisati:

```
13   while (i<100 && fscanf(ulaz, "%f,", &niz[i]) == 1)
```

odnosno dodali smo znak zareza nakon `%f` ali unutar navodnika. Na taj način će fscanf očitati i zarez. Ako se nakon posljednjeg broja ne nalazi znak zarez, funkcija fscanf i dalje vraća jedinicu (jer je taj posljednji broj uspješno učitan) a u sljedećem prolazu ćemo dobiti naravno `EOF` i petlja se prekida.

1/15/01198

Zadatak 2. *U datoteci "temperature.txt" nalaze se mjerenja tempera-*

ture u gradovima BiH. Svaki red ima sljedeći oblik: ime grada (bez razmaka, najviše 20 znakova), dan, mjesec (cijeli brojevi) i izmjerena temperatura (realan broj). Ovi elementi su razdvojeni znakom razmaka. U datoteci se nalazi najviše 1000 mjerenja. Napisati program koji učitava podatke u memoriju, a zatim pronalazi najveću izmjerenu temperaturu te na ekranu ispisuje grad u kojem je temperatura izmjerena i datum u obliku `dan.mjesec.2017.`.

Ovaj zadatak bi se mogao riješiti tako da u istoj petlji u kojoj čitamo iz datoteke pronalazimo maksimum te ga poslije petlje ispisujemo na ekran. No, česti su i zadaci gdje to nije moguće, pa ćemo ići nešto dužim rješenjem gdje se čitanje i traženje maksimuma obavlja u odvojenim petljama.

Najprije uočavamo da svaki red datoteke predstavlja jedan slog tj. svaki red sadrži isti skup podataka koji je opisan u tekstu zadatka. Za slogove je prirodno koristiti strukturni tip podataka,↑ pa ćemo definisati `struct Mjerenje` koje sadrži navedene podatke o jednom mjerenju:

↑ Poglavlje 9.3, str. 303.

```
struct Mjerenje {
  char grad[21];
  int dan, mjesec;
  double temperatura;
};
```

Ime grada smije sadržavati najviše 20 znakova, pa ostavljamo još jedno mjesto za znak \0. Pošto se u datoteci može nalaziti najviše 1000 mjerenja, deklarisaćemo niz vrijednosti tipa `struct Mjerenje` od 1000 elemenata:

```
struct Mjerenje mjerenja[1000];
```

Zatim otvaramo datoteku za čitanje, što se radi na uobičajen način, te formiramo petlju koja čita formatirane podatke iz datoteke koristeći `fscanf`. Format jednog reda datoteke glasi: `"%20s %d %d %lf\n"`. Obratite pažnju da smo postavili dozvoljenu maksimalnu dužinu stringa od 20 znakova, što će imati očekivani rezultat kod rada sa datotekom (za `scanf` je nedjelotvorno). Što se tiče rada sa nizom struktura, podsjetimo se rada sa nizovima struktura,↑ te prethodnog primjera za čitanje niza brojeva iz datoteke. U svakom prolazu kroz petlju čitaćemo i-ti član niza `mjerenja` i to pristupajući članovima strukture preko operatora tačka:

↑ Poglavlje 9.3.7, str. 313.

```
while (i<1000 && fscanf(ulaz, "%20s %d %d %lf\n", mjerenja[i].grad,
  &mjerenja[i]].dan, &mjerenja[i].mjesec, &mjerenja[i].temperatura) == 4)
```

Za promjenljivu `mjerenja[i].grad` ne treba nam znak ampersanda jer je u pitanju niz, pa je njegovo ime samo po sebi adresa. Funkcija `fscanf` će vratiti 4 u slučaju uspješnog čitanja, jer smo čitali četiri vrijednosti.

U samom tijelu petlje nećemo obavljati nikakav zadatak jer se to nalazi u sljedećoj petlji, tako da ovdje imamo samo `i++`. Nakon petlje, `i` nam je broj članova niza, pa ćemo ga sačuvati u promjenljivu

1755701198

`vel` kako bismo mogli ponovo koristiti `i` za for petlju. Slijedi standardno pronalaženje maksimuma, te na kraju ispisujemo podatke tako pronađenog grada sa najvećom temperaturom. Kompletno rješenje se nalazi ispod.

```
#include <stdio.h>
struct Mjerenje {
  char grad[21];
  int dan, mjesec;
  double temperatura;
};

int main() {
  struct Mjerenje mjerenja[1000];
  int i, vel, max;

  FILE* ulaz = fopen("temperature.txt", "r");
  if (!ulaz) {
    printf("Greska pri otvaranju.\n");
    return 1;
  }

  i=0;
  while (i<1000 && fscanf(ulaz, "%20s %d %d %lf\n", mjerenja[i].grad,
    &mjerenja[i]].dan, &mjerenja[i].mjesec, &mjerenja[i].temperatura) == 4)
      i++;
  fclose(ulaz);

  vel = i;
  max = 0;
  for (i=1; i<vel; i++)
    if (mjerenja[i].temperatura > mjerenja[max].temperatura)
      max = i;
  printf("Najveca temperatura je izmjerena u gradu %s na dan %d.%d.2017.\n",
      mjerenja[max].grad, mjerenja[max].dan, mjerenja[max].mjesec);

  return 0;
}
```

10.3 *Baze podataka*

Pojmom **program za upravljanje bazom podataka** označavamo softver koji omogućuje pisanje i čitanje strukturiranih podataka iz neke jedinice eksterne memorije. Osnovna razlika između datoteke i baze podataka je što datoteka ne mora imati fiksnu strukturu. Baza podataka osigurava da je struktura podataka uvijek očuvana, te nudi i različite mogućnosti pretraživanja.

Kada smo u ranijem zadatku kreirali niz strukturnog tipa te ga pohranili u datoteku, mi smo ustvari počeli implementirati jednu vrlo jednostavnu bazu podataka. U nastavku ovog poglavlja diskutovaćemo neke strategije implementacije baza podataka.

1/15/01/98

Zadatak 3. *Napraviti aplikaciju "Imenik" koja omogućuje pohranu telefonskog imenika u datoteku što uključuje: dodavanje, promjenu i brisanje*

telefonskih brojeva iz imenika, te pretragu po različitim kriterijima.

Potpuna implementacija ovog zadatka bi bila preobimna za ovu knjigu, tako da ćemo se posvetiti samo nekim aspektima rada aplikacije.

Prije svega, kako predstaviti telefonski imenik u memoriji računara? Jasno je da nam treba za svaki slog u imeniku ime, prezime i broj telefona, što opisuje neki strukturni tip `Osoba`, pa ćemo definisati niz podataka tog tipa:

```
#define KAPACITET 1000
struct Osoba {
  char ime[20], prezime[20], brtel[15];
};
struct Osoba imenik[KAPACITET];
```

Kapacitet tj. veličina niza mora biti konstantan. Definisali smo simboličku konstantu zato da bismo kasnije mogli po potrebi jednostavno proširiti ovaj kapacitet.

Čitanje podataka iz datoteke možemo realizovati sljedećom petljom:

```
int i = 0;
FILE* dat = fopen("baza.txt", "r+");
while (i < KAPACITET && fscanf(dat, "%20s%20s%15s\n",
  imenik[i].ime, imenik[i].prezime, imenik[i].brtel) == 3)
    i++;
broj_osoba = i;
```

Prije svega, datoteku otvaramo u modu `"r+"`, što znači da kada pročitamo podatke iz datoteke možemo se vratiti (koristeći `fseek`), te u istu datoteku i pisati. Ne smijemo pročitati više od `KAPACITET` slogova da ne bi došlo do prekoračenja dimenzije niza, pa postavljamo uslov `i<KAPACITET` kao prvi uslov u while petlju zbog kratkospojne evaluacije. Format `"%20s"` ograničava učitani podatak na 20 znakova. U slučaju uspješnog čitanja funkcija vraća broj 3. U tijelu petlje nalazi se samo naredba `i++`, jer je sav ostali posao obavljen u uslovu. Nakon petlje, `i` nam je broj uspješno učitanih podataka, pa ćemo ga prepisati u promjenljivu `broj_osoba`.

Ne smijemo zaboraviti zapisati podatke u datoteku kako ih ne bismo izgubili uslijed prestanka rada programa (krahiranje, potrošena baterija itd.) Petlja za zapis u datoteku bi mogla izgledati ovako:

```
for (i=0; i<broj_osoba; i++) {
  fprintf(dat, "%-20s%-20s%-15s\n", imenik[i].ime, imenik[i].prezime,
   imenik[i].brtel);
  if (ferror(dat)) {
    printf("Greska prilikom pisanja datoteke.\n");
    break;
  }
}
```

1 // 2 // 0 4 // 1 9 8

Broj podataka koje zapisujemo u datoteku je fiksno `broj_osoba`, pa koristimo for petlju. Format `"%-20s"` osigurava da su podaci fino poravnati ulijevo po kolonama, tako da datoteka lijepo izgleda i kada je

otvorimo u npr. Notepadu. Provjeru greške radimo kao i ranije funkcijom `ferror`.

Kada pozivati čitanje a kada pisanje? Preporučujemo da napravite funkciju `ucitaj` koja čita podatke iz datoteke, te funkciju `zapisi` koja podatke piše u datoteku, tako da ostaje pitanje kada pozivati koju od ovih funkcija. Preporučena strategija implementacije je sljedeća: pozvaćemo funkciju `ucitaj` odmah na početku programa, a funkciju `zapisi` nakon svake promjene. Ili, ako su performanse problem, možemo funkciju pozvati samo jednom prilikom završetka programa. U obje ove funkcije će se vršiti i otvaranje i zatvaranje datoteke. Razlog za to je da datoteke ne bi ostale otvorene u slučaju krahiranja programa, jer bi to moglo dovesti do gubitka podataka.

Moguća je implementacija kod koje se podaci uopšte ne drže u memoriji, nego se čitaju direktno iz datoteke, svaka izmjena se zapisuje u datoteku itd. Programi su se radili ovako u situacijama kada količina interne memorije nije dovoljna da se spreme svi podaci. Također, na ovaj način se rade profesionalne baze podataka koje mogu biti velike više terabajta. Kod takvih implementacija posebna pažnja se posvećuje različitim tehnikama koje osiguravaju brz rad, ali uz malu vjerovatnoću gubitka podataka. U našem telefonskom imeniku to bi bilo pretjerano.

Daćemo sada funkciju `ucitaj`:

```
1  int ucitaj(struct Osoba* imenik) {
2    int i = 0;
3    FILE* dat = fopen("baza.txt", "r");
4    if (dat == NULL) {
5      printf("Greska pri otvaranju datoteke za citanje.\n");
6      return 0;
7    }
8    while (i < KAPACITET && fscanf(dat, "%20s%20s%15s\n",
9      imenik[i].ime, imenik[i].prezime, imenik[i].brtel) == 3)
10       i++;
11   fclose(dat);
12   return i;
13 }
```

Od interesantnih novih stvari skrećemo pažnju da se u slučaju greške kod otvaranja datoteke poziva `return 0` (linija 6), što *neće* prekinuti program jer se nalazimo u funkciji. Funkcija vraća broj učitanih osoba, tako da će ovo značiti da nije ništa učitano, ali program dalje nastavlja rad normalno, tako da će datoteka biti kreirana prilikom poziva funkcije `zapisi`. Ako postoji neki drugi problem u radu sa diskom, to ćemo nažalost saznati tek tada. Zbog toga je poželjno pozivati funkciju `zapisi` što češće. Ako dođe do greške zbog koje je pročitan samo dio podataka iz datoteke (korumpirana datoteka), nastavljamo rad sa uspješno učitanim dijelom.

1757018

Funkcija `zapisi` bi izgledala ovako:

```
void zapisi(struct Osoba* imenik, int broj_osoba) {
  int i;
  FILE* dat = fopen("baza.txt", "w");
  if (dat == NULL) {
    printf("Greska pri otvaranju datoteke za pisanje.\n");
    exit(1);
  }
  for (i=0; i<broj_osoba; i++) {
    fprintf(dat, "%-20s%-20s%-15s\n", imenik[i].ime, imenik[i].prezime,
   imenik[i].brtel);
    if (ferror(dat)) {
      printf("Greska prilikom pisanja datoteke.\n");
      break;
    }
  }
  fclose(dat);
}
```

Ovdje pak neuspješno otvaranje datoteke predstavlja katastrofalnu grešku, zbog koje odmah prekidamo program pozivom funkcije `exit(1)`.↑ Obratite pažnju da u `ucitaj` koristimo mod "r", a u `zapisi` mod "w" jer se datoteke odmah otvaraju i zatvaraju. Funkcija `zapisi` će pobrisati sav stari sadržaj datoteke i zamijeniti ga novim imenikom. Ovo bi moglo biti sporo ako je imenik velik više stotina megabajta, tako da se tu pravi kompromis sa čestim pozivanjem funkcije.

↑ Poglavlje 6.1.1, str. 176.

Main bi mogao izgledati ovako:

```
int main() {
  broj_osoba = ucitaj(imenik);
  zapisi(imenik, broj_osoba);
  meni();
  return 0;
}
```

Vidimo da se u globalnom opsegu nalazi niz `imenik` i neka promjenljiva `broj_osoba` koja čuva broj slogova u imeniku. Odmah nakon poziva funkcije `ucitaj` pozivamo i `zapisi` da otkrijemo da li postoji neki problem sa eksternom memorijom. Nakon toga ulazimo u meni gdje će se funkcija `zapisi` pozivati nakon svake izmjene podataka. Funkcija `meni` omogućuje korisniku da izabere šta želi raditi sa imenikom:

```
void meni() {
  int opcija;
  do {
    printf("1 - Unos osobe\n2 - Ispis\n0 - Izlaz\n");
    scanf("%d", &opcija);
    switch(opcija) {
      case 1: unos_osobe(); break;
      case 2: ispis(); break;
      case 0: break;
      default: printf("Nepoznata opcija\n");
    }
  } while (opcija != 0);
}
```

1/19/98

Naš meni ima samo dvije opcije, ali jednostavno ga je proširiti na više. Ove funkcije su već poznate, ali ne smijemo zaboraviti pozvati funkciju

`zapisi`. Npr. funkcija za unos osobe bi izgledala ovako:

```
void unos_osobe() {
  printf("Ime: ");
  unesi(imenik[broj_osoba].ime, 20);
  printf("Prezime: ");
  unesi(imenik[broj_osoba].prezime, 20);
  printf("Broj telefona: ");
  unesi(imenik[broj_osoba].brtel, 15);
  broj_osoba++;
  zapisi(imenik, broj_osoba);
}
```

Nova osoba koju unosimo nalaziće se na indeksu nakon posljednje unesene osobe, a to je globalna promjenljiva `broj_osoba`. Na kraju funkcije uvećaćemo `broj_osoba` za jedan i zapisati imenik u datoteku.

Kada imenik naraste, može se desiti da zapisivanje imenika traje predugo i usporava program. U tom trenutku možemo izbaciti pozive funkcije zapisi iz drugih funkcija, a pozvati je samo u mainu nakon svih ostalih operacija. Na ovaj način je i vrlo lako prepraviti neki postojeći program koji je zamišljen da koristi isključivo internu memoriju tako da podatke drži u datoteci.

```
int main() {
  broj_osoba = ucitaj(imenik);
  zapisi(imenik, broj_osoba);
  meni();
  zapisi(imenik, broj_osoba);
  return 0;
}
```

Vidimo da se `zapisi` poziva po izlasku iz menija a pozive ove funkcije bi trebalo izbaciti iz ostalih funkcija. Umjesto poziva funkcije `meni()` može se nalaziti i sama glavna petlja programa.

10.4 *Binarne datoteke*

U mnogim udžbenicima možete čitati o razlici između *tekstualnih* i *binarnih datoteka*. Sve datoteke se sastoje iz niza bajta koje možemo tumačiti kao cjelobrojne vrijednosti 0-255, ili ih možemo tumačiti kao ASCII vrijednosti koje odgovaraju znakovima. Ako ih tumačimo na prvi način u pitanju su binarne datoteke, a ako koristimo drugi način onda su to tekstualne datoteke. Dakle, razlika je samo u intepretaciji podataka.

Istina, ako obratite pažnju na ASCII tabelu, primijetićete da neke vrijednosti ne bi trebale da se pojavljuju u tekstualnim datotekama. ASCII tabela ne definiše znakove sa vrijednostima 128-255. Na većini operativnih sistema ove vrijednosti se tradicionalno koriste za predstavljanje znakova iz nacionalnih (ne-engleskih) alfabeta, tako da ih možemo naći u tekstualnim datotekama.

U opsegu 0-31 nalaze se neki znakovi koje je logično očekivati u

17570198

tekstualnim datotekama, npr. znak za prelazak u novi red, ali većina stavki u tabeli nema smisla na savremenim sistemima. Npr. ASCII vrijednost 7 (bel) bi trebala da proizvede zvučni signal na računaru, dok 6 (ack), 21 (nak), 1 (soh) itd. služe za kontrolu u prenosu podataka preko nekog telekomunikacijskog kanala. Ni ASCII vrijednost 0 koja nam označava kraj stringa nema previše smisla u datoteci. Ako naiđemo na neki od ovih karaktera, možemo biti sigurni da je datoteka binarna tj. da ne trebamo ove vrijednosti tumačiti kao znakove, nego kao brojeve, ili nešto treće. Neki programi za obradu teksta (Notepad) neće raditi ispravno sa datotekom koja sadrži ove "nelegalne" znakove, odnosno ako otvorite datoteku i zapišete je bez promjene nećete dobiti potpuno istu datoteku kao što je polazna i time ćete je praktično oštetiti.

U funkciji `fopen` možemo u mod (režim) dodati slovo "b" da označimo da je datoteka binarna. Npr. mod "rb" označava čitanje iz binarne datoteke, "wb" pisanje u binarnu datoteku i slično. No, sve funkcije za rad sa datotekom na nivou karaktera dostupne su i sa binarnim datotekama i rade vrlo slično. Razlika koju izaziva korištenje moda "b" je isključivo u interpretaciji podataka koja je ovisna o operativnom sistemu. Ustvari, na savremenim sistemima postoje samo dvije bitne razlike:

- ASCII vrijednost oznake za novi red (\n) razlikuje se između operativnih sistema. Na operativnom sistemu Windows za novi red se koriste dva karaktera čija su ASCII vrijednosti 13 i 10, na UNIX-like sistemima (Linux, Android, OSX...) koristi se samo 10, dok na nekim starijim operativnim sistemima (npr. stari MacOS) koristio se samo znak 13. Kod binarnih datoteka vrijednosti 13 i 10 se tumače kao i sve druge.

- Operativni sistem Windows uvodi oznaku EOF u ASCII tabelu sa kodom 26. Ako se datoteka otvori u tekstualnom modu, čitanje će se zaustaviti kada se u datoteci naiđe na ovaj znak i sve poslije njega će biti ignorisano, a ako je otvorite u binarnom modu vrijednost 26 će biti očitana kao i svaka druga. Na drugim operativnim sistemima ovo ograničenje ne postoji.

10.4.1 *Funkcije za rad sa binarnim datotekama*

Pored funkcija koje rade na nivou karaktera, za binarne datoteke definisane su dvije posebne funkcije: `fread` i `fwrite`. Ove dvije funkcije služe za *serijalizaciju podataka*. Serijalizacija je direktan zapis sadržaja memorije u datoteku. Kada zapisujete tekstualne podatke npr. funkcijom `fprintf`, vrši se transformacija podataka u tekstualni zapis, npr. broj 153 će biti transformisan u niz karaktera '1', '5', '3'. Kod binarne serijalizacije u datoteku će biti zapisana vrijednost onako kako je

1755198

predstavljena u memoriji računara.↑

↑ Poglavlje 3.1, str. 65.

Funkcija `fread` služi za čitanje podataka iz binarne datoteke, a njen prototip glasi:

✓ `fread`

```
size_t fread(void* pok, size_t vel, size_t broj, FILE *dat);
```

Prije svega da se podsjetimo da je tip `size_t` cjelobrojni tip koji koristimo da označimo veličinu memorije koju zauzima neki podatak. Prvi parametar funkcije `fread` je pokazivač na neki slobodan blok memorije u koji ćemo učitati podatke iz datoteke. Možemo uočiti da je pokazivač tipa `void*`, što označava da funkciju `fread` ne interesuje kojeg tipa je taj pokazivač. Tip podataka ustvari je određen drugim parametrom `vel` koji predstavlja veličinu jednog elementa odgovarajućeg tipa u bajtima. Npr. ako `pok` pokazuje na niz tipa `int`, `vel` će biti 4 jer tip `int` zauzima 4 bajta (na PC platformi).

Parametrom `broj` definišemo koliko članova datog tipa se nalazi u nizu, a četvrti parametar je pokazivač na datoteku iz koje čitamo podatke. Funkcija vraća broj uspješno učitanih elemenata iz datoteke npr. u slučaju da se u datoteci nalazi manje elemenata od očekivanog. Značenje parametara postati će jasnije na primjerima.

```
1 FILE* dat = fopen("binarna.dat", "rb");
2 char ime[100];
3 int ucitano = fread(ime, 1, 99, dat);
4 ime[ucitano] = '\0';
5 fclose(dat);
```

U liniji 1 datoteka je otvorena kao binarna datoteka za čitanje (mod "rb"). Deklarisali smo niz od 100 vrijednosti tipa `char` pod imenom `ime` (linija 2). Prvi parametar funkcije `fread` (linija 3) je ime niza, što je ujedno adresa prvog člana niza. Ova adresa je tipa `char*`, ali će ta adresa biti pretvorena u pokazivač tipa `void*`. Veličina jednog elementa je 1 bajt, što je količina memorije koju zauzima tip `char`. Učitaćemo iz datoteke maksimalno 99 elemenata, jer moramo ostaviti i jedno mjesto za \0. Čita se iz upravo otvorene datoteke `dat`.

Ako se u datoteci nalazi 99 ili više znakova, biće učitano prvih 99 znakova i kursor će se nalaziti na stotom znaku, a promjenljiva `ucitano` će imati vrijednost 99. Ako se nalazi manje od 99 znakova, svi će biti učitani, a njihov broj će se nalaziti u promjenljivoj `ucitano`. Dakle, promjenljiva `ucitano` je uvijek broj učitanih znakova i možemo je koristiti kao indeks mjesta gdje ćemo postaviti znak \0 što vidimo u liniji 4. Konačno, datoteku treba zatvoriti (linija 5).

Ovaj primjer je ustvari vrlo sličan pozivu funkcije `fgets` koju smo opisali na str. 332. Jedina bitna razlika je u tome što će se `fgets` zaustaviti na znaku \n, a `fread` će pročitati i taj znak i nastaviti dalje do 99-tog znaka ili kraja datoteke.

Pogledajmo zato bolji primjer: probaćemo bazu podataka opisanu u poglavlju 10.3 implementirati pomoću binarnih datoteka. Neka je

1/1/1/0/1/98

`struct Osoba` definisana kao na str. 339 te deklarisan niz od `KAPACITET` osoba:

```
#define KAPACITET 1000
struct Osoba {
  char ime[20], prezime[20], brtel[15];
};
struct Osoba imenik[KAPACITET];
```

Podatke iz binarne datoteke možemo učitati u imenik sljedećom funkcijom `ucitaj`:

```
1  int ucitaj(struct Osoba* imenik) {
2    FILE* dat = fopen("baza.dat", "rb");
3    if (dat == NULL) {
4      printf("Greska pri otvaranju datoteke.\n");
5      return 0;
6    }
7    int ucitano = fread(imenik, sizeof(struct Osoba), KAPACITET, dat);
8    fclose(dat);
9    return ucitano;
10 }
```

U liniji 2 otvorena je binarna datoteka, a zatim smo u linijama 3-6 provjerili da li je datoteka uspješno otvorena (npr. da li postoji). Linija 7 sadrži čitanje iz binarne datoteke. Obratite pažnju da ovdje *nema petlje*, jer funkcija `fread` odjednom učitava *čitav niz* u memoriju. Veličinu jednog člana niza dobijamo pozivajući operator `sizeof`.↑ Kada određujemo veličinu tipa podataka kao što je strukturni tip, stavljamo ga u male zagrade: `sizeof(struct Osoba)`. Maksimalan broj elemenata koje ćemo pročitati iz datoteke je veličina deklarisanog niza, a to je `KAPACITET`. Naravno u datoteci se može nalaziti manje osoba, što ćemo dobiti kao promjenljivu `ucitano` koju vraćamo iz funkcije.

↑ Poglavlje 3.2.5, str. 83.

Kada iz binarne datoteke želimo učitati samo jednu vrijednost (a ne niz), kao parametar `broj` naravno šaljemo vrijednost 1, ali trebamo obratiti pažnju da funkcija `fread` prima adresu, pa bismo recimo jedan realan broj čitali ovako:

```
float podatak;
int ucitano = fread(&podatak, 4, 1, dat);
```

Dakle dodali smo adresni operator↑ `&` ispred imena promjenljive `podatak` da bismo dobili njenu adresu. Vrijednost `ucitano` bi trebala biti 1 ako je sve ispravno.

↑ Poglavlje 7, str. 221.

Funkcija `fwrite` zapisuje podatke u binarnu datoteku, te ima isti prototip kao `fread`:

✓ `fwrite`

```
size_t fwrite(void* pok, size_t vel, size_t broj, FILE *dat);
```

Razlika je što sada `broj` nije deklarisana veličina niza nego stvarni broj elemenata koje ćemo zapisati u izlaznu datoteku. Funkcija vraća broj uspješno zapisanih elemenata koji se može razlikovati od `broj`, npr. ako je nestalo slobodnog prostora na disku ili došlo do neke druge

1/15/01/98

greške, ali obje te situacije možemo detektovati funkcijom `ferror`, kao i ranije. Pogledaćemo i primjer funkcije `zapisi` za binarne datoteke:

```
1  void zapisi(struct Osoba* imenik, int broj_osoba) {
2    FILE* dat = fopen("baza.dat", "wb");
3    if (dat == NULL) {
4      printf("Greska pri otvaranju datoteke.\n");
5      exit(1);
6    }
7    int zapisano = fwrite(imenik, sizeof(struct Osoba), broj_osoba, dat);
8    if (ferror(dat)) {
9      printf("Greska prilikom pisanja datoteke.\n");
10     printf("Zapisano je samo %d osoba\n", zapisano);
11   }
12   fclose(dat);
13 }
```

U linijama 8-11 vidimo primjer spomenute detekcije greške, a sve ostalo bi trebalo biti jasno samo po sebi.

Kao i sa funkcijom `fread` kod zapisivanja pojedinačnih promjenljivih u binarnu datoteku moramo voditi računa da šaljemo adresu promjenljive:

```
float pi = 3.1415926;
int zapisano = fwrite(&pi, 4, 1, dat);
```

10.4.2 *Pomjeranje kursora*

Kao što smo rekli, kursor se automatski pomjera kroz datoteku kako pozivamo odgovarajuće funkcije. Međutim, u nekim slučajevima ima smisla da ručno pomjeramo kursor, pogotovo ako koristimo neki od modova za ažuriranje (sa znakom `+`). Pomjeranje kursora se vrši pozivom funkcije `fseek`. Prototip ove funkcije glasi: ✓ `fseek`

```
int fseek(FILE* datoteka, long int offset, int origin);
```

Prvi parametar je pokazivač na datoteku, dok drugi parametar predstavlja pomak (u bajtima), a njegovo tačno značenje ovisi o trećem parametru. Treći parametar `origin` može biti jedna od tri konstante: `SEEK_SET` označava početak datoteke, `SEEK_END` kraj datoteke, a `SEEK_CUR` trenutnu lokaciju. Pomak se broji od lokacije `origin`. Tako npr. ako želimo postaviti kursor na treći bajt od početka, pisali bismo:

```
fseek(dat, 3, SEEK_SET);
```

a ako želimo da pomaknemo kursor za 15 bajta unaprijed (prema početku datoteke), pisali bismo:

```
fseek(dat, -15, SEEK_CUR);
```

Funkcija `fseek` vraća 0 ako je pomjeranje uspjelo, a 1 ako nije.

Funkcija `ftell` daje trenutnu lokaciju kursora, odnosno broj bajta ✓ `ftell` od početka datoteke. Interesantno je da standard ne specificira da je vraćeni broj ustvari broj bajta od početka, nego samo da je u pitanju

1/15/01198

vrijednost tipa `long int`, ali u pravilu jeste. Primjer korištenja `fseek` i `ftell` možemo vidjeti na sljedećem isječku koda koji dopisuje neki podatak na kraj datoteke, a zatim vraća kursor tamo gdje se nalazio prije ove promjene:

```
long pos = ftell(dat);

/* Dopisujemo podatak na kraj datoteke */
fseek(dat, 0, SEEK_END);
fwrite(&imenik[i], sizeof(struct Osoba), broj_osoba, dat);

/* Vracamo se na originalnu poziciju */
fseek(dat, pos, SEEK_SET);
```

Obratite pažnju da ovakvi pozivi funkcije `fseek` imaju smisla samo za binarnim datotekama. Oni će često raditi i sa tekstualnim datotekama, međutim C standard dozvoljava isključivo `SEEK_SET` kao moguću vrijednost `origin` za tekstualne datoteke, a kao vrijednost offseta dozvoljena je ili 0 ili vrijednost dobijena sa `ftell`.

10.4.3 *Tekstualne vs. binarne datoteke*

Da li koristiti tekstualnu ili binarnu datoteku? Ne postoji jedinstven tačan odgovor, obje vrste datoteka imaju svoje prednosti i nedostatke, a programer treba napraviti odluku na osnovu konkretnog slučaja primjene.

