Uvod u programiranje kroz programski jezik C

Dragan Matić

Elektronska verzija

Prirodno – matematički fakultet
Univerzitet u Banjoj Luci
2018.

Dragan Matić
Uvod u programiranje kroz programski jezik C

Izdavač
Prirodno-matematički fakultet
Univerzitet u Banjoj Luci

Recenzenti
prof. dr Ilija Lalović, vanredni profesor,
Prirodno-matematički fakultet, Univerzitet u Banjoj Luci
prof. dr Vladimir Filipović, vanredni profesor,
Matematički fakultet, Univerzitet u Beogradu

Lektura
Slobodan Marković, profesor srpskog jezika i književnosti

CIP - Каталогизација у публикацији
Народна и универзитетска библиотека
Републике Српске, Бања Лука

004.42(075.8)
004.432.2C(075.8)

МАТИЋ, Драган, 1977-
Uvod u programiranje kroz programski jezik C / Dragan Matić. - Banja Luka : Univerzitet u Banjoj Luci, Prirodno-matematički fakultet, 2018 (Banja Luka : Akademac BL). - VIII, 261 str. : graf. prikazi, tabele ; 25 cm

Tiraž 150. - Bibliografija: str. 261.

ISBN 978-99955-21-70-7

COBISS.RS-ID 7788824

Elektronska verzija

Uvodna riječ

Ovaj udžbenik je namijenjen studentima koji slušaju uvodni kurs programiranja zasnovan na programskom jeziku C. Po svojoj strukturi i sadržaju udžbenik najviše prati nastavni plan i program za predmet Uvod u programiranje za studente prve godine nastavnog smjera na Studijskom programu tehničko vaspitanje i informatika Prirodno-matematičkog fakulteta Univerziteta u Banjoj Luci. Pored toga, udžbenik mogu koristiti svi oni koji žele da steknu elementarno znanje iz programiranja.

Pretpostavljamo da čitaoci ovog udžbenika u dovoljnoj mjeri vladaju elemenatarnim znanjem iz osnova informatike, koji uključuje rad sa folderima i datotekama, te rad u nekom od jednostavnijih razvojnih okruženja koja su namijenjena razvoju programa pisanih u programskom jeziku C.

Cilj udžbenika nije da on bude samo "vodič" za programski jezik C, već da se jednim sistematskim pristupom, uz upotrebu mehanizama koje nam pruža ovaj programski jezik i kroz prezentaciju odgovarajućih karakterističnih primjera, obrade najvažniji koncepti elemenata programiranja uopšte, kao i najvažniji dodatni koncepti koji su direktno vezani za sam programski jezik C.

Uz pomoć znanja stečenog iz ovog udžbenika, polaznik će biti sposoban da upravlja osnovnim i složenijim tipovima podataka, modeluje i implementira algoritme zasnovane na uslovnim i iterativnim naredbama i rekurziji, koristi mehanizam pokazivača za upravljanje memorijom i dinamičkim strukturama, upravlja ulaznim i izlaznim tekstualnim datotekama, te koristi funkcije standardne biblioteke programskog jezika C.

Unutar svakog poglavlja prikazan je i značajan broj primjera koji mogu pomoći lakšem razumijevanju gradiva. Neki primjeri su urađeni kompletno, od početka do kraja, dok neki drugi sadrže prikaze koraka koji su ključni za razumijevanje datog koncepta. Treba pomenuti da u programiranju vrlo često postoji više podjednako dobrih i efikasnih načina da se riješi neki problem ili zadatak. Stoga se od čitaoca ne očekuje da predložena rješenja usvoji kao jedina moguća, već da analizom ponuđenih rješenja i povezivanjem raznih koncepata i tehnika programiranja sam dođe do svojih rješenja, koja su podjednako ispravna i efikasna.

Na kraju svakog poglavlja dat je veći broj pitanja i zadataka. Cilj ovih zadataka nije da čitalac samo reprodukuje stečeno znanje, već da sam stekne

Elektronska verzija

sposobnost da modeluje i implementira algoritme kojima rješava zadatke, procjenjuje kvalitet rješenja, pronalazi rješenja za razne probleme, koji, na prvi pogled, nisu u bliskoj vezi sa gradivom iznesenim u udžbeniku.

Mnogi koncepti programiranja se međusobno preplića. Na primjer, teško je u potpunosti shvatiti pojam promjenljive dok se ne savladaju tipovi podataka i oblasti važenja identifikatora. Nizovi se ne mogu u potpunosti razumjeti dok se ne povežu sa pokazivačima. Princip formatiranog unosa podataka sa standardnog ulaza ne možemo u potpunosti shvatiti dok ne savladamo koncepte funkcija i pokazivača, prenosa argumenata u funkciju itd. Stoga je preporuka da se ovaj udžbenik ne čita isključivo redom, već da se, čak i mnogo češće nego što je to u tekstu sugerisano, sadržaj više poglavlja izučava u isto vrijeme, vraća na prethodna poglavlja ili preskače na neka naredna.

Za uspješno savladavanje elemenata programiranja potreban je strpljiv i studiozan rad. Čitaocu se preporučuje i da sve prezentovane primjere testira na svom računaru, da dodatno analizira programski kôd, mijenja uslove i vrijednosti parametara i razmišlja unaprijed kako izmjene kôda utiču na ponašanje programa. Neki zadaci podrazumijevaju i dodatni angažman, koji uključuje prikupljanje potrebnih informacija i iz drugih izvora (na primjer iz pouzdanih internet izvora), što je u programerskoj praksi standardan i uobičajen metod za učenje i napredovanje u struci.

Na kraju, želim da se zahvalim recenzentima, prof. dr Iliji Laloviću i prof. dr Vladimiru Filipoviću, koji su svojim sugestijama i komentarima poboljšali kvalitet ovog udžbenika. Takođe, zahvaljujem se na podršci i kolegama sa Katedre za računarstvo i informatiku Prirodno-matematičkog fakulteta Univerziteta u Banjoj Luci, kolegama sa Katedre za računarstvo Matematičkog fakulteta u Beogradu, kao i kolegama sa Matematičkog instituta Srpske akademije nauka i umjetnosti.

Posebno se zahvaljujem koleginici Milani Grbić na nesebičnoj pomoći u pripremi velikog broja zadataka koji su uključeni u ovaj udžbenik.

Zahvaljujem se svojoj porodici, supruzi Dragani i svojoj djeci Nataši i Milanu, koji su moj najveći životni uspjeh.

Elektronska verzija

Sadržaj

1 Uvod **1**
- 1.1 Verzije i standardizacija programskog jezika C 2
- 1.2 Organizacija izvornog kôda i prevođenje programa 4
  - 1.2.1 Pisanje izvornog kôda 4
  - 1.2.2 Prevođenje programa 6
- 1.3 Nekoliko jednostavnih programa 7
- 1.4 Pitanja i zadaci 11

2 Osnovni elementi programskog jezika C **13**
- 2.1 Identifikatori 13
- 2.2 Ključne riječi 14
- 2.3 Promjenljive. Deklaracija promjenljivih 15
  - 2.3.1 Konstante 16
- 2.4 Osnovni tipovi podataka 17
  - 2.4.1 Cjelobrojni tipovi podatka 18
  - 2.4.2 Realni tipovi podataka 20
  - 2.4.3 Znakovni tip podatka 22
- 2.5 Izrazi i operatori 24
  - 2.5.1 Izrazi 24
  - 2.5.2 Operatori 24
  - 2.5.3 Konverzija tipova 38
  - 2.5.4 Enumerativni tipovi 42
  - 2.5.5 Ključna riječ typedef 43
- 2.6 Pitanja i zadaci 44

3 Vrste programskih naredbi **47**
- 3.1 Naredbe izraza 47
- 3.2 Složene naredbe 48
- 3.3 Naredba grananja 49
  - 3.3.1 if-else naredba 49
  - 3.3.2 switch naredba 51
- 3.4 Naredbe ponavljanja 56
  - 3.4.1 Naredba ponavljanja while 57

Elektronska verzija

3.4.2 Naredba ponavljanja for . . . 59
3.4.3 Naredba ponavljanja do-while . . . 62
3.4.4 Naredbe break i continue . . . 63
3.4.5 Beskonačne petlje . . . 68
3.4.6 Ugniježdene petlje . . . 70
3.5 Pitanja i zadaci . . . 72

4 Funkcije **77**
4.1 Deklarisanje i definisanje funkcija . . . 77
4.2 Rekurzivne funkcije . . . 80
4.3 Prenos argumenata u funkciju . . . 81
4.4 Funkcije sa promjenljivim brojem argumenata . . . 84
4.5 Pitanja i zadaci . . . 87

5 Naredbe ulaza i izlaza **91**
5.1 Funkcije getchar i putchar . . . 92
5.2 Funkcije gets i puts . . . 93
5.3 Funkcije formatiranog unosa i ispisa . . . 95
5.3.1 Funkcija formatiranog unosa . . . 95
5.3.2 Funkcija formatiranog ispisa . . . 97
5.3.3 Funkcije sscanf i sprintf . . . 101
5.4 Pitanja i zadaci . . . 102

6 Nizovi **105**
6.1 Deklaracija niza . . . 105
6.2 Nizovi kao argumenti funkcija . . . 112
6.3 Višedimenzionalni nizovi . . . 114
6.4 Niske karaktera . . . 120
6.5 Pitanja i zadaci . . . 122

7 Doseg i vijek trajanja promjenljivih **127**
7.1 Doseg promjenljivih . . . 127
7.1.1 Lokalne promjenljive . . . 128
7.1.2 Globalne promjenljive . . . 131
7.2 Vijek trajanja promjenljivih . . . 133
7.2.1 Automatske promjenljive . . . 133
7.2.2 Statičke promjenljive . . . 135
7.3 Organizacija programa u više datoteka . . . 137
7.3.1 Spoljašnje promjenljive i globalne statičke promjenljive . . . 138
7.3.2 Pristup funkcijama iz drugih datoteka . . . 139

Elektronska verzija

7.3.3 Pristup promjenljivima iz drugih datoteka . . . 141
7.4 Pitanja i zadaci . . . 144

8 Pokazivači **147**
8.1 Sintaksa pokazivača . . . 148
8.2 Pokazivači kao argumenti funkcija . . . 151
8.3 Pokazivačka aritmetika . . . 154
8.4 Konstantni tipovi i pokazivači . . . 157
8.5 Nizovi i pokazivači . . . 158
8.6 Pokazivači i niske . . . 161
8.6.1 Zaglavlje string.h . . . 163
8.7 Pokazivači i višedimenzionalni nizovi . . . 168
8.8 Argumenti komandne linije . . . 170
8.9 Pokazivači na funkcije . . . 172
8.10 Pitanja i zadaci . . . 177

9 Strukture i unije **181**
9.1 Strukture . . . 181
9.1.1 Operacije na strukturama . . . 185
9.1.2 Pokazivači na strukture . . . 186
9.1.3 Strukture kao argumenti funkcija . . . 187
9.1.4 Nizovi struktura . . . 188
9.2 Unije . . . 190
9.3 Pitanja i zadaci . . . 192

10 Datoteke **195**
10.1 Otvaranje i zatvaranje datoteka . . . 195
10.2 Funkcije za čitanje i pisanje u i iz datoteke . . . 198
10.2.1 Čitanje i pisanje pojedinačnih karaktera . . . 198
10.2.2 Čitanje i pisanje linije teksta . . . 200
10.2.3 Prepoznavanje grešaka prilikom rada sa datotekama . . . 202
10.2.4 Funkcije formatiranog ulaza i izlaza . . . 203
10.3 Pitanja i zadaci . . . 204

11 Algoritmi za sortiranje nizova **207**
11.1 Osnovni pojmovi o sortiranju . . . 207
11.1.1 O analizi vremenske složenosti algoritma . . . 210
11.2 Sortiranje izborom . . . 211
11.3 Sortiranje umetanjem . . . 213
11.4 Sortiranje pomoću mjehurića . . . 214

Elektronska verzija

11.5 Brzi sort 215
11.6 Sortiranje spajanjem 217
11.7 Pitanja i zadaci 219

12 Dinamička alokacija memorije 223
12.1 Organizacija memorije C programa 224
12.1.1 Segment za smještanje programskog kôda 224
12.1.2 Segment za smještanje podataka 224
12.2 Funkcije za dinamičku alokaciju memorije 226
12.2.1 Funkcija malloc 226
12.2.2 Funkcija calloc 227
12.2.3 Funkcija realloc 228
12.2.4 Funkcija free 229
12.3 Dinamičke strukture podataka 230
12.3.1 Dinamički nizovi 230
12.3.2 Dinamički alocirane matrice 233
12.3.3 Liste kao dinamičke strukture 236
12.4 Pitanja i zadaci 248

13 Preprocesorske naredbe 251
13.1 Naredba #include 251
13.2 Naredba #define 252
13.2.1 Parametrizovana naredba #define 254
13.3 Uslovno prevođenje 257
13.4 Pitanja i zadaci 259

Literatura 261

Elektronska verzija

1 Uvod

Kada god se srećemo sa novim programskim jezikom, novom platformom ili novim okruženjem za razvoj programa, bilo da smo početnici ili već iskusni programeri, nepisano je pravilo da prvo napišemo i testiramo neki od najlakših i najjednostavnijih programa. U terminologiji programiranja, ovaj ili ovakvi programi se obično nazivaju“Hello World”, ili na našem jeziku, “Zdravo svijete” programi.

Evo programa “Zdravo svijete” u programskom jeziku C.

```c
#include <stdio.h>

int main()
{
  printf("Zdravo svijete! \n");
  return 0;
}
```

Svaki samostalan program pisan u programskom jeziku C sadrži glavnu (engl. main) funkciju, koja je početna tačka za izvršenje programa. Funkcije mogu, ali i ne moraju da “vrate” neku vrijednost kao rezultat. U slučaju da funkcija vraća neki rezultat, onda se ispred naziva funkcije mora navesti tip podatka rezultata. `main` funkcija standardno vraća cio broj kao rezultat (zbog toga ispred riječi `main` stoji riječ **`int`**), a uobičajeno je da se kao rezultat izvršenja `main` funkcije vrati rezultat nula, ukoliko je program uspješno došao do kraja. Rezultat funkcije se vraća korištenjem naredbe **`return`**. Tijelo funkcije, tj. naredbe koje se izvršavaju prilikom poziva funkcije se “grupišu” u vitičaste zagrade.

Funkcija `printf` se koristi za ispisivanje podataka na standardni izlaz (ekran). U petom redu programa naveden je najjednostavniji način upotrebe funkcije `printf`, gdje se u okviru malih zagrada, unutar znaka navaodnika navodi tekst koji će se ispisati na ekranu. Funkcija `printf` je dio standardne biblioteke funkcija (engl. standard library) koje se isporučuje zajedno se instalacijom samog prevodioca. Definicija te funkcije, zajedno sa drugim funkcijama koje se koriste za ispisivanje i unošenje podataka je smještena unutar fajla `stdio.h`. Da bi se biblioteke funkcija uključile u korisnikov program, koristi

Elektronska verzija

se tzv. preprocesorska direktiva `#include <naziv_biblioteke>`. Preprocesorske direktive su instrukcije programskog jezika C, koje se izvršavaju u početnoj (nultoj) fazi kompajliranja i podrazumijevaju uključivanje kompletnog sadržaja navedene biblioteke. Datoteke koje se uključuju u zaglavlju programa, prije programskog koda koji se odnosi na sam program se i zovu datoteke zaglavlja, ili na engleskom jeziku header files.

Komentari su dijelovi programskog koda koji su ignorisani od strane prevodioca, a koristimo ih kada je potrebno da dodatno pojasnimo naredbe i druge elemente programa koje koristimo. U upotrebi komentara treba biti umjeren. Iako ni prevelika količina komentara neće uticati na sam tok i izvršenje programa, pretjerano komentarisanje očiglednih elemenata nije poželjno, jer bespotrebno opterećuje programski kôd i čini ga glomaznim i težim za razumijevanje. Sa druge strane, komentarisanje ključnih dijelova programa kao što su pozivi funkcija, komplikovani blokovi iterativnih naredbi, neuobičajene naredbe dodjele, složene uslovne naredbe i slično, nekada može dovesti do značajnog skraćenja vremena koje je potrebno da se razumije programski kôd.

U većini modernih programskih jezika, pa tako i u programskom jeziku C, komentare pišemo između znakova `/*...*/`. Ukoliko pišemo komentar samo u jednom redu, onda možemo da koristimo i znak `//` i sav tekst nakon ovog znaka, pa do kraja reda, se smatra komentarom. Treba napomenuti da neki stariji prevodioci ne prihvataju ovaj način komentarisanja, ali je u današnje vrijeme pojava takvih prevodilaca previše rijetka da bi se tome pridavao značaj. U nastavku udžbenika ćemo često koristiti komentare linija kôda, kako bismo lakše i "direktnije" objasnili pojedine izraze i naredbe.

1.1 Verzije i standardizacija programskog jezika C

Programski jezik C spada u imperativne programske jezike opšte namjene. Prvobitno je programski jezik C bio namijenjen pisanju sistemskih programa i jezgra operativnog sistema UNIX, koji je praktično i napisan u jeziku C. Zamišljen kao jednostavan i fleksibilan, programski jezik C je vrlo brzo postao jedan od najpopularnijih jezika svog vremena, što je ostao i do današnjeg dana. Jedan je od najvažnijih programskih jezika u razvoju komercijalne računarske industrije, koji je prilagođen većini računarskih platformi, od malih sistema, pa do super-računara. Programi napisani u ovom jeziku su često bliski načinu rada hardvera, te od programera zahtijevaju da dobro razumije rad procesora, memorije, ulazno-izlaznih uređaja itd.

Programski jezik o kome se govori u ovoj knjizi ne nosi neki baš kreativan naziv. Dennis Ritchie, autor jezika C, je sedamdesetih godina dvadesetog vijeka

Elektronska verzija

dizajnirao ovaj jezik, a ime mu je dao kao “naredni” jezik jezika B. Značajan doprinos nastanku jezika C dali su i Ken Thompson koji je upravo autor programskog jezika B i Martin Richards, autor programskog jezika BCPL.

Programski jezik C se u određenoj mjeri i mijenjao/usavršavao tokom godina, a u nekoliko navrata je i formalno standardizovan. Važan pečat standardizaciji su dali Brian Kernighan i Dennis Ritchie, autori najpoznatije knjige o ovom programskom jeziku ”The C Programming Language”, koja datira iz 1978. godine. Smatra se da je sadržajem ove knjige napravljena i neformalna standardizacija čitavog jezika, pa se i verzija programskog jezika C koja je opisana u ovoj knjizi, prema inicijalima autora ponekad zove i **K&RC**. Kako se upotreba jezika C godinama značajno proširila na veći broj različitih platformi, javila se potreba za zvaničnom standardizacijom. Američki nacionalni institut za standardizaciju (engl. American National Standards Institute - ANSI) je 1983. godine formirao komitet za uspostavljanje standardne specifikacije programskog jezika C, a sam standard je završen i objavljen 1989. godine. Ova standardizovana verzija se često nazivaju i **ANSI C** ili **C89**. Isti standard je naredne, 1990. godine potvrđen i od strane Međunarodne organizacije za standardizaciju (engl. International Organization for Standardization - ISO), te se ta ista verzija jezika nekada označava i sa **C90**.

Intenzivan razvoj računarskih tehnologija u devedesetim godinama dvadesetog vijeka uticao je na potrebu za novom standardizacijom, koja je usvojena 1999. godine. Ovim standardom, koji se često označava **C99**, uvedene su neke izmjene u odnosu na prethodne verzije, kojima se postiže bolja usklađenost sa drugim programskim jezicima (najprije sa C++-om), ali i sa drugim standardima u računarstvu (kao što je dodatna podrška za IEEE 754 standard koji se odnosi na brojeve u pokretnom zarezu).

Verzijom **C11** izvršena je nova standardizacija koja se najprije odnosi na jasan i precizan opis modela za višenitno (engl. multithreading) izračunavanje.

Posljednja verzija standarda, pod zvaničnim nazivom ISO/IEC 9899:2018, objavljena je u junu 2018. godine. Ovim najnovijim standardom obezbijeđena je podrška nekim široko korištenim kolekcijama platformi za prevođenje programa pisanim u raznim programskim jezicima.

U nastavku udžbenika, kada budemo naglašavali specifičnosti jezika u odnosu na pojedine verzije, koristićemo odgovarajuće skraćene nazive verzija (na primjer C89, C99 i sl.).

Elektronska verzija

1.2 Organizacija izvornog kôda i prevođenje programa

U većini viših programskih jezika, u koje spada i programski jezik C, programe pišemo u takozvanom izvornom kôdu (engl. source code). Izvorni kôd programa je zapravo tekstualna datoteka, kojoj je pridružena odgovarajuća ekstenzija. Ekstenzija datoteka koje sadrže izvorni kôd programa pisanih u programskom jeziku C ima ekstenziju `.c`. Izvorni kôd se dalje u procesu prevođenja (u našem jeziku često koristimo i engleski izraz kompajliranje, koja potiče od engleske riječi compile) prevodi u drugi oblik - izvršni program, koji je napisan na mašinskom jeziku, tj. jeziku koji računar "razumije", tj. može da pokrene i izvrši.

1.2.1 Pisanje izvornog kôda

Kao i druge tekstualne datoteke, izvorni je kod moguće pisati u bilo kojem programu za obradu teksta. Sa druge strane, u programerskom svijetu je uobičajeno da se programi za pisanje izvornog koda objedinjuju u istu cjelinu sa prevodiocem i povezivačem. Ovakve programe nazivamo integrisana razvojna okruženja (od engleskog integrated development enviroment - IDE). Izvorni kôd programa u programskom jeziku C čuvamo, dakle, kao tekstualnu datoteku pod nekim smislenim imenom, vodeći računa da datoteka dobije ekstenziju `*.c`. Datoteka koja sadrži izvorni kôd se dalje prevodi (kažemo i kompajlira) i, u slučaju uspješnog prevođenja, može da se pokrene i izvrši.

Prilikom pisanja programa treba poštovati veliki broj "pisanih" i "nepisanih" pravila. Najprije se moraju poštovati pravila koja proističu iz standarda samog jezika. Evo samo nekoliko primjera takvih pravila. Programski jezik C pravi razliku između velikih i malih slova. Na primjer, naredbu grananja pišemo malim slovima **`if`**, nakon čega slijedi uslov koji se piše u maloj zagradi, dok riječi `IF` ili `If` nisu ključne riječi i mogu se čak koristiti i kao identifikatori (što doduše nije preporučljivo, jer može dovesti do zabune). Ključne riječi ne možemo koristiti kao identifikatore. Identifikator (npr. naziv promjenljive) ne može početi cifrom. Postoje i druga pravila, o kojima će kasnije biti riječi, kojima je definisano šta može, a šta ne može biti identifikator. Dalje, ako želimo da se više naredbi izvrši unutar bloka (više jednostavnijih naredbi grupišemo u jednu složenu), moramo koristiti vitičaste zagrade `{` i `}`. Uglaste zagrade `[` i `]` koristimo za pristup elementima niza i ne možemo ih (kao na primjer u složenijim izrazima u matematici) koristiti za definisanje redoslijeda izvršavanja matematičkih operacija. Za grupisanje manjih izraza u većem koristimo male zagrade `(` i `)`. Argumente funkcije odvajamo zarezom. Niske karaktera pišemo u okviru dvostrukih navodnika, a pojedinačne karaktere u okviru jednostrukih.

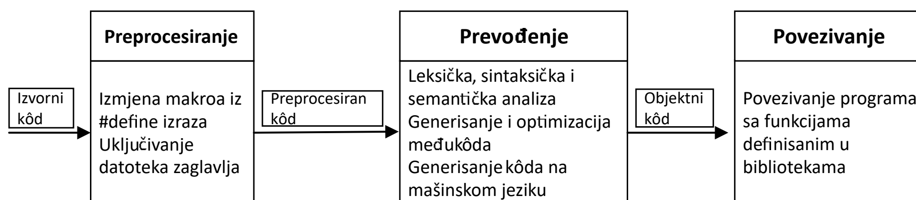
Elektronska verzija

O ovim, ali i o mnogim drugim pravilima će biti puno više riječi u nastavku udžbenika.

Pored pravila samog programkog jezika, koje svaki programer mora poštovati, postoje i razne konvencije i običaji pisanja izvornog kôda koje olakšavaju kako samo programiranje, tako i razumijevanje programa. Kod manjih programa, koji se sastoje od nekoliko desetina redova izvornog kôda, nije toliko uočljiva potreba za "lijepim pisanjem". Međutim, ako se u okviru programa radi sa desetinama promjenljivih, velikim brojem korisnički definisanih tipova podataka, velikim brojem funkcija koje se međusobno pozivaju itd. poštovanje kodeksa postaje ključno, kako za sam razvoj, tako i za kasnije održavanje i ponovnu upotrebu programa. Pored toga, postoje i pravila koja omogućavaju efikasno izvršavanje programa na određenom računaru, kojima se omogućava ili olakšava pristup memoriji ili se omogućava istovremeno izvršavanje više programa. Stoga, neka pravila organizacije kôda zavise i od samog računara (operativnog sistema, hardvera i mašinskog jezika) na kom se planira pokretanje i izvršavanje programa.

Neke konvencije su prirodne i očekivane, tako da ih se pridržavamo, čak i ako ranije nismo čuli za njih. Na primjer, sasvim je prirodno da identifikatorima dajemo takva imena koja ukazuju na njihovu namjenu. Imena promjenljivih, koje su indeksi niza, su obično `i`, `j`, `k` itd. Ako funkcija računa zbir brojeva, prirodno je da je nazovemo `suma` ili `zbir` i slično. Pored toga, prirodno je da naredbu započinjemo u novom redu (posebno one duže) i da deklaracije prototipa funkcija pišemo u jednom redu. Izbjegavamo i pisanje predugih linija kôda. Komentare u kôdu pišemo tako da oni budu smisleni i koncizni, ali i pravopisno ispravno napisani. Dalje, prirodno je da ključne riječi i druge dijelove samog jezika nećemo "redefinisati" preprocesorskim direktivama, Neke druge konvencije, kojih možemo, ali i ne moramo da se pridržavamo, mogu uključivati sljedeće: upotreba vitičastih zagrada za svaki blok naredbi, čak i onda kada taj blok sadrži samo jednu naredbu, vitičaste zagrade (i otvorenu i zatvorenu) pišemo u zasebnim redovima, između naziva promjenljivih i simbola operatora ubacujemo razmak. Male zagrade kojima grupišemo izraze pišemo što je češće moguće, kako bismo izbjegli neželjene situacije u kojima se operacije u izrazu izvršavaju na neočekivan način. Konvertovanje podataka iz jednog tipa u drugi dodatno komentarišemo u slučaju da postoji mogućnost "gubljenja" informacija. Pored ovih konvencija, postoje i konvencije koje ukazuju na sam stil programiranja, kao što su: rijetko korištenje pojedinih naredbi (na primjer naredbe **`goto`**, ali i drugih naredbi skoka, na primjer naredbe **`continue`**), izbjegavanje upotrebe nekih ključnih riječi (na primjer **`auto`** ili **`register`**) itd. Postoje i konvencije koje se mogu poštovati i kada radimo sa postojećim tipovima podataka i operacijama na njima, kada uvodimo nove tipove podataka, kada

Elektronska verzija



Slika 1.1: Faze prevođenja programa

definišemo i pozivamo funkcije, itd. O nekim od ovih konvencija će biti govora i u nastavku udžbenika.

1.2.2 Prevođenje programa

Prevođenje programa se vrši u nekoliko faza, od kojih se i svaka faza sastoji od nekoliko koraka. Prevođenje programa koji su pisani u programskom jeziku C počinje preprocesiranjem (engl. preprocessing), nakon koje ide glavni proces prevođenja, za koji se nekada baš i koristi termin prevođenje, kompilacija ili kompajliranje, te, na kraju, povezivanje (engl. linking). Pojednostavljen prikaz, u kome su naznačene samo osnovne faze je predstavljen na Slici 1.1.

Faza preprocesiranja

U okviru faze preprocesiranja vrše se jednostavne operacije nad samim tekstom izvornog kôda, kao što su uklanjanje komentara i interpretacija specifičnih naredbi (preprocesorskih direktiva).

Detaljniji opis ove faze biće prikazan u Poglavlju 13.

Faza prevođenja

Kako i sam naziv faze ukazuje, u ovoj fazi se vrši prevođenje izvornog kôda. Prevođenje se sastoji od nekoliko podfaza.

Najprije se vrši leksička analiza, u okviru koje se izvorni kôd čita karakter po karakter i identifikuju se osnovni jezički elementi koje nazivamo *leksemima*. Svakom od leksema se pridružuje odgovrajuća leksička kategorija i jedinstvena oznaka, koja se naziva *token*. Na primjer, tokeni su: identifikator, operator, separator i dr.

Po završetku leksičke, slijedi sintaksna analiza, u okviru koje se provjerava da li su leksemi sklopljeni u skladu sa pravilima programskog jezika. Rezultat koji

Elektronska verzija

se dobija na kraju sintaksne analize je kreiranje tzv. sintaksnog stabla (engl. syntax tree, a koristi se i izraz parse tree).

U sljedećoj fazi se vrši semantička analiza, tj. provjera tipova i deklaracija u sintaksnom stablu. Takođe, vrši se i implicitna konverzija kod operatora.

Nakon završene sintaksne i semantičke analize, prevodilac generiše tzv. *međukôd* koji je bliži mašinskom kôdu. Takva reprezentacija kôda treba da olakša sljedeće faze, a to su optimizacija kôda i prevođenje kôda na jezik koji odgovara ciljnoj mašini.

Faza povezivanja

U fazi povezivanja (engl. linking) se u jedan izvršni program povezuju datoteke objektnog kôda koje su nastale prevođenjem izvornog kôda i objektni moduli koji sadrže podatke iz biblioteka (standardnih i drugih biblioteka). U toku ove faze se pronalaze i identifikuju simboli koji se koriste u jednoj datoteci, a definisani su u nekoj drugoj.

O povezivanju datoteka će biti dodatno riječi u Poglavlju 7.

1.3 Nekoliko jednostavnih programa

U ovom dijelu ćemo unaprijed uraditi nekoliko jednostavnih programa. Čitaoci koji već posjeduju elementarno znanje iz programiranja mogu samostalno da pokušaju da urade sve, ili barem neke od navedenih primjera.

Primjer 1.1. Napisati program u kome se primjenjuju osnovne aritmetičke operacije.

```
1  #include <stdio.h>
2
3  main() {
4
5     int a = 21;
6     int b = 10;
7     int c ;
8
9     c = a + b;
10    printf("Linija 1 - vrijednost c: %d\n", c );
11
12    c = a - b;
13    printf("Linija 2 - vrijednost c: %d\n", c );
14
15    c = a * b;
```


Elektronska verzija

```
16     printf("Linija 3 - vrijednost c: %d\n", c );
17 
18     c = a / b;
19     printf("Linija 4 - vrijednost c: %d\n", c );
20 
21     c = a % b;
22     printf("Linija 5 - vrijednost c: %d\n", c );
23 
24     c = a++;
25     printf("Linija 6 - vrijednost c: %d\n", c );
26 
27     c = a--;
28     printf("Linija 7 - vrijednost c: %d\n", c );
29 }
```

Primjer 1.2. Sa tastature se unose koordinate 4 tačke $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$ i $D(x_4, y_4)$ u koordinatnoj ravni. Odrediti presjek prave p, koja sadrži tačke A i B i prave q, koja sadrži tačke C i D.

Zadatak najprije moramo riješiti matematički. Formula za jednačinu prave p, koja sadrži dvije tačke sa koordinatama $A(x_1, y_1)$ i $B(x_2, y_2)$ je

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1).$$

odakle dobijamo

$$y(x_2 - x_1) - y_1(x_2 - x_1) = x(y_2 - y_1) - x_1(y_2 - y_1)$$

tj.

$$(y_2 - y_1)x + (x_1 - x_2)y = x_1(y_2 - y_1) + y_1(x_1 - x_2)$$

Obilježimo sad sa $a_1 = y_2 - y_1$, $b_1 = x_1 - x_2$ i $c_1 = a_1 x_1 + b_1 y_1$ i dobijamo da jednačina prave izgleda ovako:

$$a_1 x + b_1 y = c_1$$

Slično dobijamo i jednačinu prave q. Obilježimo sa
$a_2 = y_4 - y_3$, $b_2 = x_3 - x_4$ i $c_2 = a_2 x_3 + b_2 y_3$
i tada je jednačina prave q

$$a_2 x + b_2 y = c_2$$

Elektronska verzija

Da bismo odredili tačku presjeka dvije prave p i q, trebamo da riješimo sistem jednačina

$$a_1x + b_1y = c_1$$
$$a_2x + b_2y = c_2$$

Neka je $D = a_1b_2 - b_1a_2$ determinanta sistema. Ako je determinanta sistema različita od nule, sistem ima tačno jedno rješenje (x_0, y_0) koje se računa pomoću

$$x_0 = (c_1b_2 - c_2b_1)/D$$
$$y_0 = (a_1c_2 - a_2c_1)/D.$$

Tačka (x_0, y_0) je tačka presjeka pravih p i q.

U slučaju da je determinanta jednaka nuli, prave su paralelne.