Osnovne *prednosti binarnih datoteka* su manje zauzeće prostora na disku i brži rad.

Binarne datoteke u pravilu zauzimaju manje prostora tj. može se optimizirati zapis tako da je manje zauzeće kako diska tako i interne memorije. Ova optimizacija svodi se na odabir najmanjeg tipa podataka koji je dovoljan za smještaj očekivanih podataka, npr. ako radimo sa brojevima [-100,100] nećemo koristiti tip `int`, dovoljan je `signed char`. U tom slučaju jedan broj na datom intervalu zauzima uvijek tačno 1 bajt. Ako koristimo tekstualnu datoteku, biće korišten po jedan bajt za svaku cifru broja kao i eventualni predznak minus, osim toga moramo navesti još i neki znak kojim razdvajamo taj broj od drugih podataka (npr. zarez, razmak i slično) tako da u najgorem slučaju zauzimamo čak 5 bajta za jedan broj.

Rad sa binarnim datotekama je brži, jer se podaci prilikom čitanja i pisanja direktno kopiraju sa diska u memoriju i obrnuto, ne dešava se konverzija ili interpretacija podataka. Naime, u prethodnom slučaju kada učitavamo broj iz tekstualne datoteke u promjenljivu funkcijom `fscanf`, ova funkcija će uraditi konverziju niza karaktera u `int` na način kako smo to radili sa stringovima.↑

↑ Poglavlje 8.2.2, str. 282.

No to je ujedno i nedostatak binarnih datoteka. Naime, binarne datoteke u pravilu su vezane za platformu (računarsku arhitekturu, operativni sistem) na kojoj su nastale, njihovo čitanje na drugim platfor-

1715701198

mama je otežano i ponovo zahtijeva korak konverzije.

Tekstualna datoteka se može otvoriti bilo kojim editorom npr. Notepad. Kada programer otvori tekstualnu datoteku u editoru može pokušati razumjeti njenu strukturu. Postoje standardni formati za tekstualno predstavljanje podataka kao što je spomenuti CSV, zatim XML, JSON i slični. Kod binarnih datoteka, ova ručna analiza je dosta teža jer morate poznavati kako su podaci interno predstavljeni u memoriji, koji tipovi podataka su korišteni.

Kada binarnu datoteku otvorite u tekstualnom editoru, dobićete ASCII reprezentaciju njenog sadržaja koja najčešće ne odgovara realnosti, npr. vrijednost tipa `int` će biti predstavljena kao četiri karaktera. Za rad sa binarnim datotekama koristimo *hex editor*, posebnu aplikaciju koja nam daje binarnu, decimalnu i heksadecimalnu predstavu (odatle naziv) svakog bajta, bloka od dva, četiri ili osam bajta, što programeru olakšava da prepozna npr. da su neke vrijednosti predstavljene tipom `int`. Naravno, programer mora imati barem grubu sliku šta očekuje da se u datoteci nalazi.

Kako napreduje tehnologija i dobijamo sve brže procesore i sve veće eksterne memorije, prednosti binarnih datoteka vremenom prestaju biti važne, dok prednosti tekstualnih datoteka postaju važnije nego ikada. Ipak, postoje specijalne oblasti primjene gdje je i dalje poželjno koristiti binarne datoteke.

Pitanja za razumijevanje gradiva

1. Šta se u putanjama na Windowsu označava početnim slovom C:, D: itd.?

2. Ako je tekući direktorij "C:\Program Files\test", a želimo pokrenuti program koji se nalazi na lokaciji "C:\Program Files\CodeBlocks\bin\gcc.exe", kakvu relativnu putanju trebamo koristiti?

3. Na Windows operativnom sistemu separator za putanje je backslash, a šta se koristi na Linuxu?

4. Koje sve vrste situacija mogu izazvati da funkcija `fopen` vrati nul-pokazivač?

5. Šta predstavlja mod (režim) "r+"? Moželi se u tom režimu pisati u datoteku?

6. Kojom naredbom postavljamo kursor na početak datoteke?

7. Zašto funkcija `fgetc` vraća vrijednost tipa `int` a ne tipa `char`?

17570198

8. Šta možemo koristiti umjesto provjere znaka `EOF`?

9. Šta vraća funkcija `fscanf`?

10. Šta su prednosti a šta nedostaci binarnih datoteka u odnosu na tekstualne?

11. Šta je prvi parametar funkcije `fread` i kojeg je tipa?

Zadaci za vježbu

Zadatak 4. *Napišite program koji čita tekstualnu datoteku i na ekranu ispisuje: broj redova u datoteci, broj riječi (pretpostavite da su riječi odvojene jednim ili više znakova razmaka), broj znakova, broj samoglasnika, broj velikih slova, broj cifara.*

Zadatak 5. *Data je datoteka "ispiti.txt" čiji redovi imaju strukturu* `brindexa predmet ocjena`*, gdje* `brindexa` *predstavlja broj indeksa studenta koji je polagao ispit,* `predmet` *je broj koji označava predmet iz kojeg je upisana ocjena, a* `ocjena` *je konačna ocjena koju je student osvojio. Primjer datoteke:*

```
12345 3 10
12345 4 9
12233 4 10
13579 2 8
13579 3 8
```

Prvi red označava da je student sa brojem indexa 12345 iz predmeta broj 3 (npr. to mogu biti Osnove računarstva) dobio ocjenu 10 itd. Ako je isti student položio više predmeta, jednostavno se dodaje još jedan red sa istim brojem indexa, ali drugim predmetom. Kreirajte ovu datoteku i napunite je nekim testnim podacima. Zatim napravite program koji na ulazu traži broj predmeta, te ispisuje prosječnu ocjenu i prolaznost (procenat studenata koji su položili tj. dobili ocjenu 6 ili više).

Zadatak 6. *Dopunite prethodni program tako da korisnik umjesto broja predmeta može unijeti tekstualni naziv predmeta. Da biste to postigli, kreirajte datoteku pod nazivom "predmeti.txt" koja sadrži imena predmeta zajedno sa brojevima. Kada korisnik unese naziv predmeta, najprije se provjerava da li predmet postoji u datoteci. Ako ne postoji, ispisuje se greška. U suprotnom ispisuju se podaci za taj predmet.*

1755/0198

Zadatak 7. *Dopunite posljednji zadatak sa vježbu iz prethodnog poglavlja* `struct Osoba` *tako da podatke drži u tekstualnoj datoteci. Datoteka se treba zvati "imenik.txt" i nalaziti u tekućem direktoriju, a format treba biti najprije kolona za ime, prezime i broj telefona. Cilj ove dopune je da se imena*

ne moraju ponovo unositi svaki put kada se pokrene program, nego da program pamti imena unesena posljednji put. Zatim napravite varijantu ovog zadatka u kojoj se koristi binarna datoteka.

17570198

A
Tablica operatora

Prioritet	Operator	Opis (primjer korištenja)	Poglavlje	Asocijativnost	Binarni?
1 (najveći)	`++`	Postfiksni inkrement (`a++`)	3.2.2	Lijevo	Ne
	`--`	Postfiksni dekrement (`a--`)	3.2.2		Ne
	`[]`	Indeksiranje nizova (`niz[i]`)	5		Da
	`.`	Pristup elementu (`a.b`)	9.3.4		Da
	`->`	Indirektni pristup (`a->b`)	9.3.8		Da
	`()`	Funkcijski operator (`f(x)`)	6		Da/Ne
2	`++`	Prefiksni inkrement (`++a`)	3.2.2	Desno	Ne
	`--`	Prefiksni dekrement (`--a`)	3.2.2		Ne
	`+`	Unarni plus (`+a`)	1.2.8		Ne
	`-`	Unarni minus (`-a`)	1.2.8		Ne
	`!`	Negacija (`!a`)	2.4		Ne
	`~`	Binarni komplement (1kk) (`~a`)			Ne
	`(tip)`	Kalup (`(int)a`)	3.1.8		Ne
	`*`	Operator dereferenciranja (`*a`)	7.1.3		Ne
	`&`	Adresni operator (`&a`)	7		Ne
	`sizeof`	Veličina u memoriji (`sizeof a`)	3.2.5		Ne
3	`*`	Množenje (`a*b`)	1.2.8	Lijevo	Da
	`/`	Dijeljenje (`a/b`)	1.2.8		Da
	`%`	Ostatak pri dijeljenju (`a%b`)	3.2.6		Da
4	`+`	Sabiranje (`a+b`)	1.2.8	Lijevo	Da
	`-`	Oduzimanje (`a-b`)	1.2.8		Da
5	`>>`	Desni shift (`a>>2`)		Lijevo	Da
	`<<`	Lijevi shift (`a<<2`)			Da
6	`<`	Manje od (`a<b`)	2.2	Lijevo	Da
	`<=`	Manje ili jednako (`a<=b`)	2.2		Da
	`>`	Veće od (`a>b`)	2.2		Da
	`>=`	Veće ili jednako (`a>=b`)	2.2		Da
7	`==`	Jednako (`a==b`)	2.2	Lijevo	Da
	`!=`	Nije jednako (`a!=b`)	2.2		Da

1750198

8	`&`	Binarno i (bitwise and) (`a&b`)		Lijevo	Da
9	`^`	Binarni xor (bitwise xor) (`a^b`)		Lijevo	Da
10	`|`	Binarno ili (bitwise or) (`a|b`)		Lijevo	Da
11	`&&`	Logičko i (`a&&b`)	2.4	Lijevo	Da
12	`||`	Logičko ili (`a||b`)	2.4	Lijevo	Da
13	`?:`	Kondicionalni operator (`a?b:c`)	3.2.3	Desno	Ternarni
14	`=`	Dodjela (`a=b`)	1.2.6	Desno	Da
	`+=`	Sabiranje i dodjela (`a+=b`)	1.2.8		Da
	`-=`	Oduzimanje i dodjela (`a-=b`)	1.2.8		Da
	`*=`	Množenje i dodjela (`a*=b`)	1.2.8		Da
	`/=`	Dijeljenje i dodjela (`a/=b`)	1.2.8		Da
	`%=`	Ostatak i dodjela (`a%=b`)	1.2.8		Da
	`<<=`	Lijevi shift i dodjela (`a<<=b`)			Da
	`>>=`	Desni shift i dodjela (`a>>=b`)			Da
	`&=`	Binarno "i" i dodjela (`a&=b`)			Da
	`^=`	Binarni xor i dodjela (`a^=b`)			Da
	`|=`	Binarno "ili" i dodjela (`a|=b`)			Da
15 (najmanji)	`,`	Zarez operator (`a,b`)	3.2.4	Lijevo	Da

Tablica A.1: Svi operatori u programskom jeziku C

17570198

B
Naredbe za ulaz i izlaz

Na početku knjige dali smo osnovne principe korištenja funkcija `printf` i `scanf` što smo smatrali dovoljnim. Kroz primjere u nastavku knjige mogli ste upoznati još neke načine korištenja ovih funkcija. U ovom prilogu pokušaćemo predstaviti *sve* što je bitno za ulaz i izlaz u programskom jeziku C kao svojevrsnu referencu za sve one situacije kada vam je potrebna tabela svih mogućih vrijednosti.

B.1 `printf` i `scanf`

B.1.1 Formatni stringovi (formati)

U tabeli ispod dati su svi poznati formati, tj. formatni stringovi koje možete koristiti u funkcijama `printf` i `scanf`.

Tablica B.1: Formatni stringovi za funkcije printf i scanf

Format	Značenje
`%d`, `%i`	Cijeli broj (**`int`**)
`%u`	Nepredznačni cijeli broj (**`unsigned`**)
`%o`	Cijeli broj u oktalnom brojevnom sistemu
`%x`, `%X`	Cijeli broj u heksadecimalnom brojevnom sistemu
`%f`, `%F`	Realan broj (**`float`**, **`double`**)
`%g`, `%G`	Ispisuje realan broj koristeći minimalan broj decimala
`%e`, `%E`	Realan broj u naučnoj notaciji npr. `3.9265E+2`
`%a`, `%A`	Realan broj u heksadecimalnom sistemu npr. `-0XC.90FEP-2`
`%c`	Znak (**`char`**)
`%s`	String (niz tipa **`char`**)
`%p`	Adresa u memoriji (pointer)
`%n`	Ništa se ne ispisuje/učitava, broj obrađenih znakova se upisuje u neku promjenljivu
`%%`	Znak postotka

175/0198

Kod formatnih stringova koji imaju varijantu sa malim i velikim slovom (npr. `%f` i `%F`) razlika je isključivo u tome da li će se vrijednosti `inf` (beskonačno) i `nan` (ne-broj) ispisati malim ili velikim slovima (`INF`

i `NAN`). Format `%f` očekuje da heksadecimalni broj bude dat malim slovima (npr. 0xf358a), a format `%F` velikim slovima (0xF358A).

Tabela je važeća za `printf` i `scanf`, ali postoji nekoliko sitnih razlika između načina kako ove dvije naredbe tumače pojedine formate. Spominjali smo ranije da za tip **`double`** u `scanf` trebamo koristiti format `%lf`, dok u `printf` takav format uopšte ne postoji nego se za sve realne tipove treba koristiti `%f`. Formati `%d` i `%i` su sinonimi u funkciji `printf`, a u funkciji `scanf` format `%i` dozvoljava unos oktalnih i heksadecimalnih brojeva tako što započnete unos nulom. Tako npr. ako je korisnik u promjenljivu tipa **`int`** unio 0123, sa formatom `%d` vrijednost promjenljive će biti 123, a sa formatom `%i` će biti 83 jer će se unos tumačiti kao oktalni broj pošto počinje nulom.

Naglasimo da format `%n` i u funkciji `printf` očekuje *adresu promjenljive* a ne promjenljivu (npr. `&broj`) jer će se u tu promjenljivu upisati broj obrađenih znakova, tj. u slučaju printf-a broj znakova ispisanih na ekran:

```
printf("%d plus %d daje %d%n\n", a, b, a+b, &duzina);
printf("Linija iznad je duga %d znakova\n", duzina);
```

B.1.2 Podspecifikatori

Pojmom **podspecifikator** označavamo sve ono što navodimo između znaka postotka i slova kako bismo pobliže uticali na format. Svi mogući podspecifikatori dati su u tabeli ispod.

Tablica B.2: Podspecifikatori formatnih stringova

Podspecifikator	Funkcije	Značenje
`-` (minus)	`printf`	Poravnanje ispisa ulijevo
`+` (plus)	`printf`	Obavezan ispis predznaka broja (čak i kad je to plus)
`.` (tačka)	`printf`	Preciznost realnog broja npr. `%5.2f`
`0` (nula)	`printf`	Popunjava predviđeni prostor nulama
`*` (zvjezdica)	`printf`	Prostor i/ili preciznost su dati kao parametri npr. `printf("%*.*f", 5, 2, x);`
`*` (zvjezdica)	`scanf`	Zanemaruje učitanu vrijednost
`hh`	Obje	Tip **`char`**
`h`	Obje	Tip **`short`**
`l`	Obje	Tip **`long`** ili **`double`**
`ll`	Obje	Tip **`long long`**
`L`	Obje	Tip **`long double`**
`j`	Obje	Tip `intmax_t`
`z`	Obje	Tip `size_t`
`t`	Obje	Tip `ptrdiff_t`

17/17/01 98

Možemo uočiti da su podspecifikatori podijeljeni u tri grupe. Prva četiri utiču na način ispisa na ekran (`printf`) a u funkciji `scanf` nisu dostupni. Primjeri njihovog korištenja su nam poznati od ranije.↑

↑ Poglavlje 1.2.4, str. 31.

Podspecifikator zvjezdica ima potpuno različito značenje u funkcijama `printf` i `scanf` pa smo ga naveli dvaput. U funkciji `printf` se može

koristiti i sa drugim formatima, pa tako sljedeća naredba ispisuje broj `br` u prostoru od `sir` znakova: `printf("%*d", sir, br);` Ovaj podspecifikator ima efekat "zanemarivanja" u funkciji `scanf`. Npr. sljedeća naredba:

```
scanf("%d%*d%d", &a, &b);
```

omogućuje korisniku da unese tri cijela broja razdvojena razmakom, pri čemu će prvi i treći biti uneseni u promjenljive `a` i `b`, a drugi broj će biti zanemaren.

Treća grupa podspecifikatora dostupna je u obje funkcije i služi da se pobliže odredi tip podatka, pri čemu osnovni format ostaje d (ili i) za cijeli broj, u za nepredznačni cijeli broj, a f za realan broj. Npr. format `%hhu` odgovara tipu **`unsigned char`**, format `%ld` tipu **`long int`**, dok je za **`long double`** uveden poseban format jer je ovaj tip uveden tek standardom C99. Ako u funkciji `printf` zaboravite ovaj podspecifikator, najčešće se neće desiti ništa strašno, no za `scanf` je praktično garantovano da neće raditi ispravno bez korektne oznake tipa.

B.1.3 Escape sekvence

Escape sekvence (eng. sequence)[1] nam služe kako bismo u funkciji `printf` na ekran ispisali neke znakove koji se ne mogu direktno navesti unutar navodnika. Npr. ako želimo da na ekran ispišemo dvije riječi jednu ispod druge, mogli bismo probati ovako:

```
printf("Zdravo
svijete");
```

Međutim, ovaj program se u pravilu neće moći prevesti, jer unutar string literala ne smijemo preći u novi red. Rješenje za ovaj problem glasi:

```
printf("Zdravo\nsvijete");
```

Oznaka za prelazak u novi red `n` se naziva *escape sekvenca za novi red*. Naziv je dobila po tome što počinje znakom "backslash" odnosno "obrnuta kosa crtica" `\` koji ovdje predstavlja "escape karakter"[2] tj. "znak za bijeg" jer njime "bježimo" iz normalnog procesa uređivanja teksta.

Nakon znaka "backslash" navodimo znak koji označava naredbu koju želimo da izvršimo, a dostupne naredbe date su u tablici B.3.

Obratite pažnju da neke od navedenih sekvenci nemaju nikakvog smisla na savremenim računarima. Zvučni signal najčešće nećete čuti, a `\f` i `\v` nemaju efekta na ekranu i korisni su samo za štampanje na papir. Što se tiče znakova `\b` i `\r` oni bi mogli imati opisani efekat, ali obratite pažnju da stringovi `"Zdravo"` i `"Zdd\bravo"` nisu jednaki iako će se jednako ispisati na ekranu. Sistemi za automatsko testiranje programa koji upoređuju izlaz vašeg programa sa očekivanim neće prepoznati ova dva teksta kao jednake.

[1] Na žalost u domaćoj literaturi nije šire usvojen neki standardan prevod za ovaj termin. Bukvalan prevod bi bio "nizovi za bijeg", međutim analogija koju ovo povlači već jako dugo vremena nije adekvatna i smatramo da bi takvo imenovanje bilo štetno za razumijevanje.

[2] Na starim tekstualnim terminalima, unos znaka za bijeg (najčešće pritiskom na tipku Esc) označavao je izlazak iz editora teksta i ulazak u komandni režim.

1/ELFZG198

Tablica B.3: Dostupne escape sekvence za naredbu printf

Sekvenca	Naziv	Značenje (efekat)
`\a`	Alert	Proizvodi zvučni signal ako to podržava izvršno okruženje
`\b`	Backspace	Brisanje znaka lijevo od unesenog
`\f`	Form feed	Prelazak na sljedeću stranicu
`\n`	Line feed	Novi red
`\r`	Carriage Return	Vraćanje na početak reda
`\t`	Horizontal Tab	Pomjera kursor na imaginarnu okomitu liniju na ekranu
`\v`	Vertical Tab	Pomjera kursor na imaginarnu vodoravnu liniju na ekranu
`\\`	Backslash	Ispisuje znak backslash
`\'`	Apostrof	Ispisuje znak jednostrukih navodnika
`\"`	Navodnik	Ispisuje znak dvostrukih navodnika
`\?`	Upitnik	Ispisuje znak upitnika
`\nnn`	ASCII	Ispisuje znak čija je vrijednost u ASCII tabeli `nnn` oktalno
`\xhh`	ASCII	Ispisuje znak čija je vrijednost u ASCII tabeli `hh` heksadecimalno
`\uhhhh`	Unicode	Ispisuje znak čija je vrijednost u Unicode tabeli `hhhh` heksadecimalno

Mnogi bi očekivali da će naredba `printf("abc\01234");` na ekranu ispisati samo "abc" jer znak \0 predstavlja oznaku za kraj stringa i sve nakon njega će biti zanemareno.↑ Ali ustvari na ekranu ćemo dobiti:

↑ Str. 264.

```
abc
34
```

Naime, `\012` će biti protumačeno kao oktalna vrijednost znaka u ASCII tabeli, a to je znak čija je ASCII vrijednost 10 koji je jednak prelasku u novi red (`\n`). Igrom slučaja `\0` ima vrijednost nule i u oktalnom sistemu, a to je nul-karakter. Stoga morate paziti ako iz nekog razloga navodite neki tekst nakon znaka `\0` da taj tekst ne počinje ciframa.

Escape znak `\u` (ili `\U`) će za efekat imati da će na to mjesto biti uvršten znak iz tabele Unicode znakova koja sadrži sve međunarodne karaktere, pa tako i kinesko pismo, naša slova itd. Npr. naredba `printf ("Grad Biha\u0106");` je jedini način da na ekranu ispišete tekst "Grad Bihać" koji je sukladan C standardu. Preduslov za to je da prozor za ispis koristi Unicode.

Ukoliko navedete neku escape sekvencu u pozivu funkcije `scanf`, ona će najvjerovatnije biti zanemarena.

B.1.4 Povratna vrijednost

17570198

Napomenimo još da i `printf` i `scanf` vraćaju vrijednost. Međutim, funkcija `printf` vraća broj *uspješno zapisanih znakova* na ekran, dok funkcija `scanf` vraća broj *uspješno učitanih promjenljivih*. Ovaj fenomen se može ilustrovati sljedećim isječkom koda:

```
int x,y,ps,pp;
ps = scanf("%d %d", &x, &y);
pp = printf("%d %d", x, y);
printf("\n ps=%d pp=%d\n", ps, pp);
```

Vrijednost koju vraća `scanf` (promjenljiva `ps`) bi uvijek trebala biti 2, jer ste unijeli dva broja u dvije promjenljive. Ona će biti manja od dva ako probate unijeti nešto što nije broj. S druge strane, ako ste unijeli npr. brojeve 123 i 45, funkcija `printf` (promjenljiva `pp`) će vratiti broj 6, jer je na ekranu ispisano šest znakova (3+2 cifre te jedan znak razmaka).

17570198

1757018 8

C
Napredne teme o pokazivačima

Prilikom pisanja poglavlja 7. "Pokazivači" nastojali smo da ovu tematiku, koja je ionako zahtjevna početnicima, ne činimo nepotrebno još težom. Zbog toga su izostavljene mnoge bitne i interesantne teme bez kojih priča o pokazivačima u programskim jezicima C i C++ ne bi bila kompletna. U ovom dodatku ćemo se vratiti nazad i dopuniti ovo poglavlje sa nekoliko interesantnih tema.

C.1 Dopune uvodnog poglavlja o pokazivačima

C.1.1 Napomene o adresnom modelu

Model memorije na kojem se program izvršava može varirati od računara do računara i programi ne smiju da ovise od konkretnog memorijskog modela (takvi programi nisu prenosivi između različitih računara). Danas je najkorišteniji *linearni memorijski model* gdje je memorija kontinualna i zna se šta prethodi i šta slijedi određenoj memorijskoj lokaciji, ali to opet ne znači da naš program treba da se oslanja na to! Memorija je podijeljena na ćelije, a svaka ćelija ima svoju adresu koja je unikatna. Na taj način je omogućeno jedinstveno pristupanje memorijskim ćelijama (na kojima se, između ostalog, nalaze i naše promjenljive). Iako je *pogrešno* pretpostaviti da su memorijske adrese brojčane prirode, radi jednostavnosti ćemo upravo to uraditi kako bismo objasnili kako adresiranje funkcionira.

Postoje arhitekture koje memoriju ne adresiraju brojčano, nego npr. asocijativno; također postoje drugi načini adresiranja. Npr. kod 8086 procesora korištenog u starim PC računarima, adresa je data sa parom brojeva segment:offset. Ovakav model naziva se matrični model.

17570198

C.1.2 Pokazivač na element iza kraja niza

Standard jezika C definiše pokazivač na element *tačno* iza niza (odnosno iza posljednjeg elementa niza) kao validan pokazivač, iako ne

pokazuje ni na jedan element niza. Postoji nekoliko razloga za ovu "anomaliju", a to su: jednostavnije prosljeđivanje cijelog niza funkcijama, dobivanje broja elemenata između dva pokazivača (sljedeće poglavlje), kao i to da se neki zadaci ne mogu jednostavno uraditi bez ovakve mogućnosti. Dalje, garantirano je da je pokazivač na element iza kraja niza "veći" od svih ostalih pokazivača na elemente toga niza (poređenje pokazivača i značenje poređenja ćemo objasniti kasnije). Konačno, blok dužine nula se jednostavno može predstaviti sa pokazivačem iza kraja koji je jednak pokazivaču na prvi element.

Prikažimo sad jednu "olakšicu" zahvaljujući tome što je takav pokazivač validan. Naime, ako želimo funkciji koja prihvata dva pokazivača (na početak i na kraj niza) proslijediti cijeli niz, morali bismo pisati sljedeće, pretpostavljajući da je `N` veličina niza:

```
#define N 50
...
/* fun zna da prihvata pokazivac na kraj */
void fun (int *, int *); /* prototip */
...
int niz[N] = {1, 2, 3, ...};
...
fun (&niz[0], &niz[N - 1]); /* cijeli niz */
```

Vidimo da moramo slati adresu posljednjeg elementa i pri tome indeksirati niz sa `N - 1`. Ukoliko ta ista funkcija `fun` radi sa pokazivačem na početak i pokazivačem *iza* kraja niza, onda bismo je jednostavno mogli pozvati kao

```
/* fun zna da prihvata pokazivac IZA kraja */
fun (&niz[0], &niz[N]);
```

Ili čak jednostavnije zahvaljujući činjenici da se ime niza upotrijebljeno samo za sebe implicitno pretvara u pokazivač na prvi element:

```
fun (niz, niz + N);
```

Iako je pokazivač iza kraja niza validan pokazivač u smislu da mu smijemo čitati vrijednost (odnosno adresu koju sadrži), njega nikako *ne smijemo* dereferencirati! Smijemo ga porediti, smijemo pokazivačkom aritmetikom "napraviti" takav pokazivač, ali dereferenciranjem takvog pokazivača program ulazi u nedefinisano ponašanje. Razlog je jednostavan: mjesto iza kraja niza *ne pripada* nizu a vjerovatno ne pripada niti našem programu, te je čitanje (i još opasnije: pisanje) strogo zabranjeno!

```
int niz[3] = {-4, 2, 5};
int *pok = niz + 3; /* iza kraja */
int broj = *pok; /* greska! */
*pok = 100; /* jos gore - greska */
```

C.1.3 Tip `ptrdiff_t` i "negativne" adrese

Pokažimo sada primjer kako iskoristiti razliku pokazivača da dobijemo broj elemenata između. Također, objasnimo kako se ispisuje tip koji predstavlja tu razliku i zašto je uopšte uveden.

Naime, znamo da adrese ne moraju biti numeričke što odmah dovodi do toga da ni razlika pokazivača ne mora biti numeričke prirode. Zbog toga je uveden alias za tip (`typedef`)↑ koji će predstavljati tu razliku i on je implementacijski definisan što znači da ovisi od kompajlera do kompajlera i platforme na kojoj se kompajlira. Vrlo vjerovatno je na 32 bitnim sistemima definisan kao `int` odnosno da je urađeno nešto poput sljedećeg `typedef int ptrdiff_t`, a za 64 bitne sisteme možda ovako `typedef long ptrdiff_t`. Naravno, ne smijemo se osloniti na ove pretpostavke i ovdje su pokaznog karaktera.

↑ Poglavlje 9.1, str. 299.