Prikažimo sada izvorni kôd programa.

```
1  #include <stdio.h>
2
3  int main(){
4
5     float x1, x2, x3, x4, y1, y2, y3, y4;
6     float D;
7     float x0, y0;//za presjecnu tacku
8     float a1, a2, b1, b2, c1, c2;
9     printf("Tacke A i B: x1, y1, x2, y2: ");
10    scanf("%f %f %f %f",&x1, &y1, &x2, &y2);
11    printf("Tacke C i D: x3, y3, x4, y4: ");
12    scanf("%f %f %f %f",&x3, &y3, &x4, &y4);
13
14    a1 = y2 - y1;
15    b1 = x1 - x2;
16     c1 = a1 * x1 + b1 * y1;
17
18    a2 = y4 - y3;
19    b2 = x3 - x4;
20    c2 = a2 * x3 + b2 * y3;
21
22     D = a1 * b2 - a2 * b1;
23     if( D != 0){
24       printf("Presjecna tacka je: ");
25         x0 = (b2 * c1 - b1 * c2) / D;
26         y0 = (a1 * c2 - a2 * c1) / D;
27       printf("(%.3f,%.3f)\n",x0,y0);
28    }
29    else
```

Elektronska verzija

```
      printf("Prave su paralelne.\n");
   return 0;

}
```

Primjer 1.3. Na ekranu ispisati prvih 128 karaktera ASCII kôda.

```
#include <stdio.h>
int main(){
   int i;
   printf("Prvih 128 karaktera ASCII koda:\n");
   for(i = 0; i < 128; i++)
      printf("%d: %c\n",i,i);
   return 0;
}
```

Primjer 1.4. Sa tastature se unosi niska karaktera koja sadrži samo cifre i eventualno karakter `'-'` na početku. Odrediti brojnu vrijednost unesene niske.

```
#include <stdio.h>
int main(){
   char niska[20];
   int broj = 0, znak = 1, i = 0;
   char cifra;
   gets(niska);
   if(*niska == '-')
      {
         printf("Broj je negativan.\n");
         znak = -1;
         i++;
      }
   while(*(niska+i) != '\0'){
      broj = 10 * broj + *(niska+i) - '0';
      i++;
   }
   broj = znak * broj;
   printf("Formiran je broj: %d\n",broj);
   return 0;
}
```

Elektronska verzija

Primjer 1.5. Koristeći razvoj funkcije e^x u red, odrediti vrijednost broja e na zadatu tačnost $\epsilon = 0.00001$.

```
1  #include <stdio.h>
2  int main(){
3     int i = 1;
4     float stari, rez = 0.0, epsilon =0.00001, fakt = 1, razlika = 1.0;
5     while(razlika > epsilon){
6        stari = rez; //pamtimo stari rezultat
7        rez += (1.0) / fakt; //racunamo novi rezultat
8        razlika = rez - stari;
9        //printf("%.8f %.8f %.8f\n",stari, rez, razlika);
10       fakt *= i++; //racunamo sljedeci faktorijel
11    }
12    printf("Priblizna vrijednost broja e: %f\n",rez);
13
14    return 0;
15 }
```

1.4 Pitanja i zadaci

1. Šta bi trebalo promjeniti u programu "Zdravo svijete", pa da se na ekranu umjesto te poruke ispisala poruka "Mi volimo programiranje"?
2. Za šta se koristi preprocesorska direktiva `#include`?
3. Koja funkcija je obavezni dio svakog kôda napisanog na jeziku C?
4. Kako se pišu komentari u programskom jeziku C, za šta služe i kako se prevode?
5. Kako je programski jezik C dobio ime?
6. Navesti verzije standarda programskog jezika C i objasniti razlike između njih.
7. Objasniti pravila i konvencije koje se moraju i trebaju poštovati prilikom pisanja izvornog kôda.
8. Na primjeru sljedećeg kôda

```
#include <stdio.h> //ovim redom je ukljuceno zaglavlje stdio.h
#define MAX 20

int main(){
    int a = 5; //deklaracija promjenljive a
```

Elektronska verzija

```
...
}
```

objasniti šta će se desiti u fazi preprocesiranja, a šta u fazi prevođenja kôda na mašinski jezik.

9. Od kojih podfaza se sastoji faza prevođenja kôda? Ukratko objasniti svaku od podfaza.

10. Putem interneta istraži koje verzije prevodioca za programski jezik C postoje i koje su trenutno aktuelne.

11. Istraži različita integrisana okruženja za razvoj programa u programskom jeziku C. Izaberi neka od njih i instaliraj na svoj računar. Nakon toga formiraj mišljenje o upotrebljivosti pojedinih okruženja i izaberi neko koje ti se čini najpogodnijim za dalji rad. Svoj izbor kasnije možeš i promijeniti.

12. Istraži tzv. “online prevodioce”, tj. prevodioce koji omogućavaju online prevođenje kôda i pokretanje programa. Formiraj mišljenje o upotrebljivosti online prevodioca, kao i o njihovim prednostima i nedostacima.

Elektronska verzija

2 Osnovni elementi programskog jezika C

U ovom poglavlju ćemo se najprije upoznati sa identifikatorima i ključnim riječima, a potom ćemo objasniti kako se definišu promjenljive, koji su osnovni tipovi podataka i koje operacije možemo vršiti nad njima.

2.1 Identifikatori

Identifikatori su imena koja koristimo da bismo odredili - identifikovali različite elemente programa. Na primjer, identifikatore (kao imena) pridružujemo promjenljivima, konstantama, funkcijama itd. Koristimo ih i za identifikovanje novih elemenata programa, kao što su imena struktura, unija ili enumerativnih tipova.

U većini programskih jezika koji su danas u upotrebi postoji slična konvencija za pisanje identifikatora. U programskom jeziku C, identifikatori mogu da se sastoje od slova (malih i velikih), cifara i znaka '_', s tim što nije dozvoljeno da počinju cifrom. Pošto znak razmak (*space*) nije dozvoljen, ukoliko se identifikator sastoji od više riječi, uobičajeno je da se one razdvajaju znakom '_', ili da se identifikator sastoji od malih slova, dok svaka nova riječ počinje velikim slovom.

Evo nekoliko primjera pravilne upotrebe identifikatora.

```
a       slovo            Slovo       SLOVO
c1      vrijednost_10     ispis       brojSuglasnika
```

kao i nekoliko primjera nepravilne upotrebe

```
1broj    "rijec"     ab-cd
```

Kod prvog primjera nepravilne upotrebe uočavamo da identifikator ne smije početi cifrom, a kod drugog i trećeg primjera koriste se nedozvoljeni karakteri (dupli navodnik i znak minus).

Kao identifikatori se ne mogu koristiti ni ključne i rezervisane riječi, koje će biti razmatrane u sljedećoj sekciji. Takođe, pošto se znak '_' obično koristi za početni znak u nazivima sistemskih funkcija ili promjenljivih, nije preporučljivo

Elektronska verzija

(iako je dozvoljeno) da se taj znak koristi kao početni znak identifikatora definisanih od strane programera.

Iako dužina identifikatora može biti proizvoljna, prilikom njihovog uvođenja, ipak, trebamo voditi računa da broj karaktera koje koristimo bude razuman. Na primjer, potpuno je bespotrebno i nepraktčno promjenljivoj koja nosi informaciju o broju parnih brojeva dati naziv

```
broj_parnih_brojeva_koji_se_unose_sa_tastature
```

Pored nepraktične upotrebe predugačkih naziva, treba naglasiti i da se u ANSI C standardu (C89) garantuje da se barem prvih 31 karakter imena provjerava prilikom poređenja jednakosti imena. Standardom C99 taj broj je povećan na 63. Dakle, ako dvjema promjenljivima dodijelimo imena koja imaju više od 63 početna karaktera ista, nemamo garanciju da će te dvije promjenljive biti smatrane različitim. Za spoljašnje promenljive ove vrijednosti su 6 (C89), odnosno 31 (C99).

Većina modernih programskih jezika, kao što su C, C++, Java, Python, C#... spadaju u takozvane "case sensitive" jezike, tj. u ovim jezicima se razlikuju velika i mala slova. Na primjer, `broj`, `Broj` i `BROJ` su tri različita identifikatora.

2.2 Ključne riječi

Ključne ili rezervisane riječi su one riječi koje u programskom jeziku imaju posebno značenje. Već smo pomenuli da se ključne riječi ne mogu koristiti kao identifikatori. ANSI C standardom (C89) propisane su ukupno 32 ključne riječi. To su:

```
auto double int struct
break else long switch
case enum register typedef
char extern return union
const float short unsigned
continue for signed void
default goto sizeof volatile
do if static while
```

Treba primijetiti da su sve ključne riječi sastavljene samo od malih slova engleskog alfabeta.

Novijim standardima uvedene su još neke ključne riječi koje se, između ostalog, odnose i na rad sa nekim novim tipovima podataka (logički podaci, kompleksni brojevi i sl.). Kasnije ćemo u udžbeniku pomenuti neke od njih.

Elektronska verzija

2.3 Promjenljive. Deklaracija promjenljivih

Promenljive su osnovni objekti koje koristimo u programiranju. Svakoj promjenljivoj je pridružen određen memorijski prostor, koji služi za skladištenje (čuvanje) odgovarajućeg podatka, koji predstavlja vrijednost promjenljive. Tokom izvršenja programa, toj vrijednosti može da se pristupi, tj. ta vrijednost može da se pročita. Ako nije pretpostavljeno drugačije, vrijednost promjenljive (na šta ukazuje i sama riječ promjenljiva) može da se i promijeni. U programskom jeziku C svaka promjenljiva ima svoj tip (o tipovima podataka će biti dosta riječi u nastavku ovog poglavlja).

Podsjetimo se da su promjenljive identifikatori, što znači da za određivanje naziva promjenljivih važe opšta pravila (ali i razne druge konvencije) koja smo već pominjali da važe za identifikatore. Ponovo navodimo nekoliko ispravnih i nekoliko neispravnih imena promjenljivih.

Ispravna imena promjenljivih:

```
a, b, c, a1, xy, broj, _prom, Brojac, rijec, niz, matrica
```

Neka neispravna imena promjenljivih:

```
suma brojeva
$tring
#ime
goto
1broj
"promjenljiva"
int
```

Korištenje promjenljivih podrazumijeva deklaraciju same promjenljive, dodjelu vrijednosti, eventualne kasnije izmjene te vrijednosti, kao i upotrebu dodijeljene vrijednosti u izrazima i funkcijama.

Deklaracija promjenljive uključuje određivanje tipa same promjenljive i njenog imena. Prilikom deklaracije, promjenljivoj se može odmah dodijeliti i odgovarajuća vrijednost, tj. može se izvršiti *inicijalizacija* promjenljive. Takođe, moguće je u okviru jedne deklaracije deklarisati više promjenljivih.

U programskom jeziku C sve promjenljive moraju biti deklarisane prije upotrebe. Iako je standardom C99 uvedeno da promjenljiva može biti deklarisana bilo gdje unutar bloka naredbi, čime je omogućeno da se deklaracije i izvršne naredbe uzajamno mogu preplitati, ustaljeno je pravilo da se deklaracija promjenljivih najčešće radi na početku funkcija i prije izvršnih naredbi.

Primjer 2.1. Evo nekoliko najjednostavnijih primjera deklaracije promjenlji-

Elektronska verzija

vih, sa i bez definisanja početne vrijednosti (inicijalizacije).

```
int a;
int i, j;
double x1 = 3.4, y1 = 5.33;
float x, y, z;
char slovo1 = 'a', slovo2;
```

Ako neku promjenljivu definišemo unutar neke funkcije, tada kažemo da je ta promjenljiva lokalna promjenljiva unutar te funkcije. Druge funkcije ne mogu da koriste tu promjenljivu. Sa druge strane, moguće je da u različitim funkcijama koristimo ista imena za lokalne promjenljive. Za razliku od lokalnih promjenljivih, koje su "vidljive" samo u okviru funkcije u kojoj su definisane, postoje i globalne promjenljive, koje definišemo van svih funkcija. Globalne promjenljive su "vidljive", tj. mogu se koristiti u više funkcija.

Vidljivost promjenljivih, kao i vidljivost identifikatora u opštem slučaju je određena pravilima dosega identifikatora. Najjednostavnije situacije, koje su u ovom trenutku početniku dovoljne su sljedeće: ako je promjenljiva definisana unutar nekog bloka, onda je ta promjenljiva "vidljiva" samo unutar tog bloka. Pod blokom sada možemo smatrati dio programa koji se nalazi unutar vitičastih zagrada. Takve promjenljive ćemo zvati lokalne promjenljive. Ako je promjenljiva definisana van svih blokova, onda je riječ o globalnim promjenljivima, koje su vidljive na nivou čitavog programa. Treba napomenuti da postoje i komplikovaniji slučajevi, posebno oni koji podrazumijevaju rad sa više datoteka. Tada se "vidljivost" promjenljivih mora dodatno regulisati odgovarajućim mehanizmima. O tome će biti više riječi u Poglavlju 7.

2.3.1 Konstante

Kako i sama riječ ukazuje, konstante su podaci koji imaju konstantnu, stalnu vrijednost. Razlikujemo konstante koje su brojevi (na primjer `1`, `24`, `-1000`, `4.5`, `3.33` itd.), karakteri (koje zapisujemo u okviru jednostrukih navodnika `'a'`, `'!'`, `'M'`, `'7'` itd.), niske karaktera (`"tabla"`, `"Danas je lijep dan."`, `"1"` itd.), a od konstanti možemo da gradimo i konstantne izraze, koji se sastoje od konstantnih vrijednosti i odgovarajućih operatora (na primjer, `27*139`).

Pod konstantama (kao identifikatorima) podrazumijevamo one identifikatore kojima jednom dodijeljena vrijednost ne može da se mijenja. Kao i promjenljive, i konstante u programskom jeziku C imaju svoj tip, a i deklarišu se slično kao i promjenljive, s tim što se ispred naziva tipa piše riječ **`const`**. U opštem slučaju definisanje konstante se vrši na sljedeći način:

Elektronska verzija

```
const tipPodatka ime = vrijednost;
```

Moguće je da tip podatka i riječ `const` zamijene mjesta, te je deklaracija konstante i na ovaj način

```
tipPodatka const ime = vrijednost;
```

dozvoljena, ali je gornji način više uobičajen.

Ako bismo, na primjer, htjeli da definišemo cjelobrojnu konstantu `maksimum` i dodijelimo joj vrijednost 1000, to ćemo uradi na sljedeći način:

```
const int maksimum = 1000;
```

Ako bismo negdje dalje u programu pokušali da promijenimo vrijednost konstante `maksimum`, recimo

```
maksimum = 5000; //greska
```

napravili bismo grešku, jer se u toku izvršenja programa jednom dodijeljena vrijednost konstanti ne može naknadno mijenjati.

Prilikom deklaracije konstante moguće je izostaviti vrijednost koja se dodjeljuje, ali to nema nekog velikog smisla, jer se dodjela vrijednosti kasnije svakako ne može izvršiti.

2.4 Osnovni tipovi podataka

Svakom podatku je pridružen odgovarajući **tip**, koji prevodiocu ukazuje na to na koji način planiramo da koristimo taj podatak. Određivanjem tipa nekog podatka, definiše se i veličina memorijskog prostora potrebnog za smještanje i način reprezentacije tog podatka.

U programskom jeziku C u osnovne tipove podataka uključujemo:

- tipove kojima predstavljamo cijele brojeve (cjelobrojni tipovi)
- tipove kojima predstavljamo realne brojeve (realni tipovi)
- tipove kojima predstavljamo znakove (znakovni tip)

Treba naglasiti da u verzijama programskog jezika C koje se široko koriste nema posebnog tipa podatka za predstavljanje logičkih vrijednosti tačno i netačno (`true i false`), već se umjesto njih najčešće koriste cjelobrojni tipovi, tako što se smatra da broj 0 ima vrijednost netačno, dok sve vrijednosti različite od

Elektronska verzija

nule imaju vrijednost tačno. Treba napomenuti da je standardom C99 uveden i logički tip podatka, a podaci ovog tipa mogu da uzimaju vrijednost `true` i `false`. Međutim, pri programiranju u programskom jeziku C, ustaljena je i česta praksa da se logički izrazi zasnivaju samo na brojevnim vrijednostima (0 – netačno, broj različit od nule – tačno).

2.4.1 Cjelobrojni tipovi podatka

Osnovni tip podatka za predstavljanje cijelih brojeva je `int` (riječ int je zapravo skraćeno od engleske riječi integer, što znači cio broj). Podrazumijeva se da pomoću ovog tipa možemo predstavljati *označene* (engl. signed) cijele brojeve, tj. i pozitivne i negativne brojeve (naravno i broj nula). Za predstavljanje označenih brojeva najčešće se koristi potpuni komplement. Standardom nije definisana tačna veličina podatka tipa `int` (u bitovima), već je definisano samo da se za ovaj tip podatka koristi najmanje dva bajta (16 bita). Tačna veličina obično zavisi od konkretne mašine, tj. samog hardvera računara i operativnog sistema. Na današnjim računarima, podatak tipa `int` se zapisuje pomoću 4 bajta (32 bita) ili 8 bajtova (64 bita).

Ako za predstavljanje podatka tipa `int` koristimo 32 bita, zaključujemo da se pomoću ovog tipa podatka može predstaviti ukupno 2^{32} različitih cijelih brojeva. Polovina od njih (oni kod kojih je krajnji lijevi bit jednak 1) su negativni, a druga polovina pozitivni (tu treba uključiti i broj 0, koji se predstavlja 32-bitnim nizom nula). Najmanji cio broj koji se može predstaviti tipom `int` koji je veličine 32 bita je broj $-2^{31} = -2147483648$, dok je najveći $2^{31} - 1 = 2147483647$.

Pored osnovnog cjelobrojnog tipa, moguće je koristiti i "kraće" i "duže" tipove. Ako osnovnom tipu pridružimo kvantifikator `short`, u zapisu `short int`, uvešćemo podatak koji je po veličini potencijalno manji (kraći) od podatka tipa `int`. Ako tipu `int` pridružimo kvantifikator `long`, u zapisu `long int`, uvešćemo podatak koji je potencijalno veći od osnovnog podatka tipa `int`. Od standarda C99 uveden je i kvantifikator `long long` koji se koristi za predstavljanje cijelih brojeva koji su po veličini potencijalno još veći od podatka `long int`. Standardom nije propisana tačna veličina ovih tipova, ali su definisana neka ograničenja. Tako podatak tipa `short int` zauzima barem dva bajta, a podatak tipa `int` mora biti veće ili jednake veličine kao `short int`. Podatak tipa `long int` je veličine najmanje koliko i `int` i zauzima barem četiri bajta. Veličina tipa `long long int` je najmanje koliko i veličina `long int`. Imena ovih tipova se mogu i kraće zapisati, što se u praksi i koristi: umjesto `short int` možemo pisati samo `short`, umjesto `long int` samo `long`, dok se `long long int` može pisati samo `long long`.

Elektronska verzija

Kao i osnovni cjelobrojni tip (tip **int**), i kraći i duži cjelobrojni tipovi (**short**, **long** i **long long**) su označeni, tj. mogu biti i pozitivni i negativni. Dodavanjem kvalifikatora **unsigned** ispred naziva tipa dobijamo *neoznačene* (engl. unsigned) cijele brojeve, kojima se predstavljaju samo pozitivni brojevi (uključujući i nulu). Na taj način dobijamo mogućnost da neoznačenim tipom predstavimo još veće pozitivne brojeve (praktično neoznačenim tipom možemo predstaviti duplo više pozitivnih brojeva u odnosu na odgovarajuči označeni). Na primjer, ako je veličina podatka **int** 32 bita, tada se tipom **unsigned int** predstavljaju brojevi iz intervala $[0, 2^{32}-1]$. Ispred naziva tipa može se koristiti i kvantifikator **signed** (na primjer **signed int** ili **signed long**), kojim se naglašava da se radi o označenim brojevima. Ako se uz naziv tipa ne navede nijedan od kvantifikatora **unsigned** ili **signed**, podrazumijeva se da se radi sa označenim brojevima.

Primjer 2.2. Da bismo vidjeli veličinu svakog od navedenih tipova, možemo pokrenuti sljedeći program

```
1  #include <stdio.h>
2  int main(){
3
4     printf("char: %d\n",sizeof(char));
5     printf("short: %d\n",sizeof(short));
6     printf("int: %d\n",sizeof(int));
7     printf("long: %d\n",sizeof(long));
8     printf("long long: %d\n",sizeof(long long));
9     return 0;
10
11 }
```

Ispis na ekranu zavisi od verzije sistema i prevodioca, ali bi mogao da izgleda ovako:

```
char: 1
short: 2
int: 4
long: 8
long long: 8
```

Poznavanje tačnog opsega pojedinih cjelobrojnih tipova može biti od velike važnosti kada se rješavaju problemi (zadaci) kod kojih je potrebno voditi računa o veličini ukupne zauzete memorije. Na primjer, ako radimo sa podacima koji su reda veličine nekoliko stotina ili nekoliko hiljada, nije potrebno koristiti podatke

Elektronska verzija

tipa **long**, već je dovoljno koristiti osnovni tip **int**. Međutim, ako se radi sa podacima koji su reda veličine nekoliko milijardi, tada nam skup podataka koji su predstavljeni tipom **int** nije dovoljan, te u tom slučaju koristimo tip **long**.

Primjer 2.3. Jednostavnim primjerom ilustrujmo pojavu prekoračenja pri radu sa cjelobrojnim tipovima.

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

   int a = 123456789;
   int b = 1000;
   int c = a * b;//namjerno dodijelimo preveliku vrijednost
   printf("%d\n",c);

   int x = 2147483647;//najveci oznacen cio broj
   printf("%d\n",x);//nije problem da ga ispisemo

   int y = x + 1;//pravimo prekoracenje
   printf("%d\n",y);//sad je vec problem, ispisuje se negativan broj

   char c1 = 127;//probajmo sa karakterima

   printf("%d\n",c1);//ovo je u redu
   char c2 = c1 + 1;//pravimo prekoracenje
   printf("%d\n",c2);//ispisuje se negativan broj

   unsigned int x1 = 2147483647;//probajmo neoznacene brojeve
   unsigned int y1 = x1 + 1;
   printf("%u\n",y1);//sad nema prekoracenja

   return 0;
}
```

2.4.2 Realni tipovi podataka

Za predstavljanje realnih brojeva, odnosno preciznije brojeva u pokretnom zarezu (engl. floating point numbers), koristimo tip osnovne preciznosti **float**, tip dvostruke preciznosti **double** i od standarda C99 i tip **long double**, koji se nekada naziva i podatak četvorostruke preciznosti.

Slično kao i kod cjelobrojnih tipova, standardnom nije tačno definisana veličina realnih tipova, ali je propisano da podatak tipa **double** koristi najmanje

Elektronska verzija

Tip	Veličina	Opseg	Min. poz. broj	Preciznost
`float`	4 bajta	$\pm 3.4 \cdot 10^{38}$	$1.2 \cdot 10^{-38}$	6 cifara
`double`	8 bajta	$\pm 1.7 \cdot 10^{308}$	$2.3 \cdot 10^{-308}$	15 cifrara
`long double`	10 bajta	$\pm 1.1 \cdot 10^{4932}$	$3.4 \cdot 10^{-4932}$	19 cifara

Tabela 2.1: Informacije o realnim tipovima podataka

onoliko bajtova koliko i `float`, a da podatak tipa `long double` koristi barem onoliko bajtova koliko i `double`. Na većini današnjih računara podaci tipa `float` se zapisuju pomoću 4 bajta, podaci tipa `double` pomoću 8, a podaci tipa `long double` pomoću 10 bajtova. Osobine sistema brojeva sa pokretnim zarezom su definisane IEEE754 standardom, koga se u današnje vrijeme pridržava većina proizvođača hardvera. Ukratko, ovim standardom su definisani formati, formati za razmjenu, pravila zaokruživanja, aritmetičke i druge operacije i obrada izuzetaka (na primjer dijeljenje nulom, prekoračenje i dr.). Kada govorimo o mogućim vrijednostima koje podaci predstavljeni brojevima u pokretnom zarezu mogu da imaju, najčešće mislimo na sljedeće veličine: opseg, tj. najmanji i najveći broj, najmanji pozitivan broj i preciznost. U Tabeli 2.1 prikazujemo ove vrijednosti za sva tri realna tipa podatka.

Pomenutim standardom IEEE754 uvedene su i specijalne vrijednosti: beskonačna vrijednost (engl. infinity) i vrijednost koja nije broj (engl. not a number - ili skraćeno `NaN`). Razlikuje se i pozitivna i negativna beskonačna vrijednost ($+\infty$ i $-\infty$). Na primjer, vrijednost izraza $5.0/0.0 = +\infty$, a vrijednost izraza $-2.5/0.0 = -\infty$. Ovdje spomenimo, a kasnije ćemo se na to i vratiti, da pokušaj dijeljenja nulom kod cijelih brojeva nije defisan i program će u tom slučaju najvjerovatnije prekinuti sa radom. Vrijednost `NaN` se koristi u slučajevima kada vrijednost izraza nije definisana (na primjer 0.0/0.0). Ove specijalne vrijednosti (beskonačnost i `NaN`) mogu dalje da se koriste u izrazima, što ćemo vidjeti u narednom primjeru. Neki od mogućih ispisa beskonačne vrijednosti su: `1.#INF` ili `1.#INF00` (najčešće na Windows operativnom sistemu), odnosno `inf` na Linux-u, dok se vrijednost koja nije broj štampa kao `1.#IND` ili `1.#IND00` (na Windows-u), odnosno `nan` na Linux operativnom sistemu.

Primjer 2.4. Odštampajmo na ekranu vrijednosti nekih izraza u kojima učestvuju beskonačna veličina i veličina koja nije broj.

```
1 #include <stdio.h>
2 #include <math.h>
3 int main(){
4
```

Elektronska verzija

```
    float a = 1.1, b=0.0;
    float c = a / b;
    float d = sqrt(-1);
    float e = b / b;
    float f = (-a) / b;
    float g = 1.0 / c;
    float h = c + 15.25;
    float m = e + 15.25;
    printf("%f\n",c);
    printf("%f\n",d);
    printf("%f\n",e);
    printf("%f\n",f);
    printf("%f\n",g);
    printf("%f\n",h);
    printf("%f\n",m);
     printf("%f\n",c+d);
    return 0;
}
```

Kao što smo pomenuli, ispis ovih veličina se razlikuje u zavisnosti od operativnog sistema, a jedan od mogućih ispisa je

```
1.#INF00
-1.#IND00
-1.#IND00
-1.#INF00
0.000000
1.#INF00
-1.#IND00
-1.#IND00
```

Primijetimo da je promjenljiva `d` dobila vrijednost koja nije broj, jer je korijena funkcija definisana samo za nenegativne brojeve. Promjenljiva `g` je dobila vrijednost nula, jer je izraz $1/\infty$ jednak nuli. Beskonačna vrijednost sabrana sa konačnim brojem će opet dati beskonačnu vrijednost. Vrijednost izraza u kome učestvuje vrijednost koja nije broj će opet biti vrijednost koja nije broj.

2.4.3 Znakovni tip podatka

Znaci - karakteri (velika i mala slova, cifre, znaci interpunkcije itd.) se u programskom jeziku C predstavljaju tipom **`char`**, (skraćeno od engleske riječi character, što znači znak, karakter, simbol). Tip **`char`** je zapravo cjelobrojni tip podatka, koji je veličine jedan bajt. Karakteri se kodiraju odgovarajućim bro-

Elektronska verzija

jevnim kodovima, te se i manipulacija karakterima praktično svodi na manipulacije brojevima. Drugim riječima, karakteri se predstavljaju odgovarajućim cjelobrojnim tipom relativno malog opsega, gdje podaci ovog tipa mogu da uzmu neku od ukupno 256 različitih vrijednosti. Kao što je slučaj i sa drugim cjelobrojnim tipovima, i podatak tipa `char` može biti označen i neoznačen. Podatak tipa `signed char` (ili samo `char`, jer, ako se izostave kvalifikatori `signed` odnosno `unsigned`, podrazumijeva se da je podatak označen) uzima vrijednosti iz intervala $[-128, 127]$, dok neoznačeni podatak tipa `unsigned char` uzima vrijednosti iz intervala $[0, 255]$.

Postoje različiti načini kodiranja karaktera, a jedan od nastarijih i najpoznatijih je ASCII kôd (skraćenica ASCII je akronim od American Standard Code for Information Interchange). Standard programskog jezika C ne propisuje koje kodiranje se koristi, ali je na skoro svim savremenim računarima i implementacijama programskog jezika C kodiranje karaktera zasnovano na ASCII kôdu. Poznato je da se u ASCII kôdu koristi ukupno 7 bita za predstavljanje karaktera, gdje se svakom karakteru dodjeljuje odgovarajuća brojevna vrijednost iz intervala $[0, 127]$. Neke od važnih osobina ASCII kôda koje treba poznavati su sljedeće: ASCII kod počinje takozvanim terminalnim karakterom (koji se zapisuje `'\0'`), tj. ovaj karakter u ASCII kodu ima redni broj 0. Karakter Enter (prelazak u novi red) je redni broj 10, karakter `Space` (razmak) 32. Velika slova engleske abecede su poredana redom, počev od rednog broja 65 (to je redni broj velikog slova `'A'`), pa do rednog broja 90 (redni broj velikog slova `'Z'`). Mala slova engleske abecede su takođe poredana redom, počev od rednog broja 97 (malo slovo `'a'`), do rednog broja 122 (malo slovo `'z'`). I cifre su poredane redom, počev od cifre `'0'`, koja je na rednom broju 48, do cifre `'9'` koja je na rednom broju 57. Na preostalim mjestima nalaze se znaci interpunkcije (`'.'`, `','`, `'!'`, `'?'`...), simboli za matematičke operacije (`'+'`, `'-'`, `'='`...), te ostali karakteri.

Ovdje treba napomenuti da se karakteri kao konstante zapisuju u okviru jednostrukih (malih) navodnika, tj. unutar `'` i `'`. Tako, na primjer, zapis `'n'` predstavlja malo slovo n engleske abecede, dok bi zapis bez malih navodnika (samo `n`) najvjerovatnije predstavljao neku promjenljivu koja je dobila naziv `n`. Slično, zapis `'1'` predstavlja zapis konstante koja je karakter - cifra i koja se u ASCII kôdu kodira brojem 49, dok zapis bez malih navodnika (samo 1) predstavlja brojevnu cjelobrojnu konstantnu vrijednost, tj. broj 1.

Standard dozvoljava da se u okviru jednostrukih navodnika navodi i više od jednog karaktera (na primjer `'abc'`), ali vrijednost takvog podatka nije definisana. Stoga, takve zapise treba potpuno izbjegavati.

Elektronska verzija

2.5 Izrazi i operatori

2.5.1 Izrazi

Izraze formiramo kombinovanjem operatora i operanada. Podrazumijeva se da su takve kombinacije ispravno napisane, te ih kao takve programski jezik dozvoljava. Operandi u izrazima su promjenljive ili konstante, dok se operatorima predstavljaju osnovne operacije i relacije koje se mogu izvršavati na odgovarajućim podacima. Na primjer, za cjelobrojne podatke, operandi su cjelobrojne promjenljive (na primjer promjenljive tipa **`int`** ili **`long`**), dok se kao operatori mogu koristiti aritmetički operatori (operatori $+, -, *, /, ++, --$ itd.), relacijski operatori ($<, >, <=, >=$ itd.), ili logički operatori ($!, ||, \&\&$).

Primjer 2.5. U ovom primjeru navodimo nekoliko izraza.

```
100
322 + 558
x = a + 134 * y
a1 * (b + c / d) % 2
i--
vrijednost = c - '0'
x = broj % 6
c > 5
(3 < y) && (y < 10)
```

Osim kombinovanja konstanti, promjenljivih i operanada, u elementarne izraze možemo uvrstiti i rezultate poziva funkcija, operacija pristupa elementima niza, čitanje sadržaja sa nekog memorijskog mjesta, pristup pojedinim elementima struktura i slično. O ovim konceptima će biti riječi u nastavku. U programskom jeziku C, svakom izrazu je pridružena vrijednost koja se dobija izvršavanjem svih operacija prisutnih u izrazu, u skladu sa definicijima samih operacija i pravilima koja određuju redoslijed izvršavanja operacija.

2.5.2 Operatori

Slično kao i u matematici, operacije i relacije koje se mogu izvršavati na podacima osnovnih tipova predstavljamo odgovarajućim operatorima. Tako operator $+$ koristimo za operaciju sabiranja, operator $-$ za oduzimanje, operator $*$ za množenje, a operator $/$ za dijeljenje. Pored operacija koje se primjenjuju na osnovnim tipovima, podržane su i operacije na pojedinačnim bitovima.

Po broju operanada koji učestvuju u operaciji, odgovarajuće operatore dijelimo na unarne (učestvuje samo jedan operand), binarne (dva operanda) ili

Elektronska verzija

ternarne (tri operanda). Unarni operatori mogu biti prefiksni, kada se navode prije operanda (na primjer `-x`), ili postfiksni, kada se prvo navodi operand, pa onda operator (na primjer `k++`). Binarni operandi su najčešće infiksni, tj. navode se između dva operanda.

U situacijama kada unutar izraza imamo više operatora, važno je znati kojim redoslijedom se odgovarajuće operacije izvršavaju. Najsigurniji način za upravljanje redoslijedom izvršenja operacija je upotreba "malih" zagrada (i), pomoću kojih grupišemo dijelove izraza za koje želimo da se prvi izvrše. Sa druge strane, u mnogim slučajevima zagrade možemo i izostaviti, jer postoje konvencije o prioritetima operatora kojima je definisan redoslijed izvršenja operacija.

U programskom jeziku C se uglavnom poštuju prioriteti koji važe i u matematici. Na primjer, važi da su operacije množenja i dijeljenja starije od operacija sabiranja i oduzimanja. Recimo, u izrazu `a+(b*c)` operacija množenja, svakako, ima viši prioritet od operacije sabiranja, te se zagrade mogu izostaviti i može se pisati samo `a+b*c`. Sa druge strane, u složenijim izrazima često dolazimo u situacije kada nije lako procijeniti koji operator se izvršava prije nekog drugog, te je stoga potrebno poznavati i druge principe kao što su: unarni operatori su višeg prioriteta u odnosu na binarne, aritmetički operatori su višeg prirotita u odnosu na relacijske, relacijski operatori su višeg prioriteta u odnosu na logičke, operatori dodjele su niskog pririteta, operatori kojima se pristupa pojedinačnim elementima struktura (to su operatori `.` i `->`, o kojima će biti riječi kasnije) su visokog prioriteta itd.

U slučaju da je riječ o operatorima istog prioriteta, ako su zagrade izostavljene, uglavnom se koristi tzv. lijeva asocijativnost, tj. operatori istog prioriteta se u izrazu izvršavaju s lijeva na desno.

Za početnika je važno da na osnovu konvencija o prioritetu operatora na ispravan način formira složenije izraze, vodeći računa da zagrade (i) koristi onda kada je to potrebno i/ili kada se stavljanjem odgovarajućih izraza u zagrade isključuje mogućnost "nagađanja" koji operator ima prioritet. Na primjer, u izrazu

```
z + x < y || k + m >= 10
```

nije na prvi pogled jasno koje operacije su istog prioriteta (a koje različitog) i šta će se izvršiti prije čega. Pored toga, rezultati relacijskih i logičkih operatora su u programskom jeziku C cijeli brojevi, tako da je ovaj izraz sa sintaksne strane sigurno ispravan, te prilikom prevođenja prevodilac sigurno neće prijaviti grešku. Stoga ostaje mogućnost da usljed nekorištenja zagrada izraz ne daje onu vrijednost koju programer očekuje. Ovakve situacije treba izbjegavati, tj.

Elektronska verzija

u slučajevima kada nije sasvim jasno, ili nije potpuno očigledno koja operacija ima prioritet, za grupisanje izraza je poželjno koristiti zagrade. Prirodno bi bilo da je gornji izraz napisan na primjer ovako:

```
((z + x) < y) || ((k + m) >= 10)
```

gdje se sada tačno vidi da će se prvo izvršiti obje operacije sabiranja, nakon njih dva relacijska operatora (`<` i `>=`) i, na kraju, operator disjunkcije (o ovim operatorima će biti riječi u nastavku).

Aritmetički operatori

Za sabiranje, oduzimanje, množenje, dijeljenje i računanje ostatka pri dijeljenju koristimo (binarne) infiksne operatore +, -, *, / i %. Treba odmah napomenuti da za operator dijeljenja važe pravila: ako su oba operanda cijeli brojevi, tada se primjenjuje cjelobrojno dijeljenje, tj. rezultat dijeljenja je cio dio količnika. U ostalim slučajevima (kada barem jedan od operanada nije cio) primjenjuje se dijeljenje realnih brojeva, tj. dijeljenje brojeva u pokretnom zarezu. Dijeljenje nulom nije definisano. Operator % se primjenjuje samo na operande cjelobrojnog tipa.

Unarni prefiksni operator - se koristi za promjenu znaka (na primjer, ako je promjenljiva `x` imala vrijednost -5, tada `-x` ima vrijednost 5). Postoji i unarni prefiksni operator + (koji se koristi u sintaksi `+x`), ali on suštinski ne radi ništa (na primjer, izraz `+x` ima istu vrijednost kao i `x`).

Prefiksni unarni operatori + i - su višeg prioriteta u odnosu na bilo koji binarni operator. Operatori *, / i % su istog prioriteta i imaju viši prioritet u odnosu na + i -. Za sve ove navedene binarne operatore važi lijeva asocijativnost, tj. ako drugačije (na primjer zagradama) nije definisan drugi prioritet, operacije istog prioriteta se izvršavaju sa lijeva na desno.

U programskom jeziku C, kao i u drugim savremenim programskim jezicima, postoje (i veoma se često koriste) i unarni operatori uvećanja za 1 odnosno umanjenja za 1. Ovi operatori se često nazivaju i operatori inkrementiranja i dekrementiranja (što su zapravo engleske riječi increment i decrement). Operator uvećanja za jedan se zapisuje pomoću dva uzastopna znaka + bez razmaka između (++), dok se operator umanjenja zapisije pomoću znaka –. Oba ova operatora se mogu primijeniti i na cijele i na realne brojeve. Ove operatore koristimo u prefiksnoj ili u sufiksnoj formi. U prefiksnoj formi (`++x`, `--x`) vrijednost promjenljive če biti promijenjena (uvećana odnosno umanjena za 1) prije nego što se ona iskoristi u složenom izrazu. U postfiksnoj formi (`x++`, `x--`) u složenom izrazu se prvo iskoristi stara vrijednost, te se nakon toga vrijednost

Elektronska verzija

promjenljive uvećava, odnosno umanjuje za 1. Na primjer, ako promjenljiva `a` ima vrijednost 10, onda po izvršenju izraza `b=++a` obje promjenljive `a` i `b` imaju vrijednost 11 (vrijednost promjenljive `a` se prvo uvećala sa 10 na 11, pa je ta nova vrijednost 11 dodijeljena promjenljivoj `b`). Sa druge strane, ako promjenljiva `a` ima vrijednost 10, nakon izvršenja izraza `b=a++` promjenljiva `b` ima vrijednost 10, jer je stara vrijednost promjenljive `a` (vrijednost 10) iskorištena za dodjelu, pa je tek nakon toga `a` uvećano za 1 (promjenljiva `a` svakako dobija vrijednost 11).

Operatori poređenja na brojevima

Na cijelim brojevima i brojevima u pokretnom zarezu definisani su binarni relacijski operatori za poređenje. To su operatori

- `<` operator manje
- `<=` operator manje ili jednako
- `>` operator veće
- `>=` operator veće ili jednako
- `==` operator jednakosti
- `!=` operator nejednakosti (operator različito)

Kao što je i za očekivati, ovi operatori funkcionišu po istom principu kao i odgovarajući operatori iz matematike. Kada neki od operatora poređenja primijenimo na dva operanda (dva broja) dobijamo vrijednosti 0 (koja odgovara logičkoj vrijednosti netačno) ili vrijednost 1 (tačno). Rezultat primjene operatora `==` je tačno (odnosno vrijednost 1), ako je lijevi operand jednak desnom, a inače je rezultat 0. Kod operatora `!=` situacija je obrnuta, ako su operandi različiti rezultat je 1, a ako su jednaki rezultat će biti 0. Operatori `<`, `>`, `<=`, `>=` su istog prioriteta, koji je viši u odnosu na operatore jednakosti (`==`) i nejednakosti (`!=`). Za sve ove operatore važi lijeva asocijativnost.

Primjer 2.6. Uvedimo tri cjelobrojne promjenljive `x`, `y` i `z`,

```
int x = 10, y = 5, z = 1;
```

i analizirajmo vrijednosti izraza

```
x < y == z >= y;
```

Elektronska verzija

Pošto operacije `<` i `>=` imaju veći prioritet u odnosu na operaciju `==`, ovaj izraz je ekvivalentan izrazu

```
(x < y) == (z >= y);
```

Oba izraza `x < y` i `z >= y` će imati vrijednost 0, pa će vrijednost čitavog izraza `(x < y) == (z >= y)` biti jednaka vrijednosti izraza `0 == 0`, a to je jednako 1.

Zbog činjenice da je u programskom jeziku C rezultat primjene ovih operatora cio broj (0 ili 1), treba napomenuti da se "nadovezivanje" operatora (na primjer, `a < b < c <= 6`) ne smatra sintaksnom greškom. Sa druge strane, izvršavanje ovakvih izraza često dovodi do potpuno neočekivanih rezultata. Na primjer, ako bismo za vrijednost promjenljive `x = 3` posmatrali vrijednost izraza `5 < x < 8`, sa matematičkog aspekta bismo očekivali da je izraz netačan (broj 3 nije između brojeva 5 i 8). Međutim, pošto bi se u programu operatori u izrazu `5 < 3 < 8` izvršavali slijeva na desno, postepeno bismo dobijali `5 < 3 < 8 = (5 < 3) < 8 = 0 < 8 = 1`, odnosno rezultat izvršavanja izraza bi bio 1 (tačno).

Treba napomenuti i da binarni operatori poređenja imaju niži prioritet u odnosu na aritmetičke operatore. Tako je, na primjer, izraz

```
x + y + 5 < a * b
```

ekvivalentan izrazu

```
(x + y + 5) < (a * b)
```

jer će se i u prvom slučaju prvo izvršiti aritmetičke operacije, pa tek onda operacija poređenja. Stoga je izbor koji će se pristup koristiti ostavljen programeru. Svaki programer, pa i početnik treba da formira svoje mišljenje (i argumentuje ga) kada je upotreba zagrada u ovakvim situacijama korisna, tj. kada ona pomaže preglednosti kôda, a kada ga opterećuje.

Logički operatori

Pod logičkim operatorima podrazumijevamo one operatore koji za operande uzimaju logičke izraze (tj. izraze za koje smatramo da imaju jednu od dvije vrijednosti: tačno ili netačno, odnosno u programskom jeziku C, vrijednost različitu od 0 ili vrijednost 0). U programskom jeziku C postoje tri logička operatora:

&& operator konjunkcije

Elektronska verzija

|| operator disjunkcije

! operator negacije

Prisjetimo se da se u matematici operacije konjunkcije, disjunkcije i negacije primjenjuju na logičke vrijednosti (`true i false`) i kao rezultat opet daju logičku vrijednost. S obzirom na to da su u programskom jeziku C logičke vrijednosti predstavljene brojevima, možemo reći da su i logički operatori (&&, || i !) takvi da se primjenjuju na brojeve i kao rezultat daju podatak koji je broj (tipa **`int`**) i to ponovo: vrijednost 0, ukoliko je istinitosna vrijednost logičkog izraza netačna, odnosno vrijednost 1, ako je istinitosna vrijednost logičkog izraza tačna. Način funkcionisanja ovih operatora u programskom jeziku C (kao i u svim drugim programskim jezicima) je isti kao i u matematici. Operatori konjunkcije i disjunkcije su binarni infiksni operatori, dok je operator negacije unarni (prefiksni) operator.

Ponovimo da će konjunkcija dva izraza će dati vrijednost tačno (vrijednost 1) samo ako oba izraza koji su operandi imaju vrijednost tačno. U ostalim slučajevima vrijednost konjunkcije je netačno (odnosno vrijednost 0). Disjunkcija dva izraza če biti tačna, ako je barem jedan od izraza koji su operandi tačan. Ako su oba operanda netačni izrazi i disjunkcija je netačna. Negacija tačnog izraza daje netačno, a negacija netačnog izraza tačno. S obzirom na to da ova pravila odgovaraju odgovarajućim pravilima iz matematičkih definicija ovih operacija, treba pomenuti da za operacije konjunkcije, disjunkcije i negacije u programskim jezicima važe sva ona pravila koja važe i u matematici, kao na primjer: komutativnost i asocijativnost obje binarne operacije, distributivnost jedne binarne operacije u odnosu na drugu, de Morganovi zakoni itd.

U programskom jeziku C operator konjunkcije ima viši prioritet u odnosu na disjunkciju. Stoga bi izraz

```
a || b && c
```

bio ekvivalentan izrazu

```
a || (b && c)
```

Treba naglasiti da ova osobina ne važi za odgovarajuće operatore u matematici (u matematici su oba operatora istog prioriteta, a matematički izraz $a \vee b \wedge c$ ne bi bio ispravno napisan, jer bi vrijednost izraza zavisila od izbora načina kojim redoslijedom se izvršavaju navedene operacije).

Sa aspekta redoslijeda izvršenja, oba ova binarna operatora su lijevo asocijativni. Binarni logički operatori su nižeg prioriteta i u odnosu na binarne relacijske operatore poređenja i u odnosu na aritmetičke operatore. Operator

Elektronska verzija

negacije (operator !) ima viši prioritet u odnosu na sve binarne operatore i on je desno asocijativan.

Primjer 2.7. Uvedimo nekoliko cjelobrojnih promjenljivih koje koristimo za predstavljanje logičkih vrijednosti (koristimo 1 za tačno, 0 za netačno).

```
int a = 1, b = 0, c = 1, d, e;
d = (b && a) <= !c;
e = b && a <= !c;
```

Operator negacije !, kao unarni, je najvišeg prioriteta. Dalje, pošto je izraz `b && a` jednak nuli, zaključujemo da će vrijednost promjenljive `d` biti jednaka 1. Sa druge strane, pošto je relacijski operator `<=` višeg prioriteta, u odnosu na konjunkciju, izraz `b && a <= !c` je ekvivalentan izrazu `b && (a <= !c)`, te se, uz malo računanja, lako može odrediti da promjenljiva `e` ima vrijednost 0.

Lijeno izračunavanje

Pošto za proizvoljan logički izraz `a` važi

```
1 || a = 1       0 && a = 0
```

zaključujemo da u nekim situacijama, da bismo odredili krajnju vrijednost logičkog izraza, nije neophodno računati vrijednost svih pojedinačnih izraza koji su u njemu. Način izračunavanja izraza, i to tako da se računa samo ono što je u datom trenutku neophodno, je u programiranju poznat kao *strategija lijenog izračunavanja* (engl. lazy evaluation). Ova strategija se primjenjuje u mnogim modernim programskim jezicima, uključujući i programski jezik C.

Posmatrajmo sljedeći primjer.

Primjer 2.8. Deklarišimo cjelobrojne promjenljive `x` i `z` i promjenljivoj `x` dodijelimo početnu vrijednost 10.

```
int x = 10, z;
```

Ako promjenljivoj `z` dodijelimo vrijednost na sljedeći način i pozovemo funkciju `printf` za ispis

```
z = (3 < 0) && (x++);

printf("z= %d x = %d \n", z,x);
```

Elektronska verzija

na ekranu ćemo uočiti da je vrijednost promjenljive `x` ostala 10. To je upravo zbog "lijenog" izračunavanja, jer je vrijednost čitavog izraza `(3<0) && (x++)` postala poznata već nakon računanja podizraza `(3<0)`, te se ostatak izraza nije ni računao. Međutim, ako bismo promjenljivoj `z` dodijelili vrijednost na sljedeći način

```
z = (3 > 0) && (x++);

printf("z= %d x = %d \n", z,x);
```

na ekranu bismo uočili da je vrijednost promjenljive `x` uvećana i zaključujemo da se sada izraz `x++` izvršio. Zamjenom znaka `<` znakom `>` smo značajno promijenili situaciju, jer u novim okolnostima vrijednost promjenljive `z` zavisi upravo od desnog operanda konjunkcije (izraza `(x++)`).

Operatori na nivou bitova

Programski jezik C ima i operatore koji djeluju na pojedinačnim bitovima. Primjenjuju se na cjelobrojne tipove podataka, a preporučljivo je da koristimo neoznačene brojeve (na primjer **`unsigned char`** ili **`unsigned int`**) koji mogu biti samo pozitivni. Podsjetimo se da se negativni brojevi memorišu u potpunom komplementu, pa je manipulacija bitovima negativnih brojeva zbog toga komplikovana za praktičnu upotrebu.

Na bitovima su definisani sljedeći operatori:

~	komplement
&	bitovska konjunkcija
|	bitovska disjunkcija
^	bitovska ekskluzivna disjunkcija
<<	pomjeranje ulijevo
>>	pomjeranje udesno

Operator komplement `~` je unarni i djeluje tako što se svaki bit u broju koji je njegov operand invertuje, tj. svaki bit 1 se mijenja u 0, a svaki bit 0 u bit 1. Preostali operatori su binarni i infiksni.

Operatori `&`, | i ^ za operande uzimaju po dva broja i obavljaju odgovarajuće logičke operacije (operacije konjunkcije, disjunkcije i ekskluzivne disjunkcije) na pojedinačnim bitovima, koji se u operandima nalaze na odgovarajućim mjestima. Definicije operacija su prikazane u Tabeli 2.2.

Elektronska verzija

b1	b2	b1&b2	b1^b2	b1|b2
1	1	1	0	1
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Tabela 2.2: Operacije na bitovima

U narednom primjeru prikažimo kako ovi operatori djeluju na odgovarajuće argumente.

Primjer 2.9. Neka je cjelobrojni podatak `x` binarno zapisan

`x = 00010101`
a podatak
`y = 00010011`
Tada je
`~x = 11101010`
`~y = 11101100`
`x & y = 00010001`
`x | y = 00010111`
`x ^ y = 00000110`
Logičke operatore na bitnim zapisima koristimo za transformacije binarnog zapisa na odgovarajući način. Te transforacije zovemo i "maskiranje" (od engleske riječi mask). Pošto za neki bit `B` u bitovskoj aritmetici važi:

`B & 1 = B, B & 0 = 0`

`B | 1 = 1, B | 0 = B,`

`B ^ 0 = B, B ^ 1 =` ¬`B,`

`(B ^ A) ^ A = B`, odnosno operator ^ je inverzan samom sebi

možemo zaključiti sljedeće. Ako želimo da:

- neke bitove postavimo na 0, a druge da sačuvamo, tada primjenjujemo `&`
- neke bitove postavimo na 1, a druge da sačuvamo, tada primjenjujemo |
- neke bitove invertujemo, a druge da sačuvamo, tada primjenjujemo ^

Elektronska verzija

Razmotrimo sljedeći primjer: neka je `x = MMSSSMMS` broj tipa **`unsigned char`** zapisan u binarnom sistemu pomoću 8 bita , i neka `M` označava bit koji želimo da promijenimo, a `S` koji želimo da sačuvamo (ostaje isti). Tada

a. izrazom `x & 00111001`, `M` postavljamo na 0, `S` je sačuvano

b. izrazom `x | 11000110`, `M` postavljamo na 1, `S` je sačuvano

c. izrazom `x ^ 11000110`, `M` inverujemo, a `S` je sačuvano.

Operatori pomjeranja (operatori `<<` i `>>`) za operande uzimaju cio broj (koji je lijevi operand) i vrijednost za koliko se bitova vrši pomjeranje (desni operand). Kod operatora `<<`, na osnovu vrijednosti desnog operanda, binarni zapis lijevog operanda se pomiče za odgovarajući broj mjesta u lijevo, dok se sa desne strane dopisuju nule. U slučaju operatora `>>` vrši se pomjeranje bitnog zapisa lijevog operanda u desno, dok se sa lijeve strane dopisuju nule.

Mora se voditi računa da vrijednost desnog operanda ne bude veća od broja bitova u lijevom operandu.

Neka su `x` i `y` definisani kao u prethodnom primjeru
`x=00010101` i `y=00010011`. Tada je
`x << 1 = 00101010`
`y << 2 = 01001100`
`x >> 2 = 00000101`
`y >> 3  = 00000010`

Operatori dodjele

Osnovni operator dodjele (operator =) najčešće koristimo kada nekoj promjenljivoj želimo da dodijelimo neku vrijednost. Najjednostavniji primjer u kom cjelobrojnoj promjenljivoj `x` dodjeljujemo vrijednost je sljedeći:

```
x = 10;
```

Lijevi operand može da bude promjenljiva, element niza ili pokazivač na neku memorijsku lokaciju (o čemu će biti govora kasnije). Inače, objekti kojima može biti dodijeljena neka vrijednost se zovu l-vrijednosti (na engleskom jeziku se *l*-vrijednost naziva left-value ili *l*-value).

Treba napomenuti da i operator dodjele djeluje kao i drugi binarni infiksni operatori: on zahtijeva dva operanda i produkuje rezultat, koji je onog tipa kao i lijevi operand, koji ima vrijednost koja je lijevom operandu dodijeljena (koja ne mora uvijek da bude jednaka vrijednosti desnog operanda). U prethodnom primjeru, rezultat primjene operatora = je cio broj 10. Kao posljedicu toga što

Elektronska verzija

operator dodjele produkuje i rezultat, ovaj operator možemo i nadovezivati, na primjer:

```
z = (x = 10);
```

U izrazima kod kojih egzistira samo operator dodjele, izračunavanje ide zdesna na lijevo, te je stoga i ovakav zapis dozvoljen:

```
a = b = c = d;
```

Vrijednost promjenljive `d` se dodjeljuje promjenljivoj `c`, rezultat te dodjele se dodjeljuje promjenljivoj `b`, itd.

Pored osnovnog operatora dodjele imamo i složenije operatore dodjele, koji obično uključuju osnovni operator dodjele i neki aritmetički operator. Na primjer, izraz `a = a + 10` se može napisati i pomoću složenog operatora dodjele += na sljedeći način: `a += 10`. Odgovarajući operatori dodjele postoje za većinu aritmetičkih operatora. Tako imamo složene operatore dodjele `+=`, `-=`, `*=`, `/=`, `%=`, koje koristimo kada radimo sa cijelim, odnosno realnim brojevima. Postoje i složeni operatori dodjele koji kombinuju operatore na bitovima: `&=`, |=,`<<=` i `>>=`.

U većini slučajeva, složeni operatori dodjele funkcionišu kao i odgovarajući izrazi koji koriste posebno aritmetički operator, pa onda osnovni operator dodjele. Kao što je već pomenuto, izraz `a=a+10` ostavlja potpuno isti efekat kao i izraz `a+=10`. Međutim, u nekim specifičnim situacijama (koje u većini slučajeva nisu "prirodne", ali se teoretski mogu javiti) može doći i do pojave da ta dva zapisa nisu isti. Na sljedećem primjeru koji uključuje i nizove, koji će biti razmatrani u nastavku udžbenika, može se vidjeti razlika u efektu ova dva zapisa. U izrazu

```
a[i++] += 1;
```

promjenljiva `i` će biti uvećana za 1 samo jednom, dok će u zapisu

```
a[i++] = a[i++] + 1;
```

promjenljiva `i` biti uvećana za 1 dva puta.

Ipak, ovakve situacije su same po sebi rijetke, a pošto mogu dovesti do konfuzije, treba ih izbjegavati. U slučajevima kada oba zapisa postižu isti efekat, izbor koji od njih upotrijebiti ostaje na programeru. Može se procijeniti da će iskusniji programeri radije koristiti kraći zapis, mada to i ne mora biti pravilo. Na primjer, ako je neku običnu cjelobrojnu promjenljivu `a` potrebno uvećati za 2, podjednako je prirodno koristiti bilo koji od dva zapisa:

Elektronska verzija

```
a = a + 2;
```

ili

```
a += 2;
```

Međutim, ako neki element cjelobrojnog trodimenzionalnog niza (nizovi, pa i oni višedimenzionalni će biti razmatrani u nastavku) potrebno uvećati za dva, onda je prirodnije koristiti "skraćeni" zapis:

```
niz[i][j][k] += 2;
```

nego

```
niz[i][j][k] = niz[i][j][k] + 2;
```

iako oba zapisa imaju potpuno isti efekat.

Uslovni operator

Programski jezik C ima samo jedan ternarni operator (operator koji uzima tri operanda), koji se koristi u sljedećem zapisu:

```
izraz1 ? izraz2 : izraz3
```

Izraz koji je formiran ovim operatorom se naziva uslovni izraz (ovaj operator se zove i uslovni operator). Uslovni operator funkcioniše po sljedećem principu: prvo se izračunava `izraz1`. U slučaju da je istinitosna vrijednost rezultata tog izraza tačno (različita od nula), onda se računa izraz `izraz2` i ta vrijednost je ujedno i vrijednost čitavog uslovnog izraza. Ako je istinitosna vrijednost rezultata tog izraza `izraz1` netačno, izračunava se `izraz3` i vrijednost tog izraza `izraz3` je i vrijednost čitavog uslovnog izraza. Na primjer, rezultat izraza

```
3 < 4 ? 5 : 6
```

će biti 5.

Izračunavanjem izraza

```
y = (x >= 0) ? x : -x
```

promjenljiva `y` će dobiti vrijednost apsolutne vrijednosti promjenljive `x`. Primijetimo da smo u prethodnom primjeru izraz `x >= 0` stavili u zagradu, kako

Elektronska verzija

bismo otklonili nedoumicu da li će operator `>=` imati prioritet u odnosu na simbol `?`, koji se koristi u okviru sintakse uslovnog operatora.

Postoje razne kombinacije tipova koji se mogu koristiti za `izraz2` i `izraz3`, na osnovu kojih zavisi i tip rezultata kompletnog uslovnog izraza. Najlakši slučaj je kada su rezultati izraza `izraz2` i `izraz3` istog tipa, jer je onda jasno da će i rezultat biti tog istog tipa. Međutim, u mnogim drugim slučajevima tip rezultata nije toliko očigledan, tako da nejasne situacije treba izbjegavati. Tako u primjeru

```
x == 3 ? 4.5 : 6
```

vidimo da je predviđeno da rezultat uslovnog operatora bude jednak realnom podatku 4.5 ukoliko promjenljiva `x` ima vrijednost 3, odnosno cjelobrojnom podatku (vrijednost 6) ukoliko to nije slučaj. U ovom slučaju, vršiće se konverzija cjelobrojnog podatka u realni tip i rezultat je (koliko-toliko) očekivan. Sa druge strane, u mnogim drugim slučajevima (posebno kada se "u igru" uvedu i složeniji tipovi podataka, kao što su pokazivači o kojima će biti riječi u Poglavlju 8), mogu se javiti slučajevi u kojima nije potpuno jasno kakav će biti tip rezultata uslovnog operatora. Stoga je dobra programerska praksa da se takve situacije izbjegavaju.

Operator zarez

Operator zarez (zapravo operator koji je predstavljen karakterom zarez ,) se najčešće koristi kada je dva izraza potrebno spojiti u jedan (jedinstven izraz), posebno na onim mjestima gdje sintaksa zahtijeva navođenje samo jednog izraza. Ovaj operator je najnižeg prioriteta i smatra se binarnim operatorom. U najvećem broju slučajeva (kada ovaj operator uopšte ima smisla koristiti), lijevi i desni operand su neki izrazi dodjele. Rezultat operatora zarez je jednak vrijednosti desnog operanda (a i ta vrijednost se najčešće zanemaruje). Na primjer, dozvoljeno je pisati

```
a = 1, b = 2;
```

gdje vidimo da su lijevi i desni operandi operatora , izrazi dodjele (`a = 1` i `b = 2`). Rezultat čitavog izraza jednak je rezultatu izraza `b=2`, (a to je cio broj 2), koji u ovom slučaju nema nikakvu dalju ulogu.

Treba napomenuti da se simbol , u programskom jeziku C koristi i za druge stvari (na primjer za odvajanje argumenata unutar liste argumenata funkcije), te da u tim slučajevima zarez ne smatramo operatorom, već pomoćnim simbolom koji učestvuje u gradnji drugih izraza.

Elektronska verzija

Operator sizeof

Unarni operator **sizeof** daje za rezultat broj koji predstavlja veličinu memorijskog prostora (u bajtovima) potrebnu za smiještanje svog operanda. Može se primjenjivati na promjenljive, tip podatka ili izraz. Ako **sizeof** primijenimo na neku promjenljivu, kao rezultat ćemo dobiti broj bajtova koliko je u memoriji rezervisano za čuvanje te promjenljive. Ako se **sizeof** primijeni na tip podatka, kao rezultat se vraća veličina tog tipa, odnosno broj bajtova potrebnih za smiještanje podataka tog tipa, a ako se primijeni na izraz, onda kao rezultat vraća veličinu izraza. Tip rezultata operatora **sizeof** je cio broj, za koji je u programskom jeziku C obezbijeđen zaseban tip **size_t**. Tipom **size_t** se predstavljaju neoznačeni cijeli brojevi i on se najčešće koristi prilikom rada sa funkcijama koje manipulišu količinom memorije. Ovaj tip je dovoljno širok da može da sadrži informaciju o veličini bilo kog objekta u datoj implementaciji programskog jezika. Stoga se ovaj tip ne poistovjećuje ni sa jednim do sada razmatranim tipom za neoznačene cijele brojeve (na primjer sa tipom **unsigned int**).

Za operator **sizeof** kažemo da je *compile-time* operator, što znači da se vrijednost izračunava za vrijeme prevođenja programa.

Primjer 2.10. Na većini verzija prevodioca, sljedećim programom će se na ekranu ispisati redom vrijednost: 1 4 4 i 8.

```
1
2  #include<stdio.h>
3  int main()
4  {
5      printf("%d\n",sizeof(char));
6      printf("%d\n",sizeof(int));
7      printf("%d\n",sizeof(float));
8      printf("%d", sizeof(double));
9      return 0;
10 }
```

Ako operator **sizeof** primijenimo na neki složeniji podatak koji se sastoji od više pojedinačnih podataka, onda je rezultat jednak zbiru veličina pojedinačnih podataka.

Primjer 2.11. Iako još nismo uveli nizove podataka i niske karaktera, možemo primijeniti operator **sizeof** na neke podatke ovih tipova i razumjeti šta dobijamo za rezultat. Ako pokrenemo naredni program, vidjećemo da će se na ekranu ispisati vrijednosti 6 i 24.

Elektronska verzija

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

   printf("%d\n",sizeof("tabla"));
    int niz[] = {10,20,30,40,50,60};
    printf("%d\n",sizeof(niz));
    return 0;
}
```

Riječ `"tabla"` se sastoji od pet karaktera, ali je (pošto je riječ o niski karaktera) na kraju pridodat i teminalni karakter `'\0'` (o tome ćemo detaljnije govoriti u Poglavlju 6, kada budemo razmatrali nizove), pa, stoga, dobijamo da je rezultat primjene operatora **`sizeof`** na nisku `"tabla"` jednak 6. Niz brojeva smo definisali tako što smo njegovih 6 elemenata inicijalizovali odgovarajućim vrijednostima. Kako je veličina svakog cijelog broja (podatka tipa **`int`**) 4, ukupan broj bajtova koje je ovaj niz zauzeo je $6 \cdot 4 = 24$.

Elektronska verzija

2.5.3 Konverzija tipova

U programskom jeziku C je u mnogim slučajevima dozvoljeno "miješanje" podataka koji su različitih tipova. Na primjer, dozvoljeno je sabiranje (i druge osnovne aritmetičke operacije) cjelobrojnog i realnog podatka, upoređivanje cijelog i realnog broja, dodjela cjelobrojne vrijednosti realnoj promjenljivoj, dodjela vrijednosti tipa **`char`** promjenljivoj tipa **`int`**, itd. U svim ovim slučajevima, ali i u mnogim drugim, govorimo o *konverziji* jednog tipa podatka u drugi. Neke konverzije se obavljaju automatski (praktično bez kontrole programera) i takve konverzije zovemo **implicitne konverzije**, dok je za druge odgovoran programer. Konverzije koje se izvršavaju na zahtjev programera zovemo **eksplicitne konverzije** ili **kasting** (engl. casting). Kod konverzije podataka iz jednog tipa u drugi moramo biti dosta oprezni, jer često može doći do gubitka podataka. Na primjer, pretpostavimo da promjenljiva tipa **`short`** ima vrijednost 1000 i da je želimo konvertovati u podatak tipa **`char`**. Pošto podatak tipa **`char`** nema dovoljno bitova da bi se predstavio broj 1000 (broj 1000 se u binarnom zapisu predstavlja sa 10 binarnih cifara, a kod podatka tipa **`char`** je na raspolaganju svega 8 bitova), prilikom pokušaja ovakve konverzije doći će do gubljenja informacija, tj. podatak tipa **`char`** neće moći da sadrži originalnu vrijednost.

S obzirom na to da je programski jezik C prilično fleksibilan kada je riječ o slučajevima u kojima je moguća konverzija, poznavanje pravila za razne tipove

konverzije podataka je veoma važno, kako zbog pravilnog korištenja ovog mehanizma, tako i zbog sticanja sposobnosti korištenja tehnika konverzije u situacijama kada je to neophodno ili poželjno.

Tehnike konverzije koje ćemo objasniti u ovom dijelu se odnose na brojevne podatke (cijele i realne brojeve). Kako programski jezik C dopušta rad i sa podacima koji su drugačije prirode u odnosu na brojeve (na primjer sa pokazivačima, o kojima će biti riječi kasnije), postoje i druga pravila za konverziju, kojima, takođe, u odgovarajućem trenutku, treba posvetiti pažnju.

Implicitne konverzije

Jedan od najčešćih implicitnih oblika konverzije je tzv. **promocija** (engl. promotion - unapređenje), koja podrazumijeva konverziju vrijednosti "nižeg tipa" u vrijednost "višeg tipa". Prilikom promocija, uglavnom ne dolazi do gubljenja informacija, jer je skup vrijednosti tipa podatka u koji se stari tip "promoviše" po pravilu nadskup vrijednosti starog tipa. Na primjer, promocijom se bez gubitka informacija podatak tipa `short` može konvertovati u `int`, `int` u `long`, `float` u `double` itd.

Primjer 2.12. Posmatrajmo karakterističan primjer implicitne konverzije - promocije. Na prvi pogled, čini se da će prozvod brojeva `a` i `b` već izaći iz opsega, jer je riječ o podacima tipa `char`, te bi dalje izračunavanje dovelo do greške. Međutim, u međuvremenu se desila implicitna konverzija, te je pojava prekoračenja izbjegnuta, a promjenljiva `d` dobija vrijednost 100.

```
char a = 20, b = 50, c = 10;
char d = (a * b) / c;
```

Naglasimo da važi opšte pravilo, koje smo prethodnim primjerom ilustrovali, da se aritmetički operatori ne primjenjuju na tipove podataka `char` i `short`, jer je u slučaju rada sa ovim, "malim" tipovima vjerovatnije da će doći do prekoračenja. Stoga se prije primjene operatora ovi tipovi promovišu u podatke tipa `int`.

U slučaju kada se cjelobrojni tip (`int`) konvertuje u realni (`float`), u nekim slučajevima ipak može da dođe do gubljenja informacija. Na primjer, ako podatak tipa `float` koristi 32 bita za zapis po standardu IEEE 754 (što je i slučaj sa velikim brojem današnjih računarskih sistema), tada se za mantisu koriste 3 bajta (24 bita), te svi cijeli brojevi do 2^{24} mogu predstaviti ovim realnim tipom. Međutim, samo neke cjelobrojne vrijednosti veće od 2^{24} mogu biti predstavljene ovim podatkom, dok neke i ne mogu. Tako se broj 16777217, koji je jednak $2^{24} + 1$ binarno zapisuje pomoću 25 cifara $((16777217)_{10} =$

Elektronska verzija

$(100000000000000000000001)_2$) i on se jednostavno podatkom tipa `float` ne može predstaviti.

Primjer 2.13. Jedan od najtipičnijih primjera promocije cijelog broja u realan imamo prilikom dodjele cjelobrojne vrijednosti realnoj promjenljivoj. Na primjer

```
float f = 5;
```

gdje se cio broj 5 implicitno konvertuje u realan broj 5.0, te se ta vrijednost dodjeljuje promjenljivoj `f`.

Moguća je i konverzija kada se "širi" tip podatka konvertuje u "uži" (na primjer `double` u `int` ili `long` u `short`). U slučaju da se polazna vrijednost ne može predstaviti podatkom koji je "uži", nastaju situacije kada dolazi do gubitka informacija. Ovakav tip konverzije zovemo democija (od engleske riječi demotion). Neke od ovih situacija prevodioci registruju kao potencijalni problem i javlja odgovarajuće upozorenje.

Primjer 2.14. Navedimo dva primjera democije su sljedeći.

```
int b1;
b1 = 10.25;
int c1 = 123456789;
char d1 = c1;
```

Ako se vrši konverzija realnog podatka u cjelobrojni, tada se cio broj od realnog najčešće dobija odsjecanjem decimala, tj. uzima se cio dio realnog broja. Stoga će promjenljiva `b1` zapravo dobiti vrijednost 10. U drugom slučaju je očigledno da se podatkom tipa `char` ne može predstaviti vrijednost promjenljive `c1`, te će u ovoj situaciji doći do gubitka polaznog podatka.

Implicitne promocije mogu da se jave i prilikom izvršavanja nekih operacija, kada se vrijednosti koje imaju operandi promovišu iz jednog tipa u drugi. Ako je potrebno, primjenom operatora dodjele već može da se vrši promocija desne strane u odgovarajući tip lijeve strane. Na primjer:

```
int novac = 1000;
float zarada = novac;
```

Implicitna konverzija se vrši i u slučajevima kada operandi aritmetičkih operacija nisu istog tipa. Ovakve konverzije se nazivaju uobičajene aritmetičke konverzije (od engleskog izraza usual arithmetic conversions).

Elektronska verzija

Primjer 2.15. Klasične primjere ovog tipa konverzije lako možemo vidjeti u aritmetičkim izrazima u kojima se kombinuju operandi različitog tipa.

```
int  i = 17;
char c = 'c'; // ascii vrijednost je 99
float suma;
suma = i + c;
```

Prilikom određivanja vrijednosti promjenljive `suma`, najprije se vrši konverzija promjenljive `c` u podatak tipa `int`, te se nakon toga, zbir `i+c` konvertuje u podatak tipa `float`.