Prvo pitanje koje se može postaviti jeste *zašto je taj tip predznačen (odnosno, zašto može biti negativan)*? Prvo, razlika dva pokazivača je asimetrična pa nije isto `p2 - p1` i `p1 - p2` odnosno oduzimanjem dva pokazivača možemo dobiti negativnu vrijednost (čija apsolutna vrijednost je broj elemenata između ta dva pokazivača). Drugo, i adrese mogu biti `negativne`! To je slučaj kod x86-64 linija procesora (koju vjerovatno i vi koristite) gdje je adresni prostor podijeljen na dva potprostora (adrese su simetrične u odnosu na 0). Također, ne smijemo pretpostaviti da je razlika pokazivača tipa `int` i zbog toga što na 64 bitnim sistemima (gdje se mogu smjestiti mnogo veće količine podataka i iste adresirati) razlika dva pokazivača može biti veća od tipa `int` (odnosno, ne može "stati" u njega). Zbog svega ovog navedenog je i uveden navedeni `ptrdiff_t` koji se, po potrebi, implicitno može konvertovati u cijeli broj. Tip `ptrdiff_t` je definisan u standardnoj biblioteci "stddef.h".

```
int niz[] = {1, 3, -2, 5, 1, 6, 8};
int *p1 = &niz[0], *p2 = &niz[5];
/* mozda problem? 64bit? */
printf ("Izmedju je %d elemenata!\n", p2 - p1);
```

Ako ovo kompajliramo na 64-bitnom sistemu gdje su uključena sva upozorenja od strane kompajlera, vjerovatno će nas kompajler upozoriti da koristimo pogrešan formatni string `%d`. To je zato što na 64-bitnom sistemu (sa 64-bitnim kompajlerom) razlika između pokazivača je vjerovatno tipa `long`, a znamo da se `long` mora ispisivati sa `%ld`. Zato imamo sljedeće:

```
/* mozda problem? 32bit? */
printf ("Izmedju je %ld elemenata!\n", p2 - p1);
```

I onda ćemo na 64-bitnom sistemu kompajlirati bez upozorenja... a šta ako ovo kompajliramo na 32-bitnom sistemu? Dobit ćemo upozorenje da se tip `int` ne ispisuje sa `%ld`, nego sa `%d`. Ovo znači da

1/0/0/1/9/8

nemamo portabilan kôd! Rješenje je ispisati razliku (tipa `ptrdiff_t`) pomoću specijalnog formatnog stringa `%td`:

```
#include <stddef.h> /* za ptrdiff_t */
#include <stdio.h>
int main() {
  int niz[] = {1, 3, -2, 5, 1, 6, 8};
  int *p1 = &niz[0], *p2 = &niz[6];
  /* ovo ispod je OK (^_^) */
  printf ("Izmedju je %td elemenata!\n", p2 - p1);
  ptrdiff_t n = p2 - p1;
  printf ("n: %td\n", n); /* I ovo je OK */
  return 0;
}
```

Formatni string `%td` je sastavljen od para "velicina - tip" gdje je "velicina" zapravo `t` što je specijalno namijenjeno isključivo za tip `ptrdiff_t` (isto kao što je `z` namijenjen za `size_t` i `j` za `intmax_t`), a "tip" je cjelobrojnog tipa `d`. Vrijedno je napomenuti da je formatni string `%td` uveden u standard C99, a u ranijim standardima (C89 i C90) potrebno je izvršiti konverziju u cjelobrojni tip, što veći, to bolje (najbolje `long`), i treba biti pažljiv sa ovim konverzijama. I dalje je ispisivanje razlike pokazivača sa `%d` ili sa `%ld` nedefinisano ponašanje ukoliko *nismo izvršili konverziju*!

Konačno, izlaz programa će izgledati ovako:

```
Izmedju je 6 elemenata!
n: 6
```

C.1.4 *Konstantni pokazivači i pokazivači na konstantne objekte*

Već smo u prethodnim poglavljima vidjeli da promjenljive mogu biti konstantne (što je lingvistički paradoksalno: *konstantne promjenljive*) te su spomenute dvije vrste "konstantnosti": simboličke konstante tj. makro naredbe (pomoću `#define`), te konstantne promjenljive deklarisane modifikatorom `const`.

Također je spomenuto da promjenljive deklarisane na ovaj drugi način (pomoću `const`) i *nisu prave konstante*, pa recimo:

```
const int n = 100; /* nije "prava" konstanta */
/* ne moze ovo u C-u */
int niz[n];
```

Tačno je da vrijednost promjenljive `n` ne možemo direktno mijenjati kroz naš program, ali se ta promjenljiva može promijeniti na drugi način. Problem leži u tome što je `n` prava promjenljiva, odnosno ona je smještena u memoriju koju neko drugi (npr. hardver) može mijenjati (čak iako je to memorija samo za čitanje). U C++u to ipak nije slučaj i promjenljive deklarisane sa `const` su prave konstante,[1] a memoriju zauzimaju tek po potrebi (ako se npr. postave pokazivači da pokazuju na

[1] Kad god to mogu biti, tačnije kad su inicijalizirane konstantom ili konstantnim izrazom.

1755701198

njih). Kako je već spomenuto, ako želimo deklarisati niz sa veličinom koja je konstantna, moramo koristiti `#define` sintaksu.

Ako ipak želimo konstantne pokazivače ili pokazivače na konstantne objekte, zadovoljiti ćemo se ključnom riječi `const`, jer ćemo pomoću nje imati garanciju da mi, programeri (i oni koji koriste naše funkcije i naš napisani kôd), ne možemo mijenjati navedene objekte, a to je primarna svrha ovakvih pokazivača.

Prije nego pređemo na deklaraciju i svrhu konstantnih pokazivača i pokazivača na konstantne objekte, podsjetimo se da imamo slobodu u deklaracijama mijenjati redoslijed ključnih riječi. Obje konstrukcije ispod su potpuno ekvivalentne.

```
const int x = 10;
/* isto kao */
int const x = 10;
```

Ako želimo deklarisati **pokazivač na konstantni objekat**, to možemo uraditi ovako:

✓ Pokazivač na konstantni objekat

```
int broj = 5;
const int *pok = &broj;
/* ili */
int const *pok = &broj;
```

Efekat kojeg smo postigli jeste da je izraz `*pok` tipa `const int` (kao i ranije, kada dereferenciramo pokazivač dobijamo vrijednost tipa navedenog prije znaka zvjezdica u deklaraciji). Drugim riječima, pomoću pokazivača `pok` *ne možemo* mijenjati ono na šta on pokazuje, odnosno, u primjeru iznad, promjenljivu `broj`. Pokušaj izmjene bit će prijavljen kao kompajlerska greška, pogledajmo:

```
*pok = 3; /* greska! */
```

Naravno, i dalje je moguće mijenjati *promjenljivu* `broj` konstrukcijama poput `broj = 3` pa čak i pomoću novih pokazivača:

```
int *obicni_pok = &broj;
*obicni_pok = 4; /* OK */
```

Možete se pitati "zašto će mi, onda, ta mogućnost ako ipak mogu promijeniti promjenljivu?". Pitanje je "na mjestu", ali pokazivači na konstantne objekte se većinom koriste kao parametri funkcija. Npr. mi želimo napisati funkciju koja će ispisati niz cijelih brojeva. Jasno je da ta funkcija ne bi trebala da mijenja elemente toga niza. Shodno tome, mi ćemo taj parametar deklarisati kao "pokazivač na konstantni `int`.

```
void ispisi (const int *niz, int vel) {
  while (vel--)
    printf ("%i ", *niz++);
}
```

Na ovaj način onaj koji poziva funkciju može biti siguran da funkcija neće modificirati njegov niz, potrebno je samo da pogleda prototip funkcije i da utvrdi da prihvata pokazivač na konstantni `int`.

Također, čak i kada drugi neće koristiti naše funkcije, vrlo je korisno pokazivačke parametre deklarisati konstantnim – radi nas samih. Ako smo prilikom pisanja prototipa funkcije odlučili da ta funkcija ne treba mijenjati parametar, onda ćemo spriječiti *nehotično* modificiranje parametra ako taj parametar proglasimo konstantnim. Čak postoji mišljenje poznatih programera i organizacija da *sve* promjenljive treba proglasiti konstantnim, osim samo nekih koje ćemo baš koristiti kao promjenljive odnosno mijenjati njihove vrijednosti.

Napomenimo da, bez eksplicitne konverzije, *nije* moguće dodijeliti pokazivač na *konstantan* objekat pokazivaču na *nekonstantan (običan)* objekat. Kompajler za C će nas "samo" upozoriti, dok C++ kompajler neće ni kompajlirati. Ovo je i intuitivno jer ne bismo imali nikakve koristi od takvih pokazivača ako ih jednostavno možemo učiti da pokazuju na obične objekte koje možemo modificirati.

```
int broj = 100;
int *na_obican = &broj;
const int *na_konstantan = &broj;
na_obican = na_konstantan; /* Greska! */
```

Obrnuto je ipak moguće, jer garantira da nećemo praviti nikakvu štetu (od čitanja ovdje neće biti štete, ali od pisanja kao u pogrešnom kôdu iznad može biti).

```
int broj = 100;
int *na_obican = &broj;
const int *na_konstantan = &broj;
na_konstantan = na_obican; /* OK (^_^) */
```

Upoznali smo se sa pokazivačima na konstantne objekte, odnosno pokazivačima preko kojih *ne možemo* mijenjati ono na šta oni pokazuju. Ali ti pokazivači su i dalje obični u smislu da njih možemo modificirati:

```
int niz[] = {1, 2, 3, 4, 5};
const int *pok = niz;
/* ovo NE mozemo: */
*pok = 10; /* ne ide... */
/* ali ovo MOZEMO */
pok++;
pok = niz + 2; /* i ovo */
pok -= 1; /* i ovo */
--pok; /* i ovo */
```

Znači, iako je konstantno ono na šta pokazivač pokazuje (odnosno, ne možemo to mijenjati), pokazivač kao obična promjenljiva nije konstantan. To je, također, moguće postići (ako ništa, zbog definicije da je pokazivač obična promjenljiva, pa i on zaslužuje isti tretman). Deklaraciju **konstantnog pokazivača na (običan) objekat** radimo na sljedeći način:

✓ Konstantni pokazivač

```
int broj = 10;
int * const pok = &broj;
/* neki "priblize" zvjezdicu uz const: */
int *const pok = &broj; /* isto kao i iznad! */
```

1753098

Na ovaj način smo deklarisali konstantni pokazivač na običan `int` što znači da pomoću njega možemo mijenjati ono na šta on pokazuje, ali ne možemo mijenjati sam pokazivač.

```
int niz[] = {1, 2, 3, 4, 5};
int * const pok = niz;
pok++; /* greska */
--pok; /* greska */
pok = &niz[3]; /* greska */
...
```

Međutim, možemo mijenjati ono na šta on pokazuje:

```
*pok = 3;
(*pok)++;
...
```

Konstantne pokazivače, kao i obične konstantne promjenljive, moramo inicijalizirati odmah prilikom deklaracije (nije ispravno pisati npr. liniju `int * const p;`). Također, ne treba miješati deklaraciju konstantnog pokazivača i pokazivača na konstantni objekat kada se koristi druga notacija (prvo tip pa onda `const`), pogledajmo:

```
int *const p = &broj; /* konstantni pok na int */
int const *p = &broj; /* pok na konstantni int */
```

Ova vrsta pokazivača (konstantni) nema veliku primjenu. Ime niza upotrijebljeno samo za sebe je konstantni pokazivač, a sad vidimo razlog zašto smo mogli pisati

```
int *obicni_pok = niz;
```

a obrnuto nije bilo moguće. Navedimo dvije primjene konstantnih pokazivača (obje su preventivne prirode – sprječavaju potencijalne greške).

```
int x = 10;
int * const p = &x;
...
/* npr. hocemo da povecamo 'x' */
/* i slucajno napisemo ovako nesto */
*p++; /* kompajler javlja gresku */
/* i znat cemo da smo pogrijesili */
/* te cemo se ispraviti: */
(*p)++; /* OK */
```

Druga primjena je kada funkcijama prosljeđujemo adrese običnih promjenljivih (a ne elemenata niza). Na taj način sprječavamo slučajan ulazak u *nedefinisano ponašanje*.

```
void zamijeni (int * const a, int * const b) {
  /* kompajler prijavljuje gresku
  ako neko napise npr. */
  a++; /* greska */

  /* sjetimo se POGRESNE funkcije Zamijeni
  gdje je programer pisao ovako nesto: */
  a = b; /* odnosno "prvi = drugi;" */
  /* sto je rezultiralo da funkcija nije ispravna...
```

```
  da su parametri bili KONSTANTNI pokazivaci
  takva dodjela ne bi bila moguca!
  odnosno - pogram ne bi kompajlirao i
  ne bi bilo greske :) */
}
```

Objasnimo još zašto je nedefinisano ponašanje pisanje `a++` u funkciji iznad (u slučaju da pokazivačke parametre nismo deklarisali konstantnim – da jesmo ne bismo mogli ni pisati `a++`). Naime, recimo da smo proslijedili običnu promjenljivu, te da pokazivač `a` pokazuje na nju. Na šta će pokazivati `a` nakon `a++`? Ovo je nedefinisano ponašanje i program je neispravan:

```
int broj = 10;
int *pok = &broj;
pok++; /* Nedefinisano ponasanje! */
/* gdje sad 'pok' pokazuje!? */
```

Upoznali smo se sa pokazivačima na konstante objekte, upoznali smo se sa konstantnim pokazivačima na obične objekte... možemo li ovo spojiti i deklarisati **konstantni pokazivač na konstantni objekat**? Odgovor je potvrdan:

✓ Konstantni pokazivač na konstantni objekat

```
int broj = 10;
const int * const pok = &broj;
/* ili */
int const * const pok = &broj;
```

Iznad deklarisani `pok` pokazivač je takav da ne možemo mijenjati ni njega (npr. `pok++`), niti ono na šta on pokazuje (npr. `(*p)++` ili `*p = 4`).

Već smo spomenuli da pokazivače na konstantne objekte ne možemo dodjeljivati pokazivačima na obične (nekonstantne) objekte. Ipak, to je moguće, ali uz eksplicitno naglašavanje! To treba izbjegavati ukoliko nemamo baš dobar razlog.

```
int broj = 2;
int *obican;
const int *na_konstantan = &broj;
/* dobro paziti zasto ovo radimo! */
obican = (int *)na_konstantan;
```

Ovakve eksplicitne konverzije nisu preporučljive i one znače "*hej, kompajleru, ja ovo radim na svoju ruku, znam šta radim i snosit ću sve posljedice*" te onda kompajler ne prikazuje upozorenja (vođen programerovom preporukom da on "drži sve konce u rukama").

Rezimirajmo priču o konstantnosti pokazivača i onoga na šta oni pokazuju prikazujući sve tipove na jednom mjestu:

17570198

Deklaracija	Značenje
`int *pok`	**običan** pokazivač na **običan** `int`
`const int *pok`	**običan** pokazivač na **konstantan**
`int const *pok`	`int`

`int * const pok`	**konstantan** pokazivač na **običan** `int`
`const int * const pok` `int const * const pok`	**konstantan** pokazivač na **konstantan** `int`

C.2 *Dinamička alokacija memorije*

Prilikom deklarisanja promjenljivih, te promjenljive su kreirane *statički*. U ovom slučaju, statički znači *poznato za vrijeme kompajliranja*. Kada napišemo nešto poput sljedećeg

```
int a;
double b = 3.1;
char niz[] = {'O', 'R'};
```

ove promjenljive su kreirane i za njih je rezervirana memorije prije nego se naš program i počne izvršavati. Odnosno, kompajler "zna" koliko ima ovih promjenljivih, koje su veličine i kako ih izbrisati. Pored toga, sve navedene promjenljive su kreirane na steku (engl. *stack*).[2] Ovakva alokacija memorije gdje su sve promjenljive kreirane i poznate za vrijeme kompajliranja naziva se *statička alokacija memorije*. Moguće je zatražiti dodatnu memoriju od operativnog sistema i za vrijeme rada programa i to u količinama koje nisu poznate za vrijeme kompajliranja, nego ih npr. korisnik unese. Takva alokacija se naziva *dinamička alokacija memorije*.

[2] Statičke i globalne promjenljive nisu kreirane na steku, one su fiksno alocirane.

Prije nego se upoznamo sa mehanizmima dinamičke alokacije memorije i načinima kako da to izvedemo, najprije moramo predstaviti pokazivače na "sve tipove podataka", jer funkcije za dinamičku alokaciju upravo tako i rade – one unaprijed ne mogu znati kojeg tipa su promjenljive koje mi želimo alocirati. Sve ovo će biti jasno kasnije, nakon što se upoznamo sa pojmovima vezanim za dinamičku alokaciju.

C.2.1 *Generički pokazivači (`void *`)*

Naučili smo da možemo imati pokazivač na bilo koji tip. Kada imamo pokazivač na `int`, tada njemu možemo dodijeliti adrese cjelobrojnih promjenljivih (i to samo tipa `int`). Slično vrijedi za sve ostale pokazivače – nije ih moguće dodjeljivati ostalim pokazivačima (drugih tipova) bez eksplicitne konverzije tipa.[3] Ipak, postoji jedan tip pokazivača koji može pokazivati na *bilo koji tip*. Ti pokazivači se nazivaju **generički** pokazivači i tipa su `void *`.

[3] U C-u prevodilac će nam dozvoliti da međusobno pridružujemo pokazivače različitog tipa uz upozorenje.

✓ *Pokazivači koji mogu pokazivati na bilo koji tip nazivaju se **generički pokazivači**. Njihov tip je `void *` i on pojednostavljeno znači "ja sam pokazivač na memorijsku lokaciju, ali ne znam kakvi podaci se nalaze tu".*

✓ Generički pokazivač

1757019 8

Generički pokazivači ne vrše nikakve pretpostavke *šta* se nalazi tamo gdje oni pokazuju! Jednostavno pokazuju na memoriju. Zbog ovoga ne možemo vršiti nikakve operacije nad generičkim pokazivačima poput dereferenciranja ili pokazivačke aritmetike, barem dok ih ne "vratimo" u odgovarajući tip.[4]

Interesantno je spomenuti i da je ranije spomenuta konstanta `NULL`↑ tipa `void*` u C-u.[5] Ta makro konstanta je definisana u nekoliko standardnih biblioteka (`stdlib.h`, `stddef.h` i slično) a njena vrijednost je implementacijski definisana (odnosno, ovisi od arhitekture do arhitekture). Najčešće je definisana na sljedeći način

```
#define NULL ((void *)0)
```

Odnosno, broj 0 je konvertovan u pokazivač na `void` tip koji smo upravo opisali. Ideja ove konverzije je da se "kaže" da nul-pokazivač nije broj 0! Ipak, još jednom naglasimo da nul-pokazivač uopšte ne mora sadržavati vrijednost 0, nego sadrži neku vrijednost koja je specijalna za određenu arhitekturu. Npr. postoje arhitekture kod kojih je 0 *validna adresa*, poput CDC 6600 koji je adresu 0 koristio za indikaciju uspješnosti sistemskih poziva ili DEC VAX računara kojem je 0 bila obična adresa kao i svaka druga...

Da bismo opravdali potrebu za generičkim pokazivačima, ponudićemo jedan zanimljiv zadatak.

Zadatak 1. *Napisati funkciju "prekopiraj" sa 3 parametra koja kopira jedan niz cijelih brojeva u drugi. Izvorišni niz je ograničen pokazivačima na početak i iza kraja. Treći parametar predstavlja pokazivač na početak odredišnog niza. Pretpostavlja se da odredišni niz ima dovoljno prostora da smjesti sve elemente izvorišnog niza. Funkcija treba da vrati pokazivač iza kraja odredišnog niza*

U ovom zadatku trebamo raditi sa cijelim brojevima. Neka to bude tip `int`. Funkcija za kopiranje treba da izgleda prilično jednostavno, iteriramo kroz jedan niz i kopiramo elemente u drugi niz. Funkciji nije bio potreban 4. parametar (npr. pokazivač iza kraja odredišnog niza) zbog toga što *znamo* koliko ćemo elemenata smjestiti u odredišni niz (a to je isto elemenata kao što ima i izvorišni niz).

```
1  #include <stdio.h>
2  int *prekopiraj (const int *izv_p, const int *izv_k, int *odr) {
3    while (izv_p < izv_k)
4      *odr++ = *izv_p++;
5    return odr; /* vracamo iza kraja! */
6  }
7  int main() {
8    int niz1[] = {1, 2, 3, 4, 6};
9    int niz2[50]; /* dovoljno mjesta... */
10   int *iza_kraja = prekopiraj(niz1, niz1 + 5, niz2);
11   /* ispis niza 2*/
12   int *poc = niz2;
13   while (poc < iza_kraja)
```

[4] Ipak, neki prevodioci dozvoljavaju i to, pretpostavljajući da ako dereferenciramo `void*` on će se tumačiti kao `char*`.

↑ Poglavlje 7.1.2, str. 225.

[5] U C++-u NULL je tipa `int`, odnosno ima vrijednost 0.

```
14     printf("%i ", *poc++);
15   return 0;
16 }
```

Analizirajmo prvo prototip funkcije. Stavljeno je da su parametri pokazivači na konstante brojeve, jer je očito da naša funkcija neće modificirati niz. Naravno, `odr` nije mogao biti takav pokazivač jer smo taj niz modificirali. Dalje, možemo primijetiti važnost onoga što je funkcija vratila (pokazivač iza kraja odredišnog niza). Naime, mi smo tu informaciju kasnije (u `main` funkciji) iskoristili da saznamo koliko je prekopirano elemenata. Mogli smo odmah znati da je to isto elemenata kao što ima i `niz1`, ali to uvijek nije slučaj, npr. mogli smo imati funkciju koja izbacuje sve proste brojeve iz niza koji se unosi sa tastature. Mi tada *ne znamo* koliko će elemenata sadržavati odredišni niz, a tu informaciju možemo saznati zahvaljujući pokazivaču iza kraja (npr. možemo uraditi `iza_kraja - niz2` da saznamo broj elemenata ako je to potrebno).

Sada, neka je potrebno napisati funkciju za kopiranje realnih brojeva (npr. tipa `double`). Tada *opet* moramo pisati funkciju koja ima `isto` tijelo kao iznad napisana funkcija. Čak joj moramo dati novo ime, jer u C-u ne možemo imati više funkcija istog imena. Pogledajmo takvu funkciju, nazvanu `prekopiraj_double`:

```
double *prekopiraj_double (const double *izv_p, const double *izv_k, double
     *odr) {
  while (izv_p < izv_k)
    *odr++ = *izv_p++;
  return odr; /* vracamo iza kraja! */
}
```

Tijelo funkcije je *identično*, samo su promijenjeni tipovi parametara funkcije kao i povratni tip. Ako želimo sada funkciju koja će prekopirati niz elemenata tipa `char`, ista stvar nam se dešava. Tijelo funkcije je isto, samo se prototip mijenja!

Postavlja se pitanje *da li je moguće ovo nekako automatizirati*? Da imamo samo jednu funkciju koja će moći kopirati elemente proizvoljnih tipova? Odgovor je potvrdan, barem u ovom slučaju, a zašto odgovor nije univerzalno potvrdan objasnićemo kasnije. Napisat ćemo funkciju koja kao parametre prihvata `void` pokazivače (odnosno, tip `void *`), i onda ćemo moći kopirati proizvoljne nizove. Prototip takve funkcije bi mogao biti sljedeći:

```
void *prekopiraj (const void *poc, const void *iza_kraja, void *odr);
```

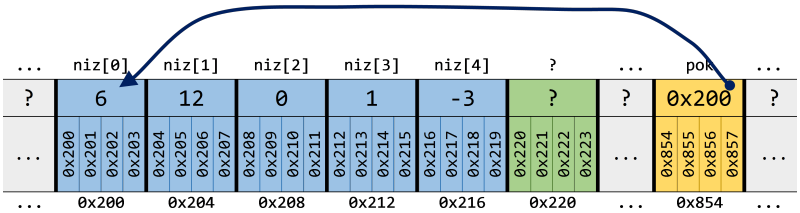
1757098

Sada ako "pohrlimo" napisati takvu funkciju naići ćemo na problem: znamo da `void` pokazivače *ne možemo* dereferencirati niti možemo primjenjivati pokazivačku aritmetiku nad njima! Neko može pomisliti da ćemo onda ove pokazivače konvertovati u npr. `int *` i onda primjenjivati pokazivačku aritmetiku nad njima. To bi možda moglo raditi,

ali je ipak u pitanju nedefinisano ponašanje, jer nije dozvoljeno konvertovati pokazivač jednog tipa u drugi a zatim ga dereferencirati.↑ Vrlo vjerovatno bismo dobili netačne rezultate npr. za `double` tip, jer je ovaj tip dvostruko veći od `int` tipa, pa bismo dodjeljivali samo "pola" elementa što će imati nepredvidljive posljedice.

↑ Poglavlje C.2.2, str. 372.

Kako, onda, riješiti navedeni problem? Rečeno je da je moguće napisati takvu funkciju, gdje je onda rješenje? Rješenje leži u garanciji C standarda da je veličina promjenljive tipa `char` tačno jedan bajt! Odnosno vrijedi `sizeof (char) == 1`. Zahvaljujući tome, svaka memorijska ćelija je tačno jedan bajt odnosno jedan `char`. Podsjetimo se slike C.1 gdje smo prikazivali pojednostavljeni model memorije.



Slika C.1: Svaka memorijska ćelija se može adresirati sa `char *`

Tada je rečeno da `p++` (gdje je `p` pokazivač na neki element niza) gotovo nikad ne znači povećanje adrese za 1, te je prikazano na slici da `p++` ukoliko `p` pokazuje na element niza tipa `int` znači povećanje adrese za 4 (u većini slučajeva). Ipak, kod pokazivača na `char` (tip `char *`) garantirano je da `p++` znači i povećanje adrese za 1. Zbog toga standard jedino za tip `char` garantira njegovu veličinu (1 bajt), kako bismo pomoću pokazivača na `char` mogli adresirati svaku memorijsku lokaciju koju i procesor može da adresira!

Zahvaljujući ovoj garanciji standarda, mi možemo prekopirati svaki bajt niza bez obzira kojeg su tipa njegovi elementi. Pogledajmo sljedeće ispravno rješenje.

```
void *prekopiraj (const void *poc, const void *iza_kraja, void *odr) {
  char *p = (char *)poc;
  char *k = (char *)iza_kraja;
  char *o = (char *)odr;
  while (p < k)
    *o++ = *p++;
  return o; /* vracamo iza kraja! */
}
```

17/57/01198

Funkcija `main` može ostati ista kao i u prethodnom rješenju. Ne trebamo vršiti konverziju u `void *` prilikom pozivanja funkcije jer, kako je već rečeno, `void` pokazivač može pokazivati na sve pa se izvrši implicitna konverzija, odnosno dovoljno je napisati samo sljedeće:

```
  int *iza_kraja = prekopiraj (niz, niz + N, niz2);
```

Može se postaviti pitanje "kada već konvertujemo u `char *` zašto onda ne stavimo da su parametri funkcije zapravo pokazivači na `char`"? Odgovor je jednostavan – ne postoji implicitna konverzija između različitih pokazivačkih tipova, nego se mora ručno konvertovati, dok kod pokazivača na `void` to se automatski radi. Da su parametri bili `char *` umjesto `void *` onda bismo morali pisati nešto poput sljedećeg:

```
int *iza_kraja = (int *)prekopiraj ((char *)niz,
                 (char *)(niz + N), (char *)niz2);
```

Pošto je kompajler za jezik C fleksibilan povodom ovih konverzija, dobili bismo samo upozorenje ukoliko ne izvršimo ove konverzije, ali u C++-u bi to bila sintaksna greška. Konverzija pokazivača se vrši isto kao i konverzija običnih promjenljivih: u zagradama ispred promjenljive ili izraza se navede željeni tip u kojeg želimo konvertovati. Više o ovim konverzijama (te njihovim ograničenjima) će biti rečeno u sljedećem potpoglavlju.↑ Spomenimo također da `void` pokazivačima možemo dodijeliti nul-pokazivač i tada se kaže da "ne pokazuje nigdje":

↑ Poglavlje C.2.2, str. 372.

```
void *p = NULL;
```

Sada nam može biti jasno zašto je makro konstanta `NULL` većinom definisana kao `#define NULL ((void *)0)` u potpoglavlju koje objašnjava nul-pokazivače.↑ Kako se svaki pokazivač može implicitno konvertovati u `void` pokazivač, tako će konstrukcija `p = NULL` raditi kada je `p` pokazivač na bilo koji tip! Također, na samom početku ovog poglavlja o pokazivačima, napisana je jedna linija (promjenljiva `x` je tipa `int`):

↑ Poglavlje 7.1.2, str. 225.

```
6 printf("Adresa od x je %x\n", &x);
```

Tada je rečeno da ovakav način ispisivanja nije ispravan jer pretpostavlja da su adrese numeričke prirode i spomenuto je da je `%p` *jedini* ispravan način za ispisivanje pokazivača (odnosno adresa). Ono što nas je tada sprječavalo da napišemo kako se ispravno ispisuju pokazivači jeste da tada nismo znali ništa o generičkim (`void`) pokazivačima, kao ni o konverzijama između pokazivača (što detaljnije objašnjava sljedeće potpoglavlje C.2.2). Naime, nismo mogli ovako ispisati adresu (kako bi neko pomislilo):

```
int x = 3;
printf ("Adresa: %p\n", &x);
```

Iako će ovaj kod raditi, on nije ispravan zato što nije po standardu (i predstavlja nedefinisano ponašanje sa nepredvidljivim posljedicama). Razlog za to je što standard traži da se pokazivači ispisuju sa `%p` formatnim stringom, ali da se prosljeđuju *isključivo* `void` pokazivači! Ispravno rješenje bi bilo da konvertujemo pokazivač prije prosljeđivanja funkciji `printf`:

```
int x = 3;
printf ("Adresa: %p\n", (void *)&x);
```

1757/0198

```
/* ili */
int *p = &x;
printf ("Adresa: %p\n", (void *)p);
/* ili cak: */
void *vp = &x;
printf ("Adresa: %p\n", vp);
```

Još da razjasnimo izjavu u vezi potvrdnog odgovora kada smo spominjali funkciju koja će raditi za sve tipove. Tada je bilo rečeno "odgovor je potvrdan, barem u ovom slučaju". Naime, nažalost, ne možemo jednostavno pisati univerzalne funkcije koje će prihvatati bilo koji tip. To smo mogli uraditi sa `prekopiraj` funkcijom jer ona nikad nije *analizirala šta se nalazi gdje pokazivač pokazuje*, nego je samo kopirala sadržaj. Da je trebalo napisati generičku funkciju (funkciju koja prihvata sve tipove) koja će ispisivati nizove proizvoljnih elemenata, to ne bismo mogli uraditi jer ne znamo kojeg tipa su elementi. Naravno, *ne smijemo* ispisati elemente sa `%c`, jer bi to ispisivalo besmislice. Slična je situacija sa funkcijama koje trebaju npr. odstranjivati neke elemente niza ili sortirati niz. Generalno, čim funkcija *mora* znati veličinu elementa niza, loša je ideja praviti takvu funkciju generičkom.

Napomenimo još da je moguće ipak (uz dodatne informacije) napisati navedene funkcije (a funkciju za sortiranje ćemo napisati kasnije kada se upoznamo sa pokazivačima na funkcije↑), ali se na prvom mjestu treba postaviti pitanje *treba li to raditi*? Ako funkcija ispisuje elemente, puno je jasnije i bolje napisati više funkcija sa različitim parametrima nego jednu funkciju koja će koristiti raznorazne trikove da sazna tip. Kasnije ćemo vidjeti kako se greške vrlo lahko mogu potkrasti kada koristimo generičke funkcije. Finalni savjet je: ne koristiti generičke funkcije i `void` pokazivače ako baš ne morate!

↑ Poglavlje C.4, str. 403.

C.2.2 *Konverzije pokazivača*

U dosadašnjim poglavljima smo intenzivno koristili raznorazne konverzije između različitih tipova. Neke od njih se dešavaju implicitno (programer ih ne mora navoditi), i takve konverzije su većinom "sigurne" (u smislu da vrlo vjerovatno neće prouzokovati greške). Takve konverzije su npr. sljedeće:

```
double d = 1; /* iz int u double */
int i = 3.14; /* iz double u int */
double d2 = i; /* iz int u double */
int i2 = d2; /* iz double i int */
```

Ovakve konverzije vrlo vjerovatno neće napraviti štetu (pogotovo kad se konvertuje iz *manjeg* u *veći* tip, npr. iz `int` u `double` ili iz `char` u `int`). Znamo da se dešava odsijecanje decimala kada konvertujemo iz `double` u `int`.↑ Znamo također probleme cjelobrojnog dijeljenja (npr. `33 / 7` nije ≈ 4.71 nego je 4, pa ako želimo ispravan rezultat moramo napisati npr. `33.0 / 7`). Sve ovo su problemi, ali ih možemo klasifi-

↑ Poglavlje 3.1.8, str. 76.

1715170198

cirati u "lakše" probleme. Tačno je da ih je nekad teško detektovati, ali najčešće će program raditi (ali neispravno), te ispravnost programa možemo provjeriti analizom finalnih rezultata. Neke konverzije su eksplicitne, npr. kada želimo izbjeći cjelobrojno dijeljenje, pa pišemo npr. `(double)broj/i`.

Ukoliko pričamo o pokazivačima i konvertovanjima između različitih tipova pokazivača, tada može doći do problema (i, prema Murphyjevom zakonu, *i doći će*). Otežavajuća okolnost je ta što je standard jezika C fleksibilan povodom ovih konverzija i dobiti ćemo samo upozorenje ukoliko ih napišemo (a ne sintaksnu grešku kao npr. u jeziku C++).

```
int i = 123;
double d = 32.13;
int *pi = &d; /* upozorenje */
double *pd = &i; /* isto... */
pd = pi; /* isto... */
short *sp = pd; /* samo upozorenje...
```

Po standardu, smijemo vršiti konverzije između pokazivača pod uslovom da *ne dereferenciramo* tako konvertovane pokazivače. Također, nikakva pokazivačka aritmetika nije dozvoljena sa njima jer kao i dereferenciranje rezultira nedefinisanim ponašanjem te su posljedice nepredvidljive. Konkretan primjer:

```
double d = 3.14;
int *p = &d; /* samo upozorenje... */
*p = 5; /* mijenja samo "pola" doublea? */
```

Iznad opisani primjer će jasno ilustrirati zašto ne smijemo dereferencirati (kao ni primjenjivati pokazivačku aritmetiku) na tako konvertovanim pokazivačima. Na većini arhitektura tip `int` je dvostruko manji od tipa `double`, pa bi posljednja linija iznad mijenjala samo prvu polovicu `double` broja. Pod pretpostavkom da se `double` brojevi čuvaju u IEEE754 formatu (na većini arhitektura je to slučaj), mi bismo konstrukcijom `*p = 5` izbrisali informacije o predznaku broja kao i njegovom eksponentu. Posljedice su, kako je već rečeno, nepredvidljive. Primjer će se kompajlirati (barem u C-u) i proći ćemo samo sa upozorenjem, a već nam je poznata činjenica da programeri jako često ignoriraju upozorenja (a evo dokaza da *ne trebaju to raditi*).

Možete se pitati zašto su onda takve konverzije i dozvoljene ako samo štete. Odgovor je da ne štete uvijek! Ako se pažljivo koriste mogu se uz pomoć konverzija rješavati neki hardverski ovisni problemi, a već smo se upoznali sa konverzijom u `char *` koja nam je pomogla da prekopiramo nizove. Naravno, konverzije su moguće i u C++-u, ali se moraju eksplicitno naglasiti tipa `int *p = (int *)double_broj;`. Tada se kompajleru kaže "znam šta radim, nemoj me upozoravati" i kompajler će "poslušati" programera.

Iako konverzije i međusobne dodjele pokazivača različitih tipova

1757098

nisu dozvoljene (odnosno rezultiraju upozorenjima), konverzije onoga na šta pokazivači pokazuju su vrlo često korištene. Npr.

```
int i = 5;
double d = 3.435;
int *pi = &i;
double *pd = &d;
pd = pi; /* ne valja! pokazivaci... */
*pd = *pi; /* savrseno legalno */
...
if (pd == pi) /* upozrenje.. ne valja */
if (*pd == *pi) /* ok! */
```

Savršeno je validno vršiti konverzije pokazivača u bilo koji tip te ponovo vratiti u originalni tip.[6] Znači, možemo konvertovati npr. `double*` u `int*` te kasnije ponovo vratiti iz `int*` u `double*` pod uslovom da ne vršimo dereferenciranje niti primjenjujemo pokazivačku aritmetiku nad pokazivačem dok je tipa `int *` (to se naziva "srednji tip" ili "pomoćni tip"). Svrha ovoga je historijske prirode i zadržana je zbog kompatibilnosti. Kako je već rečeno, konverzije u `void` pokazivač i obrnuto su također legalne (i ne zahtijevaju eksplicitnu konverziju). Još je jedna konverzija *uvijek* dozvoljena: dozvoljeno je konvertovati pokazivače u `char` pokazivač (pošto je on najmanja jedinica koju je moguće adresirati).

U potpoglavlju koje objašnjava konstantne pokazivače i konstantnost onoga na šta oni pokazuju↑ rečeno je da ne možemo legalno dodjeljivati pokazivače na konstantne objekte običnim pokazivačima odnosno da nije moguće sljedeće (dobit ćemo samo upozorenje, ali nije legalno ovo raditi)

```
int broj = 120;
const int *p = &broj;
int *obicni = p; /* ne valja! */
```

U jeziku C++ ovo neće ni kompajlirati. Ipak, moguće je "ukloniti" konstantnost pomoću operatora konverzije odnosno konverzije između jednog u drugi pokazivač:

```
int *obicni = (int *)p;
```

Ovakva konverzija pokazivača je legalna i u C-u i u C++-u jer je eksplicitno naglašeno da programer "zna šta radi" te da kompajler ne prijavljuje upozorenje ili grešku.

Vidjećemo kasnije da ponekad čak i naizgled *isti tipovi* pokazivača (čak i bez `const` modifikatora) zahtijevaju međusobnu konverziju (ako je ona moguća). Npr. pokazivač na prvi element niza i pokazivač na cijeli niz sadrže *istu* vrijednost (ista adresa), oba u svojem tipu sadrže `int`, ali je njihov ukupni tip različit. Više o ovome u potpoglavlju C.3.3.

Na kraju ovog potpoglavlja navedimo jednu anegdotu koju koriste C programeri. Anegdotu ćemo moći razumjeti zahvaljujući do sada naučenom gradivu (posebno značenju `void` pokazivača i šta znači kon-

[6] Ipak, postoji jedan izuzetak kada ne smijemo konvertovati pokazivače u proizvoljne tipove. Tada nemamo garancije da će ponovno konvertovanje u originalni tip biti validno odnosno moguće je da izgubimo informacije o tipu. To je kada konvertujemo pokazivače *različitih memorijskih poravnanja*. Neki tipovi su npr. poravnati u memoriji na 4 bajta, neki na 8... zbog toga ako konvertujemo pokazivače različitih tipova koji *nisu* isto poravnati, može doći do gubitka informacija. Zbog toga i iznad navedeni primjer (`double` pokazivač u `int` pokazivač) može da ne bude ispravan ukoliko poravnanje nije isto!

↑ Poglavlje C.1.4, str. 362.

1/15/0198

verzija pokazivača – u anegdoti navedena kao "cast" od engleske riječi za konverziju).

Stari C programeri nikad ne umiru. Oni samo budu pretvoreni u void pokazivače. (originalni engleski prijevod: *Old C programmers never die. They are just cast into void* *)

C.2.3 *Funkcije* `malloc` *i* `free`

Ovo poglavlje o dinamičkoj alokaciji memorije zahtijevalo je još dodatna potpoglavlja da razjasnimo gradivne elemente potrebne za dinamičku alokaciju memorije. Sada ćemo moći i dinamički alocirati memoriju te iskoristiti njene pogodnosti. Kako je već rečeno, dinamička alokacija memorije je proces traženja *dodatne* memorije od operativnog sistema, ukoliko takve ima (odnosno, ukoliko ima slobodne memorije). Taj proces je moguće izvršiti preko raznoraznih sistemskih funkcija koje ovise od operativnog sistema, ali puno je jednostavnije (i takav program će raditi na raznim operativnim sistemima) to uraditi pomoću standardizovanih funkcija dostupnih u jeziku C. Sve funkcije za dinamičku alokaciju memorije se nalaze u biblioteci "stdlib.h". Najkorištenija je `malloc` funkcija (skraćeno od eng. *memory allocate*), a ostale će biti spomenute u nastavku.

Pošto `malloc` funkcija ne može znati koji tip mi želimo alocirati (jer su u C-u podržani i korisnički definisani tipovi), ona kao parametar prihvata broj koji predstavlja *količinu memorije u bajtima* koju želimo zauzeti odnosno alocirati. Prateći poznatu izreku "primjer vrijedi 1000 riječi", prelazimo na primjer alociranja niza.

✓ `malloc`

```c
#include <stdio.h>  /* za printf i scanf */
#include <stdlib.h> /* za malloc i free */
int main() {
  int n, i;
  printf ("Unesite broj n: ");
  scanf ("%i", &n);

  /* sada mozemo pisati nesto poput "int niz[n]" */
  int *niz = (int *)malloc(n * sizeof (int));
  if (!niz)   {
    printf("Nema dovoljno memorije!\n");
    return 1; /* program nije uspjesno zavrsio */
  }

  /* sada je "niz" kao i obican niz! */
  for (i = 0; i < n; ++i)
    niz[i] = i;

  /* oslobodimo zauzetu memoriju */
  free (niz);
  return 0;
}
```

Detaljnim analiziranjem ćemo shvatiti sve segmente dinamičke alokacije memorije te se podsjetiti zašto su neke konstrukcije uopšte i do-

zvoljene. Prvo, uključena je biblioteka "stdlib.h" u kojoj se nalaze sve funkcije za zauzimanje (alokaciju) memorije te za njeno oslobađanje (dealokaciju).

Dalje, ovdje sada vidimo da ipak možemo imati "niz" sa proizvoljnim brojem elemenata (što do sada nije bilo moguće), pri čemu se taj broj elemenata unosi sa tastature i poznat je tek za vrijeme izvršavanja programa.

Funkcija `malloc` vraća pokazivač na `void` (odnosno, povratni tip funkcije `malloc` je `void *`), te smo zbog čitljivosti izvršili konverziju povratnog tipa u `int *`. Ovo nismo morali raditi u C-u (što je objašnjeno u prethodnom poglavlju↑) ali bismo u C++-u morali izvršiti ovu konverziju, te se inače preporučuje da se ovakva konverzija radi i zbog čitljivijeg kôda a i zbog prebacivanja programâ iz C-a u C++. Napomenimo, radi kompletnosti, da je `malloc` funkcija vraćala `char *` prije standarda C89 kojeg koristimo u ovom udžbeniku (uz par proširenja iz novijih standarda).[7]

↑ Poglavlje C.2.2, str. 372.

Već je spomenuto da `malloc` zahtijeva veličinu memorije u bajtima, a da ne bismo pretpostavljali da `int` zauzima 4 bajta (pa pisali `4 * n`), koristimo univerzalan i portabilan način pomoću `sizeof` operatora koji vraća broj bajta određenog tipa ili promjenljive (čak i izraza).

Pošto se može desiti da ne bude dovoljno slobodne memorije na raspolaganju, funkcija `malloc` će u tom slučaju vratiti nul-pokazivač. Stoga uvijek, ali baš uvijek, moramo provjeriti povratnu vrijednost `malloc` funkcije, te poduzeti određene akcije (najčešće se korisnik obavijesti da nema slobodne memorije i završi izvršavanje programa). Konstrukcija `if (!pok)` je obično poređenje pokazivača `pok` sa nul-pokazivačem. Znamo da je `if (promjenljiva)` uvijek zadovoljeno kada promjenljiva ima vrijednost različitu od 0, a `if (!promjenljiva)` je zadovoljeno jedino ako `promjenljiva` ima vrijednost nula. Slično je sa pokazivačima, pri čemu se pokazivač ne poredi sa brojem 0 nego sa nul-pokazivačem (u ovom slučaju ovo je specijalno poređenje, nije poređenje sa nulom!).

[7] Spomenuto je u poglavlju "Nizovi proizvoljne dužine" (5.1.5) da nikada ne smijemo pisati nešto poput sljedećeg: `int niz[n]` gdje je `n` obična promjenljiva (a ne makro konstanta). Kompajler za vrijeme kompajliranja mora znati veličinu niza kojeg će smjestiti na stek a to nikako ne može znati kada se ta veličina unosi za vrijeme izvršavanja programa. To što će ovaj program "raditi" je nesretna okolnost da neki kompajleri dodaju proširenja koja nisu po standardu. Ipak, uz pomoć dinamičke alokacije memorije možemo postići ovaj efekat – da imamo nizove proizvoljne dužine!

```
if (!pok)
/* isto kao */
if (pok == NULL)
... ... ...
if (pok)
/* isto kao */
if (pok != NULL)
```

Zahvaljujući činjenici da pokazivače možemo indeksirati,↑ pokazivač `niz` možemo koristiti isto kao i običan niz, što nam omogućava efekat niza sa proizvoljnom veličinom.

↑ Poglavlje 7.3.2, str. 240.

Sve što je od operativnog sistema zatraženo odnosno sva memorija koja je zauzeta mora se i osloboditi! Naime, kod dinamičke alokacije memorije kompajler ne može znati kada i koliko će alocirana memorija nama trebati, te je mi moramo ručno osloboditi. Također,

1/15/7/0/1/98

nakon što kompajler prevede naš program u izvršni kôd (izvršnu datoteku), kompajler više nema utjecaja na ono što se dešava za vrijeme rada programa. Sva dinamički alocirana memorija se nalazi na *memorijskoj gomili* (eng. *memory heap*), koja se uglavnom dodjeljuje cijeloj aplikaciji. Već je spomenuto da je drugi vid alokacije memorije statički i da se takva memorija smješta na stek (eng. *stack*). Stek je u većini slučajeva *manjeg* kapaciteta pa pomoću dinamičke alokacije memorije (na gomili) možemo alocirati nizove puno većih dimenzija. Radi poređenja, na Windows operativnim sistemima veličina steka je većinom oko 8MB (može se podesiti u kompajlerskim postavkama) na novijim računarima, dok je veličina gomile ograničena raspoloživom memorijom računara (najčešće RAM-om, ali standard ne definiše da se stek i gomila moraju nalaziti u RAM-u, pogotovo kod mikrokontrolera).

✓ Curenje memorije

U slučaju da zaboravimo osloboditi memoriju zauzetu naredbom `malloc`, a pokazivač na tu memoriju je prestao da postoji (npr. zato što se funkcija u kojoj smo alocirali memoriju završila), dolazi do *curenja memorije*. Naime, memorija je i dalje zauzeta ali mi više nemamo načina da je oslobodimo, odnosno kažemo da je "memorija iscurila". Curenje memorije je veliki problem u često korištenim računarskim programima, posebno onim pisanim u C-u i C++u. Simptom curenja je da, kako program stoji pokrenut, on postepeno zauzima sve više i više memorije, te se na kraju krahira, a u međuvremenu računar korisnika postaje usporen.

Ako sve ovo rezimiramo – programer je taj koji mora voditi računa o oslobađanju zauzete memorije! Ukoliko se zauzeta memorija ne oslobađa, program će u jednom trenutku vjerovatno krahirati, jer će nestati slobodne memorije i operativni sistem će zabraniti izvršavanje našeg programa. Pored toga, ostalim programima će biti uskraćena slobodna memorija, pa se i oni mogu naći u raznoraznim problemima.

✓ `free`

Funkcija `free` oslobađa zauzetu memoriju. Njoj se prosljeđuje pokazivač *na početak* niza kojeg treba osloboditi. Stoga ne smijemo izgubiti informaciju o početku dinamički alociranog niza:

```
int n, i;
...
int *niz = (int *)malloc (n * sizeof (int))
for (i = 0; i < n; ++i)
  scanf ("%i", niz++);
...
/* pokusava osloboditi memoriju na kraju niza! */
free (niz); /* GRESKA! posljedice nepredvidljive! */
```

1/157/01198

Rješenje iznad predstavljenog problema je kreirati još jedan pokazivač i postaviti ga da pokazuje na početak niza, te `free` funkciji proslijediti njega. Ako funkciji `free` proslijedimo nul-pokazivač, ona neće uraditi ništa. Također, nakon oslobađanja zauzete memorije pomoću `free` funkcije, pokazivač je nevalidan u smislu da ga ne smijemo deref-

erencirati niti čitati njegovu vrijednost (npr. ispisivati, porediti), ali mu smijemo dodjeljivati nove vrijednosti (npr. da pokazuje na neku već kreiranu promjenljivu ili čak na novu dinamički alociranu memoriju). Dobra je praksa nakon oslobađanja dodijeliti `NULL` pokazivaču kako ne bismo imali problem *dvostrukog brisanja* (kao u pogrešnom primjeru ispod):

```
int *p = (int *)malloc (256);
...
free (p);
...
/* mozda u petlji ili u rekurziji */
/* ili cak novoj funkciji */
free (p); /* dvostruko brisanje! */
/* posljedice nepredvidljive! */
```

Da smo nakon prvog brisanja postavili `p` da pokazuje na `NULL`, znamo da `free (NULL)` ne bi ništa uradio, odnosno ne bi bilo nikakve štete. Brisanje prostora koji nije zauzet proizvodi haos, odnosno uobičajena je izreka *"double delete is worse than no delete"*.

Naravno, `malloc` funkcija (i njoj slične) ne mora se uvijek koristiti za alokaciju niz, može se alocirati i samo jednu promjenljivu:

```
double *var = (double *)malloc (sizeof (double));
...
free (var);
```

Primjena dinamički alociranih promjenljivih je nešto rjeđa od dinamički alociranih nizova, ali se neki problemi ne mogu riješiti bez njih (npr. brojanje referenci kod optimizacije kopiranja). Jedna česta primjena je alokacija pojedinačnih čvorova u povezanoj listi ili stablu. Naravno, neko uvijek može alocirati niz od jednog elementa umjesto promjenljive.

Zahvaljujući činjenici da operator `sizeof` *ne evaluira* ono što se njemu proslijedi (odnosno, koristi samo informaciju o veličini izraza, ali ne izvršava izraz), moguće je pisati nešto poput sljedećeg, kao kraću (i univerzalniju verziju) iznad napisanog kôda (jer ne moramo tip pisati dva puta):

```
double *var = (double *)malloc (sizeof *var);
...
free (var);
```

Izraz `*var` bi izvan operatora `sizeof` bio pogrešan, jer u tom trenutku pokazivač `var` još uvijek nije inicijaliziran (ne pokazuje nigdje), štaviše nije niti deklarisan (tek se treba desiti njegova deklaracija!), ali unutar `sizeof` je potpuno legalan jer se neće evaluirati njegova vrijednost.

✓ `calloc`

Postoje još dvije standardne funkcije za alokaciju memorije: `calloc` i `realloc`. Prva se razlikuje od `malloc` što još dodatno *inicijalizira* zauzetu memoriju nulama (zbog toga i naziv "*clear alloc*"). Ona zahtijeva još jedan parametar koji predstavlja veličinu jednog elementa niza (iz razloga da zna kako će popuniti niz). Ima sljedeći prototip:

1/15/01/98

```
void *calloc (size_t broj_elemenata, size_t velicina_jednog_el);
```

Primijetimo da ova funkcija ne prihvata broj bajta nego broj elemenata, a broj bajta može saznati zahvaljujući drugom parametru koji predstavlja veličinu jednog elementa. Parametri su tipa `size_t` (kao i jedini parametar `malloc` funkcije), zbog toga što je garantovano da `size_t` najveći (pozitivan) broj koji predstavlja veličinu nekog objekta (`size_t` je alias za neki ugrađeni tip, najčešće **`unsigned int`** ili **`unsigned long`**).

✓ `realloc`

Funkcija `realloc` mijenja veličinu alociranog bloka u memoriji. Prihvata pokazivač na već postojeći (alocirani) blok i novu veličinu. Ukoliko specificiramo veću količinu nego što je već alocirano, funkcija će pokušati da proširi već postojeći blok. Ako to nije moguće, prekopirati će sav sadržaj bloka na drugu lokaciju gdje će rezervirati veći blok. U slučaju da se proslijedi manja veličina od trenutnog, bit će sačuvan sadržaj koji zadovoljava manju veličinu (ali je i dalje moguće pomjeranje sadržaja na novu lokaciju). Prototip funkcije je sljedeći:

```
void *realloc (void *stari_pok, size_t nova_velicina);
```

Ukoliko se kao prvi parametar (`stari_pok`) proslijedi nul-pokazivač, funkcija ima ponašanje identično kao `malloc` funkcija. Ukoliko proslijedimo 0 kao drugi parametar (`nova_velicina`), bit će oslobođena memorija na koju je pokazivao `stari_pok` pozivom `free` funkcije. Funkcija vraća adresu novog alociranog prostora koji ima veličinu `nova_velicina`, a sadrži sve podatke koji su se nalazili na adresi `stari_pok` (do najviše `nova_velicina` bajta). Ta nova adresa može biti ista kao i `stari_pok` (pogotovo npr. ako je nova veličina manja od stare), ali i ne mora, odnosno ne smijemo se na to osloniti.

U slučaju da funkcija ne uspije iz bilo kojeg razloga, vraća nul-pokazivač a sadržaj na kojeg je pokazivao `stari_pok` ostaje netaknut.[8]

[8] Ovo vrijedi za standarde C89 i C90. U novijim C99 i C11 standardnima povratna vrijednost `realloc` funkcije u slučaju da je drugi parametar nula je *nedefinisana*! Funkcija može vratiti nul-pokazivač ili neki drugi pokazivač kojeg ne smijemo dereferencirati.

Dakle, ispravna upotreba funkcije `realloc` bi mogla izgledati kao u sljedećem primjeru: recimo da imamo petlju u kojoj korisnik unosi potencijalno neograničen broj članova niza. Pošto ne možemo znati unaprijed koliko članova će korisnik unijeti, zauzećemo najprije 1000 elemenata:

```
int vel = 0, kapacitet = 1000, broj; /* Default kapacitet niza */
int* niz = (int*)malloc( sizeof(int) * kapacitet );
if (!niz) {
  printf("Nema memorije!\n");
  exit(0);
}
while(1) {
  printf("Unesite clan niza ili -1 za kraj: ");
  scanf("%d", &broj);
  if (broj == -1) break;
  niz[vel++] = broj;
  if (vel == kapacitet) {
    /* Dosegli smo kapacitet, prosirujemo */
    int* novi = (int*)realloc(niz, sizeof(int) * kapacitet * 2);
```

198

```
      if (!novi) {
        printf("Sva memorija je potrosena.\n");
        break; /* Prekidamo petlju */
      }
      niz = novi;
      kapacitet *= 2;
    }
  }
```

Napišimo sada dvije korisne funkcije kojima možemo automatizirati dinamičko alociranje i oslobađanje memorije. Funkcije ćemo nazvati `alociraj_niz` i `oslobodi_niz`. Ujedno napišimo i testni program te sve ovo spojimo u zadatak.

Zadatak 2. *Napisati program koji omogućuje unos proizvoljnog broja elemenata sa tastature u niz te obrće taj niz i ispisuje njegove elemente u obrnutom redoslijedu. Program mora raditi ispravno u slučaju grešaka.*

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int *alociraj_niz (int broj_el) {
4    int *pok = (int *)malloc (sizeof (int) * broj_el);
5    return pok;
6  }
7  void oslobodi_niz (int *niz) {
8    free (niz);
9  }
10 int main () {
11   int n = 0;
12   printf ("Unesite broj elemenata niza: ");
13   scanf ("%d", &n);
14   int *niz = alociraj_niz (n);
15   if (!niz) { /* kao i: if (niz == NULL) */
16     printf ("Alokacija nije uspjela!\n");
17     return -1;
18   }
19   printf ("Unesite elemente niza:\n");
20   int a = 0;
21   for (; a < n; ++a) {
22     printf ("niz[%i] = ", a);
23     scanf ("%i", &niz[a]);
24   }
25   printf ("Uneseni niz naopako ispisan: \n");
26   for (a = n; a --> 0; )
27     printf ("%i, ", niz[a]);
28   printf("\n");
29   /* Moramo osloboditi zauzetu memoriju: */
30   oslobodi_niz (niz);
31   return 0;
32 }
```

Sve u ovom rješenju je već objašnjeno, te je ono ovdje postavljen radi kompletnosti i afirmacije gradiva. Ipak, u 26. liniji možemo vidjeti simpatično "igranje":

```
26   for (a = n; a --> 0; )
```

Naredba `a --> 0` vizualno izgleda kao "sve dok `a` teži nuli" ili "dok `a` ne postane nula". Kako je ovo uopšte moguće? Zahvaljujući tome što

1/15/01/98

se izraz nalazi u srednjem dijelu petlje (koja predstavlja uslov izvršavanja). Kako C ignorira sve nepotrebne razmake, navedeni izraz se zapravo ljepše piše ovako `a-- > 0`. Značenje je jasno – poredi `a` sa 0, te umanji `a`. Ovo umanjivanje se izvršava prilikom svake iteracije petlje (čak i prije prve iteracije jer se mora provjeriti da li petlju ikako izvršiti). Zbog toga je `a` inicijaliziran na `n` (a ne `n - 1`). Navedena linija je identična sljedećoj:

```
for (a = n - 1; a >= 0; --a)
```

Primijetimo još da je `alociraj_niz` funkcija vratila pokazivač, ali on nije viseći! To znači da dinamički alocirana memorija "živi" sve dok je ne oslobodimo.

C.3 *Višestruki pokazivači*

Nekoliko puta do sada je spominjana činjenica da su i pokazivači obične promjenljive; zauzimaju memoriju, imaju vrijednosti... nalaze se na vlastitim adresama i slično. Da li je moguće deklarisati pokazivače da pokazuju na te pokazivači? Odgovor je potvrdan! Moguće je kreirati pokazivač na pokazivač i takav pokazivač se naziva **dvostruki pokazivač**. Naravno, tip takvog pokazivača nije sam `TIP *`, jer je to tip običnog (jednostrukog) pokazivača. Za dvostruke pokazivače imamo *dva* nivoa indirekcije, i zbog toga je potrebno *dvostruko dereferenciranje* da dobijemo ono na šta je prvi pokazivač pokazivao. Pređimo na primjer:

✓ Dvostruki pokazivač

```
int b = 104;
int *pb = &b;
int **ppb = &pb; /* pok na pok */

*pb = 200; /* mijenjamo b = 200 */
**ppb = 300; /* mijenjamo: *pb = 300 ili b = 300 */

printf ("b: %i, *pb: %i, **ppb: %i\n", b, *pb, **ppb);
```

Program ispisuje:

```
b: 300, *pb: 300, **ppb: 300
```

Moguće je u deklaraciji navesti i više od dvije zvjezdice, čime dobivamo **višestruke pokazivače**. Standardom jezika C mora biti podržano minimalno 12 indirekcija odnosno 12-orostruki pokazivač, ali rađeni su razni testovi koji su uspjeli kreirati pokazivače sa gotovo 100000 indirekcija, što je u tom momentu na toj mašini bilo ograničeno načinom na koji kompajler parsira[9] pokazivače (parsira ih desno-rekurzivno što znači da mora smještati na stek koji je ipak ograničen za trenutni program). Ipak, preporučuje se programerima da nikada ne koriste više od trostruke indirekcije, jer to dovodi do nečitljivog kôda i vrlo čestih grešaka (pogotovo kod dodjela pogrešnog tipa). Vidjeli smo

[9] *Parsiranje* (eng. parsing) je uobičajen termin za sintaksnu analizu. Parser je računarski algoritam koji prevodi kod u nekom jeziku u oblik pogodan za daljnju obradu npr. sintaksno stablo.

1755/2018

neke programe u kojima se "opravdano" koriste četverostruki pokazivači, ali to je rijetkost. Prikažimo primjere kako deklarisati višestruke pokazivače:

```
#include <stdio.h>
int main() {
  int b = 104;
  int *pb = &b;
  int **ppb = &pb; /* pok na pok */
  int ***pppb = &ppb; /* pok na pok na pok*/
  int ****ppppb = &pppb; /* 4-struki pok */
  /* ANALIZIRATI ovaj ispis! */
  printf ("%p == %p == %p \n",
           (void *)*ppb, (void *)pb, (void *)&b);
  /*
  "*ppb" je isto sto i "pb" a to je "&b"
  "**pppb" je isto sto i "pb" sto je isto kao i "&b"
  ...
  */
  printf ("b == ****ppppb == %i\n", ****ppppb); /* 104 */
  return 0;
}
```

Izlaz programa će varirati od operativnog sistema do operativnog sistema (u zavisnosti kako se ispisuju adrese). Također će izlaz programa varirati između dva različita pokretanja programa, pošto naše promjenljive budu kreirane na različitim (slobodnim) memorijskim lokacijama (adresama).

Ovako je izgledao izlaz programa kod pokretanja za vrijeme pisanja ovog pasusa (naravno, posljednja linija će uvijek ispisivati "104"):

```
0x7ffc36ccfa84 == 0x7ffc36ccfa84 == 0x7ffc36ccfa84
b == ****ppppb == 104
```

Možda sada može biti jasno zašto smo i mi u primjerima adrese obilježavali u notaciji "0*x*". To je zbog toga što će većina prevodilaca na većini arhitektura u gornjem programu ispisati adrese u heksadecimalnom formatu. Također, primijetite da je konverzija u `void` pokazivače bila obavezna prilikom njihovog ispisa.↑

↑ Poglavlje C.2.1, str. 367.

Ako vam još nije jasno sve ovo – ne brinite. Višestruki pokazivači su dosta apstraktni i potrebno je više vremena za njihovo savladavanje. Probajte više puta pokretati primjer iznad te ispisujte različite pokazivače sa različitim nivoom dereferenciranja (npr. `**ppppb`) kako biste shvatili "šta na šta pokazuje". Da bismo još bolje utvrdili ove indirekcije, prikazat ćemo kako to sve izgleda u memoriji. Prikazani primjer je sa dvostrukim pokazivačima te se čitaocima preporučuje da prateću sliku (sl. C.2) *pažljivo* analiziraju i shvate značenja strelica te vrijednosti u tabeli. Ne trebate smatrati da ste savladali višestruke pokazivače (čak ni dvostruke) ako ne razumijete zašto su vrijednosti u tabeli sa slike baš onakve (gledajući stanje u memoriji kao iznad tabele)!

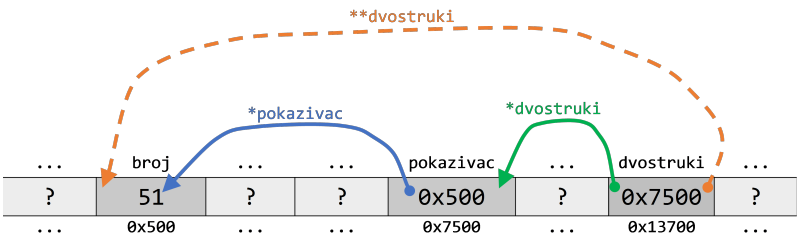
1757019 8

```
int broj = 51; /* &broj: 0x500 */
int *pokazivac = &broj; /* pokazivac: 0x500 */
```

```
int **dvostruki = &pokazivac; /* dvostruki: 0x7500 */
/* &pokazivac == dvostruki */
/* pokazivac == *dvostruki == &broj */
/* *pokazivac == **dvostruki == broj */
```

Za ovaj kôd iznad stanje u memoriji (pojednostavljeno) je kao na sljedećoj slici (sl. C.2). Tu se baš može pokazati da *slika vrijedi 1000 riječi*!

Slika C.2: Pojednostavljenje memorijskog stanja – dvostruki pokazivači



Izraz	Vrijednost	Isto kao...
broj	51	-
&broj	0x500	pokazivac
pokazivac	0x500	&broj
&pokazivac	0x7500	dvostruki
***pokazivac**	51	broj, **dvostruki
dvostruki	0x7500	&pokazivac
&dvostruki	0x13700	-
*dvostruki	0x500	pokazivac
****dvostruki**	51	broj, *pokazivac

Analizirajmo sliku C.2. Pokazivač `pokazivac` pokazuje na promjenljivu `broj` koja ima vrijednost 51. Dalje to znači da vrijedi
`pokazivac == &broj` odnosno `*pokazivac == broj == 51`
Dvostruki pokazivač `dvostruki` pokazuje na promjenljivu `pokazivac`, pa vrijedi `dvostruki == &pokazivac` odnosno
`*dvostruki == pokazivac == &broj`
Konačno, imamo izraz kao sljedeći koji obuhvata sve
`**dvostruki == *pokazivac == broj == 51`

Ako sada ponovo analiziramo sliku i prateći kôd, vjerovatno ćemo shvatiti šta se zapravo "dešava", odnosno kako funkcionišu ovi nivoi indirekcije. U slučaju da smo dvostruke pokazivače shvatili kako treba, analogiju možemo proširiti i na trostruke pokazivače, a dalje i na višestruke. Tačno je da "težina" praćenja ovoga rapidno raste, a to je i osnovni razlog zašto u programima ne trebamo imati (bez dobrog razloga) više od tri nivoa indirekcije (trostrukog pokazivača).

1/5/0198

C.3.1 *Neke primjene dvostrukih i trostrukih pokazivača*

Zašto su nam potrebni dvostruki (ili višestruki) pokazivači? Postoji više razloga, a mi ćemo spomenuti par za dvostruke i trostruke

pokazivače.

Dvostruke pokazivače koristimo kada funkcija treba da zamijeni vrijednost dva pokazivača; kada imamo pokazivač na neki element niza pokazivača (da, i to je moguće – niz pokazivača, vidjećemo kasnije svrhu), te kod kreiranja dinamički alociranih matrica (2D nizova), što je objašnjeno u poglavlju C.3.3. Trostruke pokazivače možemo koristiti da razmijenimo sadržaj dvostrukih pokazivača, te za kreiranje 3D nizova. Naravno, možemo ovako ići dalje, ali rijetko da nam trebaju četverostruki pokazivači osim da razmijenimo sadržaj trostrukih pokazivača (koji se ipak češće nalaze u programima i čiji sadržaj se često razmjenjuje). Dvostruki pokazivač se također koristi kao drugi parametar `main` funkcije (ukoliko se odlučimo pisati `main` funkciju da prihvata parametre "izvana").

Sve ovo je dovoljan razlog da pišemo o dvostrukim i trostrukim pokazivačima jer će nam oni vjerovatno trebati, pogotovo dvostruki pokazivači.

Zadatak 3. *U programu koji intenzivno koristi (jednostruke) pokazivače javila se potreba za razmjenom njihovog sadržaja. Napišite funkciju koja će upravo to raditi, te demonstrirajte njen rad u glavnoj funkciji.*

Već znamo kako razmijeniti sadržaj dvije promjenljive, pa slijedimo analogiju i sa pokazivačima. Prvo ćemo napisati isječak kôda koji bi razmijenio sadržaj dva pokazivača, a ukoliko nam je češće potrebna ta razmjena onda pišemo funkciju.

```
int a = 5, b = 7;
int *p1 = &a, *p2 = &b;
/* razmjena... */
int *temp = p1;
p1 = p2;
p2 = temp;
...
/* p1 sad pokazuje na 'b' */
*p1 = 1; /* isto kao: b = 1*/
```

Ukoliko želimo napisati funkciju (a to je traženo zadatkom), onda toj funkciji moramo kao parametre navesti upravo *dvostruke* pokazivače! Dvostruke zbog toga što ćemo funkciji prosljeđivati adrese pokazivača (kako bi funkcija mogla da razmijeni vrijednosti), a znamo da je adresa pokazivača tipa dvostruki pokazivač. Prikažimo kompletno rješenje zadatka.

```
1 #include <stdio.h>
2 void zamijeni (int **a, int **b) {
3   /* '*a' je tipa 'int *' */
4   int *temp = *a;
5   *a = *b;
6   *b = temp;
7 }
8 int main() {
9   int b1 = 5, b2 = 7;
```

```
10  int *p1 = &b1, *p2 = &b2;
11  zamijeni (&p1, &p2); /* prosljedjujemo adrese... */
12  *p2 = 100; /* mijenja 'b1'! */
13  printf ("%i\n", b1);   /* 100 */
14  printf ("%i\n", *p2); /* 100 */
15  return 0;
16 }
```

Sjetimo se da smo u poglavlju o konstantnim pokazivačima↑ govorili da bi pokazivači trebali biti konstantni kada god ih ne mijenjamo. U iznad napisanoj funkciji ne mijenjamo pokazivače nego samo ono na šta oni pokazuju, i to nakon jedne indirekcije. Nismo mijenjali vrijednosti promjenljivih `b1` i `b2` (a mogli smo iz navedene funkcije), nego `p1` i `p2`. Stoga bi ove pokazivače trebalo deklarisati konstantnim.

↑ Poglavlje C.1.4, str. 362.

Naravno, ako napišemo npr. `const int **a` to *nije* konstantni pokazivač nego je konstantno ono na šta on pokazuje (npr. promjenljiva `b1`).[10] Ali ako želimo deklarisati konstantan pokazivač, vidjećemo da kod dvostrukih pokazivača imamo dva nivoa konstantnosti: nakon prve (`pok`) i nakon druge (`*pok`) indirekcije! Ispravna deklaracija bi bila kao što je prikazano ispod:

[10] Tačnije, `a` je pokazivač na pokazivač `const int* b1`, koji je pak pokazivač na konstantni objekat.

```
void zamijeni (int ** const a, int ** const b)
```

Sa ovim smo zabranili izraze poput `a++` što smo i htjeli. Na taj način je zabranjeno mijenjanje "vanjskog" pokazivača.

Ipak, i dalje je moguće mijenjati na šta taj "vanjski" pokazivač pokazuje. Izraz `*a` je *opet* pokazivač, ali njega *možemo mijenjati* odnosno moguće je pisati `*a = &pok` ili `(*a)++`. Ukoliko i to želimo zabraniti, onda moramo i "unutrašnji" pokazivač proglasiti konstantnim. Pogledajmo primjer čisto radi kurioziteta:

```
int b = 100;
int *p1 = &b;
int ** const p2 = &p1;
/* sada... */
p2++; /* NE MOZE */
(*p2)++; /* Ali ovo MOZE! */
/* ako zelimo zabraniti oboje... */
int * const * const p3 = &p1; /* yep (^_^) */
/* sada... */
p3++; /* ne moze! */
(*p3)++; /* ni ovo ne moze! */
```

Navedeno razmatranje je prikazano čisto radi kompletnosti, te da onim programerima koji se žele držati (ispravne) prakse korištenjem konstantnih izraza pokažemo kako se to ispravno radi. Iz istog razloga ćemo još malo biti znatiželjni i napisati ***konstantni*** *dvostruki pokazivač (i "unutrašnji" i "vanjski") na* ***konstantan*** *cijeli broj*:

```
1   int a = 3;
2   /* i 'pok' ima 'const', razmisliti zasto! */
3   const int *pok = &a;
4   const int * const * const p = &pok; /* yaaaaay */
5   /* isto kao i iznad, samo 3 'const' zaredom */
```

1777198

```
6    int const * const * const p = &pok; /* (O_O) */
7    /* zabranjeno:
8    p++;
9    (*p)++; ili *p = &neki_pok...
10   **p = 10; (u cilju da promijeni promjenljivu 'a')
11   */
```

Ovo se može jednostavno protumačiti ako deklaraciju pokazivača čitamo zdesna nalijevo (od `p`). Pročitajmo 6. liniju (ona je najjasnija), sa desna: `p` je konstantni pokazivač na konstantni pokazivač na konstantni `int`.

Napomenimo još da ovdje treba biti posebno oprezan kod konverzija, jer možemo dozvoliti konverzije iz skroz različitih tipova što rezultira katastrofom! Npr. ako napišemo `int *obicni = (int *)p;`, ovdje smo konstantni dvostruki pokazivač na konstantni objekat pretvorili u obični jednostruki – i kompajler se, naravno, neće buniti!

C.3.2 *Nizovi pokazivača*

Podsjetimo se da možemo deklarisati niz elemenata bilo kojeg poznatog tipa, te da je deklaracija takvog niza oblika:

```
TIP ime[duzina];
```

pri čemu pojedinačni članovi niza imaju tip `TIP`. Možemo li onda imati i niz pokazivača? Zašto ne! Recimo na sljedeći način bismo deklarisali niz od 10 pokazivača na cijele brojeve (vrijednosti tipa `int`):

```
int* nizp[10] = {0};
```

Ovaj niz smo odmah inicijalizirali na nul-pokazivače. Upotreba konstante `NULL` ovdje nije bila potrebna jer nju svakako koristimo radi konverzije, a numerička vrijednost te konstante je 0.

Šta bismo mogli raditi sa ovakvim nizom? Pa možemo npr. postaviti članove tog niza da pokazuju na pojedinačne promjenljive ili na članove nekog drugog niza tipa `int`:

```
int niz[100];
int i, j=0, broj=0;
...
nizp[j++] = &broj;
for (i = 0; i < vel; i++)
  if (niz[i] == 0 && j < 10)
    nizp[j++] = &niz[i];
```

for petljom smo postigli da članovi niza `nizp` pokazuju na prvih 10 članova niza `niz` čija je vrijednost 0.

Ako imamo niz, onda možemo deklarisati i pokazivač na prvi član niza koji će biti tipa `int**`, odnosno *dvostruki pokazivač* tipa `int`, što nam omogućuje da naprijed pokazanu petlju realiziramo koristeći pokazivačku aritmetiku.

1755701198

```
1  int** p = nizp;
2  for (i = 0; i < vel; i++)
```

```
3     if (niz[i] == 0 && p < nizp + 10)
4       *p++ = &niz[i];
```

Proučimo malo ovaj kod. Pokazivač `p` u liniji 1 je postavljen na prvi član niza `nizp` (niza pokazivača). Rekli smo da ako imamo niz deklarisan sa `TIP ime[duzina]`, tada pokazivač na prvi član niza treba biti tipa `TIP*`, šta god bio `TIP`.↑ U našem slučaju `TIP` je `int*`, pa samim time i pokazivač `p` deklarišemo sa `int** p`. U liniji 3 koristimo već poznati trik sa adresom iza posljednjeg člana niza.↑ Pošto `nizp` ima 10 elemenata, adresa elementa nakon posljednjeg je `nizp + 10`.

↑ Poglavlje 7.3, str. 238.

↑ Poglavlje C.1.2, str. 359.

Obratimo sada pažnju na liniju 4. Poznato je da izraz `*p++` dereferencira pokazivač `p` i ujedno ga pomjera na sljedeći član niza.↑ Također znamo da derefenciranjem pokazivača tipa `TIP*` dobijamo vrijednost tipa `TIP` koja može biti i l-vrijednost, tj. može se nalaziti i sa lijeve strane dodjele. Dakle, pošto je naš pokazivač tipa `int**`, njegovim dereferenciranjem dobijamo izraz tipa `int*`, odnosno pokazivač na promjenljivu tipa `int`. Njemu pridružujemo adresu i-tog člana niza `niz`.

↑ Poglavlje 7.1.3, str. 226.

Kada smo govorili o pobrojanom tipu,↑ diskutovali smo različite strategije kako na ekranu možemo ispisati ime dana u sedmici, pa smo spomenuli da bi efikasno rješenje bilo da koristimo vrijednost pobrojanog tipa kao indeks u nizu imena dana deklarisanom na sljedeći način:

↑ Poglavlje 9.2, str. 300.

```
1  enum DanUSedmici { Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota,
      Nedjelja };
2  char imena_dana[7][12] = { "Ponedjeljak", "Utorak", "Srijeda", "Cetvrtak",
       "Petak", "Subota", "Nedjelja" };
3  ...
4  enum DanUSedmici dan = Subota;
5  printf("%s", imena_dana[dan]);
```

Prepoznajemo da je promjenljiva `imena_dana` matrica tipa `char`. Interesantna osobina matrica u C-u je da redove matrice možemo posmatrati kao nizove određenog tipa. U primjeru iznad, promjenljiva `dan` ima numeričku vrijednost 5 koja odgovara danu `Subota`. Izraz `imena_dana[5]` će se posmatrati kao niz tipa `char` čija je vrijednost "Subota", pa će se taj string ispisati na ekranu. Drugim riječima, na matricu možemo gledati i kao na *niz stringova*.

✓ Niz stringova

Matrica ima 7 redova koji odgovaraju imenima dana, a svaki string tj. svaki red matrice ima fiksno 12 elemenata, jer je 12 znakova potrebno da se predstavi najduže ime dana ("Ponedjeljak") zajedno sa pratećim obaveznim znakom \0. U drugim imenima dana preostali znakovi su neiskorišteni tj. matrica će zauzimati 84 bajta memorije. Uočavamo da se članovi ove matrice mogu mijenjati, što ne bi trebalo biti slučaj, odnosno imena dana bi trebala biti konstante. Stoga bi svakako bolje bilo u liniji 2 koristiti tip `const char`.

1755/0198

Umjesto toga, razmotrimo sljedeću deklaraciju, vrlo sličnu prethodnoj:

```
2  const char* imena_dana[7] = { "Ponedjeljak", "Utorak", "Srijeda", "
     Cetvrtak", "Petak", "Subota", "Nedjelja" };
```

Ovdje ćemo se podsjetiti još jedne programske strukture u C-u, a to je *pokazivač na znakovni literal*.↑ Naime, rekli smo da možemo deklarisati konstantni pokazivač koji pokazuje na literal koji je po definiciji konstantan. Nova varijanta niza `imena_dana` je upravo niz konstantnih pokazivača koji su usmjereni na literale sa imenima dana. Ostatak koda koji radi sa članovima ovog niza, bilo da ih indeksira ili da se koristi pokazivačka aritmetika, potpuno je identičan kao u prethodnom slučaju. Ako napišemo `imena_dana[dan]`, to je potpuno isto što i `*(imena_dana+dan)`, dakle dobićemo običan pokazivač tipa `const char*`. Također možemo deklarisati i pomoćni pokazivač `const char*p=imena_dana[dan]`, pa ga zatim proslijediti funkcijama kao što su `strlen`, `strstr` i slične.↑

↑ Poglavlje 8.1, str. 264.

↑ Str. 268.

Još jedna interesantna osobina je da se string literali ne moraju nalaziti niti na stacku niti na heapu (gomili). Na mnogim kompajlerima literali se ugrađuju u samu izvršnu datoteku (tj. njenu "data" sekciju), pa bismo mogli reći da ne zauzimaju nimalo memorije, mada to naravno nije tačno striktno govoreći. Mada uočavamo i da u verziji zadatka sa matricom tipa `char` imamo iste ove literale koji se zatim u deklaraciji matrice kopiraju na stack, tako da je memorija ustvari zauzeta dvaput.

Rad sa nizovima pokazivača je po mnogo čemu praktičniji od rada sa matricama. Pogledajmo isječak koda koji treba da sortira neki niz stringova, npr. niz imena u telefonskom imeniku, koristeći algoritam *selection sort* koji smo radili ranije.↑

↑ Poglavlje 5.2.5, str. 158.

```
1  for (i=0; i<n; i++) {
2    min=i;
3    for (j=i+1; j<n; j++) {
4      if (strcmp(niz[j], niz[min]) < 0)
5        min = j;
6    }
7    temp = niz[i];
8    niz[i] = niz[min];
9    niz[min] = temp;
10 }
```

Pošto je u pitanju niz stringova, u liniji 4 nismo mogli koristiti operator < za leksikografsko poređenje dva stringa, nego smo iskoristili funkciju `strcmp`.

Najprije se postavlja pitanje kako deklarisati promjenljivu `temp`? Prirodno to mora biti string, ali koliko velik? Dalje, kada bismo probali ovaj kod primijeniti nad matricom tipa `char`, u liniji 7 bismo imali sintaksnu grešku, jer se ne može red matrice pridružiti nizu tipa `char`. Ali, ako je `niz` deklarisan kao niz pokazivača npr. `char* niz[100]`, tada možemo `temp` deklarisati kao običan pokazivač `char* temp;`, i kod iznad radi bez problema! Trik je u tome što se u linijama 7-9 ne razmjenjuju

1777011198

stringovi nego pokazivači na njih, tj. samo se mijenjaju adrese u nizu pokazivača.

Još jedna prednost niza pokazivača je što kod deklaracije funkcije ne moramo fiksno navesti jednu od dvije dimenzije (o čemu ćemo govoriti ubrzo).

Kada govorimo o nizovima stringova i dvostrukim pokazivačima, red je da spomenemo i jedan vrlo poseban dvostruki pokazivač.

✓ Parametri funkcije `main` (argc i argv)

Naime, u čitavom ranijem tekstu knjige pretpostavljali smo da funkcija `main` nema parametara, no ovo ustvari nije tačno. Da budemo precizni, moguće su dvije varijante funkcije `main`: funkcija bez parametara, i funkcija sa dva parametra čiji je prototip kao u sljedećem programu:

```
#include <stdio.h>
int main(int argc, char** argv) {
  int i;
  printf("Program ima %d parametara.\n", argc);
  printf("Ti parametri su:\n");
  for (i=0; i<argc; i++)
    printf("%s\n", argv[i]);
  return 0;
}
```

Kada pokrećete vaš program iz komandne linije, imate mogućnost da mu proslijedite određene parametre. Recimo, ako se vaš program zove "program.exe", možete kucati:

```
C:\>program.exe p1 p2 p3
```

Sa naprijed datim C programom dobili biste sljedeći izlaz:

```
Program ima 4 parametara.
Ti parametri su:
program.exe
p1
p2
p3
```

Promjenljiva `argc` sadrži broj parametara sa kojim je program pozvan, tj. broj članova niza `argv`, pri čemu je prvi (nulti) član tog niza uvijek samo ime programa. Promjenljiva `argv` predstavlja niz pokazivača na stringove (nizove tipa `char`) koje možemo ispisati na ekran kao u primjeru iznad, ili ih možemo porediti, dodjeljivati drugim stringovima itd.

Pogledajmo još jedan primjer u kojem nizovi pokazivača mogu biti vrlo korisni.

Recimo da želimo definisati strukturu `Indeks` koja sadrži indeks jednog studenta. Pored imena i prezimena studenta i broja indeksa, ova struktura treba sadržavati sve ocjene studenta, što uključuje naziv predmeta i brojčanu ocjenu. Student može imati najviše 50 ocjena, a

17570198

ime predmeta nije duže od 100 znakova. Recimo dalje da želimo čuvati podatke o indeksima svih studenata na fakultetu (najviše 2000) u vidu niza struktura. Takva deklaracija bi mogla izgledati ovako:

```
struct Indeks {
  char ime[20], prezime[20], broj_indeksa[10];
  char imena_predmeta[50][100];
  int ocjene[50];
} fakultet[2000];
```

Koliko smo memorije zauzeli ovom deklaracijom? Jedna promjenljiva tipa `struct Indeks` zauzima ukupno 50 bajta za ime, prezime i broj indeksa, $50 \cdot 4$ bajta za ocjene, te $50 \cdot 100$ bajta za imena predmeta, čime dobijamo ukupno 5250 bajta. Ako pomnožimo to sa 2000 studenata, dobijamo 10.500.000 bajta odnosno preko 10 MB! Po svoj prilici ovoliku količinu podataka uopšte nećemo moći alocirati na stacku.

Zar ne bi bilo bolje da, umjesto 2000 studenata, u nizu držimo podatke za onaj broj studenata koliko ih zaista ima na fakultetu? U pomoć nam dolazi dinamička alokacija. Jedan čest pristup u programiranju je da se deklariše niz pokazivača tipa `struct Indeks`, a da se pojedinačni elementi dinamički alociraju.

Recimo, jedan dio programa za rad sa nizom struktura bez dinamičke alokacije bi mogao izgledati ovako:

```
  printf("Unesite: 1 - dodavanje indeksa... 0 - kraj programa\n");
  scanf("%d", &izbor);
  switch(izbor) {
    case 1:
      if (vel == KAPACITET) {
        printf("Nema vise mjesta u memoriji");
      } else {
        fakultet[vel++] = unesi_indeks();
      }
      break;
    ...
    case 0:
      exit(0);
  }
```

Korisnik unosi broj 1 za dodavanje novog studenta, 2 za promjenu, 3 za brisanje itd. Kod dodavanja pozivamo gotovu funkciju `unesi_indeks` koja će vratiti vrijednost tipa `struct Indeks`, koja će se zatim upisati u niz `fakultet` na indeksu `vel`, te će se `vel` uvećati.

Pri radu sa nizom pokazivača imali bismo ovako:

```
1  struct Indeks* fakultet[2000];
2  int vel=0, i;
3  ...
4  printf("Unesite: 1 - dodavanje indeksa... 0 - kraj programa\n");
5  scanf("%d", &izbor);
6  switch(izbor) {
7    case 1:
8      if (vel == KAPACITET) {
9        printf("Nema vise mjesta u memoriji");
10     } else {
11       fakultet[vel] = (struct Indeks*)malloc(sizeof(struct Indeks));
```

```
12        *fakultet[vel++] = unesi_indeks();
13      }
14      break;
15    case 0:
16      for (i=0, i<vel; i++)
17        free(fakultet[i]);
18      exit(0);
19    ...
20  }
```

Funkcija `unesi_indeks` i dalje vraća vrijednost tipa `struct Indeks`, ali se sada prilikom unosa novog indeksa najprije dinamički alocira memorija dovoljna za smještaj jednog indeksa (`sizeof(struct Indeks)`). Pokazivač na tu memoriju se uvrštava u niz na lokaciji `vel` (linija 11). Zatim se isti pokazivač dereferencira kako bi se u tu memorijsku lokaciju moglo upisati ono što je vratila funkcija `unesi_indeks` (linija 12).

Po završetku programa ne smijemo zaboraviti osloboditi svu memoriju koja je na ovaj način zauzeta, što se postiže petljom u linijama 16 i 17.

Prednost ovakvog načina rada je što je manipulacija nad pokazivačima znatno brža i manje zauzima memorije od rada sa strukturama. Recimo, ako bi se ukazala potreba da se ovaj niz sortira po nekom kriteriju, vršili bismo zamjenu pokazivača a ne kompletnih struktura.

C.3.3 *Pokazivači na višedimenzionalne nizove*

Rekli smo da red matrice možemo posmatrati kao niz. Možemo li se onda kretati kroz redove matrice koristeći pokazivačku aritmetiku?

Mogući su i pokazivači na višedimenzionalne nizove, a u ovom potpoglavlju ćemo se fokusirati na pokazivače na dvodimenzionalne (2D) nizove odnosno matrice. Već smo naučili da su su 2D nizovi sastavljeni od elemenata koji su tipa 1D nizovi odnosno da je dvodimenzionalni niz zapravo *niz nizova*.

Podsjetimo se da smo pod "pokazivač na niz" smatrali *pokazivač na prvi element niza*, pa smo pisali sljedeće:

```
int niz[] = {1, 2, 3, 4, ...};
int *pok = niz;
/* pokazivac na element... */
pok = &niz[3];
```

Da bismo "iterirali" kroz niz odnosno pristupali elementima niza, pokazivač sa kojim smo to radili je pokazivao na *jedan* element niza. Ista priča je i sa pokazivačima na dvodimenzionalne nizove: naši pokazivači na njih će pristupati *samo jednom* njihovom elementu a to je zapravo niz (vodeći se analogijom da je matrica niz nizova). Ovdje je važno da tip toga pokazivača mora biti *pokazivač na cijeli niz* ili drugačijom terminologijom izraženo *pokazivač na niz kao cjelinu*. Njega deklarišemo na sljedeći način:

✓ Pokazivač na niz kao cjelinu

```
1  TIP (*pok)[BROJ_ELEMENATA];
```

1755/0198

```
2  /* primjer: */
3  int (*pok)[10];
```

Iznad napisana linija sadrži deklaraciju pokazivača `pok` na cijeli niz koji ima 10 elemenata tipa `int`. Prije svega uočavamo zagrade oko izraza `*pok` što djeluje prilično neobično. Ustvari, primijetimo da bismo deklaracijom bez zagrada npr. `int *pok[10]` dobili ustvari *niz pokazivača* – koji smo upravo obradili u prethodnom potpoglavlju.

Nakon toga slijede uglaste zagrade u kojima je navedena veličina niza, koja se mora ispravno navesti. Odnosno, kod višedimenzionalnih nizova moramo navesti sve dimenzije osim prve, zbog načina na koji se višedimenzionalni nizovi interno predstavljaju u memoriji. Naš pokazivač `pok` *mora znati* koliko niz na kojeg pokazuje ima elemenata kako bismo pomoću pokazivačke aritmetike mogli da pređemo na npr. sljedeći element matrice odnosno – sljedeći niz.

Isto pravilo važi i kada želimo poslati višedimenzionalni niz u funkciju. Naime, objasnili smo da kada običan niz probamo poslati u funkciju, ono što ćemo dobiti kao parametar je ustvari pokazivač na prvi član niza.↑ U svemu tome, navođenje uglastih zagrada kao parametra funkcije nam samo sakriva (ili uljepšava, kako hoćete) ovu činjenicu. Recimo, ako bismo željeli napraviti funkciju koja ispisuje matricu cijelih brojeva na ekranu, ona bi mogla izgledati ovako:

✓ Slanje višedimenzionalnog niza u funkciju

↑ Poglavlje 7.3.4, str. 245.