Primjer 2.16. U nekim situacijama kada to može da olakša izračunavanje, programer može da "natjera" program da izvrši automatsku konverziju. Posmatrajmo sljedeću slučaj. Potrebno je odrediti prosječnu vrijednost neka tri cijela broja. Ukoliko bismo prosjek računali na sljedeći način

```
int a = 4, b = 5, c = 7;
float f = (a + b + c) / 3;
```

vrijednost promjenljive `f` bi bila 5.0, jer bi se prvo izvršilo sabiranje tri cijela broja `a`, `b` i `c`, te bi se nakon toga taj zbir (koji je cio broj) podijelio sa 3, ali cjelobrojno. Nakon što bi se, prema zadatim vrijednostima promjenljivih dobio rezultat cjelobrojnog dijeljenja (rezultat je cio broj 5), taj broj bi se dalje implicitno konvertovao u podatak tipa `float` u vrijednost 5.0. Međutim, ako bismo umjesto prethodnog imali

```
int a = 4, b = 5, c = 7;
float f = (a + b + c + 0.0) / 3;
```

dobili bismo situaciju da se u zagradi sabiraju tri cijela i jedan realan broj 0.0, te je rezultat takvog sabiranja realan broj. Dalje ponovo imamo implicitnu konverziju koja nastaje usljed dijeljenja realnog broja cijelim (sada se realan broj 16.0 dijeli cijelim brojem 3), pa se broj 3 konvertuje u realan 3.0, a količnik sada računamo tako što sada dijelimo dva realna broja. Dobijamo rezultat 5.333333.

Eksplicitne konverzije

Eksplicitna konverzija ili kasting je proces koji je definisan od strane programera. U opštem slučaju se vrši na sljedeći način:

```
(tip)izraz
```

Elektronska verzija

gdje `izraz` postaje tipa `tip`.

Primjer 2.17. Jednostavan primjer eksplicitne konverzije možemo vidjeti u sljedećem zapisu.

```
double x = 1.2, y = 4.8;
int suma = (int)x + (int)y;
```

Promjenljiva suma će dobiti vrijednost 5, jer je prije sabiranja izvršena eksplicitna konverzija promjenljivih `x` i `y` u cio broj, koja se vrši "odsjecanjem" decimalnog dijela (ne zaokruživanjem). Praktično, sabrani su brojevi 1 i 4.

2.5.4 Enumerativni tipovi

Pomoću enumerativnih ili nabrojivih tipova nam je omogućeno da se nabrajanjem deklarišemo simbolička imena koja uzimaju cjelobrojne vrijednost. Opšti oblik sintakse upotrebe enumerativnih tipova izgleda ovako

```
enum {ime1 [=konstanta1], ime2 [=konstanta2],... }
```

Dodjeljivanje konstantnih vrijednosti je opcionalno. Ako se vrijednost izostavi, ona će biti automatski dodijeljena i to na sljedeći način: ako se izostavi vrijednost prvoj simboličkoj konstanti u listi (u našem zapisu to je simbolička konstantna `ime1`), tada će njoj automatski biti dodijeljena vrijednost 0. Ostale konstante kojima nije eksplicitno dodijeljena vrijednost, dobijaju vrijednost za 1 veću od vrijednosti prethodne konstante.

Na primjer, ako bismo imali sljedeću situaciju

```
enum podaci {jedan, dva = 10, tri, cetiri = -1, pet, sest = 15};
```

Tada bi simboličke konstante `jedan`, `dva`, `tri`, `cetiri`, `pet` i `sest` redom imale vrijednost: 0, 10, 11, -1, 0 i 15.

Upotreba enumerativnih tipova nije baš česta. Za to ima nekoliko razloga. U radu sa enumerativnim tipovima smo ograničeni na relativno mali skup vrijednosti (odnosno ovaj tip ima onoliko vrijednosti koliko ih mi nabrojimo). Pored toga, ograničeni smo samo na cjelobrojne podatke. Napomenimo i da ćemo, kada budemo detaljnije razmatrali preprocesorske direktive u Poglavlju 13, navesti još jedan način kako se (pomoću direktive `#define`) simboličkim imenima, takođe, može dodijeliti odgovarajuća vrijednost (i to ne samo cjelobrojna), čime se dodatno umanjuje značaj enumerativnih tipova, jer se za slične namjene mogu koristiti i druge tehnike.

Elektronska verzija

Ipak, upotreba enumerativnih tipova u nekim situacijama je praktična. Na primjer, kada nekim podacima treba da pridružimo neke simboličke vrijednosti, najpraktičnije je da takve vrijednosti predstavimo brojevima.

Primjer 2.18. Ako imamo neke objekte koje trebamo da "obojimo" sa nekoliko boja, umjesto da svakom objektu pridružujemo odgovarajuću nisku karaktera (na primjer "crvena", "plava", "zelena" itd.), praktičnije je da definišemo enumerativni tip, na primjer:

```
enum boja {crvena = 1, plava, zelena, roze, bijela, crna};
```

te da u nastavku programa raspolažemo ovakvim simboličkim konstantama (koje suštinski imaju cjelobrojne vrijednosti). Na primjer

```
enum boja x,y;
x = crvena; /* x = 1; */
...
x = roze; /* x = 4; */
...
if(x == crvena) /* if(x == 1) */
...
```

2.5.5 Ključna riječ typedef

Pri kraju ovog veoma važnog poglavlja uvedimo još i ključnu riječ `typedef`, pomoću koje postojećim tipovima podataka možemo dodjeljivati nova imena. Opšti oblik upotrebe riječi `typedef` je sljedeći:

```
typedef tip_podatka novo_ime;
```

Ovaj pristup možemo koristiti u velikom broju slučajeva, od kojih su neki manje, a neki više značajni. Na primjer, za brojevne podatke možemo uvesti novi tip na osnovu postojećeg tipa `int` na sljedeći način:

```
typedef int brojevi; //deklarisemo novi tip podatka
...
brojevi a,b,c; //deklarisemo promjenljive koje su sad tipa brojevi
...
```

i ove promjenljive koristimo kao da su tipa `int`.

Jasno je da se ovakav pristup ne čini mnogo smislenim, ali riječ `typedef` u nekim situacijama može da skrati pisanje i poveća preglednost. Vratimo se

Elektronska verzija

na prethodnu sekciju i enumerativnim tipovima. U posljednjem primjeru smo mogli uvesti ime za tip `enum boja`, na primjer na sljedeći način:

```
enum boja {crvena = 1, plava, zelena, roze, bijela, crna};
typedef enum boja boja; //uvedemo tip boja umjesto enum boja
boja x,y; //sada koristimo novo ime za deklaraciju promjenljivih
x = crvena; /* x = 1; */
//...
x = roze; /* x = 4; */
//...
```

O daljnjoj smislenoj upotrebi ključne riječi `typedef` biće govora i u nastavku udžbenika, posebno u dijelovima koji se odnose na strukture i pokazivače.

Elektronska verzija

2.6 Pitanja i zadaci

1. Objasniti zašto sljedeće riječi nisu odgovarajuće za imena promjenljivih:
   a) `goto`
   b) `1broj`
   c) `p&c`
2. Objasniti razliku između sljedećih deklaracija promjenljive `x`:
   a) `int x=10;`
   b) `const int x=10;`
3. Koji je najmanji, a koji najveći broj koji se može predstaviti podatkom tipova `short` i `unsigned short`?
4. Znajući da je $2^{10} \approx 10^3$, procijeni koliko cifara ima:
   a) najveći podatak tipa `long`
   b) najveći podatak tipa `unsigned long`
5. Može li se broj koji je veći od 5 000 000 000 predstaviti podatkom tipa `int`? Obrazloži odgovor.
6. Navesti primjere kada se kao rezultat dobijaju vrijednosti `NaN`, pozitivna i negativna beskonačnost. Objasniti razliku između ovih vrijednosti.
7. Istražiti standardnu biblioteku `limits.h`, objasniti šta sadrži i ilustrovati primjerima njenu upotrebu.

8. Istražiti standardnu biblioteku `float.h`, objasniti šta sadrži i ilustrovati primjerima njenu upotrebu.

9. Istražiti datoteku zaglavlja `math.h`, objasniti šta sadrži i ilustrovati primjerima njenu upotrebu.

10. Objasniti razliku između vrijednosti `'1'` i `1`.

11. Kako od promjenljive tipa `char` koja ima vrijednost `'5'` odrediti promjenljivu tipa `int` koji ima vrijednost 5? Objasniti princip računanja odgovarajuće brojne vrijednosti proizvoljne cifre.

12. Date su vrijednosti promjenljivih `a=10`, `b=20`, `c=30`. Kolike će biti vrijednosti tih promjenljivih nakon izvršenja sljedećih operatora: `a=b`, `b=c` i `c=a`? A kolike ce vrijednosti biti nakon izvršenja izraza `a=b=c=a`?

13. Promjenljivoj `s` dodjeliti zbir cifara petocifrenog prirodnog broja `n`.

14. Napisati operator dodjele, kombinujući ga sa ternarnim uslovnim operatorom, kojim će se manja od vrijednosti realnih promjenljivih `a` i `b` dodijeliti promjenljivoj `min`, a veća vrijednost promjenljivoj `max`.

15. Navedi primjer kada rezultujuća vrijednost operatora dodjele, koja je dodijeljena lijevom operandu, nije jednaka vrijednosti desnog operanda.

16. Odrediti rezultate koji se dobijaju izvršenjem sljedećih operatora dodjele

    a) `x = (y = 100) + 5`

    b) `x = y = 100 + 5`

    Da li su dobijeni isti rezultati? Ako nisu dobijeni isti rezultati, objasniti zašto.

17. U izrazu postaviti zagrade tako da je vidljiv redoslijed izvršenja operacija `a + 3 < b || b > c * d && a >= d`.

18. U izrazu `x = (y = (z / w) + j) * k` odrediti redoslijed izvršenja operacija.

19. Odrediti vrijednost promjenljivih u sljedećim izrazima:

    a)
    ```
    int a, x=-1, y, z;
    a = y = z = 1;
    a = ++x && (++y || ++z);
    ```

Elektronska verzija

b) `float z; z= 7 / 2;`

c) `float z; z= 7.0 / 2;`

d) `float x = (int) 3.4 + (int) 8.1;`

e) `int a;`

`(110 % 2) ? a = 12.5 - 5.1 : a = 12.5 + 5.1`

20. Šta će se ispisati na ekranu pokretanjem sljedećeg kôda?

```
#include <stdio.h>
int main()
{
   char a, b;

   printf("Velicina je: %d\n", sizeof(a));
   printf("Velicina je: %d\n", sizeof(a + b));

   return 0;
}
```

21. Kakav je odnos prioriteta aritmetičkih, relacijskih i logičkih operatora?

22. Napisati izraz koji ima vrijednost tačno ako je karakter `c` cifra.

23. Napisati operator dodjele koji ce promjenljivoj `x` koja je **`unsigned`** sačuvati `n` krajnjih desnih bitova, a ostale postaviti na nulu.

24. Napisati operator dodjele kojim se komplementira (nule zamjenjuje jedinicama, jedinice nulama) promjenljiva `x` od pozicije `p` na dužini `n`. Bitovi su numerisani od nule zdesna u lijevo.

25. Ako je niz cijelih brojeva zadat na sljedeći način

    `int x[]={1, 2, 10, 20, 30};`

    Kako odrediti broj elemenata koji se nalazi u ovom nizu, a da to ne bude "ručno" prebrojavanje?

26. Pomoću nabrojivih tipova ilustrovati primjer u kojem se predstavljaju vrijednosti karata iz standardnog špila od 52 karte. Moguće vrijednosti su kralj, dama, žandar, as, 10, 9, 8,..., 1.

Elektronska verzija

3 Vrste programskih naredbi

Svaki program se sastoji od niza pojedinačnih naredbi. Naredbe su osnovni elementi programa, kojima se opisuju i definišu izračunavanja i predstavljaju potpunu instrukciju upućenu računaru, koju računar treba da izvrši. Naredbe mogu da se izvršavaju sekvenijalno, tj. redom jedna za drugom kako su i napisane, dok se izmjena redoslijeda izvršenja naredbi postiže upotrebom naredbi grananja, naredbi skoka ili iterativnim naredbama. Svaka naredba u programskom jeziku C se završava znakom "tačka i zarez" ;.

Naredbe mogu biti jednostavne i složene. Najvažnije naredbe koje možemo smatrati jednostavnim su naredbe izraza, dok su složene naredbe sastavljene od više jednostavnijih.

3.1 Naredbe izraza

Osnovni i najjednostavniji oblik naredbi je naredba izraza. Ponovimo da u programskom jeziku C izraze formiramo od promjenljivih, konstanti, imena funkcija i operatora, dok složenije izraze dobijamo kombinovanjem više manjih, jednostavnijih izraza. Izvršavanjem izraza određuje se njegova vrijednost, u skladu sa prisutnim operacijama i njihovim prioritetima.

Pored naredbi koje podrazumijevaju izvršenje izraza, ovdje treba pomenuti i druge jednostavne naredbe, od kojih je najjednostavnija tzv. prazna naredba, tj. naredba koja sadrži samo znak ;, naredbu kojom se poziva izvršenje funkcije, kao i naredbu koja se koristi za vraćanje rezultata funkcije (naredba **`return`**).

Primjer 3.1. Navedimo nekoliko naredbi izraza kao primjer.

```
a = 0;
i = i + 1;
x *= 3;
c = sqrt(a * a + b * b);
printf("Zdravo narode!\n");
c++;
return rezultat;
; //prazna naredba, ima samo tacku zarez
c + sum(a + b); //ispravna, ali ne bas smislena naredba
```

Elektronska verzija

3.2 Složene naredbe

Često je potrebno više naredbi, koje obično slijede jedna za drugom, posmatrati kao jednu jedinstvenu složenu naredbu. Za grupisanje više naredbi u jednu složenu naredbu koristimo vitičaste zagrade { i }. Složenu naredbu još nazivamo i blok naredbi. Dozvoljeno je i da se blok sastoji od samo jedne naredbe, tj. unutar vitičastih zagrada se nalazi samo jedna naredba. U većini slučajeva jednu naredbu nije obavezno stavljati u "blok". Sa druge strane, postoje situacije u kojima je grupisanje naredbi u okviru vitičastih zagrada obavezno čak i ako se blok sastoji od samo jedne naredbe, kao što je to slučaj sa blokom naredbi koje čine tijelo funkcije, ili naredbama koje su u okviru **do-while** iterativne naredbe. Nakon zatvaranja vitičaste zagrade kojom završavamo blok naredbi nije potrebno stavljati znak ;.

Iako se blokovi naredbi mogu praviti na prilično proizvoljan način (na primjer možemo praviti blok iza bloka, ili blok u bloku), blokovi naredbi ipak trebaju da budu organizovani kao logičke cjeline, tj. grupisanje naredbi treba raditi onda kada za tim postoji stvarna potreba. Pored toga, ispravan rad sa blokovima je veoma važan i zbog tzv. d*osega promjenljivih*, o čemu će dosta biti riječi kasnije u Poglavlju 7. Ovdje samo pomenimo da se mora voditi računa da je promjenljiva definisana u okviru nekog bloka, dostupna samo u tom bloku (a ne i van njega).

Primjer 3.2. Sljedeći zadatak nije ispravno napisan. Iako bi početniku možda bilo "logično" da deklaracije promjenljivih grupiše u jedan blok, a nastavak programa u drugi, to nije ispravno, jer su promjenljive a i b definisane unutar jednog bloka i kao lokalne promjenljive su vidljive samo u tom bloku. Da bi ovaj zadatak bio sintaksno ispravan, nisu potrebni dodatni blokovi unutar bloka za početak i kraj main funkcije.

```
1  #include <stdio.h>
2  int main(){
3     {
4        int a = 10;
5        int b = 15;
6     }
7     {
8        printf("%d\n",a);//pogresno, a nije vidljiva van bloka u kome je
             deklarisana
9     }
10    return 0;
11 }
```

Elektronska verzija

3.3 Naredba grananja

Naredbe grananje su jedne od najvažnijih programskih struktura kojima se omogućava izvršavanje pojedinih grupa naredbi, u zavisnosti od rezultata postavljenog uslova. Ovu grupu naredbi čini poznata `if` uslovna naredba, te naredba višestrukog grananja - `switch` naredba.

3.3.1 if-else naredba

Naredba `if` ima sljedeću sintaksu

```
if(uslov)
    naredba1;
else
    naredba2;
```

Naredbe `naredba1` i `naredba2` mogu biti ili pojedinačne naredbe (na primjer naredba izraza ili naredba poziva funkcije), ili složena naredba (blok naredbi), tj. više naredbi grupisanih u okviru vitičastih zagrada. Ponovimo da, u slučaju da imamo samo jednu naredbu, nju možemo, ali i ne moramo "grupisati" u blok pomoću vitičastih zagrada.

Dio naredbe `else` (sama riječ `else` i naredba koja slijedi nakon nje) nije obavezan, što je inače slučaj i u drugim imperativnim programskim jezicima.

Izraz koji slijedi nakon riječi `if` je logički uslov (za koga se smatra da može da bude tačan ili netačan). Kao što je već pominjano, logički izrazi se u programskom jeziku C uglavnom predstavljaju cjelobrojnim podacima (mada mogu i brojevima u pokretnom zarezu). Izraz koji je različit od nule se smatra tačnim, dok se izraz koji je jednak nuli smatra netačnim. U programskom jeziku C ovaj izraz se obavezno stavlja u male zagrade.

Jedna uslovna naredba može da sadrži nove uslovne naredbe, čime omogućavamo grananje programa na više od dvije grane. Tako dobijamo višestruku `if` naredbu, koja sadrži više `if` ugniježdenih naredbi. U tom slučaju, naredba `else` se uvijek veže za posljednje neupareno `if`, ukoliko zagradama nije drugačije određeno.

Kroz analizu jednostavnih zadataka koji slijede mogu se uočiti i razumjeti različiti načini upotrebe `if` naredbe.

Primjer 3.3. Napisati program koji zahtijeva unos dva cijela broja `a` i `b` sa tastature, a zatim računa vrijednost `f` na sljedeći način:

`f = a + b` ako je `a` neparno

`f = a * b` ako je `a` parno

Elektronska verzija

```
1
2  #include <stdio.h>
3
4  int main()
5  {
6      int a, b;
7      printf("Unesite brojeve a i b.\n");
8      scanf("%d %d",&a,&b);
9
10     int f;
11
12     if(a % 2 == 0)
13         f = a + b;
14     else
15         f = a * b;
16
17     printf("f=%d\n",f);
18
19     return 0;
20 }
```

Primjer 3.4. Napisati program koji promjenljivoj `MAX` dodjeljuje najveću vrijednost od tri unesena cijela broja `a`,`b`,`c`.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a, b, c, MAX;
6
7      printf("Unesite brojeve a, b i c:");
8      scanf("%d %d% d",&a,&b,&c);
9      if(a > b && a > c)
10         MAX = a;
11     else
12         if(b > a && b > c)
13             MAX = b;
14         else
15             MAX = c;
16     printf("Najveci medju unesenim brojevima je %d.\n",MAX);
17     return 0;
18 }
```

Elektronska verzija

Primjer 3.5. Napisati program koji za učitanu vrijednost ugla u stepenima (koja može biti veća od 360) određuje kvadrant kome ugao pripada.

```
#include<stdio.h>
int main()
{
    int u;
    printf("Unesite ugao:\n ");
    scanf("%d",&u);

    u = u % 360;

    if(0 <= u && u < 90)
        printf("I kvadrant.\n");
    else
        if(u >= 90 && u < 180)
            printf("II kvadrant.\n");
        else
            if(u >= 180 && u < 270)
                printf("III kvadrant.\n");
            else
                printf("IV kvadrant.\n");
    return 0;

}
```

Po ovom primjeru vidimo da je nekada potrebno ugnijezditi nekoliko **if** naredbi da bi se pokrili svi slučajevi. Umjesto toga, možemo koristiti naredbu **switch**, koju ćemo objasniti u narednoj sekciji.

3.3.2 switch naredba

Naredba **switch** se koristi za višestruko grananje i najčešće ima sljedeći opšti oblik:

```
switch(izraz)
{

    case konstanta1:
        naredba1; break;
    case konstanta2:
        naredba2; break;
...
```

Elektronska verzija

```
    default:
        naredbaN; break;

}
```

Kao što se vidi, ova naredba počinje riječju `switch` nakon koje slijedi izraz, koji pišemo u malim zagradama, a koji je najčešće cjelobrojnog ili nabrojivog tipa. Nakon ovog izraza slijedi niz od nekoliko dijelova koji počinju riječju `case`. Iza svake riječi `case` slijedi konstantna vrijednost koja mora biti istog tipa kao i vrijednost izraza u malim zagradama, potom znak :, te, nakon toga, jedna ili više naredbi, koje su odvojene `;`. Idući od jednog do drugog slučaja, program ispituje da li je izraz jednak konstantnoj vrijednosti koja slijedi nakon neke riječi `case`. Kada program naiđe na vrijednost koja je jednaka vrijednosti `izraz`, izvršavaju se naredbe koje slijede nakon dvije tačke, sve dok se ne dođe do kraja čitavog bloka ili do naredbe `break`. Naredbu `break` nije obavezno navoditi, ali nam ona koristi da, kada pronađemo slučaj koji nam je povoljan, prekinemo dalje izvršavanje naredbe `switch`, a program nastavlja sa radom od prve naredne naredbe koja slijedi nakon `switch` bloka. Napomenimo da ćemo naredbu `break` uskoro ponovo pominjati, kada budemo razmatrali načine kako možemo prekinuti izvršenje iterativnih naredbi.

Na slučaj `default` se prelazi ako se nigdje ranije ne desi da je vrijednost izraza `izraz` jednaka nekom od konstantih izraza. Slučaj `default` je opcionalan, što znači da se ne mora navesti. Ako nije naveden, a ranije se nije desilo "poklapanje" vrijednosti izraza `izraz` sa nekim od konstantnih, onda se u okviru `switch` bloka neće izvršiti ništa.

Primjer 3.6. Uradimo sada ponovo, pomoću `switch` naredbe zadatak sa uglovima, iz Primjera 3.5.

```
1  #include<stdio.h>
2
3
4  int main()
5  {
6      int u;
7      printf("Unesite ugao:\n ");
8      scanf("%d",&u);
9
10     u = u % 360;
11
12     switch(u / 90)
13     {
```

Elektronska verzija

```
14          case 0: printf("I kvadrant.\n");break;
15          case 1: printf("II kvadrant.\n");break;
16          case 2: printf("III kvadrant.\n");break;
17          case 3:printf("IV kvadrant.\n");break;
18      }
19
20      return 0;
21
22  }
```

Kao što smo pomenuli, naredba **break** nije obavezna, ali bi njenim izostavljanjem došlo do drugačijeg toka izvršenja programa. Ako izostavimo naredbu **break**, program nastavlja da izvršava i sve ostale naredbe koje slijede nakon ostalih slučajeva, ili dok ne naiđe na naredbu **break** na nekom drugom mjestu, ili dok ne dođe do kraja čitavog bloka.

Primjer 3.7. Posmatrajmo ponovo prethodni primjer, ali u rješenju izostavimo naredbe **break**.

```
1   #include<stdio.h>
2
3
4   int main()
5   {
6       int u;
7       printf("Unesite ugao:\n ");
8       scanf("%d",&u);
9
10      u = u % 360;
11
12      switch(u / 90)
13      {
14          case 0: printf("I kvadrant.\n");
15          case 1: printf("II kvadrant.\n");
16          case 2: printf("III kvadrant.\n");
17          case 3: printf("IV kvadrant.\n");
18      }
19
20      return 0;
21
22  }
```

Ako bismo, na primjer, u program unijeli ugao od 135 stepeni (koji pripada II kvadrantu), na ekranu bismo dobili ispis:

Elektronska verzija

```
Unesite ugao: 135
II kvadrant.
III kvadrant.
IV kvadrant.
```

Pošto je ispunjen slučaj koji je drugi po redu (`case 1`), ispisuje se naredba koja slijedi nakon njega, ali pošto se izvršenje bloka ne prekida, izvršavaju se i sve ostale naredbe koje slijede do kraja bloka.

Uradimo i malo složeniji primjer.

Primjer 3.8. Napisati program koji za unijeti datum sa `d` dana, `m` mjeseci, `g` godina, određuje datum sljedećeg dana.

Ako nije riječ o posljednjem danu u mjesecu, datum sljedećeg dana je naredni dan u istom mjesecu. Ako jeste posljednji dan u mjesecu, naredni dan je prvi dan sljedećeg mjeseca, a ako je riječ o posljednjem danu u posljednjem mjesecu (tj. 31. decembru), naredni dan je 1. januar sljedeće godine. Dakle, u zadatku nam je potrebno i da na osnovu mjeseca odredimo broj dana u tom mjesecu. To nije teško, osim za mjesec februar, čiji broj dana zavisi od toga da li je riječ o prestupnoj godini ili ne. Podsjetimo se da je godina prestupna, ako je djeljiva sa četiri, uz dodatni uslov, da ako je djeljiva sa 100, tada mora biti djeljiva i sa 400. Na primjer, 2000. godina je bila prestupna (djeljiva je sa 4, a djeljiva je i sa 400), dok godina 1900. godina nije bila prestupna. Djeljiva je sa 4, ali je djeljiva i sa 100, dok nije djeljiva sa 400.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int d, m, g, br_dana;
6      printf("Unesite redom dan, mjesec i godinu.\n");
7      scanf("%d%d%d",&d,&m,&g);
8      //najprije odredimo da li je godina prestupna
9      int prestupna = (g % 4 == 0) && (g % 100 != 0 || g % 400 == 0);
10
11     switch(m)
12     {
13         case 1: case 3: case 5: case 7: case 8: case 10: case 12:
               br_dana = 31; break;
14         case 4: case 6: case 9: case 11: br_dana = 30; break;
15         case 2: if(prestupna) br_dana = 29; else br_dana = 28; break;
16     }
17     if(d < br_dana)
```

Elektronska verzija

```
18          d++; //nije posljednji u mjesecu
19      else{
20          d = 1; // sad je prvi datum u mjesecu
21          m++; // prebacimo se i na sljedeci mjesec
22          if(m == 13) //ako je bio decembar
23          {
24              m = 1; //prebacimo se na januar
25              g++; //povecamo godinu
26          }
27      }
28      printf("Datum sljedeceg dana je: %d.%d.%d.\n",d,m,g);
29      return 0;
30  }
```

Primijetimo da u ovom primjeru nemamo naredbu **break** nakon svakog slučaja. Zapravo, mi ovdje koristimo mehanizam da se, ako je ispunjen neki slučaj, izvršavaju naredbe koje slijede nakon tog slučaja, ali i sve ostale naredbe, do prve naredbe **break**. Ako je, na primjer, za promjenljivu `m` unesena vrijednsot 1, već će prvi slučaj biti ispunjen (**case** 1). Izvršavaju se naredbe koje slijede uz taj slučaj (kojih doduše nema!), ali i sve naredne naredbe do prve naredbe **break**. Odnosno, nama je važno da se, ako je `m` jednako 1, izvrši naredba `br_dana=31;`. Drugim riječima, koristeći osobine **switch** naredbe, izbjegli smo pisanje dužeg kôda, koji bi, uz upotrebe naredbe **break** za svaki slučaj, izgledao ovako:

```
    ...
    switch(m)
    {
        case 1: br_dana = 31; break;
        case 3: br_dana = 31; break;
        case 5: br_dana = 31; break;
        case 7: br_dana = 31; break;
        case 8: br_dana = 31; break;
        case 10: br_dana = 31; break;
        case 12: br_dana = 31; break;
        case 4: br_dana = 30; break;
        case 6: br_dana = 30; break;
        case 9: br_dana = 30; break;
        case 11: br_dana = 30; break;
        case 2: if(prestupna) br_dana = 29; else br_dana = 28; break;
    }
    ...
```

Elektronska verzija

3.4 Naredbe ponavljanja

U većini programa se javljaju situacije kada je jednu ili više naredbi potrebno izvršiti više puta. Ako je iste naredbe potrebno izvršiti relativno mali broj puta (na primjer dva ili tri puta), onda se u programu mogu koristiti linijske strukture, tako što bi se naredbe koje treba ponavljati u izvornom kôdu napisale veći broj puta. Na primjer, u zadatku koji bi počinjao tekstom “Sa tastature se unose tri broja...”, bi se mogle koristiti tri promjenljive `a`, `b` i `c`, te bi se naredba `scanf` mogla pozvati tri puta uzastopno. Međutim, ako bi zadatak počinjao tekstom “Sa tastature se unosi 40 brojeva...”, bilo bi besmisleno definisati 40 promjenljivih i pisati 40 naredbi `scanf` jednu za drugom. Pored toga, česte su situacije kada se unaprijed i ne zna koliko puta neke naredbe treba ponavljati, jer broj ponavljanja naredbi može da bude i promjenljiv i da zavisi od vrijednosti koje se izračunavaju u toku samog izvršenja programa. U takvim slučajevima ne možemo iskoristiti linijsku strukturu, već trebamo uvesti takozvane ciklične strukture (petlje). Ciklične strukture omogućavaju izvršavanje jedne ili više naredbi veći broj puta, pri čemu broj ponavljanja zavisi od uslova kojim se određuje kriterijum za prekid izvršenja.

Primjer 3.9. Evo nekih jednostavnih karakterističnih primjera koji podrazumijevaju upotrebu cikličnih struktura.

- Sa tastature se unosi 20 brojeva, odrediti njihov zbir.
- Sa tastature se unose brojevi dok njihov zbir ne pređe 100.
- Sa tastature se unosi broj `n`, a potom još i `n` brojeva, odrediti koliko je među njima parnih.
- Ispisati sve djelioce broja koji se unosi sa tastature.
- Ispitati da li je broj koji se unosi sa tastature prost.
- Odrediti koliko je u niski karaktera malih slova.
- Za nisku karaktera koja se sastoji samo od cifara odrediti njenu brojnu vrijednost.

U programerskom rječniku u našem jeziku postoji nekoliko izraza za naredbe ponavljanja. Izraz ciklična struktura ili petlja, zapravo, potiče od engleske riječi *loop* (što znači upravo petlja). Zbog samog načina funkcionisanja, ove naredbe nazivamo i naredbe ponavljanja ili “iterativne naredbe”, gdje opet koristimo englesku riječ *iterative*, što u ovom kontekstu znači upravo ponavljanje

Elektronska verzija

naredbi. U okviru procesa ponavljanja naredbi (iterativnog procesa), jedan ciklus izvršenja naredbi nakon kog se ponovo ispituje uslov da li će se naredbe opet ponavljati ili ne, zovemo iteracija. Tako ćemo reći "u prvoj iteraciji", "u narednoj iteraciji", "u petoj iteraciji", "u posljednjoj iteraciji" itd.

U programskom jeziku C raspolažemo sa tri tipa naredbi ponavljanja:

- naredba ponavljanja `while`;
- naredba ponavaljanja `for`;
- naredba ponavljanja `do - while`.

3.4.1 Naredba ponavljanja while

Naredbu ponavljanja `while`, ili kraće `while` – petlju koristimo u sljedećoj sintaksi

```
while(uslov)
blok naredbi koje se ponavljaju
```

Blok naredbi koje se ponavljaju čini tijelo petlje i može da sadrži samo jednu naredbu (tada se ona može ili ne mora stavljati u vitičaste zagrade), ili više naredbi i u tom slučaju se one pišu u okviru vitičastih zagrada, čineći tako jednu složenu naredbu.

U okviru `while` petlje prvo se ispituje vrijednost izraza `uslov`. Ako taj izraz ima vrijednost tačno, tj. vrijednost izraza uslov je različita od nule, izvršavaju se naredbe koje se ponavljaju (pojedinačna naredba ili više narebi u bloku). Nakon toga, vrijednost izraza `uslov` se ponovo provjerava i u slučaju da mu je istinitosna vrijednost ostala tačno, naredbe u tijelu petlje se opet izvršavaju. Ovaj iterativni proces (provjera uslova i izvršenje naredbi) se ponavlja sve dok istinitosna vrijednost uslova ne postane netačno (dok vrijednost izraza uslov ne dobije vrijednost nula). Tada se `while` petlja završava i program nastavlja sa radom od prve naredne naredbe koja slijedi poslije `while` petlje.

Kroz nekoliko jednostavnih primjera ilustrujmo rad `while` petlje.

Primjer 3.10. Napisati program koji sabira sve brojeve sa ulaza, dok se ne unese 0.

```
1 #include<stdio.h>
2 int main()
3 {
4    int n;
```

Elektronska verzija

```
5       int zbir = 0;
6       scanf("%d",&n);
7
8       while(n != 0){
9
10          zbir += n;
11          scanf("%d",&n);
12      }
13
14      printf("Zbir je: %d.\n",zbir);
15
16      return 0;
17  }
```

Primjer 3.11. Napisati program koji unijeti broj `u` sa tastature transformiše tako sto mu uklanja nule sa desne strane. Npr. 1200 se transformiše u 12.

```
1   #include<stdio.h>
2   int main()
3   {
4       printf("Unesite broj:\n ");
5       int u;
6       scanf("%d",&u);
7       while(u % 10 == 0)
8           u = u / 10;
9
10      printf("Transformisan broj je %d\n",u);
11      return 0;
12  }
```

Primjer 3.12. Napisati program za izračunavanje i štampanje stepena promjenljive x, koja se unosi sa tastature, počev od x^2, x^4, x^8, x^{16}...sve dok stepen od x ne dobije vrijednost veću od 10^8. Štampati i posljednji stepen od x koji obezbjeđuje izlaz iz ciklusa. Pretpostavka je $x > 1$.

```
1   #include <stdio.h>
2   #include <math.h>
3
4   int main()
5   {
6       int x;
7       printf("Unesite cijeli broj:\n");
```

Elektronska verzija

```
8       scanf("%d",&x);
9
10      int d = 2;
11
12      while(pow(x,d) <= pow(10,8)){
13          float rez = pow(x,d);
14          printf("%f\n",rez);
15          d *= 2;
16      }
17      printf("Izlaz iz ciklusa: %f\n", pow(x,d));
18
19      return 0;
20  }
```

3.4.2 Naredba ponavljanja for

Naredba ponavljanja **for** (ili kraće **for** petlja) je malo složenije sintakse u odnosu na **while** petlju:

```
for(inicijalizacija; uslov; korak)
naredbe koje se ponavljaju
```

Slično kao i kod **while** petlje, blok naredbi (tijelo **for** petlje) koje se ponavljaju može da sadrži samo jednu naredbu ili više naredbi. Pravila grupisanja u zagrade su ista kao i kod **while** petlje.