```
void ispisi_matricu (int mat[][10], int n, int m) {
  int i,j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++)
      printf("%5d", mat[i][j]);
    printf("\n");
  }
}
```

Obratite pažnju na parametar `mat[][10]`: prva dimenzija je izostavljena (kao što smo radili i ranije), ali već drugu moramo navesti, te će ova funkcija raditi *samo za matrice širine 10*. Drugim riječima, parametar `m` nam nije bio ni potreban jer je širina fiksno 10 (osim ako imamo neku logičku širinu↑ matrice `m` koja je manja od 10). Stvarna istina je da je promjenljiva `mat` pokazivač na niz kao cjelinu, te ako pozovemo funkciju sa matricom 10x10 kao argumentom, vrijednost parametra `mat` će biti pokazivač na prvi red ove matrice.

↑ Str. 142.

Zbog ovog ograničenja u C-u se mnogo češće koriste nizovi pokazivača od matrica (ili generalno višedimenzionalnih nizova): sa matricama je nemoguće pisati univerzalne funkcije koje rade sa matricama zaista proizvoljnih dimenzija. Ovaj problem rješavamo tako što višedimenzionalne strukture predstavljamo kao nizove pokazivača, o čemu ćemo više vidjeti u sljedećem potpoglavlju.

Pogledajmo primjer rada sa pokazivačem na red matrice:

1755018 98

```
1 #include <stdio.h>
2 int main() {
```

```
3   int mat[2][3] = {{1, 2, 3}, {4, 5, 6}};
4   int (*pok)[3] = &mat[0]; /* na 1. element tj. niz*/
5   pok++; /* pomjerio je 'pok' na drugi niz */
6   /* pok pokazuje na 2. red matrice */
7   printf ("(*pok)[0]: %i\n", (*pok)[0]); /* 4 */
8   printf ("(*pok)[1]: %i\n", (*pok)[1]); /* 5 */
9 }
```

Pokazivač `pok` smo inicijalizirali da pokazuje na prvi element matrice, a to je prvi red (odnosno prvi niz). Kako već znamo da se ime niza upotrijebljeno samo za sebe konvertuje u pokazivač na prvi element, to isto vrijedi i za matrice pa možemo pisati i ovako:

```
  int (*pok)[3] = mat;
```

Naravno, i dalje ne možemo pisati `&mat`, jer je to pokazivač drugog tipa, odnosno pokazivač na matricu kao cjelinu (o čemu ćemo govoriti u sljedećem potpoglavlju). Adresa prvog niza matrice je ista kao i adresa prvog elementa prvog reda matrice, odnosno *ista* je adresa za `&mat[0]` kao i `&mat[0][0]`, ali tipovi pokazivača koji sadrže tu adresu *nisu isti*. Također, pokazivač `pok` se ne može konvertovati u dvostruki pokazivač (a to je najčešća greška početnika i na nju ćemo ukazati u nastavku), jer oni nisu istog tipa, te međusobno nisu kompatibilni.

U 5. liniji imamo pomjeranje pokazivača koje i ovdje znači pomjeranje za jedan element, ali u našem slučaju je to cijeli niz (a ne samo jedan običan element tipa `int`). Zahvaljujući upravo tome, izraz `p++` će pomjeriti pokazivač da pokazuje na sljedeći red matrice (i ovo je osnovni razlog zašto taj pokazivač mora znati dužinu niza – da bi znao koliko elemenata da "preskoči")!

Linije 7 i 8 pristupaju elementima drugog reda matrice. Prvo moramo dereferencirati pokazivač (i ovdje su potrebne zagrade oko izraza dereferenciranja, jer bi izraz `*pok[1]` značio da dereferenciramo drugi element niza pokazivača, odnosno bilo bi jednako izrazu `*(pok[1])`). Nakon dereferenciranja dobili smo niz odnosno cijeli red matrice, te je potrebno odrediti kojem elementu toga niza (reda matrice) želimo pristupiti.

Također, kako znamo, izraz `*pok` je isto što i `*(pok + 0)`, tako da je umjesto izraza u linijama 7 i 8 moguće pisati i `pok[0][0]` odnosno `pok[0][1]`. Navedene linije su potpuno ekvivalentne sljedećim:

```
8   printf ("pok[0][0]: %i\n", pok[0][0]); /* 4 */
9   printf ("pok[0][1]: %i\n", pok[0][1]); /* 5 */
```

Ovo nam omogućava da pomoću pokazivača `pok` koji pokazuje na 2. red matrice pristupimo 1. redu matrice, ili bilo kojem drugom. Pogledajmo kako ćemo ispisati prva tri elementa prvog reda matrice:

```
1 #include <stdio.h>
2 int main() {
3   int mat[2][3] = {{1, 2, 3}, {4, 5, 6}};
4   int (*pok)[3] = &mat[0]; /* na 1. element tj. niz*/
5   pok++; /* pomjerio je 'pok' na drugi niz */
```

1.1.7.0.1.98

```
  /* pok pokazuje na 2. red matrice */
  /* pristupamo PRVOM redu ovako: */
  printf ("Prvi red glasi:\n"); /* ispis: 1 2 3 */
  printf ("%i %i %i\n", pok[-1][0], pok[-1][1],
                                    pok[-1][2]);
  return 0;
}
```

Vidjeli smo da i ove pokazivače možemo indeksirati sa negativnim vrijednostima,↑ čime dobivamo efekat da relativno pristupamo drugim elementima matrice (odnosno, drugim nizovima). Analogija je ista kao i za jednodimenzionalne nizove, ako je `p` pokazivao na 5. element niza, sa `p[-2]` ćemo dobiti 3. element niza; tako je i u primjeru iznad, sa `pok[-1]` dobivamo prethodni element matrice odnosno prethodni niz!

↑ Poglavlje 7.3.2, str. 240.

Ovakve pokazivače međusobno možemo dodjeljivati ako i samo ako su istog tipa tj. pokazuju na nizove sa istim brojem elemenata istog tipa!

```
  int (*p1)[10];
  int (*p2)[9];
  p1 = p2; /* ne valja, upozorenje u C, greska u C++ */
  p2 = p1; /* isto, ne valja! */
```

Ukažimo sada na najčešću grešku koju prave početnici, a koju nerijetko možemo vidjeti i u raznim udžbenicima. Probaćemo napraviti varijantu ranije date funkcije `ispisi_matricu` koristeći pokazivačku aritmetiku:

```
#include <stdio.h>
void ispisi_matricu (int mat[][10], int m, int n) {
  int i, j;
  int *p = mat; /* ovdje je upozorenje */
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
      printf ("%d ", *p++);
    printf ("\n");
  }
}
int main() {
  int mat[4][10] = {{1, 2}, {3, 4}, {5, 6}, {7, 8}};
  ispisi_matricu (mat, 4, 2);
  return 0;
}
```

U čemu je stvar, pa i mnogi udžbenici koriste ovakav neispravan pristup? Problem leži u tome da će, u jeziku C, ovo gotovo uvijek raditi. Standardom jezika C (kao i C++) garantirano je da se višedimenzionalni nizovi u memoriji predstavljaju kao jednodimenzionalni, pri čemu je poredak po redovima. Ako imamo matricu kao u primjeru iznad, garantirano je da će nakon prvog reda u memoriji doći drugi red odnosno da nakon brojeva 1 i 2 dolazi broj 3, nakon njega 4 itd. Zbog takve predstave matrice u memoriji, izraz `p++`, kada `p` pokazuje na posljednji element reda, će pomjeriti pokazivač na novi red, čime programer ima "osjećaj" da se "kreće" po matrici.

1770198

Zbog te garancije da se redovi nalaze jedan iza drugog, program će na svakoj razumnoj arhitekturi i sa svakim razumnim kompajlerom raditi. Ali on i dalje *nije ispravan*! Razlog za ovo je što smo koristili nevalidne pokazivače. Prisjetimo se da je pokazivač na element niza validan ako primjenom pokazivačke aritmetike bude pokazivao isključivo na elemente tog niza ili eventualno na jedan element iza kraja niza.↑ U rješenju iznad to nije slučaj, pokazivač `p` (koji po tipu treba da pokazuje na *jedan* element *jednog* niza) je pomjeren daleko iza granica tog niza (jer je matrica sastavljena od više nizova).

↑ Poglavlje C.1.2, str. 359.

Ista stvar bi se desila da smo *nepravilno* odredili tip parametra kao jednostruki ili dvostruki pokazivač:

```
void ispisi_matricu (int *mat, int m, int n);
void ispisi_matricu (int **mat, int m, int n);
```

Ovakav program, iako će kompajlirati i raditi,[11] biti će neispravan jer sadrži *nedefinisano ponašanje* i posljedice su nepredvidljive (npr. kompajler koji bi generirao takav program da on krahira za primjer iznad bi bio *ispravan* kompajler i bio bi u *skladu sa standardom*). Čak i ako u funkciji sa posljednjim prototipom ne koristimo pokazivačku aritmetiku, takav program bi se kompajlirao i gotovo sigurno radio, ali je neispravan jer smo prilikom poziva funkcije pridružili matricu kao cjelinu pokazivaču pogrešnog tipa, te je pristup i-tom redu te matrice nedefinisano ponašanje.

[11] Barem u C-u, u C++ neće ni kompajlirati zbog 4. linije koja glasi `int *p = mat;`

Ispravno rješenje za ispisivanje matrice pokazivačkom aritmetikom je da koristimo obje vrste pokazivača: za kretanje po redovima koristimo pokazivače na cijele nizove (jer će pomjeranje pokazivača značiti pomjeranje na sljedeći red), a za kretanje po kolonama koristimo obične pokazivače (na jedan element) jer pomjeranje takvog pokazivača znači prelazak na sljedeći element niza.

Zadatak 4. *Napisati funkciju "ispisi_matricu" koja ispisuje matricu cijelih brojeva sa proizvoljnim brojem redova i deset kolona. Elemente ispisivati na 5 razmaka obavezno koristeći pokazivačku aritmetiku.*

```
1  #include <stdio.h>
2  void ispisi_matricu (int mat[][10], int m, int n) {
3    int i, j;
4    /* pokazivac na redove */
5    int (*p_red)[10] = mat;
6    for (i = 0; i < m; i++) {
7      /* na prvi elem. trenutnog reda */
8      int *p_el = *p_red;
9      for (j = 0; j < n; j++)
10       printf ("%5d", *p_el++);
11     printf ("\n");
12     /* predjimo na sljedeci red/niz */
13     p_red++;
14   }
15 }
16 int main() {
```

```
17  int mat[4][10] = {{1, 2}, {3, 4}, {5, 6}, {7, 8}};
18  ispisi_matricu (mat, 4, 2);
19  return 0;
20 }
```

Prije nego krenemo sa analizom, napomenimo da smo za tip prvog parametra umjesto `int mat[][10]` mogli pisati i:

```
2  void ispisi_matricu (int (*mat)[10], int m, int n)
```

što je u skladu sa analogijom za jednodimenzionalne nizove kada smo pisali `int niz[]` što je bilo isto kao i pokazivač na jedan njegov element `int *niz`.

U napisanom kôdu komentari su dovoljni za razumijevanje pojedinih dijelova. Napomenimo da smo umjesto pomoćnog pokazivača `p_red` mogli koristiti i proslijeđeni parametar `mat` (a iznad smo vidjeli da su istog tipa), ali je to urađeno radi lakšeg praćenja i čitljivosti. Još je potrebno analizirati liniju 9 u kojoj kreiramo privremeni pokazivač na jedan element niza. U njoj inicijaliziramo tako kreirani pokazivač sa dereferenciranim pokazivačem na jedan red. Kako je sada *samo* dereferenciranje pokazivača dovoljno kada smo prethodno vidjeli da smo morali i indeksirati i dereferencirati pokazivač na cijeli niz (govorimo o izrazima tipa `(*pok)[0]`)? Odgovor je jednostavan, moramo koristiti i indeksiranje ako želimo dobiti element, ali ako želimo dobiti pokazivač na taj element, indeksiranje nije potrebno (analogija sa jednodimenzionalnim nizovima je sljedeća: izraz `niz` je pokazivač, ali izrazi `niz[0]` i `*niz` su element na kojeg pokazivač pokazuje).

Zahvaljujući ovom razmatranju, možemo doći do sljedećeg zaključka: dovoljno je napisati samo `**p_red` da bismo dobili vrijednost *prvog* elementa reda na kojeg pokazivač `p_red` pokazuje. Taj izraz je isti kao `(*p_red)[0]` i kao `pok[0][0]`. Ovo je zanimljivo jer smo iskoristili dvostruko dereferenciranje pokazivača koji *nije dvostruki* pokazivač!

Veliki broj programera koristi stil programiranja gdje sve što se ne treba mijenjati proglase konstantnim, što je ispravna i dobro preporučena praksa. Stoga bismo mogli napisati sljedeći prototip funkcije:

```
void ispisi_matricu (const int (*mat)[10], int m, int n)
/* ili */
void ispisi_matricu (const int mat[][10], int m, int n)
```

Naravno, i pokazivače `p_red` i `p_el` bismo morali deklarisati drugačije, stavljajući im da pokazuju na konstantne objekte:

```
const int (*p_red)[10] = mat;
...
const int *p_el = *p_red;
```

Nažalost, postoji "greška" u standardu jezika C te funkcija sa ovako napisanim zaglavljem ne bi mogla da prihvati običnu matricu (koja nije deklarisana konstantnom). Ovo je suprotno dosadašnjoj priči gdje

smo u `const` promjenljivu uvijek mogli staviti neku nekonstantnu vrijednost (odnosno pisati npr. `const int *p = obicni_pok`). Znači, u C-u nije validno sljedeće:

```
int mat[2][10] = {...};
const int (*pok)[10] = mat; /* upozorenje */
/* ni ovo ne bi moglo :( */
ispisi_matricu (mat, 2, 2);
```

Iznad napisani kôd je savršeno legalan u jeziku C++, ali u C-u izaziva upozorenje i program nije u skladu sa standardom. Većina kompajlera će prihvatiti kôd i ignorirati standard (jer je u ovom slučaju intuitivno da bi takva dodjela ili poziv funkcije bili legalni), ali, naravno, kompajleri moraju prijaviti grešku ukoliko se podese da se striktno drže standarda (što je dobra praksa).

Konkretno, problem u ovom slučaju jeste što, po C standardu, kada deklarišemo da je niz konstantan zapravo smo deklarisali samo da su njegovi elementi konstantni (ali ne i sami niz), pa tako pokazivač `pok` je *pokazivač koji pokazuje na obični niz konstantnih brojeva*, dok su po C++ standardu i niz i njegovi elementi konstantni! Drugim riječima, u C++-u, pokazivač `pok` je *pokazivač koji pokazuje na konstantni niz konstantnih brojeva*. I ovo je još jedna rijetka razlika između C i C++ jezika koju gotovo niko ne spominje.

```
const int niz[10][5];
```

Iznad deklarisani niz u C-u nije konstantan nego su konstantni njegovi elementi, dok u C++u su konstantni i niz i njegovi elementi. Ovdje nema baš razlike između ovo dvoje, ali kada trebamo to proslijediti funkcijama odnosno dodijeliti pokazivačima, nastaju nekompatibilnosti.

Postoji li kompromis? Da li je moguće pisati "sigurne" funkcije (koje koriste konstantnost), a da pišemo programe u C-u? Odgovor je potvrdan – moguće je uz pomoć konverzije (tada kompajler ne prijavljuje upozorenja jer programer vrši konverzije na svoju ruku):

```
ispisi_mat ((const int (*)[2])mat, m, n);
```

Ovdje trebamo naročito biti oprezni jer u ovom slučaju, zbog eksplicitne konverzije, možemo proslijediti *bilo kakav* pokazivač (koji npr. pokazuje na jedan `char`)!

C.3.4 *Pokazivači na matrice kao cjeline*

1777018198

Vidjeli smo da postoje pokazivači na jedan element nekog niza. Također smo vidjeli da postoje pokazivači na jedan element matrice (odnosno pokazivači na cijele nizove). Možemo li nastaviti? Postoje li pokazivači na cijele matrice odnosno pokazivači na *jedan* element nekog 3D niza? Odgovor je potvrdan! Ovdje ćemo kratko prikazati deklaraciju takvih pokazivača radi kompletnosti gradiva.

```
#include <stdio.h>
int main() {
  int niz3D[3][2][2] = {
                            {{1, 2}, {3, 4}},
                            {{5, 6}, {7, 8}},
                            {{9, 19}, {11, 12}}
                        };
  int (*pok)[2][2] = niz3D;
  printf ("Prvi el, prva mat: %i\n", ***pok); /* 1 */
  pok++; /* pomjerimo na sljedecu matricu... */
  printf ("Prvi el, druga mat: %i\n", ***pok); /* 5 */
  return 0;
}
```

Program iznad ispisuje:

```
Prvi el, prva mat: 1
Prvi el, druga mat: 5
```

Naravno, možemo ovako ići dokle god je to potrebno pa kreirati pokazivače i na cijele 5D nizove, ali ne treba pretjerivati i koristiti stvari koje zaista nisu potrebne u programima, a vrlo rijetko su potrebni pokazivači veće kompleksnosti nego što je to ovdje predstavljeno.

C.3.5 *Dinamički alocirane i "grbave" matrice*

Kao što je moguće posmatrati matricu kao niz nizova, to važi i obrnuto: niz nizova se može posmatrati kao matrica. To nam daje interesantnu mogućnost da matricu nekog tipa npr. `int` predstavimo kao niz pokazivača na nizove tipa `int`.

Zadatak 5. *Napisati program koji omogućava unos dva cijela broja koji predstavljaju dimenzije željene matrice, nakon toga omogućava unos elemenata te matrice i ispisuje ih na 5 mjesta širine. Elementi matrice treba da budu cijeli brojevi, a dimenzije matrice neograničene u smislu da korisnik može unijeti bilo koje brojeve (ne treba ograničavati matricu na neku unaprijed definisanu dimenziju!)*

Ako ste pomislili da ovdje treba koristiti dinamičku alokaciju memorije, dobro ste pomislili! Već smo naučili kako kreirati nizove proizvoljne veličine,↑ te smo u zadatku 2 (str. 380) napisali funkciju koja je dinamički alocirala memoriju, pod imenom `alociraj_niz`, te smo u tu memoriju smještali elemente niza. Iskoristit ćemo ovu funkciju u rješenju ovog zadatka.

↑ Poglavlje C.2, str. 367.

Započećemo tako što ćemo napraviti niz pokazivača, pri čemu će svaki pokazivač pokazivati na jedan dinamički alocirani niz:

```
#include <stdio.h>
int main() {
  int* matrica[20];
  int i, j, sirina;
  printf("Unesite sirinu matrice:");
  scanf ("%d", &sirina);
  for (i = 0; i < 20; i++) {
```

```
8     matrica[i] = (int*)malloc(sizeof(int) * sirina);
9   }
10
11  printf("Unesite elemente matrice:");
12  for (i = 0; i < 20; i++) {
13    for (j = 0; j < sirina; j++) {
14      printf("Element [%d,%d]: ", i, j);
15      scanf("%d", &matrica[i][j]);
16    }
17  }
18  ...
```

Najprije uočavamo da je visina matrice fiksirana na 20 redova. To je posljedica činjenice što je `matrica` ustvari niz pokazivača, a nizovi u C-u imaju fiksne dimenzije. U liniji 8 vidimo dinamičku alokaciju jednog niza od `sirina` elemenata, pa je memorija potrebna za taj niz `sizeof(int)*sirina`. Povratnu vrijednost funkcije `malloc` trebamo pretvoriti u tip člana niza `matrica`, a to je `int*`.

Konačno, u liniji 15 vidimo da se sa ovako kreiranim nizom pokazivača može raditi kao da je to zaista obična matrica. Naime, `matrica[i]` je pokazivač tipa `int*`, a kada pokazivač indeksiramo, dobijamo vrijednost člana niza za `j` mjesta pomaknutog u odnosu na lokaciju pokazivača.

Da bismo zaobišli postavljeno ograničenje, možemo li sam niz pokazivača dinamički alocirati? Što da ne, i u tom slučaju imamo dvostruki pokazivač! Naime, rekli smo da pokazivač na niz tipa `TIP` ima tip `TIP*`, a dinamički alocirani niz pridružujemo pokazivaču na prvi član takvog niza. Ovo je jasnije iz sljedećeg primjera:

```
1  int a = 1, b = 2, c = 3;
2  int *niz_pok[3] = {&a, &b, &c}; /* niz pokazivaca */
3  niz_pok[0] /* je pokazivac (kao i &a) */
4  &niz_pok[0] /* je pokazivac na prvi element
5                 a to je pokazivac na pokazivac! */
6  int **pok = &niz_pok[1];
7  ...
```

Naš program sada izgleda ovako:

```
1  #include <stdio.h>
2  int main() {
3    int** matrica;
4    int i, j, sirina, visina;
5    printf("Unesite visinu matrice:");
6    scanf ("%d", &visina);
7    printf("Unesite sirinu matrice:");
8    scanf ("%d", &sirina);
9    matrica = (int**)malloc( sizeof(int*) * visina );
10   ...
```

1/15/01/98

Ostatak programa ponovo izgleda isto, odnosno sve ostale funkcije uopšte ne zanima da li je u pitanju običan ili dinamički alocirani niz, naredbe se ponašaju isto.

U ovom programu nedostaju mnoge stvari: prije svega, nismo provjerili da li je alokacija uspjela, a nismo ni oslobodili memoriju. Cjelovit

program će biti znatno duži, ali smo ga ipak dali u nastavku radi boljeg usvajanja gradiva. Ovako složen program bilo je neophodno organizovati u više funkcija.

```
#include <stdio.h>
#include <stdlib.h>
int *alociraj_niz (int n) {
  int *p = (int *)malloc (sizeof (int) * n);
  return p;
}
int **alociraj_niz_pokazivaca (int n) {
  /* koristimo calloc kako bi svi pokazivaci
     bili nul-pokazivaci. VAZNO! */
  int **p = (int **)calloc (sizeof (int *) * n);
  return p;
}
void oslobodi_niz (int *p) { free (p); }
void oslobodi_mat (int **mat, int n) {
  /* ako je nul pok, ne radi nista */
  if (mat == NULL)
    return;
  int i;
  /* oslobodimo redove */
  for (i = 0; i < n; i++)
    oslobodi_niz (mat[i]);
  /* oslobodimo niz pokazivaca */
  free (mat);
}
int **alociraj_mat (int m, int n) {
  int **mat = alociraj_niz_pokazivaca (m);
  if (mat == NULL)
    return NULL; /* nije uspjelo */
  int i;
  for (i = 0; i < m; ++i) {
    mat[i] = alociraj_niz (n);
    /* ako nije uspjelo... */
    if (mat[i] == NULL) {
      oslobodi_mat (mat, i); /* sve pobrisi */
      return NULL; /* nije uspjelo */
    }
  }
  return mat;
}
```

Kratko analizirajmo postavljeno rješenje, a zainteresiranima se preporučuje da detaljnije analiziraju svaki aspekt rješenja te da sami isprobaju različite dinamičke alokacije (npr. 3D niza). Funkcije `alociraj_niz` i `oslobodi_niz` su već objašnjene u navedenom zadatku 2. Jednostavno pomoću `malloc` funkcije rezervišemo dovoljno prostora, te vratimo pokazivač kojeg je vratila i `malloc` funkcija. Prilikom oslobađanja, kojeg ne smijemo zaboraviti da ne bismo imali curenje memorije,↑ koristimo `free` funkciju.

↑ Poglavlje C.2.3, str. 375.

Nakon toga imamo `alociraj_niz_pokazivaca` funkciju koja alocira niz pokazivača. Postupak alociranja je sličan kao kada alociramo običan niz cijelih brojeva, samo ovaj put umjesto `n` cijelih brojeva alociramo `n` pokazivača na cijele brojeve. Povratni tip ove funkcije je dvostruki

1715701198

pokazivač na cijeli broj. Objasnimo zašto:

```
int a = 1, b = 2, c = 3;
int *niz_pok[3] = {&a, &b, &c}; /* niz pokazivaca */
niz_pok[0] /* je pokazivac (kao i &a) */
&niz_pok[0] /* je pokazivac na prvi element
               a to je pokazivac na pokazivac! */
int **pok = &niz_pok[1];
...
```

Umjesto `malloc` koristili smo funkciju `calloc` da postavimo sve pokazivače da budu nul-pokazivači odnosno da ne pokazuju nigdje. Ovo će nam koristiti u funkciji za oslobađanje memorije, jer znamo da `free` funkcija *neće napraviti štetu* ukoliko joj se proslijedi nul-pokazivač.

Sljedeća funkcija je funkcija za oslobađanje zauzete memorije za matricu. Ona prvo provjeri da li je proslijeđeni pokazivač `NULL`, te ako jeste ne radi ništa. Potreba za ovom provjerom se javila zbog petlje koja slijedi. Naime, u njoj indeksiramo proslijeđeni pokazivač, a znamo da ne smijemo indeksirati (ni dereferencirati) nul-pokazivače. Nakon toga, pobrišemo svaki element matrice. Pošto je `mat` niz pokazivača, svaki pokazivač je pokazivao na jedan dinamički alocirani niz (ovu strukturu ćemo objasniti kada budemo objašnjavali sljedeću funkciju). Nakon što smo oslobodili memoriju pojedinih nizova, onda je potrebno osloboditi memoriju i niza pokazivača, a to smo uradili konstrukcijom `free (mat)`, nije bilo potrebe pisati novu funkciju samo za ovo.

Funkcija `alociraj_mat` je nešto kompleksnija. U njoj alociramo niz pokazivača na cijele brojeve (od `m` elemenata). Ti pokazivači će pokazivati na prve elemente nizova (čime ćemo imati niz pokazivača gdje svaki element pokazuje na novi niz, a to je zapravo matrica – niz nizova). Nakon provjere da li je alokacija uspjela, potrebno je alocirati svaki pojedini niz (od `n` elemenata) i pokazivač na njegov prvi element smjestiti u niz pokazivača. U slučaju da alokacija nije uspjela, potrebno je osloboditi sve prethodno zauzete nizove. Npr. da smo uspjeli alocirati niz pokazivača koji ima 10 elemenata. Nakon toga smo uspjeli alocirati 4 reda matrice, i nestalo je memorije prilikom alociranja 5. reda. Tada mi moramo pobrisati prva 4 reda (stoga je `i` kao 2. argument funkcije), kao i osloboditi taj alocirani niz pokazivača. Ukoliko je ipak bilo memorije, sve je prošlo kako treba, mi vraćamo pokazivač na niz pokazivača iz funkcije.

Prikažimo sada i `main` funkciju koju smo "odvojili" od originalnog kôda zbog njegove dužine:

```
void unesi_mat (int **mat, int m, int n) {
  int i, j;
  printf ("Unesite elemente:\n");
  for (i = 0; i < m; ++i)
    for (j = 0; j < n; ++j)
      scanf ("%i", &mat[i][j]);
}
void ispisi_mat (int **mat, int m, int n) {
```

```
62   int i, j;
63   printf ("Matrica glasi:\n");
64   for (i = 0; i < m; ++i) {
65     for (j = 0; j < n; ++j)
66       printf ("%5d", mat[i][j]);
67     printf ("\n");
68   }
69 }
70 int main() {
71   int m, n;
72   printf ("Unesite m i n: ");
73   scanf ("%i %i", &m, &n);
74   int **mat = alociraj_mat (m, n);
75   if (mat == NULL) {
76     printf ("Alokacija nije uspjela!");
77     return 1; /* nije OK */
78   }
79   unesi_mat (mat, m, n);
80   ispisi_mat (mat, m, n);
81   /* ne zaboravimo osloboditi memoriju! */
82   oslobodi_mat (mat, m);
83   return 0;
84 }
```

Ovdje moramo provjeriti da li je alokacija uspjela. Ukoliko jeste, možemo pristupati elementima tako alocirane matrice kao i običnoj matrici (zahvaljujući tome što možemo indeksirati pokazivače↑). Kao primjer rada sa ovako kreiranom matricom, dali smo i `unesi_mat` funkciju za unos matrice, te `ispisi_mat` za ispis matrice na ekranu. Vidimo da kod slanja matrice u funkciju sada ne moramo fiksno navesti jednu od dimenzija nego je to sada još jedan parametar funkcije, što je slično načinu kako smo slali nizove u funkcije, te se prirodno proširuje na tri ili više dimenzija.

↑ Poglavlje 7.3.2, str. 240.

Gledajući dati kod, možemo si postaviti pitanje: da li se širina matrice `n` može varirati u funkciji `alociraj_mat`? Odgovor i na ovo pitanje je potvrdan. Time dobijamo matricu kod koje svaki red ima svoju širinu, što se često koristi u računarstvu, te se takva matrica naziva *grbava matrica* iz razloga što je kod ispisa ova matrica vizuelno "grbava" tj. njena desna strana nije poravnata.

✓ Grbava matrica

Problem pri radu sa ovakvom grbavom matricom je što u svakoj narednoj petlji moramo znati širinu i-tog reda da bismo mogli proći kroz njegove elemente. Jedno rješenje je da umjesto promjenljice `sirina` koristimo niz širina čija će dimenzija biti `visina`. Pa kako je matrica po visini (niz pokazivača) dinamički alocirana, tako i ovaj niz širina mora biti dinamički alociran.

Vratimo se na primjer sa početka ovog potpoglavlja:

1 7 2 7 0 1 9 8

```
1 #include <stdio.h>
2 int main() {
3   int** matrica;
4   int* sirina;
5   int i, j, visina;
6   printf("Unesite visinu matrice:");
```

```
  scanf ("%d", &visina);
  matrica = (int**)malloc( sizeof(int*) * visina );
  sirina = (int*)malloc( sizeof(int) * visina );

  printf("Unesite elemente matrice:");
  for (i = 0; i < 20; i++) {
    printf("Unesite sirinu %d-tog reda matrice:", i + 1);
    scanf ("%d", &sirina[i]);
    matrica[i] = (int*)malloc(sizeof(int) * sirina[i]);

    for (j = 0; j < sirina; j++) {
      printf("Element [%d,%d]: ", i, j);
      scanf("%d", &matrica[i][j]);
    }
  }
  ...
```

Funkcije za rad sa grbavom matricom sada primaju visinu i niz širina, npr. funkcija za ispis matrice glasi:

```
void ispisi_mat (int **mat, int *sirina, int visina) {
  int i,j;
  printf ("Matrica glasi:\n");
  for (i = 0; i < visina; ++i) {
    for (j = 0; j < sirina[i]; ++j)
      printf ("%5d", mat[i][j]);
    printf ("\n");
  }
}
```

Širina matrice se može utvrditi i programski, pa se u tom slučaju ne mora čuvati u nizu niti slati u funkciju. Npr. jedan školski slučaj primjene grbave matrice je Pascalov trougao. Kod nje širina i-tog reda je jednostavno jednaka $i + 1$.

C.4 *Pokazivači na funkcije*

Privedimo kraju priču o pokazivačima tako što ćemo uvesti i pokazivače na funkcije. I to je moguće i jako korisno, jer omogućava automatizirano pozivanje funkcija, kao i prijenos funkcija kao parametara drugim funkcijama. Prilikom deklaracije pokazivača na funkciju specificiramo prototip funkcije: povratni tip, broj i tip parametara. Također, kada navedemo samo ime funkcije, dobili smo njenu adresu odnosno pokazivač na nju.

✓ *Ime funkcije upotrijebljeno samo za sebe predstavlja pokazivač na tu funkciju (odnosno adresu mašinskog kôda gdje se ta funkcija nalazi)* ✓ Pokazivač na funkciju

17570198

Ako primijenimo ovu definiciju, isto je `fun` i `&fun` ukoliko je `fun` ime neke funkcije:

```
  int kvadrat (int a) {return x * x;}
  ...
  kvadrat; /* ovo je adresa funkcije (pok. na nju) */
  &kvadrat; /* i ovo je adresa funkcije */
```

Pošto pokazivače na funkcije možemo samo dereferencirati (jer je to jedina smislena operacija nad njima), ne postoji pokazivačka aritmetika za njih.[12] Čak, pošto je dereferenciranje *jedina* smislena operacija koja se s njima može obaviti, možemo pokazivače i "pozivati", čime izgleda kao da su "prave" funkcije. Naravno, i dalje ih je moguće dereferencirati, ali u tome slučaju moramo staviti par zagrada oko dereferenciranog pokazivača kako ne bismo pokušali dereferencirati ono što funkcija vraća:

[12] Ponovo, neki prevodioci će dopustiti pokazivačku aritmetiku nad pokazivačima na funkcije, tretirajući ih kao da su `char*`.