Kako sama riječ kaže, izraz `inicijalizacija` je naredba koja se najčešće koristi za inicijalizaciju nekih promjenljivih koje se koriste u tijelu petlje. Izraz `uslov` ima istu ulogu kao i kod **while** petlje, tj. odluka da li će se naredbe iz tijela petlje izvršavati ili ne zavisi od istinitosne vrijednosti izraza `uslov`. Naredba korak obično služi za izmjene vrijednosti promjenljivih kojima se kontroliše rad same petlje. Redoslijed izvršavanja naredbi u **for** se može prikazati na sljedeći način:

1. Izvršava se inicijalizacija
2. Izračunava se `uslov`
3. Ako nije ispunjen uslov, prelazi se na korak 7. Ako je uslov ispunjen, ide se na sljedeću stavku 4
4. Izvršavaju se naredbe koje se ponavljaju
5. Izvršava se naredba `korak`

Elektronska verzija

6. Povratak se na stavku 2
7. Kraj

Kao što vidimo, na početku `for` petlje vrši se inicijalizacija (stavka 1). Nakon toga se ispituje uslov (stavka 2). Ukoliko je vrijednost izraza `uslov` tačna, izvršavaju se naredbe koje čine tijelo petlje (stavka 4). Ako vrijednost uslova nije tačna, ide se na stavku 7 (Kraj) i program nastavlja sa radom od prve naredne naredbe koja slijedi poslije `for` petlje. Ako je uslov bio tačan, nakon stavke 4, tj. izvršenja naredbi iz tijela `for` petlje, izvršava se naredba `korak` (stavka 5), nakon čega se program ponovo vraća na stavku 2, tj. ispitivanje uslova.

Jednostavnijim riječima, nakon što se jednom izvrši inicijalizacija, program ulazi u petlju `uslov - naredbe - korak` i ponovo `uslov`, sve kod `uslov` ne postane netačan.

Petlju `for` obično koristimo kada prije početka izvršenja petlje znamo koliko puta ćemo ponavljati iteracije, tj. kada imamo mogućnost da "brojimo" iteracije. Na primjer, ako znamo da se iteracije trebaju ponavljati n puta, "brojanje" iteracija realizujemo na sljedeći način: u naredbi inicijalizacija postavimo vrijednost neke promjenljive (recimo promjenljive i) na neku početnu vrijednost (na primjer na 0), izraz uslov definišemo tako je on tačan sve dok je i manje od n, dok po završetku svake iteracije, u okviru naredbe korak, vrijednost promjenljive i uvećavamo za jedan. U programu bi to izgledalo ovako

```
int i, n;
...
for (i = 0; i < n; i++)
naredbe koje se ponavljaju
...
```

Na taj način realizujemo brojanje: za $i = 0$ imamo početnu iteraciju, za $i = 1$ drugu, za $i = 2$ treću itd. Primijetimo da smo brojanje n iteracija realizovali po principu brojanja od nule, tj: $0, 1, 2, ..., n - 1$. Naravno, brojanje n iteracija se može realizovati i počev od 1 ($1, 2, 3$ do n), ili na primjer unazad, (brojanjem od n do 1). U programu bi brojanje unazad izgledalo ovako

```
int i, n;
...
for (i = n; i >= 1; i--)
naredbe koje se ponavljaju
...
```

Elektronska verzija

Primjer 3.13. Sa tastature se unosi broj `n`. Izračunati zbir prvih `n` cijelih pozitivnih brojeva.

Ovaj zadatak realizujemo kao klasično brojanje, gdje u `i`–tom koraku (gdje se `i` kreće od 1 do `n`), na trenutni zbir sabiramo broj `i`.

```
#include <stdio.h>
int main(){
   int i, n;
   int suma = 0;
   printf("Unesite koliko brojeva se sabira:");
   scanf("%d",&n);
   for(i = 1; i <= n; i++)
      suma += i;
   printf("Zbir prvih %d brojeva je %d\n",n,suma);
   return 0;

}
```

Sljedećim primjerom ilustrujmo da u okviru naredbe **`for`** ne moramo da imamo klasično brojanje.

Primjer 3.14. Napisati program kojim se računa vrijednost funkcije $f(x) = 2x^2 + 3$ koja uzima realne vrijednosti na zatvorenom intervalu od a do b (koji se unose sa tastature) sa korakom 0.5.

```
#include <stdio.h>

int main()
{
    float a,b;
    printf("Unesite brojeve a i b:");
    scanf("%f %f",&a,&b);

    float x;

    for(x = a; x <= b; x += 0.5){
        float f = 2 * x * x + 3;
        printf("f(%.2f)=%.2f\n",x,f);
    }

    return 0;
}
```

Elektronska verzija

3.4.3 Naredba ponavljanja do-while

Sintaksa **do-while** petlje izgleda ovako

```
do{
   blok naredbi koje se ponavljaju
}while(uslov)
```

Bez obzira na to da li je u tijelu petlje jedna naredba ili blok od više naredbi, vitičaste zagrade su obavezne. Nakon što program naiđe na naredbu **do**, izvršavaju se naredbe koje slijede u vitičastim zagradama. Zatim slijedi provjera istinitosti uslova. Ukoliko je vrijednost uslova tačno, ciklus se ponavlja tako što program ponovo izvršava naredbe u tijelu petlje.

Kao što vidimo, za razliku od **while** i **for** petlji, kod **do - while** petlje se istinitosna vrijednost uslova provjerava poslije izvršenja naredbi u tijelu same petlje. Iz toga možemo zaključiti da će se, ako program naiđe na naredbu **do**, naredbe koje slijede izvršiti najmanje jednom. To nije slučaj sa druge dvije petlje, kod kojih je moguće da se naredbe nikada ne izvrše, ukoliko je uslov u startu netačan.

Ova činjenica u nekim situacijama može biti korisna. Ako programer zna da se neki blok naredbi u iterativnom procesu mora izvršiti najmanje jednom, upotreba **do-while** petlje je opravdana. Sa druge strane, takve situacije nisu toliko česte u odnosu na sve druge u kojima se javlja potreba za naredbama ponavljanja. Samim tim, možemo zaključiti da se u redovnoj programerskoj praksi naredba **do - while** ipak koristi rjeđe nego preostale dvije naredbe ponavljanja.

Uradimo nekoliko primjera.

Primjer 3.15. Napisati program kojim se određuje broj jedinica u binarnom zapisu prirodnog broja n, koji se unosi sa tastature.

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

Elektronska verzija

```
13          int i = n % 2;
14          if(i == 1)
15              brojac++;
16          n /= 2;
17      }while(n > 0);
18
19      printf("Jedinica u unesenom broju ima %d.\n",brojac);
20
21      return 0;
22  }
```

Primjer 3.16. Napisati program kojim se među brojevima: $1,\ 1+\frac{1}{2},\ 1+\frac{1}{2}+\frac{1}{3}, \ldots$ pronalazi prvi veći od broja a koji se unosi sa tastature.

```
1
2   #include <stdio.h>
3
4   int main()
5   {
6       float a;
7       printf("Unesite broj a:\n");
8       scanf("%f",&a);
9
10      float suma = 0;
11      float i = 0;
12
13      do{
14          i++;
15          suma += 1 / i;
16      }while (suma < a);
17
18      printf("Trazeni broj je %f.\n",suma);
19
20      return 0;
21  }
```

3.4.4 Naredbe break i continue

Naredbu **break** koristimo za trenutni prekid izvršavanja čitave petlje. Preciznije, nakon što se izvrši naredba **break**, prekida se izvršavanje naredbi koje se nalaze u tijelu petlje, a program nastavlja sa radom od prve naredne naredbe koja slijedi nakon petlje.

Elektronska verzija

Naredbu `continue` koristimo za bezuslovni prekid izvršavanja trenutne iteracije, dok program nastavlja sa radom sa narednom iteracijom. Praktično, kada program u okviru tijela petlje izvrši naredbu `continue`, sve naredbe koje slijede u nastavku tijela petlje se preskaču i program nastavlja sa izvršenjem naredne iteracije ukoliko su zadovoljeni uslovi za ulazak u nju.

Ovdje pomenimo još i naredbu `goto`, koja omogućava bezuslovni skok sa jednog dijela programa na neki drugi. Stariji programski jezici, nastali pod uticajem asemblerskih jezika, intenzivno su koristili ovaj mehanizam "skakanja" po programu, što sa sintaksne strane nije pogrešno, ali dovodi do težeg razumijevanja samog kôda. Zbog toga su se još šezdesetih i sedamdesetih godina dvadesetog vijeka javile inicijative i preporuke da se izbjegava upotreba ove naredbe skoka. Pored toga, na osnovu teoreme o strukturnom programiranju iz 1966. godine, (koja se po svojim autorima zove i Böhm-Jacopini teorema), važi da se svaki program može zamijeniti ekvivalentnim programom koji kombinuje sekvencijalni niz naredbi, naredbu izbora (`if - else`) i neku od naredbi ponavljanja (petlji). Na osnovu ovog rezultata, a i zdravorazumskog rezonovanja koje je u programiranju često, naredba skoka `goto` je praktično izbačena iz programerske prakse.

Treba napomenuti da se i naredbama `break` i `continue` takođe narušava strukturianost koda, što, takođe, može da dovede do otežanog razumijevanja. Međutim, za razliku od naredbe `goto`, ove dvije naredbe ne dozvoljavaju da se napravi "veliki" skok u programu, već se pomoću njih samo izlazi iz petlje, odnosno trenutne iteracije. Uz činjenicu da upotreba naredbi `break` i `continue` nekada može da dovede i do bolje čitljivosti koda, možemo zaključiti da upotreba ovih naredbi u nekim situacijama može da bude opravdana. Ipak, treba imati na umu da se izlazak iz petlje (ili iz jedne njene iteracije) uvijek može realizovati i bez upotrebi naredbi `break` i `continue`.

U narednim primjerima ilustrujemo upotrebu naredbi `break` i `continue`.

Primjer 3.17. Sa tastature se unosi broj, ispisati da li on sadrži cifru 5.

U svakoj iteraciji `while` petlje ćemo, krenuvši sa desne strane ka lijevoj, pristupati pojedinačnim ciframa broja. Ako naiđemo na cifru 5, nema potrebe da dalje išta dodatno ispitujemo i iterativni proces možemo prekinuti. Na kraju treba razumjeti na koji način ćemo moći da zaključimo šta je uzrokovalo prekid petlje.

```
1  #include <stdio.h>
2
3  int main(){
4     int a, b;
```

Elektronska verzija

```
   printf("Unesi broj:");
   scanf("%d",&a);
   b = a;
   while(b > 0){
      int cifra = b % 10;
      if(cifra == 5) //pronasli smo cifru, nema potrebe da dalje ista ispitujemo
         break;
      b /= 10; //nastavljamo dalje tako sto "skidamo" desnu cifru
   }
   if(b > 0) //iz while petlje smo izasli naredbom break
      printf("Broj %d sadrzi cifru 5\n",a);
   else
      printf("Broj %d ne sadrzi cifru 5\n",a);
   return 0;
}
```

Uradimo ovaj isti zadatak bez naredbe **`break`**. Kontrolu prekida **`while`** petlje vršimo preko dva uslova. Prvi je isti kao i u prethodnom primjeru, odnosno petlju sigurno moramo prekinuti ako "potrošimo" sve cifre broja. Pomoću drugog uslova, koji je kontrolisan promjenljivom `pronasli`, rješavamo dvije stvari. Prvo, omogućavamo da se **`while`** petlja prekine po završetku one iteracije u kojoj je pronađena cifra 5 (to je u prethodnom primjeru bilo realizovano naredbom **`break`**) i drugo, po izlasku iz petlje, u toj promjenljivom imamo informaciju da li je cifra 5 uopšte pronađena ili ne.

```
#include <stdio.h>

int main(){
   int a, b;
   printf("Unesi broj:");
   scanf("%d",&a);
   b = a;
   int pronasli = 0; //za sada smatramo da nismo nasli cifru
   while(b > 0 && !pronasli) //kad promj. pronasli postane 1 while petlja se zaustavlja
   {
      int cifra = b % 10;
      if(cifra == 5)
         pronasli = 1;//pronasli smo cifru
      b /= 10;//nastavljamo dalje tako sto "skidamo" desnu cifru
   }
   if(pronasli)
      printf("Broj %d sadrzi cifru 5\n",a);
```

Elektronska verzija

```
    else
       printf("Broj %d ne sadrzi cifru 5\n",a);
    return 0;
}
```

Uradimo još jedan duži primjer, u kome kombinujemo razne tehnike koje smo do sada koristili.

Primjer 3.18. Sa tastature se unosi broj n, a potom se korisniku omogućava unos n karaktera sa tastature, ali ako se unese karakter `'*'` dalji unos se prekida. Program kao rezultat vraća informaciju o tome koliko je uneseno velikih slova, malih slova, cifara i ostalih karaktera. Takođe, ispisati poruku da li je došlo do prekida (unosom karaktera `'*'`) ili je uneseno svih n karaktera.

```
#include <stdio.h>

int main(){
    int n;

    printf("Unesite broj n:\n");
    scanf("%d",&n);

    char c;
    int brojac = 0;
    int br_malih = 0, br_velikih = 0, br_cifara = 0;

    while(brojac < n){
        printf("Unesite karakter:\n");
        scanf("\n%c",&c);

        if(c == '*') break; //da ne bismo komplikovali, koristimo break za prekid while petlje
        if(c >= 'a' && c <= 'z')
            br_malih++;
        else if(c >= 'A' && c <= 'Z')
                    br_velikih++;
            else if(c >= '0' && c <= '9')
                br_cifara++;

        brojac++;
    }

    int br_ostalih = brojac - (br_malih + br_velikih + br_cifara);

```

Elektronska verzija

```
30      if(brojac == n)
31         printf("Uneseno je svih %d karaktera.\n",n);
32      else
33         printf("Doslo je do prekida\n");
34
35      printf("Malih slova: %d, velikih slova: %d, cifara: %d, ostalih
           karaktera: %d.\n",br_malih, br_velikih, br_cifara, br_ostalih);
36      return 0;
37   }
```

Sljedećim primjerom ilustrujmo upotrebu naredbe `continue`.

Primjer 3.19. Sa tastature se unosi cio broj. Na osnovu njega formirati novi broj kod koga su iz polaznog broja izbačene neparne cifre.

```
1   # include <stdio.h>
2
3   int main (){
4
5       int a, b, cifra, t = 1, novi;
6
7       printf("Unesi prirodan broj: ");
8       scanf("%d", &a);
9
10      b = a;
11
12
13      while (b > 0){
14        cifra = b % 10;
15        b /= 10;
16        if (cifra % 2 == 1)
17             continue;//ako je trenutna cifra neparna, preskacemo je i
                   nastavljamo dalje
18        novi = cifra * t + novi;
19        t = t * 10;
20      }
21
22      printf("Prepravljeni broj: %d", novi);
23
24      return 0;
25  }
```

Elektronska verzija

3.4.5 Beskonačne petlje

Kako i sam izraz kaže, beskonačne petlje su petlje koje se nikada ne završavaju, odnosno, uslov kojim kontrolišemo da li će petlja nastaviti sa radom ili ne je uvijek ispunjen. Pošto se programi koji sadrže beskonačne petlje nikada ne završavaju, smatramo da takvi programi nisu ispravni i samim tim, beskonačne petlje predstavljaju grešku u programu. Ove greške se uglavnom ne mogu otkriti u procesu prevođenja programa, već se one javljaju tek po pokretanju programa.

Posmatrajmo nekoliko tipičnih primjera beskonačnih petlji, koje uglavnom nastaju usljed grešaka u pisanju programa.

Primjer 3.20. Greška zbog stavljanja znaka `;` nakon zatvorene male zagrade u `while` petlji. U ovom slučaju, program smatra da se prazna naredba koja se formalno nalazi prije tog znaka `;` ponavlja dok je ispunjen uslov. Programer je vjerovatno htio da izrazom `i++` promjenljivu `i` poveća dovoljan broj puta da se `while` petlja završi kada `i` dostigne vrijednost `n`, ali do tog dijela kôda program uopšte i ne stiže, već ostaje "zaglavljen" u beskonačnoj `while` petlji.

```
int i, n;
...
i = 0;
while(i < n);//pogresno, ne treba ;
{
    ...
    i++;
}
...
```

Primjer 3.21. Slična situacija se može desiti i kod `for` petlje. Na primjer, programer je odlučio da mu brojač kreće od neke gornje vrijednosti i da se spušta do nule, ali je zaboravio da umjesto inkrementiranja brojača uključi dekrementiranje.

```
...
int i, n;
for (i = n; i >= 0; i++) //pogresno, promjenljiva i se nece spustiti
    ispod nule
{
    ...
}
```

Elektronska verzija

Pojava beskonačne petlje može da nastane i zbog pogrešnog unosa podatka, o kome programer nije vodio računa. Posmatrajmo sljedeći primjer.

Primjer 3.22. Napisati program koji za unesene brojeve `a` i `b` ispisuje sve brojeve između njih.

Sljedeći program ispravno radi u slučaju da je `a` manje od `b`.

```
1  #include <stdio.h>
2  int main()
3  {
4     int a, b;
5     scanf("%d %d",&a,&b);
6      while(a != b-1)
7      {
8          a++;
9          printf("%d\n",a);
10     }
11     return 0;
12 }
```

Međutim, u slučaju da su uneseni brojevi takvi da je `b` manje od `a`, tada uslov **`while`** petlje nikada neće postati netačan, odnosno, pošto se `a` stalno uvećava, nikada neće postati jednako `b-1`. Zadatak bi se mogao ispravno riješiti tako da se dozvoli ulazak u **`while`** petlju, samo ako je `a` manje od `b-1` (a ne različito, kao u prethodnom primjeru):

```
...
while(a < b-1)
{
...
}
```

Nekada programeri, a posebno onda kada je potrebno napraviti brzo rješenje, koriste pristup da je uslov kojim se reguliše ulazak, odnosno izlazak iz petlje uvijek ispunjen, dok se petlja prekida naredbom **`break`**. Ovakav pristup je sintaksno dozvoljen, ali se, po nekim neformalnim konvencijama, on ne preporučuje.

Primjer 3.23. Odrediti proizvod pozitivnih brojeva koji se unose sa tastature. Nule ignorisati, a na unos negativnog broja prekinuti dalji tok unosa.

Zadatak možemo uraditi na nekoliko načina, a ovdje prikažimo pristup u kome je uslov za izlazak iz petlje uvijek tačan. Petlju prekidamo naredbom **`break`**.

Elektronska verzija

```
1  #include <stdio.h>
2  int main()
3  {
4     int a, proiz = 1;
5      while(1)//za sada ne mislimo nista o uslovu
6      {
7          scanf("%d",&a);
8        if(a == 0)//ignorisemo unos nula
9            continue;
10       if (a < 0)//ovdje aktiviramo break
11           break;
12       proiz *= a;
13     }
14     printf("Proizvod: %d\n",proiz);
15     return 0;
16 }
```

3.4.6 Ugniježdene petlje

U programerskoj praksi su česte i situacije kada koristimo “petlju unutar petlje”, ili tzv. ugniježdene petlje. Tu najčešće podrazumijevamo da se unutar jedne iteracije jedne petlje (koju zovemo i vanjskom petljom), izvrše sve iteracije druge (unutrašnje) petlje. Rad sa ugniježdenim petljama ne zahtijeva poznavanje nekih novih tehnika programiranja, već se vještina u radu sa njima najviše stiče vježbom. Treba imati u vidu da rad sa ugnježdenim petljama može značajno da uspori izvršenje programa, jer se ukupan broj koraka (naredbi) koje program treba da izvrši na ovaj način može značajno povećati. Takođe, ako radimo sa ugnježdenim petljama, treba izbjegavati naredbe **`break`** i **`continue`**, jer može doći do konfuzije na koju iterativnu naredbu se odnosi naredba skoka i koja iteracija se prekida, a koja ne.

Krenimo od jednog od najjednostavnijih primjera.

Primjer 3.24. Na ekranu ispisati tablicu množenja brojeva do 20.

```
1  #include <stdio.h>
2  int main()
3  {
4     int i, j;
5     for (i = 1; i <= 20; i++){
6        for (j = 1; j <= 20; j++)
7           printf("%4d",i*j);
```

Elektronska verzija

```
8       printf("\n");
9     }
10    return 0;
11 }
```

U ovom jednostavnom primjeru smo morali voditi računa o "poravnanju" ispisa, te smo zbog toga za svaki broj zauzeli 4 mjesta (to smo u izrazu za ispis regulisali pomoću zapisa `%4d`). O formatiranju ispisa će biti puno više riječi u Poglavlju 5.

Uradimo još nekoliko primjera sa ugniježdenim petljama.

Primjer 3.25. Napisati program koji za uneseni broj n ispisuje sve proste brojeve koji nisu veći od n.

Ispitivanje da li je broj prost ćemo realizovati na najjednostavniji način. Za posmatrani broj i, za koji ispitujemo da li je prost ili ne, u unutrašnjoj petlji pokušavamo da pronađemo njegovog djelioca većeg od 1, a manjeg od $i/2$. Ako takvog pronađemo, broj i nije prost i unutrašnju petlju možemo da završimo i prelazimo na sljedeći broj. Ako ne pronađemo nijednog djelioca iz posmatranog intervala $[2, i/2]$, zaključujemo da je broj i prost i ispisujemo ga na ekranu.

```
1  #include <stdio.h>
2  int main()
3  {
4     int i,n,djelilac;
5     int prost;
6     printf("Unesi broj n:");
7     scanf("%d",&n);
8
9     for(i = 2; i <= n; i++)
10    {
11       prost = 1; //pretpostavimo da je i prost, a ako zakljucimo da
             nije, promijenicemo na 0
12       for(djelilac = 2; djelilac <= i/2 && prost; djelilac++)
13          if (i % djelilac == 0)//pronasli smo djelioca, i nije prost
14             prost = 0;
15       if(prost)
16          printf("%d\n",i);
17    }
18    return 0;
19 }
```

Primjer 3.26. Broj je savršen, ako je zbir njegovih djelilaca jednak njegovoj dvostrukoj vrijednosti. Najmanji savršen broj je 6. Njegovi djelioci su 1, 2, 3 i

Elektronska verzija

6, a važi $1 + 2 + 3 + 6 = 12 = 2 \cdot 6$. Sljedeći savršen broj je 28. Ako saberemo sve djelioce broja 28 dobićemo $1 + 2 + 4 + 7 + 14 + 28 = 56 = 2 \cdot 28$. U ovom primjeru ćemo napisati program koji na ekranu ispisuje sve savršene brojeve koji su manji od 10000.

Zadatak ćemo riješiti na najjednostavniji, ali ne i na najefikasniji način. Za svaki broj koji je manji od 10000 ćemo, bez dodatne analize i preskakanja nepotrebnih slučajeva, odrediti zbir njegovih djelilaca i na kraju uporediti sa njegovom dvostrukom vrijednošću. Unutrašnju petlju nećemo prekidati nijednim dodatnim uslovom, kojim bismo eventualno ubrzali rad.

```
1  #include <stdio.h>
2  int main()
3  {
4     int i, djelilac;
5     int suma;
6     for(i = 2; i <= 10000; i++)
7     {
8        suma = 0; //za svako i racunamo zbir djelilaca
9        for(djelilac = 1; djelilac <= i; djelilac++)
10          if (i % djelilac == 0)//pronasli smo djelioca
11             suma += djelilac;
12       if(suma == 2 * i)
13          printf("%d\n",i);
14    }
15    return 0;
16 }
```

3.5 Pitanja i zadaci

1. Dati su podaci `int s1, m1, s2, m2`, koji redom predstavljaju sate i minute dva događaja. Odrediti da li se prije desio dogadjaj u `s1` sati i `m1` minuta ili događaj u `s2` sati i `m2` minuta, a nakon toga odrediti koliko je sati i minuta prošlo izmedju dva događaja. Sati uzimaju vrijednosti između 0 i 23, minute vrijednosti od 0 do 59.

2. Napisati program koji ugao dat u sekundama izražava u stepenima, minutama i sekundama.

3. Koliku vrijednost će imati promjenljiva `a` nakon izvršenja narednog dijela kôda, ako promjenljiva `x` ima vrijednost 2?

Elektronska verzija

```
int x,a;
if(x == 1) a = 2;
if(x == 2) a = 4;
if(x == 3) a = 6;
else a = 10;
```

4. Napraviti izmjene u kôdu iz prethodnog zadatka takve da se nakon izvršenja programa važi sljedeće: ako `x` ima vrijednost 1, 2 ili 3 onda se promjenljivoj `a` dodjeljuje dvostruka vrijednost promjenljive `x`, a u suprotnom vrijednost 10.

5. Napisati program koji računa vrijednost promjenljive `y` na sljedeći način:
   - ako je `x < -5` onda je `y = x * x * 2.0`
   - ako je `-5 < x < 0` onda je `y = x + 1.0`
   - ako je `x == 0` onda je `y = 0.0`
   - ako je `x > 0` onda je `y = sqrt(x)`

   Promjenljiva `x` je tipa **`int`**. Koji tip podatka je odgovarajući za promjenljivu `y` i zašto?

6. Sljedeći dio kôda napisati pomocu **`if-else`** naredbe:

```
int x, y;
switch(x){
    case 10: y = x * 2; break;
    case 100: y = x / 2; break;
    case 1000: y = x + x; break;
    default: y = x;
}
```

7. Kockica za igru "Čovječe ne ljuti se" se baca 15 puta, odnosno 15 puta se unosi sa tastature cijeli broj iz intervala [1,6]. Odrediti koliko puta je pala jednica, dvojka, trojka, četvroka, petica i šestica. Ako se unese broj koji nije cijeli broj iz intervala [1,6] ispisuje se poruka na ekranu da unos nije odgovarajući i taj unos se ne uzima u obzir, već se unos broja ponavlja.

8. Predstaviti **`for`** petlju ekvivalentnom **`while`** petljom i ekvivalentnom **`do-while`** petljom.

9. Predstaviti **`do-while`** petlju ekvivalentnom **`for`** petljom i ekvivalentnom **`while`** petljom.

Elektronska verzija

10. Predstaviti **while** petlju ekvivalentnom **for** petljom i ekvivalentnom **do-while** petljom.

11. Dat je sljedeći kôd:

```
int x = 2, y = 5;
while(x * y < 65){
    x++;
    y += x;
}
```

Koliko puta će se izvrsiti blok naredbi i kolike će biti vrijednosti promjenljivih `x` i `y` nakon izlaska iz petlje?

12. Koliku vrijednosti ima promjenljiva `i` nakon izlaska iz date petlje?

```
int i;
for(i = 5; i > 2; i++)
    i--;
```

13. Sa tastaure se unose cijene proizvoda neke fabrike, dok se ne unese nula. Napisati program koji određuje najmanju i najveću unesenu cijenu, kao i redne brojeve proizvoda sa najmanjom i najvećom cijenom. Redni brojevi se dodjeljuju proizvodima po redoslijedu unosa.

14. Kolika će biti vrijednost promjenljive `a` nakon izvršenja sljedećeg dijela kôda?

```
int a = 12;
do{
    a++;
}while(a > 15);
```

A kolika nakon kôda?

```
int a = 12;
while(a > 15){
    a++;
}
```

Uporediti dobijene rezultate.

15. Sljedeći dio kôda izmijeniti tako da ne sadrži naredbu **break**.

Elektronska verzija

```
int suma = 0;
int i;
for(i = 0; i < 100; i += 2){
    if(i == 50) break;
    suma += i;
}
```

16. Napisati program koji sabira sve cijele neparne brojeve manje od 99, ali tako da u traženu sumu ne ulazi broj `x` koji je manji od 99 i koji se unosi sa tastature.

17. Po čemu se razlikuju naredbe **`break`** i **`continue`**? Ilustrovati primjerom.

18. Uporediti izvršenja naredna dva kôda. Da li postoji razlog zbog kojeg u kôdu broj 2 nismo u posljednjoj naredbi ispisa ispisali vrijednost promjenljive `j`?

    Kôd 1:

```
1  #include<stdio.h>
2  int main(){
3      int i;
4      int j = 5;
5      for(i = 0; i < 10; i++){
6          while(j < 12){
7              j += 2;
8              i++;
9              printf("Unutar while petlje i=%d, j=%d\n",i, j);
10         }
11         printf("Dio for petlje i=%d, j=%d\n",i, j);
12     }
13     printf("Na kraju: i=%d, j=%d\n", i, j);
14     return 0;
15 }
```

    Kôd 2:

```
1  #include<stdio.h>
2  int main(){
3      int i;
4      for(i = 0; i < 10; i++){
5          int j = 5;
6          while(j < 12){
7              j += 2;
```

Elektronska verzija

```
8            i++;
9            printf("Unutar while petlje i=%d, j=%d\n",i, j);
10       }
11       printf("Dio for petlje i=%d, j=%d\n",i, j);
12   }
13   printf("Na kraju: i=%d\n", i);
14   return 0;
15 }
```

19. Šta se ispisuje datim kôdom na ekranu?

```
int i,j;
for(i = 0; i < 10; i++)
    for(j = 2; j < 10; j++){
        if((i + j) % 2 == 0)
            printf("i=%d, j=%d\n",i,j);
    }
```

20. Uraditi Primjer 3.23 tako da ne sadrži naredbu **break**.

21. Napisati program koji za dva cijela broja određuje koliko imaju zajedničkih djelilaca.

22. Napisati program koji učitava realne brojeve, sve dok se ne unese nula. Nakon toga ispisati na ekranu koliko je puta došlo do promjene znaka. Na primjer, ako je uneseno

$$2.1, 3.6, -7.8, -9.9, -0.2, 3.77, -2.56, 0$$

onda se kao rezultat vraća 3.

23. Broj 36 ima neobično svojstvo. On je potpun kvadrat i jednak je sumi cijelih brojeva od 1 do 8. Sljedeći takav broj je 1225 jer je $1225 = 35^2$ i jednak je sumi cijelih brojeva od 1 do 49. Odrediti prvih N brojeva koji su potpuni kvadrati i za neko I jednaki su zbiru $1 + 2 + ... + I$. Broj N se unosi sa tastature.

Elektronska verzija

4 Funkcije

Funkcija je zaokruženi dio programa (često se naziva i potprogramom), koji može (ali i ne mora) da uzima ulazne podatke, izvršava niz naredbi i vraća rezultat programu iz kog je ta funkcija pozvana. Iako se veliki broj programa može napisati samo upotrebom jedne, glavne funkcije, iskusan programer će se radije opredijeliti da programski kôd izdijeli na veći broj manjih zasebnih dijelova – potprograma, od kojih svaki dio izvršava neki određeni zadatak. Ovakav pristup omogućava lakše upravljanje kôdom, prilagođavanje kôda različitim problemima, kao i bolju razumljivost i čitljivost čitavog programa.

Jednom napisana funkcija može više puta biti iskorištena u okviru jednog programa, ali se, takođe, može koristiti i pozivati u više različitih programa. Zbog toga se često kaže da se upotrebom funkcija dobija mogućnost "ponovnog korištenja" (engl. re-usability) programskog kôda, što najčešće dovodi do vremenskih ušteda u razvoju.

4.1 Deklarisanje i definisanje funkcija

Kada trebamo da pišemo funkcije, vodimo računa o sljedećim elementima: deklaraciji i definiciji funkcije, rezultatu funkcije, argumentima funkcije i načinu poziva funkcije. **Deklaraciju funkcije** shvatamo kao najavljivanje funkcije ili kreiranje prototipa funkcije, gdje navodimo tip podatka koji funkcija vraća kao rezultat, ime funkcije, te broj i tip podataka koji se funkciji predaju kao argumenti. Opšti oblik deklaracije funkcije bi izgledao ovako

```
tipRezultata ime_funkcije(tip1 arg1, ... ,tipN argN);
```

gdje tipRezultata predstavlja tip podatka koji funkcija vraća kao rezultat svog izvršavanja. Nakon tipa rezultata, navodi se ime funkcije. Imena funkcija su identifikatori, te, stoga, i za njih važe ista pravila kao i za imena promjenljivih. Kao što smo i ranije pominjali, poželjno je, ali ne i obavezno, da ime funkcije ukazuje na ono šta funkcija radi.

Kao i kod deklarisanja funkcije, prilikom **definisanja funkcije** navodimo tip rezultata funkcije, ime funkcije i listu argumenata, nakon čega slijedi **tijelo funkcije**, koje se sastoji od jedne ili više naredbi. **Rezultat funkcije** je vri-

Elektronska verzija

jednost koja se izračunava u okviru tijela funkcije i vraća kao rezultat na ono mjesto u programu sa kog je funkcija pozvana. Opšti oblik definicije funkcije izgleda ovako

```
tipPodatka ime_funkcije(tip1 arg1, ... ,tipN argN)
{
    tijelo funkcije
}
```

Pri deklarisanju i definisanju funkcije unutar malih zagrada koja slijedi nakon imena funkcije, navodimo **formalne argumente** ili **parametre** funkcije, najčešće u formi `tip1 arg1, ... ,tipN argN`. Ispred naziva svakog parametra navodi se njegov tip, a parametri se razdvajaju zarezom. Kao i imena promjenljivih, preporučljivo je da imena parametara funkcije oslikavaju njihovo značenje i ulogu u funkciji.

Moguće je da funkcija i ne uzima argumente, ali se i u tom slučaju moraju navesti otvorena i zatvorena mala zagrada, između kojih se može, ali i ne mora navesti riječ `void`. Ako se riječ `void` ne navodi, prevodilac prilikom poziva funkcije ne provjerava da li je ona zaista pozvana bez argumenata.

Tijelo funkcije se navodi unutar vitičastih zagrada koje slijede nakon zatvorene male zagrade.

Funkcija se izvršava tako što je "pozivamo" iz nekog drugog dijela programa. Prilikom poziva funkcije navodi se njen naziv, nakon čega slijedi lista **stvarnih argumenata**. Stvarni argumenti moraju biti navedeni u redoslijedu koji je zadan deklaracijom ili definicijom funkcije, na način da redoslijed tipova stvarnih argumenata odgovara redoslijedu formalnih argumenata.

Na primjer, ako je funkcija `f` deklarisana sa

```
int f(char c, double x, int p);
```

i ako su u programu iz koje se poziva funkcija definisane promjenljive `s` tipa `char`, promjenljiva `p` tipa `int` i realna promjenljiva `y` (tipa `double`), onda bi poziv funkcije `f` mogao da izgleda ovako:

```
vrijednost = f(s,y,p);
```

Vidimo da su stvarni argumenti `s`, `y` i `p` u pozivu funkcije navedeni istim redoslijedom kao i odgovarajući formalni argumenti. Pored toga, primjećujemo da je naziv stvarnog argumenta `p` isti kao i naziv formalnog argumenta, što je dozvoljeno. Primijetimo još iz deklaracije funkcije `f`, da je tip rezultata te funkcije podatak tipa `int`, te je, stoga, pravilno razmišljati da je promjenljiva

Elektronska verzija

`vrijednost` koja “prihvata” rezultat funkcije upravo tog tipa.

Funkcija vraća rezultat svog izvršavanja pomoću naredbe **`return`**. Oblik u kome se naredba **`return`** koristi je prilično jednostavan:

```
return rezultat;
```

gdje je rezultat upravo onog tipa koji je naveden kao tip rezultata funkcije. Po izvršavanju naredbe **`return`** funkcija završava sa svojim izvršavanjem, a vrijednost `rezultat` se vraća onom dijelu programa iz kojeg je pozvana funkcija. Po pravilu, funkcija kao rezultat može vratiti neki od aritmetičkih tipova, strukturu, uniju ili pokazivač.