```
int x = 10;
int y = kvadrat (x); /* standardan poziv... */
/* poziv kada dereferenciramo */
int z = (*kvadrat) (x);
```

Da nismo stavili zagrade, imali bismo izraz `*kvadrat (x)` a to je zapravo dereferenciranje pokazivača kojeg funkcija vraća. U ovom slučaju to bi bila greška jer `kvadrat` funkcija ne vraća pokazivač.

Kada smo se upoznali sa ovim pojmovima, pogotovo da su imena funkcija zapravo adrese početka mašinskog kôda gdje se one nalaze, napišimo kako da *deklarišemo* takve pokazivače i da u njih smjestimo adrese funkcijâ. Prvo ćemo napisati generalnu sintaksu:

```
POVRATNI_TIP (*ime_pokazivaca)([PARAMETRI...]);
```

Pod `POVRATNI_TIP` se smatra povratni tip funkcije, a ako funkcija ne vraća ništa onda stavimo `void`. Parametri su proizvoljni, ali ako ih ima navodimo samo njihove tipove.[13] Pogledajmo primjer deklaracije pokazivača na funkcije istog prototipa kao i `kvadrat` funkcija iznad napisana:

[13] Imena parametara možemo pisati ali će biti zanemarena.

```
int (*pok)(int) = kvadrat;
/* ili */
int (*pok)(int) = &kvadrat;
```

Ovako deklarisanom pokazivaču možemo dodijeliti adresu bilo koje funkcije koja vraća cijeli broj i koja prihvata jedan cijeli broj (odnosno bilo koju funkciju sa `int` povratnim tipom i jednim parametrom tipa `int`). Tako, validno je i sljedeće:

```
int kub (int x) {return x * x * x};
int je_li_prost (int x) { /* provjera */ return ...; }
...
pok = kub; /* ili: pok = &kub */
pok = &je_li_prost;
```

Ukoliko funkcija ne prihvata parametre (a npr. neka u primjeru ispod i ne vraća ništa), onda stavimo samo prazan par zagrada:

```
void ispisi_OR () {printf ("OR");}
...
void (*pok2)() = &ispisi_OR;
```

1/10/2018

Kako znamo iz poglavlja o funkcijama, ukoliko prilikom deklaracije funkcije stavimo prazan par zagrada (kao kod `ispisi_OR` funkcije iznad), tada joj možemo proslijediti proizvoljan broj parametara proizvoljnog tipa. To smo sprječavali tako što kao jedini parametar stavimo

`void`, čime smo onemogućili takve pozive (koji su u C++-u podrazumijevano takvi). Tako možemo uraditi (i to se često može vidjeti u C programima) i sa pokazivačima:

```
void (*pok2)(void) = &ispisi_OR;
/* a mogli smo biti "striktni" i kod
   definicije funkcije: */
void ispisi_OR (void) {...}
```

Dva pokazivača na funkcije različitih prototipova nisu kompatibilna i ne mogu se međusobno dodjeljivati. Tako, iznad deklarisani pokazivač `pok` (koji je pokazivao na funkcije koje prihvataju jedan cjelobrojni parametar i vraćaju cijeli broj) ne možemo dodijeliti pokazivaču `pok2` koji pokazuje na funkciju bez parametara i koja ne vraća ništa. Ni obrnuto nije moguće. Nije moguće ni ako je samo parametar drugog tipa:

```
int (*pok3)(double);
pok = pok3; /* greska! */
```

Već smo vidjeli da pokazivače na funkcije možemo dereferencirati, pa smo mogli pisati nešto poput sljedećeg:

```
int kvadrat (int x) {return x * x;}
...
int (*pok)(int) = kvadrat;
...
int k = (*pok) (5); /* k je 25 */
```

Međutim, također je naglašeno da pokazivače na funkcije možemo samo dereferencirati (to je njihova jedina operacija), tako da je onda i funkcijski poziv (kao da su obične funkcije) dozvoljen, pa čak i više korišten:

```
int k = pok (5); /* kao obicna funkcija! */
```

Zahvaljujući mogućnosti da funkcijski pokazivači (odnosno pokazivači na funkcije) mogu pokazivati na više funkcija istog prototipa, možemo pisati programe koji će "pozivati" jedan pokazivač a koji će u tome trenutku pokazivati na različite funkcije. Npr. jedna od primjena može biti da pravimo (mini) kalkulator:

```
double mnozi (double a, double b) {...}
double dijeli (double a, double b) {...}
double saberi (double a, double b) {...}
double oduzmi (double a, double b) {...}
double stepenuj (double a, double b) {...}
...
double (*operacija)(double, double); /* pok... */
...
if (unos == '*')
  operacija = mnozi;
else if (unos == '-')
  operacija = oduzmi;
...
else
  opracija = NULL;
```

```
  ...
  if (operacija) /* != NULL */
     rez = operacija (b1, b2);
```

Primjer iznad je pokaznog karaktera, te sadrži jednu zanimljivost: i funkcijskim pokazivačima se, kao i običnim, može dodijeliti nul-pokazivač, čime oni ne pokazuju niti na jednu funkciju.

Funkcijski pokazivači se intenzivno koriste kod programiranja vođenog događajima (eng. *event-driven programming*), a tom paradigmom se kreiraju skoro sve grafičke aplikacije koje odgovaraju na korisničke akcije. Pogledajmo jedan primjer:

```
void dugme_kliknuo (void *na_sta) {
  /* ovo ce se pozivati kada neko
  klikne na nase dugme... */
  struct Dugme *d = (struct Dugme *)na_sta;
  printf ("Tekst dugmeta: %s", (*na_sta).Tekst);
}
...
struct Dugme d = {...}; /* kreiranje dugmeta */
/* postavljanje dogadjaja (eventa)  za "klik" */
d.kliknuo = dugme_kliknuo;
/* kad god korisnik klikne na 'd' dugme, pozvace
  se dodijeljena funkcija. d.kliknuo je pokazivac
  na funkciju!
*/
...
```

U ovom primjeru, `Dugme` je struktura↑ koja sadrži sve osobine nekog dugmeta na formi: veličinu, tekst, boju... pa samim time bi trebalo da sadrži i ono što se treba desiti kada korisnik klikne na dugme. Taj kod se nalazi u našoj funkciji `dugme_kliknuo`. Ostaje da nekako saopštimo dugmetu `d` da se kod za klikanje na to dugme nalazi u toj funkciji. To ćemo postići tako što u strukturi `Dugme` imamo i neki pokazivač na funkciju, pa bi definicija te strukture mogla izgledati ovako:

↑ Poglavlje 9.3, str. 303.

```
struct Dugme {
  int sirina, visina;
  Boja boja;
  void (*kliknuo)(void*);
  ...
};
```

Na ovaj način, koristeći pokazivače na funkcije kao članove struktura, u programskom jeziku C možemo simulirati C++ klase! Također, ovako izgledaju dijelovi kôda koji se stvarno koristi za pisanje grafičkih aplikacija za Windows operativni sistem.

Također, još jedna važna primjena funkcijskih pokazivača je kod niza funkcija (odnosno niza pokazivača na funkcije). Tada u niz možemo smjestiti pokazivače na funkcije istog prototipa, te ih sve automatizirano pozivati (npr. u for petlji). Na ovaj način se kreiraju neki sistemi provjere ispravnosti programa automatskim testiranjem (unit testovi).

1/15/01/98

Da bismo to postigli, moramo naučiti deklarisati niz pokazivača na funkcije. Tačno je da izgleda komplikovano, i postaje sve komplikovanije kako prototipovi funkcija postaju komplikovaniji, ali to nas ne treba obeshrabriti da pokušamo. Također, postoji pravilo za čitanje kompleksnijih deklaracija koje, u skraćenoj verziji, glasi: "čitanje deklaracije promjenljive počinjemo od njenog imena, te se krećemo desno do prve (male) zagrade, nakon čega krećemo lijevo do prve zagrade i postupak ponavljamo dok ne pročitamo cijeli izraz". Ovo pravilo se naziva i pravilo "desno-lijevo", a koristeći njega ćemo protumačiti deklaraciju niza pokazivača. Prvo prikažimo njegovu deklaraciju i primjenu.

✓ Pravilo "desno-lijevo"

```
1  #include <stdio.h>
2  int kvadrat (int x) {return x * x;}
3  int negativan (int x) {return -x;}
4  int kub (int x) {return x * x * x;}
5  int paran (int x) {return x % 2 == 0;}
6  int na_petu (int x) {return x * x * x * x * x;}
7  /* .. */
8  int main() {
9    int x, i;
10   printf ("Unesite x: ");
11   scanf ("%i", &x);
12   /* niz od 5 funkcija (niz pok. na fun) */
13   /* funkcije vracaju int i prihvataju int */
14   int (*niz[5])(int) = {&kvadrat, &negativan,
15                         &kub, &paran, &na_petu};
16   niz[0] = &kvadrat; /* moze i ovako... */
17   for (i = 0; i < 5; ++i) {
18     int r = niz[i](x); /* poziv i-te funkcije */
19     /* isto kao i: */
20     r = (*niz[i])(x);
21     printf ("Fun %i, x = %i\n", i + 1, r);
22   }
23   return 0;
24 }
```

Objasnimo spornu 14. i 15. liniju. U njoj je deklarisan niz od 5 pokazivača na funkcije koje prihvataju jedan cijeli broj i vraćaju cijeli broj. Primijenimo "desno-lijevo" pravilo. Polazimo od imena promjenljive a to je `niz`. Idemo desno i vidimo da je to niz od 5 elemenata. Idemo još desno i naići ćemo na zagradu, pa moramo ići lijevo. Do sada je pročitano da je `niz` niz od 5 elemenata. Nakon kretanja nalijevo vidimo da su to pokazivači, pa imamo da je to niz od 5 elemenata koji su pokazivači. Pošto smo naišli na zagradu, idemo desno, preskačemo već korištenu zagradu, i vidimo da je to funkcija koja prihvata cijeli broj tipa `int`. Do sada imamo da je `niz` zapravo niz od 5 elemenata koji su pokazivači na funkcije koje prihvataju `int`. Vratimo se lijevo i pročitamo da te funkcije vraćaju cijeli broj (`int`).

Napomenimo također da bi u C-u bila validna i sljedeća deklaracija:

```
int (*niz[5])() = {...} /* bez "int" */
```

1/15/01198

I program bi i dalje radio isto i bio bi tačan. Ovo možemo uraditi zahvaljujući činjenici koju smo razmatrali u prethodnom dijelu ovog potpoglavlja: da funkcija bez parametara može da prihvati neograničen broj parametara neograničenog tipa, pa tako i pokazivač na funkciju koja nema parametara. Ipak, ovako deklarisan niz *neće* kompajlirati u C++-u, jer se tamo funkcija bez parametara upravo tako tretira – da nema parametara (ista stvar je sa pokazivačima na takvu funkciju). Stoga, navedeni izraz iznad je identičan sljedećem u jeziku C++:

```
int (*niz[5])(void) = {...}
```

Evo još jednog razloga zašto je poželjno za funkcije koje nemaju parametara (a pogotovo pokazivače na takve funkcije) staviti da je parametar `void`, čime se jasno daje do znanja da funkcija ne prihvata parametre. Također, takvi programi će bez izmjena moći kompajlirati i kompajlerom za jezik C++!

Ispis programa iznad ako unesemo broj 7 će biti sljedeći:

```
Unesite x: 7
Fun 1, x = 49
Fun 2, x = -7
Fun 3, x = 343
Fun 4, x = 0
Fun 5, x = 16807
```

Tumačenje ovih deklaracija je mnogo jednostavnije ukoliko kreiramo nova imena za tipove (aliase) pomoću `typedef`. Tako možemo lakše čitati (ali i kreirati) složenije izraze. Prikažimo kako bismo pomoću `typedef` mogli da čitljivije deklarišemo niz pokazivača na funkcije:

```
typedef int (*fun_pok)(int);
...
fun_pok niz[10]; /* niz 5 pokazivaca na funkcije */
```

Iznad deklarisani niz je identičan nizu kojeg smo deklarisali ranije (`int (*niz[5])(int)`), a ipak je čitljiviji. Pomoću `typedef` zaista možemo pojednostavniti kompleksnije deklaracije.

Najinteresantniju primjenu pokazivača na funkcije smo ostavili za kraj, a to je da funkcije možemo proslijediti kao parametre drugim funkcijama.

✓ Funkcija kao argument funkcije

Zadatak 6. *Napisati funkciju koja prihvata niz realnih brojeva te transformira elemente toga niza pomoću funkcije koja mu se prosljeđuje kao parametar. Funkcija treba da vrati pokazivač na početak tako transformiranog niza. Demonstrirati rad funkcije transoformiranjem niza tako što će mu biti kvadrirani elementi, zatim promijenjen predznak elemenata, pa kubirani elementi i na kraju odsječene decimale.*

1/5/0198

Pošto nije definisan način prijenosa niza u funkciju, a pošto je ovo poglavlje o pokazivačima, koristiti ćemo ograničavanje niza sa

pokazivačem na početak i pokazivačem iza kraja. Također, napisane su trivijalne funkcije za transformiranje jednog elementa, i to nakon `transformiraj` funkcije što indicira da našu funkciju može neko drugi koristiti godinama kasnije (upoznat ćemo se sa sličnim funkcijama u `stdlib.h` biblioteci).

```
#include <stdio.h>
double *transformiraj (double *p, double *ik,
                       double (*fun)(double)) {
  double *poc = p;
  while (p < ik) {
    *p = fun (*p);
    p++;
  }
  return poc;
}

double kvadrat (double x) {return x * x;}
double negativan (double x) {return -x;}
double kub (double x) {return x * x * x;}
double cijeli (double x) {return (int)x;}
```

Treći parametar `transformiraj` funkcije je funkcijski pokazivač na funkciju koja prihvata `double` i vraća `double`. Tip parametra je identičan kao kada deklarišemo pokazivač. Ipak, kod parametara koje prihvataju funkcije, možemo čak jednostavnije pisati (slično kao što smo za nizove mogli pisati `int niz[]` umjesto `int *niz` ili umjesto `int niz[][5]` smo pisali `int (*niz)[5]`), pa umjesto napisanog pisati `double fun(double)` odnosno, zaglavlje funkcije je moglo biti:

```
double *transformiraj (double *p, double *ik,
                                  double fun(double))
```

Ili čak još čitljivije koristeći alias za tip (`typedef`):

```
typedef double (*f_p)(double);
double *transformiraj (double *p, double *ik, f_p fun)
```

Poziv ovog pokazivača je isto kao i obične funkcije (čime imamo osjećaj da smo proslijedili funkciju u drugu funkciju, a ne samo pokazivač). Naravno, mogli smo pisati manje čitljiviju (ali jasniju) konstrukciju čime se naglašava da je to pokazivač:

```
  *p = (*fun)(*p); /* isto kao: *p = fun(*p); */
```

Ostatak funkcije bi trebao biti jasan, jer smo već nekoliko puta radili slične zadatke. Napisane su i trivijalne funkcije koje ćemo prosljeđivati `transformiraj` funkciji a koje vrše transformaciju proslijeđenog argumenta (odnosno, transformirat će po jedan element niza).

Napišimo sada i pomoćnu funkciju za ispis niza jer ćemo ga u `main` funkciji ispisivati više puta. U njoj smo deklarisali i statičku promjenljivu kako bismo razlikovali različite ispise, te smo ispisivali elemente niza na 10 mjesta širine sa desnim poravnanjem. Također ispred svakog broja je stavljeno da ima predznak (to je značenje znaka "+" u formatnom stringu `printf` funkcije), čak i kada su brojevi pozitivni.

1/15/01/98

Interesantno, da smo htjeli *lijevo* poravnanje, ali da i dalje želimo da imamo predznak ispred pozitivnih brojeva pisali bismo naizgled čudan formatni string: `%-+10.2f`.

```
18 /* pomocna funkcija za ispis */
19 void ispis (double *p, double *ik) {
20   static int i = 1;
21   printf ("Ispis %i: ", i++);
22   while (p < ik)
23     printf ("%+10.2f ", *p++);
24   printf ("\n");
25 }
```

Konačno, prikažimo i `main` funkciju u kojoj smo deklarisali niz od 4 elementa i kojeg smo transformirali po zahtjevima zadatka korištenjem pomoćnih funkcija. Ovdje vidimo da jednu funkciju možemo upotrijebiti više puta da radi različite stvari i da tu istu funkciju neko drugi može koristiti bez modifikacija da radi nešto što njemu odgovara (npr. da traži 18. korijen):

```
27 int main() {
28   #define N 4
29   double niz[N] = {1.1, 2.5, 4, 3.32};
30   ispis (niz, niz + N);
31   transformiraj (niz, niz + N, kvadrat);
32   ispis (niz, niz + N);
33   transformiraj (niz, niz + N, negativan);
34   ispis (niz, niz + N);
35   transformiraj (niz, niz + N, kub);
36   ispis (niz, niz + N);
37   /* moze direktno ispis: */
38   ispis (transformiraj (niz, niz + N, cijeli),
39                                       niz + N);
40   return 0;
41 }
```

Izgled izlaza napisanog programa će biti sljedeći:

```
Ispis 1:      +1.10      +2.50      +4.00      +3.32
Ispis 2:      +1.21      +6.25     +16.00     +11.02
Ispis 3:      -1.21      -6.25     -16.00     -11.02
Ispis 4:      -1.77    -244.14   -4096.00   -1339.15
Ispis 5:      -1.00    -244.00   -4096.00   -1339.00
```

Zadatak 7. *Napisati funkciju "izvod" koja računa prvi izvod proizvoljne funkcije u nekoj tački. Koristiti numerički metod za izračunavanje izvoda, a za korak diferenciranja h koristiti* 10^{-7}. *Pretpostavlja se da su funkcije diferencijabilne u proslijeđenoj tački. Testirati napisanu funkciju za računanje izvoda sljedećih funkcija:*
(1) $\sin x$ *u tački* $x = 0$;
(2) $x^3 - 5x^2 + 2x + 11$ *u tački* $x = 2.5$;
(3) $\cos^2 x$ *u tačkama* $x = 0$ *i* $x = 1$.

I u ovom slučaju ćemo prenositi funkcije kao parametre drugim funkcijama. Naša `izvod` funkcija će prihvatati dva parametra: funkciju

1757018

čiji izvod treba naći i tačku u kojoj izvod tražimo. Za računanje izvoda numeričkim metodom ćemo koristiti formulu simetrične diferencije:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Već nam je preporučena vrijednost za h i to 10^{-7}. Ostaje samo da napišemo funkciju koja će po iznad napisanoj formuli računati izvod u proslijeđenoj tački.

```
#include <stdio.h>
#include <math.h> /* za sin i cos */
double izvod (double f(double), double x) {
  const double h = 10E-7;
  double diff = f (x + h) - f (x - h);
  return diff / (2.0 * h);
}
/* funkcije za testiranje: */
double fun (double x) { /* x^3 - 5x^2 + 2x + 11 */
  return (x * x * x) - (5 * x * x) + 2 * x + 11;
}
double cos2 (double x) { /* cos^2 (x) */
  double c = cos (x);
  return c * c;
}
int main() {
  printf ("sin(0)' = %.3f\n", izvod (sin, 0));
  printf ("fun(2.5)' = %.3f\n", izvod (fun, 2.5));
  printf ("cos2(0)' = %.3f\n", izvod (cos2, 0));
  printf ("cos2(1)' = %.3f\n", izvod (cos2, 1));
  return 0;
}
```

Prikažimo još i ispis programa iznad, s obzirom da nije potrebno nikakvo objašnjavanje, jer je svrha ovog zadatka da pokaže korist funkcijskih pokazivača.

```
sin(0)' = 1.000
fun(2.5)' = -4.250
cos2(0)' = 0.000
cos2(1)' = -0.909
```

U potpoglavlju o generičkim pokazivačima↑ kao i o konverzijama pokazivača↑ smo spominjali da će nam ove konverzije biti korisne kada budemo pisali funkciju za sortiranje nizova proizvoljnog tipa. Takva funkcija će prihvatati `void` pokazivače, ali nismo je mogli napisati ranije jer nismo bili upoznati sa funkcijskim pokazivačima. Naime, u ovoj funkciji (pošto mi ne znamo u koji tip da konvertujemo `void` pokazivače) obavezno je imati još jednu funkciju koja ima dopunske informacije o *poretku* elemenata. Tu funkciju prosljeđujemo kao argument funkciji za sortiranje.

↑ Poglavlje C.2.1, str. 367.
↑ Poglavlje C.2.2, str. 372.

Zadatak 8. *Napisati funkciju "sortiraj" koja sortira proslijeđeni niza elemenata proizvoljnog tipa. Funkcija ima 4 parametra: pokazivač na početak niza, veličinu niza, veličinu u bajtima jednog elementa, te funkciju* `f` *za*

1/15/0198

poređenje dva elementa. Navedena `f` *funkcija prihvata dva generička pokazivača a vraća* 0 *ili* 1 *(logička vrijednost) u zavisnosti da li treba element na kojeg pokazuje prvi pokazivač doći prije elementa na kojeg pokazuje drugi pokazivač, ili ne.*

Rješenje ovog zadatka traži analizu njegove postavke prije svega. Naša funkcija za sortiranje će moći sortirati nizove elemenata *bilo* kojeg tipa, što znači da će prvi pokazivač biti generički pokazivač (tipa `void *`). Naravno, njega nećemo staviti da pokazuje na konstantni objekat, jer ćemo modificirati blok memorije na kojeg on pokazuje. Dalje, sljedeća dva parametra mogu biti samo pozitivna (veličina niza i bajtska veličina jednog elementa), pa ćemo staviti da su tipa `size_t`. (Mogli smo pisati i `unsigned int`, a zašto je baš odabran `size_t` bit će objašnjeno naknadno.)

Posljednji parametar je funkcija (odnosno pokazivač na nju) koja prihvata dva pokazivača na konstantne objekte. Stavljeno je da pokazuje na konstantne objekte jer takva funkcija *ne bi trebala* da modificira elemente, nego samo da ih uporedi i odredi koji je prije po definisanom poretku. Prototip takve funkcije je sljedeći:

```
int poredi (const void *p1, const void *p2)
```

Naravno, morali su biti generički pokazivači jer naša funkcija `sortiraj` neće znati kakve ćemo joj nizove prosljeđivati! To može biti niz cijelih brojeva, niz realnih brojeva ili čak niz nekih drugih objekata (npr. niz slika). Naša funkcija `poredi` će vratiti 1 ako element na kojeg pokazuje `p1` treba doći ispred elementa na kojeg pokazuje `p2`, u suprotnom će vraćati 0.

Do sada smo naučili da prototip `sortiraj` funkcije treba biti sljedeći:

```
void sortiraj (void *niz, /* pokazivac na pocetak */
  size_t broj_el, /* broj elemenata niza */
  size_t vel_el, /* vel. (u bajtima) jednog el. */
  int fun (const void *, const void *));
```

Za sortiranje ćemo koristiti već naučeni algoritam, *selection sort*. Umjesto dosadašnjeg poređenja poput `niz[i] < niz[j]` sada ćemo pisati nešto poput sljedećeg: `if (fun (niz[i], niz[j]) == 1)`. Također, kako ćemo morati mijenjati elemente niza, a ne znamo njihovu veličinu, morat ćemo njih konvertovati u pokazivač na `char`, pa mijenjati bajt po bajt jednog elementa. Za tu svrhu ćemo napisati i `zamijeni` funkciju.