Ako je tip rezultata u naredbi **`return`** različit od tipa podatka koji funkcija vraća, tada će taj rezultat biti konvertovan u odgovarajući tip podatka (pod uslovom da je to uopšte moguće), ali takve situacije treba izbjegavati.

Moguće je i da funkcija ne treba da vrati ništa kao rezultat. U tom slučaju, za tip vrijednosti koju funkcija vraća koristimo riječ **`void`**, (engl. void znači prazan, te, stoga, i mi nekada kažemo da u tom slučaju funkcija vraća prazan podatak). S obzirom na to da u ovom slučaju funkcija ne vraća ništa kao rezultat (tj. vraća prazan podatak), naredbu **`return`** ne moramo ni da koristimo. Međutim, ako ipak želimo da je koristimo, tada je koristimo bez rezultata, tj. samo **`return;`**

Ukoliko imamo grananje unutar funkcije, moguće je, a nekada i poželjno, da imamo više naredbi **`return`**, gdje iza svake naredbe **`return`** slijedi odgovarajući izraz. Funkcija završava sa radom nakon što izvrši jednu naredbu **`return`** i vrati vrijednost koja slijedi nakon te naredbe.

Posmatrajmo sljedeći primjer funkcije.

Primjer 4.1. Napisati funkciju koja vraća znak broja, tj.: 1 ako je pozitivan, -1 ako je negativan, a 0 ako je broj jednak 0.

```
int znak(int a) {
    if(a > 0)
        return 1;
    else if(a == 0)
            return 0;
         else return -1;
}
```

Ukoliko se kod funkcije koja je definisana tako da vraća neku vrijednost izostavi naredba **`return`**, prevodilac će u tom slučaju prijaviti upozorenje, a rezultat poziva funkcije će biti neodređen. Zbog toga, takve situacije treba izbjegavati.

Elektronska verzija

4.2 Rekurzivne funkcije

Rekurzivne funkcije su funkcije koje u okviru svoje definicije pozivaju tu istu funkciju. Mnoge matematičke funkcije se mogu definisati rekurzivno (na primjer funkcija faktorijel, čuveni Fibonačijev niz, stepena funkcija itd.). Pored toga, u programiranju (i računarstvu uopšte) su neke od veoma značajnih struktura definisane rekurzivno (na primjer struktura binarno stablo ili hijerarhija organizacije foldera na disku). Stoga je rekurzija kao tehnika pisanja funkcija u mnogim praktičnim situacijama veoma dobrodošla, jer se pokazuje da tako napisane funkcije nude prilično elegantna rješenja. Evo nekoliko primjera rekurzivno definisanih funkcija.

Primjer 4.2. Napisati rekurzivnu funkciju koja računa fakorijel nenegativnog broja.

```
int faktorijel(int n){
    if(n == 0) return 1;
    if(n == 1) return 1;
    return n * faktorijel(n-1);
}
```

Kao što smo napomenuli, rekurzivne funkcije nude elegantna rješenja, ali su ona često vremenski i memorijski veoma neefikasna. Posmatrajmo sljedeći primjer.

Primjer 4.3. Napisati rekurzivnu funkciju koja računa `n`-ti element Fibonačijevog niza. Za Fibonačijev niz važi sljedeća definicija:

```
fib(0) = fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)
```

odnosno dva početna elementa su jednaka jedinici, dok je svaki naredni jednak zbiru prethodna dva. Na osnovu navedenog, možemo napisati rekurzivnu definiciju funkcije:

```
int fib(int n){
    if(n == 0) return 1;
    if(n == 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

Ovakvom definicijom lako uočavamo problem "neefikasnosti" rekurzije, jer je za računanje `n`-tog elementa niza potrebno izvršiti veliki broj istih izračunavanja. Na primjer, peti element ovog niza se računa na sljedeći način:

Elektronska verzija

```
f(5) = f(4) + f(3) = f(3) + f(2) + f(2) + f(1) =
= f(2) + f(1) + f(1) + f(0) + f(1) + f(0) + 1 = f(1) + f(0) + 1 + 1 +
    1 + 1 + 1 + 1 = 1 + 1 + 6 = 8
```

Efikasnije rješenje se dobija funkcijom koja umjesto rekurzije koristi iterativni proces.

```
int fib (int n){
   int i;
   int prvi, drugi, novi;
   if (n ==0 || n == 1)
      return 1;
   prvi = 1;
   drugi = 1;
   for (i = 1; i < n; i++){
      novi = prvi +drugi;
      prvi = drugi;
      drugi = novi;
   }
   return novi;
}
```

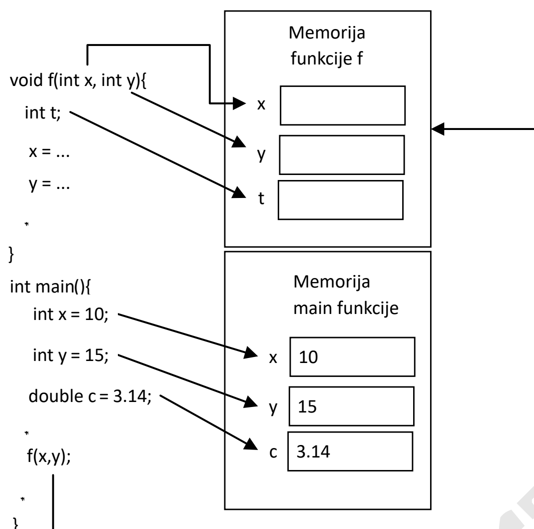
Napomenimo da ćemo sa na Fibonačijev niz vratiti još jednom, kada budemo razmatrali pojam statičkih promjenljivih, u Poglavlju 7 u Primjeru 7.11.

4.3 Prenos argumenata u funkciju

Kao što smo već pomenuli, argumenti koji su navedeni u definiciji funkcije se zovu formalni argumenti, dok se izrazi koji se nalaze na odgovarajućim mjestima prilikom poziva funkcije zovu stvarni argumenti. Prenos argumenata u funkciju se uvijek odvija **po vrijednosti**. To znači da se prilikom poziva funkcije stvarni argumenti kopiraju u formalne argumente, dok funkcija prima kopije stvarnih argumenata, ne mijenjajući originalne vrijednosti.

Sa jedne strane, ovaj pristup je koristan u situacijama kada stvarni argumenti ne treba da se mijenjaju u okviru funkcije. Zapravo, kao što je i napisano u prethodnoj rečenici, stvarni argumenti (argumenti na koje je pozvana funkcija) se i ne mogu promijeniti u okviru funkcije. Međutim, ako je u okviru funkcije potrebno izvršiti izmjenu vrijednosti parametra na koji je funkcija pozvana, tada se mora pribjeći drugom pristupu, koji podrazumijeva slanje **adrese** parametra. Tada se **adresa** podatka u funkciju prenosi po vrijednosti, a sam podatak se mijenja na osnovu činjenice da funkcija "zna" memorijsku lokaciju, gdje se taj

Elektronska verzija



Slika 4.1: Ilustracija prenosa argumenata u funkciju

Elektronska verzija

podatak nalazi, na osnovu koje taj podatak može i da izmijeni. O prenosu argumenata pomoću adrese će u udžbeniku kasnije ponovo biti riječi, kada budemo razmatrali nizove u Poglavlju 6 i pokazivače u Poglavlju 8. Ilustrativan prikaz prenosa argumenata u funkciju i raspolaganje različitim dijelovima memorije `main` funkcije i funkcije `f`, koja se poziva iz `main` funkcije može se vidjeti na Slici 4.1.

Uradimo dva primjera kojima se može ilustrovati prenos argumenata u funkciju po vrijednosti.

Primjer 4.4. Posmatrajmo program koji sadrži funkciju `uvecaj`, u okviru koje se vrijednost njenog cjelobrojnog argumenta uvećava za 1.

```
1  #include <stdio.h>
2  void uvecaj(int x){
3
4      x++;
5      printf("Vrijednost x unutar funkcije: \%d\n",x);
6  }
7  int main(){
8
9      int x = 10;
10     uvecaj(x);
```

```
11
12      printf("Vrijednost x nakon u glavnom programu nakon poziva
            funkcije: %d\n",x);
13
14      return 0;
15
16  }
```

Po pokretanju programa, na ekranu dobijamo sljedeći ispis

```
Vrijednost x unutar funkcije: 11
Vrijednost x nakon u glavnom programu nakon poziva funkcije: 10
```

Vidimo da se `x` lokalno uvećalo za 1, dok, po završetku funkcije, to uvećanje nema efekta na promjenljivu `x` na koju je funkcija pozvana. Razlog za ovo je taj što funkcija `uvecaj` raspolaže kopijom vrijednosti promjenljive `x` iz glavne funkcije (sticajem okolnosti i jednoj i drugoj promjenljivoj smo dali isto ime `x`, ali to nisu iste promjenljive). Sve izmjene koje se vrše unutar funkcije se dešavaju na lokalnoj kopiji promjenljive, dok, po završetku funkcije, izmjene nemaju efekta na promjenljivu `x` iz glavnog dijela programa.

Primjer 4.5. Napisaćemo funkciju koja računa najveći zajednički djelilac dva broja po Euklidovom algoritmu, kao i program koji je koristi.

```
1   #include <stdio.h>
2   int nzd(int a, int b){
3     while(a && b)
4       if(a > b)
5         a %= b;
6       else
7         b %= a;
8     return a + b;
9   }
10  int main(){
11    int a = 1989;
12    int b = 867;
13    int d = nzd(a,b);
14    printf("nzd(%d,%d) = %d\n",a,b,d);
15    return 0;
16
17  }
```

Kao što vidimo i iz ovog primjera, promjenljive `a` i `b`, koje su argumenti funkcije `nzd`, su lokalne kopije promjenljivih `a` i `b`, koje su definisane u okviru `main`

Elektronska verzija

funkcije. Unutar funkcije `nzd` te lokalne kopije se mijenjaju (smanjuju), sve dok jedna od njih ne dođe do vrijednosti 0, kada uslov **`while`** petlje postaje netačan. Rezultat funkcije je praktično jednak vrijednosti one promjenljive koja je različita od nule. U glavnom dijelu programa, promjenljive `a` i `b` ne mijenjaju svoje prvi put dodijeljene vrijednosti.

4.4 Funkcije sa promjenljivim brojem argumenata

U nekim situacijama korisno je definisati funkcije koje mogu da uzimaju promjenljiv broj parametara. Iako ih do sada nismo razmatrali u detalje, podsjetimo se da su funkcije `scanf` i `printf` upravo takve. Na primjer, kod funkcije `printf` u jednom pozivu možemo da šaljemo veći broj podataka na standardni izlaz, a ukupan broj tih podataka može biti proizvoljan. Pored funkcija sa promjenljivim brojem parametara koje su dio standardne biblioteke, takve funkcije možemo da razvijamo i sami.

Opšti oblik funkcije sa promjenljivim parametrima izgleda ovako

```
tipPodatka ime(tip1 arg1,...,tipN argN,...);
```

gdje je, kao i kod "običnih" funkcija, u uobičajenoj formi prvo navedena lista argumenata `tip1 arg1,...,tipN argN` koji su deklarisani, a nakon toga slijede tri tačke (`...`) koje ukazuju na to da, nakon deklarisanih argumenata, slijede još i dodatni nedeklarisani argumenti.

Na primjer, funkcija sa promjenljivim brojem parametara bi mogla biti deklarisana ovako:

```
double f(int n,...);
```

Mehanizam programskog jezika C, kojim se omogućava rad sa ovakvim funkcijama je definisan u datoteci standardne biblioteke `stdarg.h`. Unutar ove datoteke definisan je tip podataka `va_list`, koji služi za predstavljanje liste (dodatnih) argumenata koji su, pored onih fiksnih, proslijeđeni funkciji. U samom početku rada, uobičajeno je da najprije definišemo promjenljivu ovog tipa:

```
va_list poklista;
```

Pored ove liste, na raspolaganju su i posebne funkcije koje su zapravo makroi (o makroima će biti riječi u nastavku udžbenika), pomoću kojih možemo da upravaljamo listom argumenata. Makro `va_start`, koja ima deklaraciju

```
void va_start(va_list ap, last_arg);
```

Elektronska verzija

je funkcija kojom počinje obrada liste. `va_start` uzima kao prvi argument promjenljivu tipa `va_list` (u našem slučaju to bi bila `poklista`, a kao drugi, posljednji deklarisani argument funkcije (posljednji argument prije tri tačke). U našem primjeru bi to bio parametar `n`.

```
va_start(poklista,n);
```

Po uspješnom završetku funkcije `va_start`, promjenljiva koja služi za rad sa listom (u našem slučaju to je `poklista`) je povezana sa listom argumenata i spremna za dalju upotrebu. Druga funkcija (koja je takođe makro)

```
type va_arg(va_list ap, type);
```

za argumente uzima listu i tip podatka i preuzima sljedeći argument iz liste i konvertuje ga u podatak željenog tipa. Na našem primjeru, pozivom funkcije

```
va_arg(poklista,int);
```

bi se preuzelo naredni element iz liste i pokušao konvertovati u `int`.

Makro `va_end` se koristi za završetak obrade liste argumenata. Pozivanje makroa `va_end` je obavezno.

Primjer 4.6. Razmotrimo situaciju u kojoj je potrebno definisati funkciju koja računa prosječnu vrijednost cijelih brojeva, s tim što ukupan broj brojeva, čiji se prosjek računa, može biti proizvoljan. Da ne bismo za svaki tačan broj brojeva pisali po jednu funkciju koja uzima tačno toliko argumenata (a sa pravom se čini se da bi to bilo i besmisleno i nemoguće), napisaćemo funkciju `prosjek`, koja uzima promjenljiv broj argumenata.

Funkciju možemo deklarisati na sljedeći način:

```
float prosjek(int brojArgumenata, ...);
```

gdje će prvi, fiksni parametar, (koji je uvijek obavezan), sadržavati informaciju koliko slijedi brojeva čiji se prosjek računa. Koristeći gore opisanu tehniku rada sa listom argumenata, funkciju, kao i program koji je koristi, možemo napisati na sljedeći način.

```
1  #include <stdio.h>
2  #include <stdarg.h>
3
4  float prosjek(int brojArgumenata, ...)
5  {
6     float suma=0.0;
```

Elektronska verzija

```

    va_list args;
    va_start (args,brojArgumenata);

    int i;
    for (i = 0; i < brojArgumenata; i++) {
        int broj = va_arg (args,int); //podatke citamo kao int jer je
            tako navedeno u zadatku
        suma += broj; //ali sumu cuvamo kao flaot
    }

    float p=suma/brojArgumenata;
    va_end (args);
    return p;
}

int main()
{
    float p = prosjek(6,1,2,3,4,5,6);
    printf("Prosjek je %f\n",p);
    return 0;
}
```

Iako prvi argument ovakvih funkcija uvijek mora biti fiksan, on ne mora obavezno da nosi informaciju o broju preostalih argumenata. Posmatrajmo naredni primjer.

Primjer 4.7. Potrebno je ponovo napisati funkciju koja računa prosjek prvih nekoliko brojeva koji su argumenti funkcije, sve do argumenta koji ima vrijednost -1.

```
#include <stdio.h>
#include <stdarg.h>
float prosjek2(int broj1, ...)
{
    float suma = broj1;//i broj1 ulazi u prosjek
    int brojac=1;

    va_list args;
    va_start (args,broj1);

    while(1){
        int broj = va_arg (args,int);
        if(broj == -1) break; //kad dodjemo do -1 prekidamo petlju
```

Elektronska verzija

```
14            suma += broj;
15        brojac++;
16    }
17
18    float p = suma / brojac;
19    va_end (args);
20    return p;
21 }
22
23 int main()
24 {
25    float p = prosjek2(1,2,3,4,5,-1,10,11,234,34); //sve poslije -1 se
          nece razmatrati
26    printf("Prosjek je %f\n",p);
27    return 0;
28 }
```

4.5 Pitanja i zadaci

1. Kolika je vrijednost promjenjivih `x`, `y` i `z` nakon izvršenja narednog kôda?

```
1  #include<stdio.h>
2  int f(int x, int y)
3  {
4      x++;
5      y *= 2;
6      return x + y;
7  }
8  int main()
9  {
10
11     int x = 3;
12     int y = 4;
13     int z=f(x,y);
14
15     printf("x=%d, y=%d\n, z=%d\n",x,y,z);
16
17     return 0;
18 }
```

2. Šta ispisuje sljedeći kôd na ekranu? Objasni zašto razmjena vrijednosti promjenljivih unutar funkcije `zamijeni` ne utiče na njihove vrijednosti u

Elektronska verzija

`main` funkciji.

```
#include<stdio.h>
void zamijeni(int a, int b){
    int t = a;
    a = b;
    b = t;
}
int main(){
    int a = 11;
    int b = 22;
    printf("Prije zamjene a=%d, b=%d\n",a,b);

    zamijeni(a,b);
    printf("Nakon zamjene a=%d, b=%d\n",a,b);

    return 0;
}
```

3. Napisati funkciju koja za argument uzima cijeli broj, a kao rezultat vraća njegovu apsolutnu vrijednost. Testirati funkciju u glavnom dijelu programa.

4. Napisati funkciju koja za argument uzima četiri cijela broja, a kao rezultat vraća najveći od njih. Testirati funkciju u glavnom dijelu programa.

5. Napisati iterativnu i rekurzivnu definiciju funkcije dvostruki faktorijel.

6. Ako je funkcija `f` definisana na sljedeci način

```
int f(int n){
    if(n == 0 || n == 1)
        return n;
    return n + f(n-2);
}
```

   i ako je u glavnom dijelu programa zadata naredba

   `int a = f(12);`

   koliko puta će biti pozvana funkcija `f`, dok se ne izračuna vrijednost `a`? Šta predstavlja dobijena vrijednost `a` za proizvoljan cijeli broj n?

7. Napisati rekurzivnu definiciju funkcije koja računa sumu svih prirodnih brojeva od 1 do n.

Elektronska verzija

8. Date su dvije definicije funkcije `stepen`, koja računa vrijednost n^k. Uporediti date definicije, da li obje odgovaraju definiciji stepene funkcije i koja od njih ima više rekurzivnih poziva?

```
int stepen1(int n, int k){
    if(k == 0) return 1;
    if(k == 1) return n;
    if(k % 2 == 0)
        return stepen1(n * n, k / 2);
    else
        return n * stepen1(n * n, k / 2);
}

int stepen2(int n, int k)
{
    if(k == 0) return 1;
    return n * stepen2(n,k - 1);
}
```

9. Ako je funkcija deklarisana na sljedeci način

   ```
   int f(int broj1, int broj2,...)
   ```

   koliko stvarnih argumenata može da procesira?

10. Napisati funkciju koja određuje koliko argumenata funkcije je jednako nuli. Argumenti se razmatraju redom sve do argumenta koji ima vrijednost veću od 100.

11. Napisati funkciju `f(int b, char c1, char c2,...)` u kojoj `b` predstavlja broj karaktera koje treba razmatrati. Za svaki od `b` karaktera se provjerava da li je malo ili veliko slovo. Ako je karakter malo slovo na ekranu se ispisuje isto to, ali veliko slovo, a ako je karakter veliko slovo, na ekranu se ispisuje isto to, ali malo slovo. Koji tip podatka kao rezultat vraća ova funkcija?

12. Napisati funkciju koja ispituje da li je broj koji uzima za argument prost broj. U glavnom dijelu programa se unosi sa tastature cijeli broj `n` i na ekranu se štampaju svi prosti brojevi manji od n.

13. Napisati funkciju koja za argument uzima cijeli broj i iz njegovog zapisa uklanja cifru hiljada, ako takva cifra postoji, a ako ne, broj ostaje nepromijenjen. Testirati funkciju u glavnom dijelu programa.

Elektronska verzija

14. Za dva cijela broja kažemo da su relaciji ako se sastoje od istih cifara, npr. brojevi 1223 i 321123 su u relaciji. Napisati funkciju koja za argument uzima dva cijela broja, a kao rezultat vraća 1, ako su brojevi u relaciji, dok u suprotnom vraća 0.

15. Neka je broj n_1 proizvod cifara datog broja n, broj n_2 proizvod cifara broja n_1,..., broj n_k proizvod cifara broja n_{k-1}, pri čemu je k najmanji prirodan broj za koji je n_k jednocifren. Napisati funkciju koja za dato n izračunava k. Na primjer, vrijednosti ove funkcije za $10,\ 25,\ 39$ su redom $1,\ 2,\ 3$.

Elektronska verzija

5 Naredbe ulaza i izlaza

U prvim koracima temeljnog učenja i savladavanja tehnika programiranja u bilo kom programskom jeziku, veoma je važno omogućiti komunikaciju programa sa korisnikom programa. Veoma često, od korisnika se očekuje da u program unese određene podatke, dok program tokom ili nakon izračunavanja korisniku ispisuje odgovarajuće poruke kao rezultate izvršavanja.

Ulazni podaci se u programe koji su pisani u programskom jeziku C uglavnom unose, ili preko standardnog ulaza, koji je u većini slučajeva tastatura, ili putem tzv. ulaznih datoteka. Kao i u većini drugih programskih jezika, programski jezik C raspolaže odgovarajućim funkcijama pomoću kojih se ti podaci učitavaju u program.

Kada je riječ o izlaznim podacima, tu prvenstveno mislimo na izlaz podataka na standardni izlaz - ekran, ili ispis podataka u izlaznu datoteku. Kao što je slučaj i sa ulazom podataka, programski jezik C posjeduje ugrađene funkcije pomoću kojih se podaci ispisuju na ekran, odnosno u izlazne datoteke.

Ulazne i izlazne operacije u programskom jeziku C su podržane specijalizovanim funkcijama, koje pripadaju standardnoj biblioteci samog jezika. Stoga se u programe koji koriste ulazne ili izlazne funkcije (a većina programa su takvi) treba uključiti standardno zaglavlje `<stdio.h>`. Ulazni i izlazni uređaji (tastatura i ekran) se u programskom jeziku C tretiraju kao fajlovi. Ulaz i izlaz se realizuju kao *tokovi* (engl. stream) podataka (obično pojedinačnih bajtova ili karaktera). Kada se program izvršava, da bi se obezbijedio pristup tastaturi ili ekranu, automatski se otvaraju tri toka (fajla): standardni ulaz (`stdin`), standardni izlaz (`stdout`) i standardni izlaz za greške (`stderr`), na koji se obično upućuju poruke o greškama koje nastaju u toku rada programa, koje se takođe najčešće prikazuju na ekranu.

Treba pomenuti da je moguće i preusmjeriti standardni ulaz (unos podataka sa tastature) na neku ulaznu datoteku, kao što je moguće preusmjeriti i standardni izlaz (ekran) na izlaznu datoteku. Takođe, moguće je preusmjeriti i standardni izlaz za greške u izlaznu datoteku.

U velikom broju primjera koje smo do sada analizirali, koristili smo funkciju `printf`, pomoću koje možemo da na formatiran način ispisujemo podatke na ekran, a u nekoliko zadataka smo i unosili podatke sa tastature funkcijom `scanf`. U nastavku ćemo detaljno analizirati ove, ali i druge funkcije koje

Elektronska verzija

omogućavaju unos i ispis podataka. Krenimo od najjednostavnijih funkcija: `getchar` i `putchar`.

5.1 Funkcije getchar i putchar

Najjednostavniji način za unošenje podataka u program je čitanje jednog karaktera sa standardnog ulaza (tastature), pomoću funkcije `getchar`. Funkcija `getchar` ne uzima ništa za argument, a kao rezultat vraća sljedeći karakter sa ulaza, ili vrijednost `EOF`, kada dođe do kraja toka. Njen prototip je:

```
int getchar();
```

Iako je prvenstvena namjena ove funkcije, na šta nas upućuje i njeno ime, da vrati karakter koji se unosi sa ulaza, tip njenog rezultata je, ipak, cio broj. To, zapravo, ništa ne komplikuje stvari, jer je tip podatka `int` dovoljno veliki da se pomoću njega mogu zapisati svi karakteri (prisjetimo se da svakom karakteru odgovara njegov redni broj u ASCII kôdu). Pored toga, cjelobrojnom vrijednošću je moguće je predstaviti i vrijednost simboličke konstante `EOF`, kojoj je najčešće dodijeljena vrijednost -1.

Pomoću funkcije `putchar` možemo ispisivati pojedinačne karaktere na standardnom izlazu. Funkcija za argument uzima karakter koji ispisuje, dok kao rezultat vraća taj karakter (kao cio broj) ili konstantu `EOF` u slučaju da je došlo do greške prilikom ispisa. Prototip funkcije `putchar` je:

```
int putchar(int)
```

Uradimo dva primjera koji koriste ove dvije funkcije.

Primjer 5.1. Napisati program u okviru kojeg se unosi tekst sa standardnog ulaza, a određuje se podatak koliko je uneseno malih, slova, velikih slova, cifara, razmaka i redova.

```
1  #include <stdio.h>
2  int main() {
3
4    int malih = 0, velikih = 0, cifara = 0, razmaka = 0, redova = 0;
5    int c;
6    while((c = getchar()) != EOF) {
7      if(c >= 'a' && c <= 'z')
8        malih++;
9      if(c >= 'A' && c <= 'Z')
10       velikih++;
```

Elektronska verzija

```
11        if(c >= '0' && c <= '9')
12          cifara++;
13        if (c == ' ')
14          razmaka++;
15        if(c == '\n')
16          redova++;
17      }
18      printf("Broj malih slova: %d\n", malih);
19      printf("Broj velikih slova: %d\n", velikih);
20      printf("Broj cifara: %d\n", cifara);
21      printf("Broj belina: %d\n", razmaka);
22      printf("Broj redova: %d\n", redova);
23
24      return 0;
25  }
```

Primjer 5.2. Napisati program koji čita karaktere sa standardnog ulaza sve do pojave simbola `EOF` i na ekranu ispisuje karaktere koji su samoglasnici.

U ovom zadatku ćemo za ispis koristiti funkciju `putchar`.

```
1   #include <stdio.h>
2  int main(){
3    int karakter;
4    while((karakter = getchar())!= EOF){
5        if(karakter == 'a' || karakter == 'e' || karakter == 'i' ||
             karakter == 'o' ||
6            karakter == 'u' || karakter == 'A' || karakter == 'E' ||
                 karakter == 'I' ||
7            karakter == 'O' || karakter == 'U')
8            putchar(karakter);
9    }
10   return 0;
11 }
```

5.2 Funkcije gets i puts

Funkciju `gets` možemo koristiti za čitanje teksta sa standardnog ulaza sve do kraja tekuće linije (linije teksta koji se trenutno čita), ili do kraja datoteke. Pročitani karakteri se smještaju u nisku karaktera `s`. Osobine niski karaktera ćemo detaljno objašnjavati u poglavlju o pokazivačima. Za sada, dovoljno je prihvatiti da je niska karaktera podatak koji se formira na osnovu pojedinačnih

Elektronska verzija

karaktera, a da se informacija o tome gdje se ta niska nalazi u memoriji čuva u *pokazivaču* na tu nisku. Pokazivače (kasnije ćemo vidjeti da su to, takođe, promjenljive) prepoznajemo tako što uz tip podatka piše znak asterisk (zvjezdica). Umjesto pokazivača na nisku prilikom poziva funkcije `gets` možemo koristiti i niz karaktera. Prototip ove funkcije izgleda ovako:

```
char* gets(char * s)
```

Ako je znak za kraj reda posljednji koji je pročitan (i ako je on uzrokovao prekid čitanja), on se ne smješta u nisku, a na kraj niske dodaje nulti karakter (karakter sa ASCII vrijednošću nula), o čemu će, takođe, biti dosta riječi kasnije. Kao što se može i pretpostaviti iz same sintakse upotrebe funkcije `gets`, ona kao argument uzima pokazivač na niz karaktera, koji se popunjava prilikom unosa, a kao rezultat vraća upravo taj pokazivač (pokazivač `s`), ako je čitanje podataka bilo uspješno, dok, u slučaju neuspješnog čitanja, vraća pokazivač `NULL`.

Treba napomenuti da se prilikom unosa teksta ne vrši provjera da li je u memoriji obezbijeđeno dovoljno prostora za čuvanje pročitanog sadržaja. Zbog toga se smatra da ova funkcija nije preporučljiva za upotrebu u onim slučajevima kada postoji realna pretpostavka je dužina niske koja se unosi veća od odgovarajućeg memorijskog prostora predviđenog za njeno čuvanje.

Funkcija `puts` je relativno jednostavna za upotrebu i služi za ispis niske na standardnom izlazu. Njen prototip izgleda ovako

```
int puts(const char *s)
```

Prilikom ispisa, završni karakter niske (završni nulti karakter) se ne ispisuje. U slučaju neuspjelog ispisa, funkcija kao rezultat vraća `EOF`, dok se u slučaju uspješnog izvršenja vraća nenegativna vrijednost.

Primjer 5.3. Najjednostavniji primjer upotrebe funkcije `gets` i `puts` je sljedeći. Napisati program koji dozvoljava unos rečenice koja se sastoji od jedne linije teksta, a zatim unesenu rečenicu ispisuje na ekran.

```
1  #include<stdio.h>
2
3  int main(){
4
5    char rijeci[40];//napravimo niz karaktera da bismo obezbijedili
        memorijski prostor za cuvanje recenice
6    puts("Unesite recenicu:");
7    gets(rijeci);
```

Elektronska verzija

```
8
9      puts("Unijeli ste:");
10     puts(rijeci);
11     return 0;
12 }
```

5.3 Funkcije formatiranog unosa i ispisa

Pomoću funkcija za unos i ispis podataka koje smo do sada naveli (funkcije `getchar`, `putchar`, `gets` i `puts`) radimo samo sa pojedinačnim karakterima ili niskama. Pored ovih funkcija, postoje i funkcije formatiranog ulaza i ispisa, koje mogu da rade i sa nekim drugim tipovima podataka, prvenstveno sa brojevima.

5.3.1 Funkcija formatiranog unosa

Funkcija `scanf` je funkcija koja se koristi za učitavanje podataka sa standardnog ulaza koji su u nekom unaprijed očekivanom formatu. Prototip funkcije izgleda ovako

```
int scanf(const char *format, ...);
```

Tri tačke u prototipu ukazuju na to da funkcija može da uzme promjenljiv broj argumenata, a zapravo broj argumenata zavisi od broja podataka koji se unose. U opštem slučaju, funkcija se poziva na sljedeći način:

```
scanf(format, arg1, arg2, ... ,argN)
```

Prilikom čitanja, karakteri koji se unose se interpretiraju na osnovu specifikacije koja je navedena u niski `format`, dok se rezultat čitanja smješta na odgovarajuća mjesta definisana argumentima `arg1`, `arg2`, ... ,`argN`.

Kontrolna niska `format` (naziva se i format niska) je konstantan niz znakova koji se sastoji od individualnih grupa znakova koji definišu način konverzije karaktera koji se unose. Svakom argumentu funkcije `scanf`, koji slijede nakon niske `format`, je pridružena jedna grupa znakova konverzije. Svaka od tih grupa počinje znakom %, nakon koga slijedi oznaka odgovarajuće konverzije koja odgovara tipu podatka koji se učitava (na primjer `%d` ili `%s`).

Funkcija `scanf` završava sa radom kada se iscrpe sve grupe znakova konverzije u format niski, ili kada neki dio podataka, koji se unosi, ne odgovara

Elektronska verzija

Znak konverzije	Tip podatka koji se učitava
`%d`	cijeli broju dekadnom zapisu (**`int`**)
`%e,%f,%g`	broj s pokretnim zarezom (**`double`**)
`%h`	kratak cijeli broj (**`short`**)
`%i`	cijeli broj, koji može biti u dekadnom, oktalnom (ako ima vodeću nulu) ili heksadecimalnom zapisu (ako ima vodeći zapis 0x ili 0X) (**`int`**)
`%o`	cijeli broj u oktalnom zapisu (**`int`**)
`%u`	neoznačen cijeli broj (**`unsigned int`**)
`%x`	cijeli broj u heksadecimalnom zapisu (**`int`**)
`%c`	jedan karakter (**`char`**)
`%s`	niska koja se unosi bez navodnika (**`char *`**)
`%p`	pokazivač (**`void *`**)

Tabela 5.1: Pregled znakova za konverziju prilikom čitanja podataka

predviđenoj šemi koja je navedena u format niski. Funkcija kao rezultat vraća broj uspješno unesenih podataka.

Najčešći znakovi konverzije su prikazani u Tabeli 5.1. Pored oznaka za konverziju, koji su prikazani u lijevoj koloni tabele, format niska može da sadrži i druge karaktere. Karakteri "razmak" (space) i tabulator u format niski se ignorišu, dok se u slučaju prisustva nekog "običnog" karaktera očekuje da se naredni karakter u unosu bude upravo jednak njemu.

Pomenuli smo da se rezultati čitanja smještaju u argumente `arg1`, `arg2`, ... ,`argN` funkcije `scanf`. Važno je napomenuti da ovi argumenti nisu "obične" promjenljive, već su to argumenti koji predstavljaju *adrese* promjenljivih u koje se smještaju podaci koji se unose. Na taj način se omogućava da se vrijednost promjenljive, čija se adresa prenosi u funkciju `scanf`, promijeni na odgovarajuću vrijednost, koja je unesena sa tastature. Kasnije ćemo se, kada uvedemo pokazivače u Poglavlju 8, detaljnije upoznati sa mehanizmima koji se odnose na rad sa adresama podataka. Pored toga, važno je napomenuti da sve promjenljive koje pomoću funkcije `scanf` trebaju da dobiju neku vrijednost koja je unesena sa tastature, moraju prethodno biti deklarisane. Na primjer, ako želimo da u promjenljivu `a` smjestimo cio broj koji unosimo sa tastature, to ćemo uraditi na sljedeći način:

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

gdje zapis `&a` znači da funkciji `scanf` prosljeđujemo adresu promjenljive `a`.

Elektronska verzija

Primjer 5.4. U narednom primjeru koristimo funkciju `scanf` za formatirani unos nekoliko različitih tipova podataka.

```
1  #include<stdio.h>
2  int main()
3  {
4      char a;
5      int b;
6      float c;
7      double d;
8      printf("Unesi karakter: ");
9      scanf("%c",&a);
10     printf("Unesi cio broj: ");
11     scanf("%d",&b);
12     printf("Unesi realan broj (jednostruka preciznost): ");
13     scanf("%f",&c);
14     printf("Unesi realan broj (dvostruka preciznost): ");
15     scanf("%if",&d);
16     return 0;
17 }
```

5.3.2 Funkcija formatiranog ispisa

Za formatiran ispis podataka na standardni izlaz koristi se funkcija `printf`, čija deklaracija ima sljedeću sintaksu:

```
int printf(const char *format, ...);
```

Slično kao i kod funkcije `scanf`, tri tačke u prototipu ukazuju na činjenicu da funkcija `printf` može da uzme promjenljiv broj argumenata. Broj argumenata praktično zavisi od broja podataka koji se ispisuju na ekranu u okviru poziva funkcije. U opštem slučaju, funkcija se poziva na sljedeći način:

```
printf(format, arg1, arg2, ... ,argN)
```

gdje je u okviru konstantnog stringa `format` definisan način formatiranja ispisa argumenata koji slijede nakon njega. Slično kao i kod funkcije `scanf`, ali u ovom slučaju format ispisa argumenata, je definisan odgovarajućim kontrolnim znacima (koji počinju znakom %). Pojedinačne grupe kontrolnih i običnih znakova unutar niske `format` mogu se nastavljati jedna na drugu. Svi znakovi osim kontrolnih će biti ispisani onako kako su uneseni, dok će na mjestu kontrolnih znakova biti ispisana vrijednost odgovaraćeg argumenta, u formatu koji

Elektronska verzija

je definisan tim kontrolnim znacima. Praktično, kontrolni znaci definišu način konverzije odgovarajućeg podatka u niz karaktera koji će biti ispisani na ekranu.

Funkcija `printf` kao rezultat vraća broj ispisanih znakova, ili znak `EOF`, ako je došlo do greške.

Kao što je već pomenutno, kod funkcije `printf` specifikacija formata ispisa svakog argumenta je definisana odgovarajućim kontrolnim znacima. Svaka grupa kontrolnih znakova počinje karakterom %, nakon čega mogu (ali ne moraju) da slijede neki specifični karakteri kojima se definiše poravnanje, širina i preciznost ispisa i dr. Specifikacija formata ispisa argumenta se završava karakterom (ili karakterima) koji definišu konverziju.

Između znaka % i karaktera za definisanje konverzije mogu da se nađu:

- Karakter minus (-), kojim se definiše da je argument koji se konvertuje lijevo poravnat;
- Broj koji definiše širinu prostora u koji se argument ispisuje. Ako je polje šire u odnosu na to koliko karaktera je potrebno da se argument ispiše, onda se ono nadopunjuje razmacima, sa lijeve ili desne strane, u odnosu na poravnanje.
- Karakter tačka (.), pomoću koje se odvaja informacija o širini polja od preciznosti;
- Broj koji definiše preciznost, na sljedeći način: Ako se ispisije niska, definiše najveći broj karaktera koje treba štampati. Ako se ispisuje broj u pokretnom zarezu definiše broj decimala iza decimalne tačke, a ako se ispisuje cio broj, definiše najmanji broj cifara koje se štampaju.
- Karakter h, ako se cio broj štampa kao **`short`**;
- Karakter l, ako se cio broj štampa kao **`long`**.

Najčešći znakovi konverzije su prikazani u Tabeli 5.2.

Prilikom poziva funkcije `printf`, treba voditi računa da broj argumenata koji se ispisuju i njihovi tipovi odgovaraju specifikaciji koja je navedena u niski koja sadrži specifikaciju. U slučaju da se ne navede odgovarajući broj argumenata, ili da tipovi argumenata ne odgovaraju navedenim tipovima u specifikaciji, dolazi se do situacija da ponašanje programa nije definisano. Stoga, takve situacije treba izbjegavati.

Primjer 5.5. U sljedećem primjeru prikazana je neispravna upotreba funkcije `printf`. Važno je napomenuti da prevodilac u najvećem broju ovakvih slučajeva neće prijaviti grešku, ali je nepredvidivo na koji način će program reagovati i na kraju, šta će se ispisati na ekranu.

Elektronska verzija

Znak konverzije	Tip podatka koji se ispisuje
`%d`,`%i`	cijeli broj u dekadnom zapisu (**`int`**)
`%u`	neoznačen cijeli broj (**`unsigned int`**)
`%o`	neoznačen cijeli broj u oktalnom zapisu (**`unsigned int`**)
`%x`	neoznačen cijeli broj u heksadecimalnom zapisu (**`unsigned int`**)
`%e`,`%f`,`%g`	broj sa pokretnim zarezom (**`double`**)
`%c`	jedan karakter (**`char`**)
`%s`	niska (**`char *`**)
`%p`	pokazivač (**`void *`**)

Tabela 5.2: Pregled znakova za konverziju prilikom ispisivanja podataka

```c
#include <stdio.h>

int main() {
   int i = 10,j = 20;
   printf("%d %d %d\n",i,j); // ovdje nam nedostaje jedan argument
   printf("%d\n",i,j); // ovdje imamo jedan argument viska
   printf("%d\n","tabla"); // ovdje nam tipovi ne odgovaraju
   printf("%s\n",j); // i ovdje nam tipovi ne odgovaraju
   printf("%u\n",-i);//ovdje nije dobro to sto negativan broj
       ispisujemo kao unsigned
   return 0;
}
```

Evo nekoliko primjera ispravne upotrebe funkcije `printf`

Primjer 5.6. U prvom primjeru ispisujemo cijele brojeve u različitim formatima

```c
#include <stdio.h>

int main() {

    printf("%d\n",29); //ispisujemo 29 kao obican int
    printf("%u\n",29); //ispisujemo 29 kao unsigned
    printf("%u\n",-29); //ovo nece valjati (pogledati prethodni
        primjer)
    printf("%o\n",29); // ispisujemo 29 u oktalnom zapisu
    printf("%x\n",29); //ispisujemo 29 u heksadecimalnom zapisu

```

Elektronska verzija

```
11      //u naredna cetiri reda koristimo desno poravnanje i uzimamo 8
            mjesta za ispis broja
12      printf("%8d\n", 0);
13      printf("%8d\n", 123456);
14      printf("%8d\n", -10);
15      printf("%8d\n", -123456);
16
17      //u naredna cetiri reda koristimo lijevo poravnanje i uzimamo 8
            mjesta za ispis broja
18      printf("%-8d\n", 0);
19      printf("%-8d\n", 123456);
20      printf("%-8d\n", -10);
21      printf("%-8d\n", -123456);
22
23      printf("%+5d\n", 10); //ispisujemo i predznak +
24      printf("%+5d\n", -10); //ovdje je svakako ispisan predznak -
25      printf("%08d\n", 10); //prazna mjesta popunjavamo nulama
26
27      return 0;
28  }
```

Primjer 5.7. Ispišimo sada decimalne brojeve u raznim varijantama. Čitaocu ostavljamo da analizira, red po red, različite formate ispisa.

```
1   #include<stdio.h>
2   int main(){
3       float x = 123.4567;
4
5       printf("%f\n", x);
6       printf("%.3f\n", x);
7       printf("%.1f\n", x);
8       printf("%2.1f\n", x);
9
10      printf("%5.1f\n", x);
11      printf("%8.1f\n", x);
12      printf("%8.6f\n", x);
13      printf("%.2e\n", x);
14      printf("%.4e\n", x);
15      printf("%-20.7f\n" , x);
16      printf("%20.7f\n" , x);
17
18      return 0;
19  }
```

Elektronska verzija

5.3.3 Funkcije sscanf i sprintf

U standardnoj biblioteci za ulaz i izlaz (`stdio.h`) definisane su još dvije veoma korisne funkcije, koje omogućavaju formatiran unos i ispis podataka: funkcije `sscanf` i `sprintf`. Prototip ovih funkcija izgleda ovako

```
int sscanf(const char *s, const char *format, ...);
int sprintf(char *s, const char *format, ...);
```

Prototipi ovih funkcija se od funkcija `scanf` i `printf` razlikuju po prvom argumentu, koji je, zapravo, niska (označena sa `s`) iz koje se čitaju ulazni podaci u slučaju funkcije `sscanf`, odnosno, u koju se pišu podaci (u slučaju funkcije `sprintf`).

Funkcija `sscanf` je analogna funkciji `scanf`, s tim što se podaci ne učitavaju sa standardnog ulaza, već iz niske karaktera `s`. Preciznije, prilikom poziva funkcije, koji u opštem slučaju izgleda ovako

```
sscanf(s,format, arg1, arg2, ... ,argN)
```

karakteri koji su prisutni u niski `s` se interpretiraju na osnovu specifikacije koja je navedena u niski `format`. Rezultat čitanja se smješta na odgovarajuća mjesta definisana argumentima `arg1`, `arg2`, ... ,`argN`. Slično kao i kod funkcije `scanf`, važno je da sadržaj niske `s` (kao što je to bio slučaj i sa ulaznom niskom kod funkcije `scanf`) odgovara specifikaciji koja je navedena u niski `format`.

Funkcija `sprintf` je analogna funkciji `printf`, s tim što se sad podaci ne ispisuju na standardni izlaz, već u nisku `s`. Za uspješno izvršenje ove funkcije, pretpostavka je da je niska `s` dovoljno velika da se u nju može smjestiti (zapisati) odgovarajući rezultat koji se formira konvertovanjem podataka u izlaznu nisku. Preciznije, prilikom poziva funkcije

```
sprintf(s,format, arg1, arg2, ... ,argN)
```

argumenti `arg1 , arg2 , ... ,argN` se konvertuju po pravilima definisanim u niski `format` i zajedno sa eventualnim ostalim karakterima, koji su prisutni u niski `format`, upisuju u nisku `s`.

Primjer 5.8. Ilustrujmo u jednom zadatku primjer upotrebe obje ove funkcije. Najprije ćemo zadati nisku karaktera, pa ćemo funkcijom `sscanf` pažljivo pročitati podatke iz te niske. Zatim ćemo pomoću funkcije `sprintf` zapisati formatiran ispis u nisku `pisanje`. Da bismo provjerili da li je upis bio ispravan, ispisaćemo na kraju nisku `pisanje` na ekran.

Elektronska verzija

```
1  #include <stdio.h>
2  int main ()
3  {
4     char *niska = "Ovo je niska s 2 broja 123 i 444";
5     char s1[10], s2[10], s3[10], s4[10];
6     int b1, b2, b3;
7     char c1, c2;
8     char pisanje[60];
9     sscanf (niska,"%s %s %s %c %d %s %d %c
         %d",s1,s2,s3,&c1,&b1,s4,&b2,&c2,&b3);
10    printf("Zapisivanje u nisku...\n");
11    sprintf(pisanje,"%s %s %s %c %d %s %d %c
         %d",s1,s2,s3,c1,b1,s4,b2,c2,b3);
12    printf("%s",pisanje);
13    return 0;
14 }
```

5.4 Pitanja i zadaci

1. Sa standardnog ulaza se čita jedan po jedan karakter, dok se ne pročita znak `'*'`. Nakon unosa svih karaktera, na ekranu ispisati sljedeće informacije:
   a) uneseni karakter koji ima najveću ASCII vrijednost;
   b) redni broj unesenog karaktera koji ima najveću ASCII vrijednost;
   c) koliko je ukupno uneseno karaktera.
2. Sa standardnog ulaza se čita jedan po jedan karakter, dok se ne dođe do kraja toka. Napisati program koji za svaku cifru ispisuje koliko puta se pojavila među unesenim karakterima.
3. Sa tastature se unose riječi sve dok se ne unese ukupno 10 riječi. Nakon unosa svake riječi ispisati riječ na ekranu.
4. Šta se ispisuje na ekranu sljedecim kôdom?

   ```
   char c = 'z';
   printf("%c %d %o  %x\n",c,c,c,c);
   ```
5. Napisati program koji na ekranu ispisuje sljedeću poruku:

   Ana je rekla: "Danas je roba poskupila za 10% u odnosu na cijenu od prosle sedmice."

Elektronska verzija

6. Napisati program koji za dva unesena cijela broja ispisuje postavku izraza i vrijednost rezultata koji predstavlja:
   a) zbir unesenih vrijednosti;
   b) razliku unesenih vrijednosti;
   c) sumu kvadrata unesenih vrijednosti;
   d) vrijednost cjelobrojnog dijeljenja i ostatak pri djeljenju prve unesene vrijednosti drugom unesenom vrijednošću.

7. Šta se ispisuje na ekranu sljedećim kôdom?

   ```
   printf("Slova:\n%2c\n%4c\n",'x','X');
   ```

8. Sa tastature se unosi niska karaktera koja sadrži informacije o broju indeksa, imenu, prezimenu i broju osvojenih bodova studenta na nekom ispitu. Podaci su međusobno odvojeni po jednim znakom razmaka. Na osnovu podataka u unesenoj niski postaviti vrijednosti promjenljivih `int redniBroj`, `char ime[20]`, `char prezime[20]` i `int brojBodova`.

9. Napisati program koji sabira pet cjelobrojnih vrijednosti koje su zadate u niski karaktera, koja se unosi sa tastature.

10. Šta je sadržaj niske `s` nakon izvšenja narednog kôda?

    ```
    char s[100];
    int a = 5, b = 3, c;
    c = a * b;
    c++;
    sprintf(s, "Proizvod brojeva %d i %d uvecan za 1 iznosi %d", a,
        b, c);
    ```

Elektronska verzija

Elektronska verzija

6 Nizovi

Kako nam i sama riječ niz ukazuje, pod nizom u programskim jezicima podrazumijevamo skup objekata koji su "poredani" jedan pored drugog i kao takvi čine odgovarajuću cjelinu. Malo preciznije, u većini programskih jezika, niz predstavlja skup podataka istog tipa, koji nisu pojedinačno imenovani, već se oni identifikuju svojom pozicijom (indeksom) u nizu. Nizovi predstavljaju veoma važne i moćne strukture pomoću kojih na jednom, dovoljno velikom mjestu, smještamo veći broj podataka istog tipa. Umijeće rada sa nizovima je jedan od glavnih preduslova za bilo kakav ozbiljan rad u većini programskih jezika.

6.1 Deklaracija niza

Sintaksa deklaracije niza je

```
tip imeNiza[dimenzija];
```

`tip` određuje tip elemenata niza, dok `dimenzija` predstavlja broj elemenata niza. Dimenzija niza treba da bude konstantan cjelobrojni pozitivan izraz, na primjer pozitivan cio broj, ili cjelobrojna promjenljiva kojoj je dodijeljena neka pozitivna vrijednost. Elementima niza se pristupa pomoću njihovih pozicija – indeksa u nizu. Element na početnoj poziciji ima indeks 0, sljedeći element indeks 1 itd., dok posljednji element niza ima indeks `dimenzija-1`. Elementu na poziciji `indeks` se pristupa na sljedeći način

```
imeNiza[indeks];
```

Prilikom kreiranja niza, na osnovu tipa podatka elemenata niza i dimenzije niza u memoriji se zauzima odgovarajući prostor koji je potreban da se smjeste svi elementi niza. Susjedni elementi niza se smještaju na susjedne memorijske lokacije.

Naglasimo da su, iako se to dâ i pretpostaviti, nizovi indeksirani cijelim brojevima. Ako želimo da pristupimo elementima niza, trebamo voditi računa da indeks pripada datom opsegu `[0,dimenzija-1]`.

Treba napomenuti da je, za razliku od nekih drugih programskih jezika, u programskom jeziku C ponekad moguće "pokušati" pristupiti i nepostojećem

Elektronska verzija

elementu niza (na primjer elementu niza sa indeksom -1, ili elementu sa indeksom `dimenzija`). U tim slučajevima program prilikom izvršenja najvjerovatnije neće prijaviti grešku, ali posljedice takvog neopreznog pristupa mogu biti ozbiljne i najčešće dovode do neočekivanog rezultata. Stoga, takve situacije treba u potpunosti izbjegavati.

Kao i kod običnih promjenljivih, podrazumijeva se da se prilikom deklarisanja niza elementima ne dodjeljuje vrijednost. Prevodilac samo alocira odgovarajuću količinu memorije, bez provjeravanja ili mijenjanja tog memorijskog prostora.

Sa druge strane, elementi niza se mogu eksplicitno inicijalizovati navođenjem liste vrijednosti razdvojenih zarezima u okviru vitičastih zagrada. Ako želimo da prilikom deklaracije niza svim elementima niza dodijelimo i odgovarajuću vrijednost, onda nije neophodno navoditi njegovu dimenziju. Ako je navedena i dimenzija niza, a neki elementi niza se inicijalizuju, onda broj vrijednosti u inicijalizacionoj listi ne smije biti veći od dimenzije. U slučaju da je dimenzija veća od broja elemenata koji se inicijalizuju, preostali elementi dobiće vrijednost 0.

Sa druge strane, ako je broj inicijalizacijskih vrijednosti veći od dimenzije samog niza, doći će do greške.

Evo nekoliko primjera definisanja nizova.

```
int niz1[10]; // formiran je niz od 10 cijelih brojeva, ali elementi
    nisu inicijalizovani
int niz2[10] = {0}; //i ovo je niz od 10 cijelih brojeva i svi
    elementi su postavljeni na nulu
int niz3[10] = {2,4,6}; //i ovaj niz ima 10 elemenata, od kojih su
    prva tri redom jednaki 2, 4 i 6, a preostalih 7 elemenata je
    jednako nuli
char slova[] = {'t', 'a', 'b', 'l', 'a'}; // formiran je niz od 5
    elemenata sa elementima koji su karakteri redom: 't', 'a', 'b',
    'l' i 'a'
char slova2[] = "tabla"; // formiran je niz od 6 elemenata sa
    elementima koji su karakteri redom: 't', 'a', 'b', 'l', 'a', uz
    dodati posljednji, terminalni karakter '\0'
```

Iako smo to već pomenuli, nije loše i ponoviti da se prilikom deklaracija niza u izrazu

```
tip imeNiza[dimenzija];
```

mora voditi računa da vrijednost `dimenzija` bude cjelobrojan pozitivan izraz, koji je najčešće cjelobrojna pozitivna konstanta (pozitivan cio broj), ili promjenljiva za koju se zna koliko ima vrijednost u trenutku kreiranja niza.

Elektronska verzija

Primjer 6.1. Između ostalih, sljedeći način deklaracije niza je dozvoljen i često korišten:

```
int n;
...
scanf("%d",&n);//ucitamo n sa tastature, vodimo racuna da je pozitivan
int niz[n]; // napravimo niz duzine n
...
```

Posmatrajmo jedan od najjednostavnijih primjera.

Primjer 6.2. Sa tastature se unosi broj n, a potom još i n brojeva. Ispisati brojeve obrnutim redoslijedom u odnosu na unos.

```
1  #include<stdio.h>
2
3  int main(){
4      int i, n;
5      printf("Koliko elemenata ima niz: ");
6      scanf("%d",&n);
7      int niz[n];
8      for (i = 0; i < n; i++)
9     {
10       printf("Unesi %d. element:",i);
11       scanf("%d",&niz[i]);
12    }
13     printf("Ispis u obrnutom redoslijedu...\n");
14     for (i = n-1; i >= 0; i--)
15    {
16       printf("%d\n",niz[i]);
17    }
18 return 0;
19 }
```

U prethodnom primjeru možemo primijetiti da smo za unos elemenata niza (a kasnije i za ispis) koristili **for** iterativnu naredbu. Kao što smo dosta puta do sada već vidjeli, naredbu **for** je praktično koristiti kada trebamo da "brojimo" koliko puta izvršavamo neke naredbe. U slučaju rada sa nizovima, upravo nam taj pristup i odgovara. Na primjer, ako unosimo elemente niza sa tastature, prvo unosimo element na indeksu 0, pa onda element na indeksu 1, pa element na indeksu 2, itd.

Može se desiti da se dužina niza može odrediti tek nekim izračunavanjem u toku programa.

Elektronska verzija

Uradimo još nekoliko primjera kojima ćemo prikazati razne tehnike rada sa nizovima.

Primjer 6.3. Sa tastature se unosi broj n. Formirati cjelobrojni niz koji ima tačno onoliko elemenata koliko n ima cifara i nakon toga popuniti niz ciframa broja n. Ispisati elemente niza na ekranu.

U ovom zadatku, po unosu broja n još uvijek ne znamo koliko elemenata će imati niz, jer ne znamo koliko n ima cifara. Broj cifara broja n možemo odrediti u funkciji, a rezultat funkcije iskoristiti da kreiramo niz. Nakon toga elementima niza dodjeljujemo vrijednost cifara broja n.

```
1  #include<stdio.h>
2  int brojCifara(int n){
3     int brojac = 1;//ima najmanje jednu
4     while((n /= 10)!= 0)//odmah skidamo cifru i ispitujemo uslov
5        brojac++;
6     return brojac;
7  }
8  int main(){
9      int i, n;
10     printf("Unesite broj: ");
11     scanf("%d",&n);
12     int bc = brojCifara(n);
13     int niz[bc]; //tek sad pravimo niz
14     for(i = 0; i < bc; i++ )//znamo koliko tacno imamo cifara, pa
          mozemo da koristimo for petlju
15     {
16       niz[bc-i-1] = n % 10;//niz popunjavamo sa desne strane ka lijevoj
17       n /= 10;
18     }
19     printf("Ispis cifara kao elemenata niza...\n");
20     for (i = 0; i < bc; i++)
21    {
22       printf("%d",niz[i]);
23    }
24     return 0;
25 }
```

Uradimo sada zadatak gdje su elementi niza realni brojevi.

Primjer 6.4. Ako je polinom $P(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$ dat nizom svojih koeficijenata, napisati program koji izračunava vrijednost polinoma za dato x.

Elektronska verzija

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

int main(){
    int n;
    printf("Unesite stepen polinoma :\n");
    scanf("%d",&n);

    float unos[n+1];
    int i;
    for(i = 0; i < n+1; i++){
        printf("Unesite %d element \n",i);
        scanf("%f",&unos[i]);
    }
    float x;
    printf("Unesite x za koje zelite racunati vrijednost polinoma.\n");
    scanf("%f",&x);

    float vr = 0;
    for(i = 0; i < n + 1; i++)
        vr += unos[i] * pow(x,i);

    printf("Vijednost polinoma u tacki %f je %f.\n",x,vr);
    return 0;
}
```

Pored podataka koji se u program unose sa standardnog ili nekog drugog ulaza, nizove možemo da koristimo i za evidentiranje pojava nekih podataka. Posmatrajmo sljedeća dva primjera.

Primjer 6.5. Napisati program koji broji pojavljivanja svake cifre na standardnom ulazu.

Zadatak ćemo uraditi tako što ćemo formirati niz dužine 10 i svaki element niza povezati sa odgovarajućom cifrom. Svaki put kada se unese neka cifra, odgovarajući element niza uvećamo za 1. U zadataku koristimo i funkciju `isdigit`, koja vraća informaciju da li je uneseni karakter cifra ili ne. Naravno, ne bi bilo pogrešno i da smo sami napisali funkciju koja to isto radi.

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

int main() {
   int cifre[10];
```

Elektronska verzija

```
6     int c, i;
7     for (i = 0; i < 10; i++)
8        cifre[i] = 0;
9     while ((c = getchar())!= EOF) {
10       if (isdigit(c))
11          cifre[c - '0']++;//odgovarajuci element niza uvecamo za 1
12    }
13    for (i = 0; i < 10; i++)
14       printf("Cifra %d se pojavljuje %d puta\n", i, cifre[i]);
15    return 0;
16 }
```

Primjer 6.6. Napisati funkciju `prost(x)` koja ispituje da li je broj `x` prost. Napisati glavni dio programa u kojem se unosi prirodan broj n, a zatim i niz cijelih brojeva duzine n. Potom se iz niza izdvaja podniz prostih brojeva i ispisuje se na ekranu.

Ranije smo već ispisivali proste brojeve na ekran. Sada trebamo da na osnovu niza koji smo unijeli sa tastature formiramo novi niz, koji sadrži samo proste brojeve. Ovaj zadatak nije baš "zgodno" rješavati pomoću niza, pošto unaprijed ne znamo koliko ćemo prostih brojeva ukupno imati. Imamo dvije mogućnosti. Prva je da napravimo malo "komotniji" niz, koji je jednake dužine kao i početni, jer bi nam takav niz sigurno bio dovoljne dužine. Druga varijanta, koju ćemo mi koristiti, je da prvo prebrojimo koliko ćemo imati prostih brojeva, formiramo niz koji je upravo te dužine i onda taj niz popunimo prostim brojevima. Nedostatak ovog pristupa je taj što se za svaki broj dva puta ispituje da li je prost.

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

Elektronska verzija

```

    int unos [n];
    int i;
    for(i = 0; i < n; i++){
        printf("Unesite %d element \n",i);
        scanf("%d",&unos[i]);
    }

    int brojP = 0;
    //prvo prebrojimo koliko ima prostih
    for(i = 0;i < n; i++)
      if(prost(unos[i]))
        brojP++;
    //formiramo niz tacno te duzine
    int nizProstih[brojP];
    int j = 0;

    for(i = 0; i < n; i++)
      if(prost(unos[i]))
      {
        nizProstih[j] = unos[i];
        j++;
      }
    printf("Niz prostih brojeva...\n");
    for(j = 0; j < brojP; j++)
      printf("%d ",nizProstih[j]);
    printf("\n");

    return 0;
}
```

Kao što smo vidjeli iz prethodnih nekoliko primjera, elementima niza redom, počev od prvog (koji ima indeks 0), pa na dalje, najlakše pristupamo pomoću iteracija `for` petlje. U nekim slučajevima, kada unaprijed ne znamo kojim elementima niza trebamo da pristupimo, je zgodnije koristiti i `while` petlju. Posmatrajmo sljedeći primjer.

Primjer 6.7. Sa tastature se unosi cjelobrojni niz i još dva broja `x` i `y`. Odrediti proizvod elemenata u nizu, koji se nalaze između prvog pojavljivanja vrijednosti `x` i prvog pojavljivanja vrijednosti `y`. Pretpostavka je da se vrijednosti `x` i `y` nalaze u nizu i da se vrijednost `x` nalazi prije vrijednosti `y`.

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

Elektronska verzija

```
3     int n;
4     printf("Unesite broj n:\n");
5     scanf("%d",&n);
6 
7     int unos[n];
8     int i;
9     for(i = 0; i < n; i++){
10        printf("Unesite %d element: ",i);
11        scanf("%d",&unos[i]);
12    }
13    int x, y;
14    printf("Unesite brojeve x i y.\n");
15    scanf("%d %d",&x,&y);
16
17    int j = 0;
18    //pomjeramo se po nizu dok ne naidjemo na element jednak x
19    while(unos[j] != x)
20        j++;
21
22    j++; //preskocimo x
23
24    int proizvod = 1;
25    //pomjeramo se po nizu i racunamo proizvod dok ne naidjemo na
          element jednak y
26    while(unos[j] != y)
27    {
28        proizvod *= unos[j];
29        j++;
30    }
31
32    printf("Proizvod brojeva izmedju %d i %d u nizu je
          %d.\n",x,y,proizvod);
33
34    return 0;
35 }
```

6.2 Nizovi kao argumenti funkcija

Funkcije kao argument mogu uzimati i nizove. Međutim, za razliku od jednostavnijih tipova podataka, čitav niz se ne prenosi po vrijednosti, već se suštinski prenosi samo informacija o adresi početka niza (adresi prvog elementa niza) i tipu elemenata niza. Pošto funkcija dobija informaciju o adresi početka orig-

Elektronska verzija

inalnog niza, pomoću te informacije i informacije o tipu elemenata, ona može da pristupi memorijskom prostoru, gdje su smješteni i ostali elementi niza.

Primjer 6.8. Napišimo jednostavnu funkciju koja za argument uzima cjelobrojni niz i njegovu dužinu, a kao rezultat vraća zbir elemenata niza.

```
int  zbir( int niz[], int n) {
    int i, rezultat = 0;
   for(i = 0; i < n; i++)
     rezultat += niz[i];
    return rezultat;
}
```

S obzirom na to da se prilikom prenosa niza u funkciju prenosi samo informacija o adresi prvog elementa, dimenziju niza nije neophodno navoditi unutar uglastih zagrada, jer se ona svakako ne prenosi u funkciju. Kada se upoznamo sa pokazivačima i njihovom vezom sa nizovima, vidjećemo da je uobičajeno da se, prilikom prenosa niza u funkciju, ravnopravno koristi i sintaksa pokazivača.

Za posljedicu činjenice da se niz u funkciju prenosi tako što se prenosi samo informacija o tome gdje je niz smješten u memoriji imamo da je unutar funkcije moguće mijenjati vrijednosti elemenata niza, a te promjene ostaju sačuvane i nakon izvršenja funkcije.

Primjer 6.9. Veoma jednostavan primjer kojim ilustrujemo činjenicu da se unutar funkcije mogu mijenjati vrijednosti elemenata niza je prikazan u sljedećem zadatku. Funkcija `f` za argument uzima cjelobrojni niz i jedan indeks, a vrijednost elementa, koji je na tom indeksu postavlja na 1000.

```
1  #include<stdio.h>
2 void f(int niz[], int i){
3    niz[i] = 1000;
4 }
5
6 int main(){
7    int niz[] = {1,2,3,4}; //inicijalizujemo niz na bilo koji nacin
8    f(niz,3); //unutar funkcije cemo promijeniti vrijednost elementa
         koji ima indeks 3
9    printf("%d",niz[3]);
10   return 0;
11 }
```

Elektronska verzija

Činjenicu da unutar funkcija možemo mijenjati vrijednosti elementima niza koristimo dosta često (na primjer kod funkcija kojima "popunjavamo" nizove ili kod funkcija za sortiranje nizova).

Primjer 6.10. Napisati funkciju pomoću koje se sa standardnog ulaza unosi cjelobrojni niz, funkciju koja ispisuje niz na ekranu, kao i jednostavan program koji primjenjuje obje funkcije.

```
#include <stdio.h>
void unos(int niz[], int n){
    int i;
    printf("Unos niza...\n");
    for(i = 0; i < n; i++)
        scanf("%d",&niz[i]);
}
void ispis(int niz[], int n){
    int i;
    printf("Ispis niza...\n");
    for(i = 0; i < n; i++)
        printf("%d\t",niz[i]);
    printf("\n");
}
int main(){
    int i;
    int niz[10];
    unos(niz,10);
    ispis(niz,10);
    return 0;
}
```

Kao što vidimo, a to treba zapamtiti, funkcija `unos` kao rezultat ne vraća ništa (tip rezultata je `void`), ali kao posljedicu izvršenja funkcije imaćemo izmijenjen niz, tako što su elementi niza dobili vrijednosti brojeva koji se unose sa tastature. I funkcija `ispis` ne vraća ništa kao rezultat, ali ona po pravilu ne bi trebala (iako može) da mijenja elemente niza.

6.3 Višedimenzionalni nizovi

Pored običnih (jednodimenzionalnih) nizova, moguće je definisati i višedimenzionalne nizove. Iako broj dimenzija nije ograničen, najviše se koriste dvodimenzionalni nizovi (koji odgovaraju matematičkim strukturama matricama), rjeđe trodimenzionalni, dok se nizovi od četiri i više dimenzija koriste prilično rijetko.

Elektronska verzija

Višedimenzionalne nizove, zapravo, definišemo kao nizove nizova. Deklarišemo ih na sljedeći način:

```
<tip> imeNiza [dimenzija1][dimenzija2]...;
```

dok odgovarajućem elementu pristupamo preko njegovih indeksa

```
imeNiza [indeks1][indeks2]...;
```

Na primjer, ako dvodimenzionalni niz definišemo na sljedeći način

```
int matrica[2][3];
```

definisali smo strukturu koja ima ukupno 6 elemenata, koji su organizovani u dva reda po tri kolone. Uobičajeno je da dvodimenzionalne nizove posmatramo kao matrice (otuda i motivacija za davanje takvog imena). Elementi ovog dvodimenzionalnog niza (matrice) su

matrica[0][0]	matrica[0][1]	matrica[0][2]
matrica[1][0]	matrica[1][1]	matrica[1][2]

Treba voditi računa da se elementima matrice ne može pristupati tako što se indeksi vrste i kolone u okviru uglaste zagrade odvoje zarezom, kako je to već uobičajeno u matematičkoj notaciji. Na primjer, izraz `matrica[1,2]` bi bio interpretiran tako što bi se zapis `1,2`, koji piše u uglastim zagradama, smatrao izrazom koji je formiran pomoću (infiksnog binarnog) operatora zarez sa operandima brojevima 1 i 2. Takav izraz bi bio jednak vrijednosti desnog operanda (broju 2), te bi praktično zapis `matrica[1,2]` bio ekvivalentan zapisu `matrica[2]`, što je pogrešno, jer je `matrica` dvodimenzionalni niz.

Elementi višedimenzionalnih nizova se u memoriji pamte "jedan za drugim", što u suštini odgovara organizaciji memorije za pamćenje jednodimenzionalnih nizova. Na primjer, dvodimenzionalni niz (matrica) sa m vrsta i n kolona (i koji samim tim ima ukupno $m \cdot n$ elemenata) se u memoriji pamti kao jednodimenzionalni niz dužine $m \cdot n$. Elementi matrice se u memoriji čuvaju "vrsta po vrsta", tj. prvih n elemenata su redom elementi početne vrste, narednih n elemenata su redom elementi naredne vrste itd.

Tako će element matrice `matrica[i][j]` u memoriji biti na mjestu `k=i*n+j`, tj. prije njega će biti svi elementi iz svih vrsta prije `i`-te i još elementi iz `i`-te vrste koji su prije `j`-te kolone. I obrnuto, ako se jednodimenzionalni niz elemenata posmatra kao prostor za smještanje matrice `matrica` dimenzije $m \cdot n$, onda k-tom elementu niza odgovara element matrice `matrica[k/n][k%n]`. Slične tehnike se mogu primijeniti i prilikom izračunavanja odgovarajućih elemenata višedimenzionalnih nizova i jednodimenzionalnog niza.

Elektronska verzija

Činjenica da se svaka matrica (ali i višedimenzionalni nizovi) može posmatrati kao jednodimenzionalni niz, se može iskoristiti u nekim praktičnim situacijama. Ove situacije srećemo kada dimenzija matrice nije unaprijed poznata, već se memorija za višedimenzionalne nizove treba alocirati dinamički. O dinamičkoj alokaciji matrica će biti više govora u Poglavlju 12.

Inicijalizacija početnih vrijednosti elemenata višedimenzionalnog niza se može izvršiti na nekoliko načina: listom inicijalnih vrijednosti nizova niže dimenzije, ili prosto listom pojedinačnih elemenata. Na našem primjeru dvodimenzionalnog niza, inicijalizacija elemenata bi mogla da se uradi na jedan od tri načina:

```
int matrica[2][3] = {{1,2,3},{4,5,6}};
int matrica[2][3] = {1,2,3,4,5,6};
int matrica[][3] = {{1,2,3},{4,5,6}};
```

Višedimenzionalni nizovi takođe mogu da budu argumenti funkcija. Način deklaracije višedimenzionalnog niza kao argumenta funkcije se realizuje tako što se on deklariše sa svim svojim dimenzijama, osim eventualno prve. Na primjer, funkcija koja za argument uzima cjelobroju matricu sa 2 reda i 3 kolone bi mogla biti deklarisana ovako:

```
tip funkcija(int mat[2][3], int m, int n);
```

ili

```
tip funkcija(int mat[][3], int m, int n);
```

Najprije uradimo najjednostavniji primjer.

Primjer 6.11. Sa tastature se unose dimenzije matrice m i n, gdje je m broj vrsta, a n broj kolona. Nakon toga, formirati matricu datih dimenzija, unijeti elemente matrice sa tastature i ispisati matricu na ekranu u odgovarajućoj formi. Maksimalan broj vrsta i kolona je 20.

Unos i ispis matrice ćemo realizovati pomoću odgovarajućih funkcija.