```
1 #include <stdio.h>
2 void zamijeni (void *a, void *b, size_t vel) {
3   char *p = (char *)a;
4   char *q = (char *)b;
5   while (vel--) {
6     /* mijenjaj bajt po bajt */
7     char tmp = *p;
```

```
8     *p++ = *q;
9     *q++ = tmp;
10  }
11 }
12 typedef int (*f_poredi)(const void *, const void *);
13 void sortiraj (void *niz, size_t broj_el,
14                          size_t vel_el, f_poredi fun) {
15   char *p = (char *)niz;
16   size_t i, j;
17   for (i = 0; i < broj_el; ++i) {
18     max = i;
19     for (j = i + 1; j < broj_el; ++j)
20       if (fun (p + max * vel_el, p + j * vel_el) == 1)
21         max = j;
22     zamijeni (p + i * vel_el, p + max * vel_el, vel_el);
23   }
24 }
```

Analizirajmo napisane funkcije. U funkciji `zamijeni` smo mijenjali dva elementa na koje pokazuju pokazivači `a` i `b`. Pošto unaprijed ne znamo veličinu jednog elementa, mi imamo i dopunsku informaciju koja upravo to govori. U petlji unutar funkcije mijenjamo bajt po bajt određenog elementa, a to možemo uraditi jer smo naučili da tip `char` zauzima tačno jedan bajt (garancija standarda za jezik C).

U funkciji `sortiraj` je iskorišten alias za tip njenog posljednjeg parametra (funkcijskog pokazivača). Naredne dvije petlje su standardne za *selection sort* algoritam, prva iterira kroz čitav niz dok druga polazi od trenutnog elementa prve petlje te traži najveći od tog mjesta do kraja. Taj željeni poredak dobivamo pomoću parametra `fun`, odnosno pokazivača na funkciju kojom onaj koji sortira niz želi da odredi poredak. Pošto je pokazivač `p` zapravo pokazivač na `char`, onda ne možemo samo pisati `p + j` odnosno `&p[j]`, jer je element niza vjerovatno veći (zauzima više memorije) nego što to zauzima tip `char`. Zbog toga smo indeks množili sa veličinom elementa kako bismo riješili upravo spomenuti problem. Radi pojednostavljenja, ovu liniju funkcije napišimo pomoću indeksiranja:

```
22 if (fun (&p[max * vel_el], &p[j * vel_el]) == 1)
```

Ukoliko je `fun` funkcija vratila 1, to je značilo da elementi nisu u željenom poretku i potrebno ih je zamijeniti pomoću `zamijeni` funkcije. Napisanu `sortiraj` funkciju sada možemo koristiti da sortiramo proizvoljne nizove u proizvoljan poredak po proizvoljnom kriteriju. Bit će prikazano sortiranje niza cijelih brojeva u rastućem i opadajućem poretku bez mijenjanja funkcije za sortiranje:

```
25 int rastuci (const void *a, const void *b) {
26   int *p1 = (int *)a;
27   int *p2 = (int *)b;
28   if (*p1 > *p2)
29     return 1;
30   return 0;
31 }
32 int opadajuci (const void *a, const void *b) {   /* kraca verzija */
```

```
33   if (*(int *)a < *(int *)b)
34     return 1;
35   return 0;
36 }
37 void ispisi_niz (int *p, int vel) {
38   while (vel--)
39     printf ("%i ", *p++);
40   printf ("\n");
41 }
42 int main () {
43   int niz[] = {4, 2, 7, 0, -3, 7, 1, 4};
44   const int N = sizeof niz / sizeof *niz;
45   sortiraj (niz, N, sizeof *niz, rastuci);
46   ispisi_niz (niz, N);
47   sortiraj (niz, N, sizeof *niz, opadajuci);
48   ispisi_niz (niz, N);
49   return 0;
50 }
```

Funkcije `opadajuci` i `rastuci` se prosljeđuju `sortiraj` funkciji kao 4. parametar i one se nazivaju **funkcije kriterija**. U njima pretvorimo `void` pokazivače u željeni tip i odredimo poredak.

Na sličan način radi funkcija `qsort` iz "stdlib.h" biblioteke, s tim što je njen algoritam (najčešće *quick sort*) za sortiranje mnogo efikasniji. Zapravo, ona vjerovatno koristi više algoritama u zavisnosti koji je pogodniji za dati niz. Njen prototip je *identičan* prototipu naše `sortiraj` funkcije i to je glavni razlog zašto smo stavili da su parametri koji predstavljaju veličinu niza i broj bajta jednog elementa tipa `size_t` – upravo što je to tako kod `qsort` funkcije. Navedena funkcija se koristi isto kao i naša funkcija, pri čemu smo napravili jedno olakšanje kod funkcije kriterija. Naime, ona kod `qsort` funkcije ne vraća samo 0 ili 1, nego može dodatno vratiti i -1. Razlog za ovo je zahtjev algortima koji se koristi za sortiranje (pošto naš kriterij govori kada elementi nisu u željenom poretku, ali ne govori ništa kada su jednaki), pa je potrebno koristiti vrijednosti i za jednak ili pogrešan poredak. Funkcija kriterija koju prihvata `qsort` mora da vraća:

- -1 ako element na kojeg pokazuje `p1` treba da dođe *prije* elementa na kojeg pokazuje `p2`;
- 0 ako su elementi na koje pokazuju `p1` i `p2` jednaki;
- 1 ako element na kojeg pokazuje `p1` treba da dođe *nakon* elementa na kojeg pokazuje `p2`.

Zadatak 9. *Koristeći napisanu funkciju `sortiraj` napisati program koji niz cijelih brojeva sortira po zbiru cifara tako da elementi sa većim zbirom cifara dolaze ispred elemenata sa manjim zbirom cifara. Npr. broj 53 (sa zbirom cifara 8) dolazi prije broja 1211 (sa zbirom cifara 4). U slučaju da je zbir cifara jednak, prije treba doći veći broj. Također, u istom programu sortirati niz realnih brojeva u rastući poredak, a kriterij sortiranja treba*

1/17/0198

da bude sljedeći: broj sa manjim decimalnim dijelom dolazi prije broja sa većim decimalnim dijelom. Tako, broj 110.4 dolazi prije broja 0.9. U slučaju istih decimalnih dijelova ne treba raditi ništa (poredak nebitan) zbog problema poređenja realnih promjenljivih.

U ovom rješenju, radi uštede prostora, nećemo ponovo pisati funkcije `sortiraj` i `zamijeni`. Pretpostavit ćemo da se tu nalaze. Ono što trebamo uraditi jeste samo da napišemo funkcije kriterija za sortiranje nizova, te `main` funkciju u kojoj demonstriramo rad napisanih funkcija.

```
1  int zbir_cifara (int x) {
2    int c = 0;
3    while (x != 0) {
4      c += abs( x % 10 );
5      x /= 10;
6    }
7    return c;
8  }
9  int kriterij_int (const void *a, const void *b) {
10   int b1 = *(int *)a; /* pretvori i dereferenciraj */
11   int b2 = *(int *)b;
12   int c1 = zbir_cifara(b1);
13   int c2 = zbir_cifara(b2);
14   if (c1 == c2) /* ako je zbir cifara jednak... */
15     return b1 < b2; /* ...vracamo manji broj... */
16   return c1 < c2; /* ...inace vracamo manji zbir cifara */
17 }
18 int kriterij_dbl (const void *a, const void *b) {
19   double b1 = *(double *)a;
20   double b2 = *(double *)b;
21   double dec1 = b1 - (int)b1; /* uzimamo dio iza decimalnog zareza */
22   double dec2 = b2 - (int)b2;
23   return dec1 > dec2;
24 }
25 int main () {
26   int niz1[6] = {12, 1211, 104, 53, 0, 7}, i;
27   double niz2[6] = {12.1, 1.2, 104.3, 53.05, 11, 0.7};
28   sortiraj (niz1, 6, sizeof *niz1, kriterij_int);
29   sortiraj (niz2, 6, sizeof *niz2, kriterij_dbl);
30   printf ("Niz1: ");
31   for (i = 0; i < 6; ++i)
32     printf ("%8i", niz1[i]);
33   printf ("\nNiz2: ");
34   for (i = 0; i < 6; ++i)
35     printf ("%8.2f", niz2[i]);
36   return 0;
37 }
```

Funkcije su veoma jednostavne; a izraz `return dec1 > dec2` kako već znamo će vratiti 1 ili 0 u zavisnosti da li je uslov tačan ili ne, a naša `sortiraj` funkcija upravo treba da prihvati funkciju kriterija koja vraća 1 ili 0. Prikažimo i izlaz programa:

```
Niz1:       53       7    1211     104      12       0
Niz2:    11.00   53.05   12.10    1.20  104.30    0.70
```

Vidimo kako je prvi niz sortiran u opadajućem poretku po zbiru cifara. U slučaju da su dva broja jednaka po zbiru cifara (a to je slučaj

sa brojevima 1211 i 104 koji imaju zbir cifara 5), onda je veći broj (1211) prije manjeg broja. Drugi niz je sortiran u rastući poredak po necijelom (decimalnom) dijelu brojeva. Poredak je proizvoljan, koristimo istu funkciju, a sortiramo proizvoljne nizove po vlastitom kriteriju!

Već je spomenuto u potpoglavlju o generičkim pokazivačima↑ da njih ne treba upotrebljavati kada baš za to nemamo potrebe! U ovom slučaju sortiranja itekako imamo potrebe, sa jednom funkcijom možemo sortirati u proizvoljni poredak, a da ne pišemo različite verzije iste funkcije, gdje se samo uslov i tipovi razlikuju. Uostalom, i `qsort` funkcija je standardna funkcija, što znači da je opravdana potreba za njom.

↑ Poglavlje C.2.1, str. 367.

Ipak, trebamo biti *jako pažljivi* prilikom korištenja ovih funkcija, pa i naše `sortiraj` funkcije. U njoj vršimo (dozvoljenu) konverziju pokazivača, a naučili smo da konverzijom dajemo do znanja kompajleru da "znamo šta radimo", a kompajler nas neće upozoravati ni ako *ne znamo* šta radimo. Razmotrimo sada vrlo izglednu grešku koju ćemo napraviti ako ne vodimo računa, a kompajler nas neće upozoriti da pravimo grešku:

```
sortiraj (niz1, 6, sizeof *niz1, kriterij_dbl);
```

Ovdje je proslijeđen `niz1` koji ima elemente tipa `int`, a funkcija kriterija `kriterij_dbl` vrši konverziju pokazivača u tip `double`! Program se nalazi u nedefinisanom ponašanju kao što je to objašnjeno u potpoglavlju o konverzijama,↑ a posljedice su nepredvidljive. Povrh svega, najgore je što možemo imati programe koji izgledaju da rade (npr. da prosljeđujemo niz elemenata tipa `int` a funkcija za kriterij vrši konverziju u npr. `long`). Tada će to izgledno raditi na 32-bitnim sistemima (gdje `int` i `long` zauzimaju isto memorije) dok na 64-bitnim sistemima to vjerovatno nije slučaj i program neće raditi.

↑ Poglavlje C.2.2, str. 372.

Završimo priču o funkcijskim pokazivačima jednim kratkim programom koji demonstrira tačnost priče o prosljeđivanju funkcija koje će se kasnije pozivati na neke događaje. Takva funkcija je `atexit` iz "stdlib.h". Njoj možemo prosljeđivati funkcije koje će se pozivati kada se pozove `exit` funkcija za prekid rada programa. Prilikom poziva `atexit` funkcije ona *zapamti* sve proslijeđene funkcije, te ih poziva kada se pozove `exit` funkcija. Nju koristimo kada želimo osloboditi neke globalne resurse koji su zauzeti za vrijeme rada programa, kada želimo zabilježiti vrijeme završetka programa i slično. Ta funkcija može pamtiti najmanje 32 proslijeđene funkcije, a kada pozovemo `exit` funkciju, sve zapamćene funkcije će se pozvati obrnutim redoslijedom od redoslijeda dodavanja. Za pamćenje proslijeđenih funkcija koristi se niz pokazivača na funkcije (vjerovatno statički ili globalni).

Pogledajmo primjer:

```
1 #include <stdio.h>
```

1755/0198

```
#include <stdlib.h> /* atexit i exit */
void fun1() {
  printf ("Logiram podatke u app.log...\n");
  /* logiram u datoteku... */
}
void fun2()  {
  printf ("Obavjestavam korisnika o zatvaranju...\n");
  /* vjerovatno se zabiljezi i trenutno vrijeme... */
}
void ocisti () {
  /* cistim datoteke */
  printf ("Datoteke ociscene...\n");
}
void caoooo() {
  printf ("Caoooooo! Hvala na koristenju! :)");
}
int main() {
  /* funkcije se pozivaju OBRNUTIM redoslijedom */
  atexit (caoooo);
  atexit (&fun1); /* moze i bez & */
  atexit (ocisti);
  atexit (fun2);
  printf("Dobro dosli...\nUnesite bilo sta: ");
  getchar(); /* neki znak, kao scanf ("%c", &znak); */
  /* return 2;  ovo ne bi nista ispisalo... */
  /* gasi program, ALI i poziva registrovane fun. */
  exit (1);
}
```

Slijedi prikaz izlaza programa:

```
Dobro dosli...
Unesite bilo sta: e
Obavjestavam korisnika o zatvaranju...
Fajlovi ocisceni...
Logiram podatke u app.log...
Caoooooo! Hvala na koristenju! :)
```

C.5 *Greške prilikom kompajlerskih optimizacija*

U ovom poglavlju ćemo se kratko osvrnuti na neke probleme koji mogu nastati ukoliko su uključena optimiziranja od strane kompajlera. Kompajleri vrše optimizaciju kôda da bismo imali programe koji su efikasniji i čiji je izvršna datoteka manja. Čak i prilikom podrazumijevanih opcija kompajlera uključene su neke optimizacije (koje možemo naknadno isključiti). Optimizacije su dozvoljene ako kompajler "procijeni" da će izlaz programa biti isti, odnosno da će program imati iste funkcionalnosti bez optimizacija. Ipak, postoje slučajevi i gdje kompajler griješi, te zbog toga mu moramo pomoći označiti bitne promjenljive. Naravno, isključivanjem optimizacija bismo imali programe sa lošijim performansama i većim izvršnim datotekama, tako da su programi bez optimizacija većinom nepoželjni. Postoji kompromis! Moguće je i optimizirati programe i zadržati bitne promjenljive tako da programi i dalje budu ispravni.

1777098

Ovi problemi kod optimizacija su prisutni kada programiramo hardver ili višenitne (engl. *multithread*) programe (ili generalno procese koji dijele zajedničku memoriju), gdje se vrijednost neke zajedničke promjenljive može "neočekivano" izmijeniti, npr. očitavanje temperature ili kada drugi proces promijeni dijeljenu promjenljivu a kompajler ne očekuje da će biti mijenjana. Da bismo riješili ove probleme, jezik C nam omogućava dvije ključne riječi, `volatile` i `restrict` koje ćemo kratko objasniti.

C.5.1 Ključna riječ `volatile`

Sa ključnom rječju `volatile` (koja služi kao modifikator, slično kao i `const`) označavamo kompajleru da ne ukloni promjenljivu prilikom optimizacije; naglašavamo mu da se ta promjenljiva ipak može promijeniti (iako kompajler zaključuje obrnuto). Inače, "volatile" prevedeno sa engleskog, a u kontekstu jezika C, znači "nepostojan", "nestabilan" te "promjenljiv". Vrijede sva pravila za deklarisanje `volatile` promjenljivih kao i za `const` promjenljive:

```
int volatile x = 5;
/* isto... */
volatile int x = 5;
```

Dalje, možemo koristiti oba modifikatora, pa pisati nešto poput sljedećeg:

```
const volatile int x = 5;
```

Kako smo se upoznali sa načinom deklaracije `volatile` promjenljivih prikažimo problematičan primjer:

```
int *pok = (int *)0x12345;
while (*pok == 0) { radi nesto }
```

U ovom slučaju smo pokazivač `pok` usmjerili na adresu 0x12345, koja je dijeljena adresa i na kojoj npr. neki senzor upisuje vrijednosti. Ovdje smo napravili izuzetak da pokazivače smijemo inicijalizirati numeričkim vrijednostima, jer programiramo hardver i "znamo šta radimo". Kompajler, ipak, ne zna da će neko drugi pisati na adresi 0x12345 (jer, što bi, to je naš program). Zbog toga, kompajler jednom pročita ono šta se nalazi na adresi 0x12345 i npr. odredi da se tu nalazi neki broj različit od nule. Zbog toga primijeti da je petlja beskorisna! Primijeti da `*pok` neće nikad biti nula i odredi da se petlja neće ni izvršiti! U tome slučaju, ako su optimizacije uključene, on će ukloniti ovu petlju kao nepotrebnu!

Zbog toga, mi kompajleru moramo "savjetovati" da se ta promjenljiva može mijenjati izvana iako to ne izgleda za vrijeme kompajliranja. Deklarišemo promjenljivu da je `volatile` i problem je riješen! Prilikom optimiziranja kompajler će znati da promjenljiva bitna i da se njena

17570198

vrijednost može promijeniti, te da se i petlja može izvršiti! Rješenje je sljedeće:

```
volatile int *pok = (volatile int *)0x12345;
while (*pok == 0) { radi nesto}
```

C.5.2 *Ključna riječ* `restrict`

Ova ključna riječ postoji samo u jeziku C (i to od standarda C99), dok se razmatra njeno uključivanje u C++ jezik sa standardom C++17. Ona pomaže kompajleru da uradi optimizaciju odnosno da ukloni neke provjere pokazivača koje inače ne smije ukloniti. Samo se pokazivači mogu deklarisati sa ključnom riječi `restrict`, ne mogu "obične" promjenljive. Inače, u našim kôdovima se često javljaju situacije da više pokazivača pokazuje na istu promjenljivu (što se naziva *aliasing*). Kod `restrict` pokazivača, mi kompajleru označavamo da jedino taj pokazivač pokazuje na promjenljivu i da niti jedan drugi pokazivač neće pokazivati na tu promjenljivu dok `restrict` pokazivač pokazuje na nju. Garanciju održava *programer* odnosno mi! Ukoliko se desi da još neki pokazivač bude pokazivao na promjenljivu, program ulazi u nedefinisano ponašanje i posljedice su nepredvidljive. Zanimljivo je da je upravo ovo jedna od pet najčešćih grešaka gdje programeri pišu programe koji imaju nedefinisano ponašanje! (Ostala 4 slučaja su prekoračenje predznačnih tipova, pisanje bočnih efekata npr. `i = i++`, divlji pokazivači van niza i korištenje neinicijaliziranih promjenljivih.)

Kada neki pokazivač označimo da je `restrict`, tada kompajler *zna* da ono na šta on pokazuje neće biti promijenjeno, te može ubrzati odnosno optimizirati neke operacije. Prikažimo umjetni primjer gdje kompajleru možemo pomoći da optimizira kôd. Neka imamo funkciju koja množi dvije promjenljive na koje pokazuju njena dva parametra:

```
void fun (int *a, int *b)
  {
    *a *= (*b)++;
  }
```

Ipak, u ovom slučaju moguće je da i `a` i `b` pokazuju na *istu promjenljivu*:

```
int a = 5;
fun (&a, &a);
```

U tom slučaju kompajler ne može optimizirati napisani kôd jer prvo mora učitati vrijednost onoga na šta pokazuje `b` i biti siguran da to *nije ista* promjenljiva na koju i `a` pokazuje, jer bi inače imao višestruko modificiranje u jednom izrazu. Zbog toga imamo jednu asemblersku instrukciju više, te jedno učitavanje promjenljive. Ako mu mi ipak kažemo da su ovi pokazivači "ograničeni", tj. da mi garantiramo da niko drugi ne pokazuje na promjenljive, tada kompajler bez ikakve provjere može optimizirati kôd iznad i biti "siguran" da na promjenljivu ne pokazuje više pokazivača:

1/15/01/98

```
void fun (int * restrict a, int * restrict b) {
  *a *= (*b)++;
}
```

Naravno, tada *ne smijemo* napraviti aliasing (da dva pokazivača pokazuju na istu promjenljivu) kao u sljedećem pogrešnom primjeru koji ima nedefinisano ponašanje:

```
int a = 5;
fun (&a, &a); /* greska! ne smijemo! */
```

17570198

Indeks

špageti-kod, 61
1kk, 69
2kk, 69

ažuriranje (u zaglavlju for petlje), 90
abs, 73
adresa, 27, 221, 359
adresni operator, 221
algoritam, 144, 163
alias, 299
alternativa, 43, 48
ampersand, 34
anonimna struktura, 306
apostrof, 74
apsolutna putanja, 323
apsolutna vrijednost, 73
argc, argv, 389
argument funkcije, 187
arhitektura računara, 66
array, 160
ASCII, 23, 74
asocijacija, 51
asocijativnost operatora, 80

backslash, 22, 24, 33
bajt, byte, 27, 65
baza podataka, 338
beskonačna petlja, 93, 105
biblioteka, 22
binarne datoteke, 342
binarni brojevni sistem, 65
binarni operator, 36, 80
bit, 65
bitpolja, 316
blok koda, 23, 44, 102
bool, 46
Booleova algebra, 50
break, 97
broj u stringu, 282
brojač, 89, 136
buffer, 332

CamelCase, 301, 304
case, 59
case-sensitive, 272
cast, 76
char, 30, 74, 263
continue, 97
CSV, 333, 334
curenje memorije, 377

datoteka, 321
De Morganov zakon, 52
decimalna tačka, 36
definicija, 28, 40, 176, 299
dekadski brojevni sistem, 67
deklaracija promjenljive, 28
dereferenciranje, 226
dijagram toka, 43
dinamička alokacija memorije, 367
direktorij, 322
distribucija, 52
do-while, 107
dodavanje u niz, 289
dodjela, 35, 79
double, 30
DRY, 182
dvodimenzionalni niz, 160
dvostruka petlja, 100
dvostruki navodnici, 22
dvostruki pokazivač, 381
dvostruko brisanje, 378

eksterna memorija, 321
else, 44
enum, 300
EOF, 327, 329, 331
epsilon, 73
escape sekvence, 33, 326, 355
exit, 177

fabs, 73
fajl, 321
faktorijel, 119
false, 46
fclose, 325
feof, 330
ferror, 331
fgetc, 329
fgets, 332
file, 321
file pointer, 327
fizička veličina, 142
float, 30
flow chart, 43
folder, 322
fopen, 325
for petlja, 88
formatirane datoteke, 333
formatiranje, 31
formatni string, 353
fprintf, 335
fputc, 330
fputs, 332
fscanf, 333
fseek, 346
ftell, 346
funkcija, 175
funkcija kriterija, 414
funkcije bez parametara, 194

generički pokazivač, 367
getchar, 266
getline, 266
gets, 266
glavna dijagonala, 164
globalna promjenljiva, 203

17570198

gomila (memorija), 377
gornja trougaona matrica, 164
goto, 108
grananje, 43

heap, 377
heksadecimalni brojevni sistem, 68
heksadecimalno predstavljanje adresa, 233
hex editor, 348
histogram, 155

idempotencija, 51
identifikator, 30
if, 44
if nije petlja, 46
implicitna konverzija tipa, 81
indentacija, 46
inicijalizacija, 29
inicijalizacija (u zaglavlju for petlje), 90
inicijalizacija niza, 134
int, 29, 30
integer, 29
interna memorija, 321
intmax_t, 354
ispravan program, 140
istinosne tablice, 51
iterativno razmišljanje, 120
izbacivanje iz niza, 150
izlazak van opsega niza, 131
izuzetak, 171

kako rješavati zadatke?, 38, 101, 111
kalup, 76
karakter, 74, 263
klizeći pokazivač, 247, 269
kolekcija, 166
kombinovani operator, 37, 84
komentar, 41
kompajler, 25, 140
komplement dvojke, 69
komplement jedinice, 69
komutacija, 51
kondicionalni operator, 82
konstanta, 40, 362
konstantni pokazivač, 362
kontrola toka, 43
kontrolna promjenljiva, 89
konvergencija, 116
konvertovanje pokazivača, 372
korijenski direktorij, 322
krahiranje, 34, 132, 213
kratkospojna evaluacija, 54
kursor, 327, 346

l-vrijednost (l-value), 35, 226
labela, 61
leksikografski poredak, 273
LIFO - last in first out, 197
literal, 36
loader, 25
logička greška, 27, 141
logička promjenljiva, 123
logička veličina, 142
logički operator, 49, 79
logički tip, 46
lokalna promjenljiva, 187
lokalni prototip, 191
long, 30

mašinski jezik, 25
magična konstanta, 109, 271
main, 22, 389
maksimum, 145
male zagrade, 22
matematički operator, 79
matrica, 160
memorija, 27, 65, 321, 359
minimum, 145
mod, 326

n-terostruka petlja, 161
naša slova, 23
najmanji član niza, 145
najveći član niza, 145
NaN, 137
naredba dodjele, 35
nečiste funkcije, 202
nedefinisano ponašanje, 200, 228
negativna memorijska adresa, 361
negativno indeksiranje, 241
neinicijalizirana vrijednost, 29, 108, 142
neispisivi znakovi, 75
nepredznačni tip, 66
neutralni element, 51
nevalidan pokazivač, 236
niz, 129, 160
niz znakova, 263
NUL karakter, 264
nul pokazivač, 226
nul-terminiranje, 277
obične zagrade, 22
odsijecanje, 76
oktalni brojevni sistem, 68
opadajuća petlja, 96
opcija, 43
operand, 36
operator, 36, 79
operator dodjele, 79
operator ostatka, 36
operator poređenja, 46
operator tačka, 307
overflow, 78
oznake za formatiranje, 31

parametar funkcije, 187
parsiranje, 381
particija, 321
Pascalov trougao, 216
PC arhitektura, 66
petlja, 87
pobrojani tip, 300
podijeli pa vladaj, 101
podprogram, 193
podspecifikator, 354
podspecifikatori, 32
podstring, izbacivanje, 285
podstring, riječ, 284
podstring, traženje, 277
podstring, ubacivanje, 289
pokazivač, 221
pokazivač iza kraja niza, 359
pokazivač na datoteku, 327
pokazivač na niz, 238
pokazivač, dvostruki, 381
pokazivačka aritmetika, 239
polje, 129, 160
ponavljanje koda, 109
poravnanje memorije, 304
portabilnost, 251
postfiksni operator, 81
postoji barem jedan x, 124, 137
povratna vrijednost funkcije, 23
pow, 36, 73
poziv po referenci, 200, 229
poziv po vrijednosti, 199
pravilo "desno-lijevo", 407
prazan blok, 94
prebrojavanje, 136
prebrojavanje članova niza, 155
predznačni tip, 66
prefiksni operator, 81

1157018

prekidač, 65
prekoračenje, 78
preprocesorska direktiva, 22, 40
prevodilac, 25, 140
printf, 22
prioritet operatora, 80
procedura, 193
profiler, 29
promjenljiva, 27
prosjek, 136
prosti brojevi, 120
prototip funkcije, 191
ptrdiff_t, 361
punjač, 25
putanja, 323

RAM, 27
razlika pokazivača, 239
red (matematički), 116
rekurzivna funkcija, 211
relacioni operator, 46, 79
relativna putanja, 323
restrict pokazivači, 419
return, 23, 25, 177
return value, 23
rezervisana riječ, 31
riječ, 304

scope, 200
serijalizacija, 343
signed, 66
simbolička konstanta, 40, 362
simulacija indeksiranja, 246
sintaksna greška, 26, 141
size_t, 84
sizeof, 83
slog, 333, 339
sortiranje, 158
sporedna dijagonala, 164
sporedni efekat, 54, 79, 201
srednja vrijednost, 136
stablo direktorija, 322
stack, 196, 367
stdio.h, 22
stek, 196, 367
strcat, 275
strchr, 270
strcmp, 273
strcpy, 276
strelica operator, 315
string, 22, 263
string literal, 263
string.h, 267
strlen, 268
strncat, 275
strstr, 277
struct, 303
struktura, 303
strukturni tip podataka, 303
suma (matematička), 115
suma članova niza, 135
switch, 59

tačka, 307
tačka-zarez, 23
tabulator, 33
tag, 304
tekstualne datoteke, 342
tekući direktorij, 323
termininiranje rekurzije, 213
ternarni operator, 82
tijelo for petlje, 89
trodimenzionalni niz, 160
true, 46
typedef, 299

ubacivanje u niz, 153, 289
ugniježđena grananja, 57
ugniježđene funkcije, 178
ugniježđena petlja, 99
unarni operator, 36, 80
undefined behavior, 228
unesi funkcija, 266
Unicode, 356
unija, 317
union, 317
unsigned, 66
upozorenje, 26
uslov, 44
uslov (u zaglavlju for petlje), 90

valgrind, 29
varijabla, 27
velike zagrade, 22
višedimenzionalni nizovi, 160
višestruka petlja, 99
višestruki izbor, 44
viseći pokazivač, 256
vitičaste zagrade, 22
void, 183, 193, 367
volatile pokazivači, 418

while, 105

x86 arhitektura, 66

za svako x, 123, 137
zaglavlje for petlje, 89
zamjena dvije vrijednosti preko pomoćne, 159
zamjena stringa, 291
zaokruživanje, 76
znak, 74, 263
znakovni literal, 263
znakovni tip, 74

17570198

1757018 8

Popis slika

1.1 Pojednostavljeno predstavljanje memorije i promjenljive 28
1.2 Način uvrštavanja formata u pozivu `printf` funkcije 31
1.3 Osnovni princip rada svakog računarskog programa 39

2.1 Opcija 44
2.2 Alternativa 44
2.3 Višestruki izbor 44

3.1 Prekidačka logika 66

4.1 Petlja ili špageti kod? 87
4.2 Struktura for petlje 89
4.3 Zaglavlje for petlje 90
4.4 Zloupotreba zaglavlja for petlje 94
4.5 Transformacija for petlje u while petlju i obrnuto 107
4.6 Transformacija do-while petlje u while petlju 108
4.7 Redoslijed kretanja po scan-linijama 111
4.8 Crtanje oblika pomoću koordinatnog sistema 114

5.1 Izlazak iz opsega niza lako može dovesti do ovakvih situacija 132
5.2 Gornja trougaona matrica 165

6.1 Funkcije nekada zamišljamo kao crne kutije 176
6.2 Lokalne promjenljive iz `main` funkcije na steku 196
6.3 Lokalne promjenljive iz `main` i `fun` funkcija 197
6.4 I parametri `fun` funkcije su lokalne promjenljive 198
6.5 Lokalne promjenljive iz `main` i `fun` funkcija 199
6.6 Cvijet suncokreta je jedan primjer fraktalnog oblika koji se pojavljuje u prirodi 212

7.1 Memorijski model nakon deklarisanja promjenljive broj 223
7.2 Memorijski model nakon dodjele 223
7.3 Zauzeta je još jedna lokacija: pokazivac 223
7.4 Konačno stanje nakon "povezivanja" 224
7.5 `b1` i `b2` se nalaze na fiktivnim adresama 0x200 i 0x300 233

17570198

7.6 `prvi` i `drugi` pokazuju na `b1` i `b2` respektivno 233
7.7 Promjenljiva temp je na adresi 0x900 234
7.8 Korist `temp` promjenljive za zamjenu 234
7.9 Vraćena je vrijednost iz `temp` u `*drugi` (`b2`) 234
7.10 Lokalne promjenljive su uništene a promjenljive `b1` i `b2` su zamijenile vrijednosti 235
7.11 Promjenljiva `pok` pokazuje na prvi element niza 238

9.1 Izgled promjenljive `struct Datum d` u memoriji. Ovdje "adrese" predstavljaju apstraktne memorijske lokacije pri čemu svaka promjenljiva zauzima jednu lokaciju. 305
9.2 Organizacija strukture Vrijeme u memoriji 316

10.1 Windows Explorer – standardni upravitelj datotekama na operativnom sistemu Windows 322
10.2 Značenje specijalnih oznaka "." i ".." 323
10.3 Standardna hijerarhija direktorija operativnog sistema Windows 324
10.4 Standardna hijerarhija direktorija operativnog sistema Linux 325

C.1 Svaka memorijska ćelija se može adresirati sa `char *` 370
C.2 Pojednostavljenje memorijskog stanja – dvostruki pokazivači 383

17570198