```
1  #include<stdio.h>
2  #define MAX 20
3  void unos(int mat[MAX][MAX], int m, int n){
4     int i, j;
5     for(i = 0; i < m; i++)
6        for(j = 0; j < n; j++)
7        {
8           printf("Unesite A[%d][%d] element matrice: ",i,j);
9           scanf("%d",&mat[i][j]);
```

Elektronska verzija

```
10       }
11  }
12  void ispis(int mat[MAX][MAX], int m, int n){
13     int i, j;
14     printf("Matrica je:\n");
15     for(i = 0; i < m; i++)
16     {
17        for(j = 0; j < n; j++)
18           printf("%d\t",mat[i][j]);
19        printf("\n");
20     }
21  }
22  int main()
23  {
24     int m, n;
25     printf("Unesite broj vrsta i kolona matrice:\n");
26     scanf("%d %d",&m,&n);
27     int A[m][n];
28     unos(A,m,n);
29     ispis(A,m,n);
30     return 0;
31  }
```

Ove funkcije za unos i ispis matrice su prilično “univerzalne” i mogu se koristiti i u drugim zadacima. Uradimo još nekoliko primjera sa matricama.

Primjer 6.12. Zaključne ocjene učenika jednog odjeljenja date su matricom $A_{n\times m}$, $n \leq 50$, $m \leq 15$, tako da je $A[i][j]$ ocjena i-tog učenika iz predmeta j. Napisati program kojim se prvo unose ocjene učenika, a zatim se za svakog učenika ispisuje prosječna ocjena i za svaki predmet prosječna ocjena iz tog predmeta.

```
1   #include<stdio.h>
2
3   int main(){
4       int n;
5       int m;
6
7       printf("Unesite broj ucenika u odjeljenju:\n");
8       scanf("%d",&n);
9
10      printf("Unesite broj predmeta:\n");
11      scanf("%d",&m);
12
```

Elektronska verzija

```
13      int A [n][m];
14
15      int i, j;
16      for(i = 0; i < n; i++)
17          for(j = 0; j < m; j++){
18              printf("Unesite ocjenu ucenika %d iz predmeta
                    %d:\n",(i+1),(j+1));
19              scanf("%d",&A[i][j]);
20          }
21
22      for(i = 0; i < n; i++){
23          int suma = 0;
24          for(j = 0; j < m; j++)
25              suma += A[i][j];
26          float prosjek = (suma + 0.0) / m;//imlicitno konvertujemo
                podatke u float
27          printf("Prosjek ocjena ucenika %d je %f\n",(i+1),prosjek);
28      }
29
30      for(j = 0; j < m; j++){
31          int suma = 0;
32          for(i = 0; i < n; i++)
33              suma += A[i][j];
34          float prosjek = (suma + 0.0) / n;//imlicitno konvertujemo
                podatke u float
35          printf("Prosjek ocjena iz predmeta %d je %f\n",(j+1),prosjek);
36      }
37
38      return 0;
39  }
```

Primjer 6.13. Data je matrica realnih brojeva $A_{n \times n}$, $n \leq 100$. Napisati program koji u matrici A određuje minimalni element na sporednoj dijagonali.

```
1   #include<stdio.h>
2
3   int main(){
4       int n;
5
6       printf("Unesite broj vrsta i kolona matrice:\n");
7       scanf("%d",&n);
8
9       float A [n][n];
10
```

Elektronska verzija

```
11      int i, j;
12      for(i = 0; i < n; i++)
13          for(j = 0; j < n; j++){
14              printf("Unesite A[%d][%d] element matrice:\n",(i+1),(j+1));
15              scanf("%f",&A[i][j]);
16          }
17
18      printf("Matrica je:\n");
19      for(i = 0; i < n; i++){
20          for(j = 0; j < n; j++)
21              printf("%f\t",A[i][j]);
22          printf("\n");
23      }
24
25      float m = A[0][n-1];
26      for(i = 1; i < n; i++)
27          if(A[i][n-i-1] < m)
28              m = A[i][n-i-1];
29
30      printf("Minimalni element na sporednoj dijagonali je %f\n",m);
31
32      return 0;
33  }
```

Primjer 6.14. Na prijemu je n osoba, $n \leq 100$. Data je matrica $A_{n\times n}$ tako da je $A[i][j] = 1$, ako osoba i poznaje osobu j, inače 0. Napisati program koji određuje redni broj osobe koja poznaje najviše osoba.

```
1   #include <stdio.h>
2
3   int main(){
4       int n;
5       printf("Unesite broj vrsta i kolona matrice:\n");
6       scanf("%d",&n);
7
8       int A [n][n];
9
10      int i, j;
11      for(i = 0; i < n; i++)
12          for(j = 0; j < n; j++){
13              printf("Unesite A[%d][%d] element matrice:\n",(i+1),(j+1));
14              scanf("%d",&A[i][j]);
15          }
16
```

Elektronska verzija

```
17      printf("Matrica je:\n");
18      for(i = 0; i < n; i++){
19          for(j = 0; j < n; j++)
20              printf("%d\t",A[i][j]);
21          printf("\n");
22      }
23
24      int koja_poznaje = 0;
25      int broj_osoba = 0;
26      //prvo izbrojimo za prvu osobu
27      for(j = 0; j < n; j++)
28          if(A[0][j] == 1)
29              broj_osoba++;
30      //sad razmatramo ostale osobe
31      for(i = 1; i < n; i++){
32          int poznaje_i = 0;
33          for(j = 0; j < n; j++)//brojimo koliko osoba poznaje osoba i
34              if(A[i][j] == 1)
35                  poznaje_i++;
36          if(poznaje_i > broj_osoba){
37              koja_poznaje = i;
38              broj_osoba = poznaje_i;
39          }
40      }
41
42      printf("%d. osoba poznaje najvise osoba na
            prijemu.\n",(koja_poznaje+1));
43
44      return 0;
45
46  }
```

6.4 Niske karaktera

Niske karaktera, ili kraće samo niske, (u našem vokabularu koristimo i englesku riječ string) su podaci koji se izuzetno često koriste u programiranju, praktično u svakom programu. Niskama predstavljamo riječi i rečenice iz svakodnevnog govora. Uobičajeno je da se komunikacija između programa i korisnika odvija preko ulaznih i izlaznih niski. Niske koristimo za označavanje putanja ka fajlovima i folderima, putanja ka serverima sa kojima program može da komunicira itd. Zbog velike potrebe da se manipulacija niskama odvija na što efikasniji način, u svim modernim programskim jezicima ovom tipu podatka je posvećena

Elektronska verzija

posebna pažnja i obezbijeđeni su posebni mehanizmi za rad sa njima.

Kao što je već viđeno u mnogim primjerima do sada, konstantne niske se prave tako što se između dvostrukih navodnika navode karakteri koji formiraju nisku. Na primjer, `"Ovo je jedna niska"`. Pored "običnih" karaktera, u okviru niske možemo da navodimo (a i to smo do sada radili mnogo puta) i posebne sekvence karaktera (na primjer, sekvence koje formatiraju ispis: `%d`,`%c`..., sekvence koje označavaju specijalne karaktere: `\n`, `\t` itd.).

Svaka niska se interno u programu predstavlja nizom karaktera, koji se završava tzv. terminalnim karakterom `'\0'`, koji se još zove i završna, terminalna nula (engl. null terminator). Kao posljedicu ovakvog zapisa niske u memoriji, imamo da niske mogu biti proizvoljne dužine, dok se sama informacija o dužini niske ne čuva (ali se može izračunati prolaskom kroz čitavu nisku). Iako je terminalna nula sastavni dio niske, treba napomenuti da se ona "ne ubraja" u karaktere, kada je riječ o praktičnoj upotrebi same niske. Tako ćemo, na primjer, za nisku `"tabla"` reći da se ona sastoji od 5 karaktera (to su karakteri `'t'`, `'a'`, `'b'`, `'l'` i `'a'`) i njena dužina je jednaka 5, iako mi znamo da je u memoriji, da bi se sačuvala ta niska zauzeto ukupno 6 mjesta (5 za svaki od karaktera same niske plus šesti za terminalnu nulu).

Treba razlikovati pojedinačne karaktere od niski koje formiramo na osnovu samo jednog karaktera. Tako su `'A'` i `"A"` dva različita objekta: u prvom slučaju riječ je o jednom pojedinačnom karakteru (slovu A), dok je u drugom slučaju riječ o niski koja se formira od dva znaka: prvi znak je veliko slovo A, dok je drugi znak terminalna nula. Slično, treba razlikovati sljedeće konstante: `1`, `'1'` i `"1"`. U prvom slučaju riječ je o konstanti koja je broj (to je podatak tipa `int`), u drugom slučaju riječ je o karakteru koji je cifra (ne broj koji ima vrijednost 1, već karakter koji na osnovu rednog broja u ASCII kôdu ima vrijednost 49), dok je u trećem slučaju riječ o niski koja je formirana od dva karaktera: prvi karakter je cifra 1, dok je drugi karakter terminalna nula. Prazan niz znakova (prazna niska) `""` sadrži samo terminalnu nulu. Dužina prazne niske je nula.

Iako su sličnog naziva i slične strukture, a u velikom broju slučajeva služe i sličnoj svrsi, treba razlikovati nizove karaktera od niski. Niz karaktera je struktura koja je prvenstveno niz čiji su elementi karakteri, dok nisku čine karakteri koji slijede jedan za drugim (što je naravno slično kao kad kažemo da je riječ o nizu karaktera) i koja ima još jedan dodatni karakter, koji je jednak terminalnoj nuli. Ipak, u mnogim situacijama nizove karaktera i niske možemo koristiti skoro pa ravnopravno, o čemu će posebno biti riječi kada se nizovi uvedu u zajednički okvir sa pokazivačima.

Niske možemo koristiti i za inicijalizaciju elemenata niza karaktera. Evo nekih karakterističnih primjera, koji ilustruju sličnosti i razlike između nizova i

Elektronska verzija

niski.

```
char niz1[] = {'t', 'a', 'b', 'l', 'a'};
char niz2[] = "tabla";
```

Deklaracija niza `niz1` je poznata i njome je deklarisan niz koji ima pet elemenata. Drugom deklaracijom je određen niz koji ima 6 elemenata: prvih pet elemenata su redom karakteri riječi "tabla", dok je posljednji, šesti element ovog niza karakter terminalna nula. Dakle, niz karaktera koje sadrži `niz1` se ne može smatrati ispravnom niskom, jer se ne završava terminalnom nulom. Ako bismo niz htjeli da zasnujemo tako da njegovi karakteri mogu da formiraju nisku, tada bismo na našem primjeru to uradili ovako:

```
char niz3[] = {'t', 'a', 'b', 'l', 'a','\0'};
```

Dvije niske koje se u programu nalaze zapisane jedna neposredno uz drugu se spajaju. Tako je na primjer zapis,

```
"Ovo je jedna niska" " koja se nastavlja na drugu"
```

jednak zapisu

```
"Ovo je jedna niska koja se nastavlja na drugu"
```

6.5 Pitanja i zadaci

1. Šta se ispisuje na ekranu sljedećim kôdom?

```
char niz[10] = {'a', 'b'};
for(i = 0; i < 10; i++)
    printf("%c\t",niz[i]);
printf("\n");
```

2. Neka su zadata dva cjelobrojna niza `a` i `b` na sljedeći način

```
int a[6] = {1, 2};
int b[10] = {1, 2, 3, 4, 5, 6};
```

i neka su zadate naredbe dodjele

```
int i = 1;
while(i < 5){
```

Elektronska verzija

```
    a[i] = b[i+1] + 1;
    b[2*i] = 2 * a[i];
    i++;
}
```

Koji elementi su sadržani u nizovima `a` i `b` nakon izvršenja datih naredbi?

3. Da li je moguće u programskom jeziku C izvršiti sljedeće naredbe? Obrazloži odgovore.

```
int a[5] = {1, 2, 3, 3, 2};
int b[5];
a = b;
a--;
```

4. Napisati funkciju koja na ekranu ispisuje samo elemente realnog niza koji se nalaze na neparnim pozicijama. Testirati funkciju u glavnom dijelu programa na nizu koji se unosi sa tastature.

5. Napisati program koji broji pojavljivanje svakog karaktera koji je malo slovo na standardnom ulazu.

6. Napisati program koji iz unesenog cjelobrojnog niza izbacuje elemente niza čija je apsolutna vrijednost manja od apsolutne vrijednosti broja x koji se unosi sa tastature.

7. Napisati program koji određuje najmanji element cjelobrojnog niza i poziciju na kojoj se najmanji element pojavljuje u nizu. Ako se najmanji element pojavljuje više puta onda pronaći poziciju njegovog posljednjeg pojavljivanja.

8. Napisati program koji na osnovu dva cjelobrojna niza `a` i `b` jednakih dužina, formira niz `c` koji na poziciji takav da je `c[i]=max{a[i],b[i]}` za `0<=i<n`.

9. Napisati program koji na osnovu dva realna niza, duzina m i n, koji se unose sa tastature, formira novi niz koji sadrži samo one elemente, koji se nalaze u oba unesena niza.

10. Napisati program koji transformiše elemente cjelobrojnog niza na sljedeći način:

    a) ako se element nalazi na parnoj poziciji, onda se mijenja svojom najmanjom cifrom;

Elektronska verzija

b) ako se element nalazi na neparnoj poziciji, onda se mijenja svojom najvećom cifrom.

11. Napisati program kojim se unose cijeli brojevi sa tastature, dok se ne unese nula, ili dok se ne unese ukupno 20 brojeva. Nakon završenog unosa, odrediti vrijednost koja se računa tako sto se od posljednjeg unesnog elementa niza oduzme pretposljednja unesena vrijednost, zatim se dobijenoj razlici doda vrijednost elementa koji je unesen prije pretposljednjeg i tako se, naizmjenično, oduzimaju i sabiraju elementi niza, dok se ne dođe do prvog elementa niza.

12. Napisati program koji pronalazi broj pojavljivanja karaktera `c`, koji se unosi sa tastature, u nizu karaktera, koji se, takođe, unosi sa tastature.

13. Napisati program koji iz datog niza cijelih brojeva izbacuje sve duplikate.

14. Za dva cijela broja kažemo da su relaciji ako se sastoje od istih cifara, npr. brojevi 1223 i 321123 su u relaciji. Napisati funkciju koja za argument uzima dva cijela broja, a kao rezultat vraća 1, ako su brojevi u relaciji, u suprotnom 0.

15. Napisati program koji u nizu brojeva pronalazi dva broja koja su na najmanjem rastojanju. Program testirati na nizu realnih brojeva dužine n. Broj n i niz se unose sa tastature, a kao rezultat programa ispisuju se brojevi koji se nalaze na najmanjem rastojanju, kao i vrijednost tog rastojanja.

16. Napisati program koji za niz prosječnih ocjena učenika nekog odjeljenja (indeks u nizu odgovara rednom broju u dnevniku) određuje redni broj učenika, čija je srednja ocjena najbliža prosječnoj. Ako je više takvih učenika ispisati najmanji broj.

17. Napisati program kojim se na osnovu matrice cijelih brojeva A, formata $n \times n$, formira niz $b[0], ..., b[n-1]$, čiji elementi su redom jednaki razlici najmanjih i najvećih elemenata u vrsti.

18. Napisati program koji za cjelobrojnu matricu A formata $m \times n$ formira matricu istog formata u kojoj je element na poziciji `[i][j]` jednak aritmetičkoj sredini susjednih elemenata elementa `A[i][j]`.

19. Napisati program koji ispituje da li je cjelobrojna matrica ortonormirana. Matrica je ortonormirana, ako je skalarni proizvod svakog para različitih vrsta jednak 0, a skalarni proizvod vrste sa samom sobom jednak 1.

Elektronska verzija

20. Za kvadratnu matricu A formata $n \times n$ kažemo da je dijagonalno dominantna ako za svako i vrijedi

$$\sum_{j \neq i} | A[i][j] | < A[i][i].$$

Napisati funkciju d, koja za argument uzima kvadratnu matricu i vraća informaciju da li je dijagonalno dominantna.

Elektronska verzija

Elektronska verzija

7 Doseg i vijek trajanja promjenljivih

Kao što smo do sada vidjeli, u programskom jeziku C sve promjenljive, a to važi i za druge identifikatore, moramo deklarisati prije upotrebe. Već smo do sada usvojili da deklaracijom promjenljivih određujemo njen tip i ime, a eventualno joj dodjeljujemo i početnu vrijednost. Pored toga, deklaracijom definišemo još dva važna svojstva: doseg i vijek trajanja. I ova dva svojstva su vezana za sve identifikatore, ali ćemo se u ovom poglavlju najviše fokusirati na doseg i vijek trajanja promjenljivih. *Doseg* ili *oblast važenja* (engl. scope) promjenljive je dio programa u kojem je njena deklaracija vidljiva i u kojem se toj promjenljivoj može pristupiti putem njenog imena. *Životni vijek* promjenljive ili *vijek trajanja* (engl. lifetime) promjenljive je dio vremena izvršavanja programa u kojem se garantuje da je za tu promjenljivu rezervisan odgovarajući dio memorije i da se ta promjenljiva može koristiti.

Za razliku od tipa promjenljive, koji se uvijek eksplicitno određuje nazivom tipa (`int`, `double` i sl.), doseg i životni vijek promjenljive su određeni položajem deklaracije u programu, a mogu se i dodatno mijenjati ključnim riječima `static` ili `extern`.

Iako bi ovo poglavlje moglo biti i u sastavu nekih drugih, smatramo da je na koncepte koji se tiču dosega i životnog vijeka potrebno staviti dodatni naglasak, jer oni predstavljaju nadogradnju na sadržaje prethodno iznesene u udžbeniku.

Najprije se detaljno upoznajmo sa dosegom promjenljivih.

7.1 Doseg promjenljivih

Da bi se izbjegla zavisnost između dijelova programa tamo gdje to nije neophodno, ali i izbjegao neočekivan ili neželjen uticaj parametara, koji su uvedeni u jednom dijelu programa, na druge dijelove, poželjno je da se promjenljive i drugi identifikatori deklarišu upravo tako, da oblast njihovog važenja (doseg) ne prevazilazi onaj okvir u kome se ta promjenljiva zaista koristi. Konkretno, ako se neka promjenljiva koristi samo unutar neke funkcije, onda tu promjenljivu treba i deklarisati na način da ona i bude "vidljiva" samo u okviru te funkcije, a ne i van nje. Može se primjenjivati i još striktniji pristup, koji podrazumijeva da se promjenljive deklarišu u još užim oblastima (zapravo blokovima), ukoliko je potreba za njihovim postojanjem ograničena isključivo na taj blok. Na

Elektronska verzija

taj način se, uz poštovanje i mnogih drugih principa, omogućava efekat modularnosti programa. Modularnost podrazumijeva da se program sastoji od više međusobno nezavisnih dijelova, koji se po potrebi kasnije "uklapaju" u jednu cjelinu, tako što se dijelovi (moduli) pozivaju sukcesivno jedan za drugim, ili jedan unutar drugog.

Ipak, u nekim situacijama je opravdano i da različiti dijelovi programa koriste zajedničke promjenljive, jer se, na taj način, može postići ušteda u memorijskom prostoru i procesorskom vremenu, ili se implementacija značajno pojednostavljuje. Stoga je važno razumjeti mehanizme koji određuju doseg promjenljivih i prepoznati situacije kada je potrebno, ili poželjno, voditi računa o oblasti važenja.

U programskom jeziku C postoje dva najznačajnija tipa dosega promjenjivih: doseg nivoa datoteke (engl. file level scope) i doseg nivoa bloka (engl. block level scope). Kako i sami nazivi ovih dosega ukazuju, doseg nivoa datoteke podrazumijeva da ime promjenljive važi od mjesta u programu gdje je ona uvedena, pa do kraja datoteke. Doseg nivoa bloka ograničava prostor važenja promjenljive na blok, tj. ime promjenljive važi od mjesta u bloku u kome je uvedena, pa do kraja tog bloka. Pored dosega na nivou datoteke i na nivou bloka, kasnije u ovom poglavlju ćemo vidjeti da promjenljive mogu imati doseg i u više datoteka.

Promjenljive koje su definisane na nivou datoteke nazivamo *globalne*, dok su promjenljive definisane u okviru bloka *lokalne*. Pored toga, kako se blokovi mogu nalaziti unutar drugih blokova, za promjenljive koje su lokalne za jedan (spoljašnji) blok mogu postati "globalne" za sve unutrašnje blokove, koji se nalaze unutar njega. Ipak, u ovom slučaju smo tu riječ stavili pod navodnike, jer ćemo u daljem tekstu pod globalnim promjenljivima podrazumijevati one koje su definisane na nivou datoteke.

7.1.1 Lokalne promjenljive

Lokalne promjenljive su promjenljive koje su deklarisane unutar bloka i kao što smo već rekli, njihov doseg je samo unutar tog bloka. Izvan tog bloka može postojati promjenljiva istog imena. Kroz nekoliko primjera razmotrimo različite situacije koje se mogu javiti.

Primjer 7.1. Najjednostavnija situacija koja se može javiti je sljedeća. Promjenljiva `x` je definisana

```
...
if(1){
   int x = 10;
```

Elektronska verzija

```
    printf("%d\n",x++);
}
x = 15; //greska, x vazi samo u prethodnom bloku
...
```

Međutim, isto ime x možemo koristiti van bloka i definisati promjenljivu tog imena.

Primjer 7.2. U sljedećem prikazu x je definisano nakon bloka i stvari su jasne.

```
...
if(1){
    int x = 10;
    printf("%d\n",x++);
}
int x = 15; //ispravno, x je dozvoljeno ime
...
```

Može se desiti da u jednom trenutku postoji i više promjenljivih istog imena. Ako su njihovi dosezi jedan u okviru drugog, tada promjenljiva u "užoj" oblasti skriva promjenljivu iz šire.

Primjer 7.3. Posmatrajmo situaciju kada je promjenljiva x definisana prije bloka, a onda potom i u bloku koji slijedi. U ovom, pomalo "vještačkom" primjeru vidimo kako se unutar bloka **while** petlje naizmjenično koriste dvije promjenljive istog imena. Promjenljiva x koja se koristi u prvoj naredbi printf je "vanjska" za taj blok, dok je druga promjenljiva x kreirana lokalno u bloku.

```
...
int brojac = 0;
int x = 15;

while (brojac++ < 3){
    printf("%d\n",x++);
    int x = 10;
    printf("%d\n",x++);
}
...
```

Po završetku ove **while** naredbe, na ekranu će biti ispisane vrijednosti:

```
15
10
```

Elektronska verzija

```
16
10
17
10
```

Vrijednost prve promjenljive `x` se iz iteracije u iteraciju povećava, jer je njen okvir važenja i van bloka **`while`** petlje. Sa druge strane, druga promjenljiva `x` ima okvir važenja samo unutar bloka, tj. završetkom bloka ona prestaje da važi i njeno uvećanje za jedan nema uticaja po završetku bloka. Ulaskom u novu iteraciju, druga promjenljiva `x` se definiše "ispočetka" i ta nova promjenljiva `x` ponovo dobija vrijednost 10, koju mi vidimo ispisanu na ekranu.

Primijetimo još i to da je po definisanju lokalne promjenljive istog imena (u našem slučaju druge promjenljive `x`), prva promjenljiva (kod nas je to prva promjenljiva `x`, koja je prvobitno dobila vrijednost 15), postala nedostupna.

Slična situacija se dešava i kada radimo sa funkcijama. Neke od situacija koje se mogu javiti analizirajmo kroz sljedeći primjer.

Primjer 7.4. Posmatrajmo sada tri cjelobrojne promjenljive, nakon koje slijedi definicija funkcije.

```
int a, b, c;
...
void f(int a) {
    int b;
    c = a + b;
...
}
```

Formalni argument funkcije (argument `a`) je vidljiv samo unutar funkcije. Isto važi i za lokalnu promjenljivu `b`. Time smo promjenljive `a` i `b`, koje su definisane u prvom redu prikaza "sakrili" i njih ne možemo koristiti. Za razliku od njih, promjenljiva `c` nije sakrivena lokalnom promjenljivom, te se ona može koristiti u okviru funkcije.

Treba napomenuti da su, za razliku od promjenljivih, funkcije u najvećem broju slučajeva globalne, tj. one se uglavnom i definišu sa ciljem da budu vidljive u svakom dijelu programa (da bi se tako mogle i koristiti). Prije standarda C99, funkcije nisu ni mogle biti definisane lokalno (na primjer, unutar druge funkcije), ali je pomenutim standardom i to omogućeno. Slično kao i kod promjenljivih, ako je funkcija deklarisana globalno, na nivou datoteke, onda se njen doseg proteže od mjesta deklaracije, pa do kraja datoteke.

Elektronska verzija

Primjer 7.5. Ilustrujmo “vidljivost” funkcije, koja je definisana unutar druge funkcije, sljedećim kompletnim programom.

```
#include <stdio.h>
int f(){
   //prvo definisemo funkciju unutar funkcije
   int g(){
      return 1234;
   }
   int c = g();//ispravno, funkcija g je vidljiva iz funkcije f
   return c;
}
int main(){

   g(); //greska, funkcija g nije vidljiva iz main funkcije
   int x = f();//ovo je dozvoljeno
   printf("%d\n",x);

   return 0;
}
```

7.1.2 Globalne promjenljive

Promjenljive koje su definisane izvan svih blokova nazivamo globalnim promjenljivima. Njihov doseg je od mjesta definisanja do kraja datoteke.

Primjer 7.6. Najjednostavniji primjer prikazuje upotrebu globalne promjenljive u funkcijama.

```
#include <stdio.h>
    int x = 10;
    void f() {
        x++; //x je vidljivo u funkciji f
        printf("x=%d\n",x);
    }
    int main(){
        x++; //ovo je sve ista promjenljiva x
        f();
        return 0;
}
```

Promjenljiva x je globalna i dostupna je u svim dijelovima programa, od tačke njenog definisanja. U situacijama kada više funkcija koristi isti podatak,

Elektronska verzija

a izmjene vrijednosti tog podatka unutar jedne funkcije trebaju da se odraze na čitav program i sve ostale funkcije, tada je praktično taj podatak čuvati u globalnoj promjenljivoj. Pošto je globalna promjenljiva dostupna unutar svih funkcija, onda u funkcijama nije potrebno uvoditi formalni argument, koji bi "prihvatao" taj podatak, a izmjena podatka unutar jedne funkcije se, naravno, odražava na ostatak programa.

Primjer 7.7. Jednostavnim primjerom ilustrujmo upotrebu globalne promjenljive koja je niz.

```
1  #include <stdio.h>
2  int niz[10];
3  void unos(){
4     int i;
5     printf("Unos niza...\n");
6     for (i = 0; i < 10; i++)
7        scanf("%d",&niz[i]);
8  }
9  void ispis(){
10    int i;
11    printf("Ispis niza...\n");
12    for (i = 0; i < 10; i++)
13       printf("%d\t",niz[i]);
14    printf("\n");
15 }
16 void dupliraj(){
17    int i;
18    printf("Dupliranje elemenata niza...\n");
19    for (i = 0; i < 10; i++)
20       niz[i] *= 2;
21     }
22 int main(){
23    unos();
24    dupliraj();
25    ispis();
26
27    return 0;
28 }
```

Elektronska verzija

7.2 Vijek trajanja promjenljivih

Ponovimo još jednom da se pod vijekom trajanja promjenljive podrazumijeva vrijeme, za koje se garantuje da je za tu promjenljivu rezervisan odgovarajući memorijski prostor i da je promjenljiva dostupna preko svog imena. Prema vijeku trajanja, promjenljive možemo podijeliti na automatske, statičke i dinamičke. Automatske promjeljive su promjenljive koje su definisane unutar nekog bloka, tj. unutar neke funkcije. Ovdje ćemo isključiti promjenljive koje su kreirane ključnom riječju **static**, o čemu ćemo više reći u nastavku ovog poglavlja. Većina promjenljivih koje smo do sada koristili su automatske promjenljive. Statičke promjenljive su promjenljive čiji životni vijek znatno "duži" od životnog vijeka automatskih promjenljivih, tj. njihov životni vijek traje do završetka programa. Dinamičke promjenljive se kreiraju i "uništavaju" na zahtjev korisnika, eksplicitnim naredbama za alokaciju i brisanje memorije.

U nastavku ćemo se fokusirati na automatske i statičke promjenljive, dok će dinamičke promjenljive posebno biti objašnjene u Poglavlju 12.

7.2.1 Automatske promjenljive

Automatske promjenljive su promjenljive koje se najviše koriste. Kreiraju se u okviru blokova naredbi i njihov životni vijek traje od trenutka njihovog definisanja, pa do kraja bloka. Automatske promjenljive istog imena, koje su definisane u okviru različitih blokova nisu ni u kakvoj vezi, u šta smo se kroz dosadašnje primjere mogli i uvjeriti.

Primjer 7.8. Posmatrajmo dio kôda programa, koji je vrlo sličan kôdu iz Primjera 7.3. Primijetimo da smo sada promijenili naziv promjenljive u okviru bloka **while** petlje, jer nam sada nije važno da "sakrijemo" gornju promjenljivu `x`, već da analiziramo životni vijek promjenljivih (posebno promjenljivih `x` i `y`).

```
{
...
int brojac = 0;
int x = 15;

while (brojac++ < 3){
   printf("%d\n",x++);
   int y = 10;
   printf("%d\n",y++);
}
printf("%d\n",x); //zivotni vijek promjenljive x jos uvijek traje
}
```

Elektronska verzija

Posmatrajmo ispis vrijednosti na ekranu.

```
15
10
16
10
17
10
18
```

Sve promjenljive u ovom primjeru su automatske i svaka ima svoj blok, u kome ima svoj životni vijek. Promjenljive `brojac` i `x` su definisane u okviru nekog šireg bloka (to je blok koji je smješten u okviru skroz gornje i skroz donje vitičaste zagrade), a promjenljiva `y` je definisana u okviru bloka **`while`** petlje. Zaključujemo da promjenljivima `brojac` i `x` životni vijek počinje prije **`while`** petlje, traje dok se **`while`** petlja izvršava (a izvršava se tri puta), a traje i po završetku petlje, sve do završetka čitavog bloka. Za razliku od te dvije promjenljive, životni vijek promjenljive `y` traje znatno kraće, zapravo traje od mjesta definisanja (kada `y` dobije vrijednost 10), pa do kraja bloka **`while`** petlje. U narednoj iteraciji, stara promjenljiva `y`, koja se u međuvremenu i uvećala za jedan, više ne postoji, a kreira se nova promjenljiva `y`, koja ponovo dobija vrijednost 10.

Slična situacija je i sa promjenljivima koje su argumenti funkcije ili lokalne promjenljive (a koje nisu statičke) unutar funkcija.

Primjer 7.9. Posmatrajmo funkciju

```
int f(int x) {
    char c;
    static int y;
}
```

Promjenljive `x` i `c` su automatske i njihov životni vijek traje od jednog poziva funkcije, do završetka funkcije. Promjenljiva `y` je statička promjenljiva (ne automatska).

Podsjetimo se i da, ako se promjenljive koje smo mahom do sada koristili (a to su upravo automatske promjenljive) ne inicijalizuju prilikom uvođenja, one dobijaju vrijednost koju ne možemo predvidjeti. Najčešće je to vrijednost koja se zatekla na pridruženoj memorijskoj lokaciji.

Pomenimo i da ključna riječ **`auto`**, koja se, kao i drugi kvalifikatori memorijskih klasa, koristi ispred deklaracije promjenljive.

Elektronska verzija

```
auto tip ime = vrijednost;
```

Za upotebu kvalifikatora `auto` nema posebnog razloga osim iskazivanja namjere programera, jer je podrazumijevano da će promjenljive, ukoliko nije drugačije naznačeno, biti okarakterisane kao automatske. Stoga se ovaj kvalifikator, praktično, i ne koristi.

Pomenimo ovdje još jedan kvalifikator memorijskih klasa koji se rijetko koristi - kvalifikator `register`. Ovaj kvalifikator se koristi na sličan način kao i `auto`,

```
register tip ime = vrijednost;
```

Njime se prevodiocu sugeriše da promjenljivu alocira u registru procesora. Međutim, pošto moderni prevodioci u fazi optimizacije kôda veoma uspješno određuju koje promjenljive ima smisla čuvati u registrima, ovaj kvalifikator se rijetko koristi, te ga ni mi dalje nećemo analizirati.

7.2.2 Statičke promjenljive

Podsjetimo da su statičke promjenljive one čiji životni vijek traje od tačke njihovog definisanja, pa do završetka programa. Razlikujemo *globalne* statičke promjenljive i *lokalne* statičke promjenljive.

Globalne statičke promjenljive

Globalne promjenljive, koje smo uveli u sekciji 7.1.2 imaju statički životni vijek, jer je memorijski prostor za čuvanje globalnih promjenljivih rezervisan sve do završetka programa. Ove promjenljive se čuvaju u **segmentu podataka**, a ukoliko im se prilikom definisanja ne dodijeli vrijednost, podrazumijevano se inicijalizuju na nulu. Već smo pomenuli da je u globalnim promjenljivima praktično čuvati informacije koje se koriste na više različitih mjesta u programu (na primjer, u nekoliko funkcija). Pošto su globalne promjenljive "vidljive" na svim mjestima u programu, onda za njih nije potrebno obezbjeđivati odgovarajuće formalne parametre unutar funkcija, već se one u funkcijama mogu koristiti direktno, preko imena. Sa druge strane, zbog lakšeg održavanja bolje preglednosti programa, globalne promjenljive treba koristiti samo onda kada za to postoji baš dobar razlog.

Ovdje samo pomenimo (a u Sekciji 7.3.1 ćemo o tome detaljnije) da se i na globalne promjenljive, takođe, može primijeniti kvalifikator `static`, čime utičemo na doseg promjenljive kao vanjskog simbola.

Elektronska verzija

Lokalne statičke promjenljive

Koncept lokalnih statičkih promjenljivih nije teško razumjeti. To su lokalne promjenljive (znači definisane u okviru nekog bloka), kojima je pridružen kvalifikator `static`. Time je određeno da su one dostupne samo u tom bloku, a njihov životni vijek traje tokom čitavog izvršenja programa. Lokalna statička promjenljiva se inicijalizuje samo jednom i to prilikom prvog ulaska programa u dati blok. Ukoliko nije navedena inicijalna vrijednost, podrazumijeva se da će statička promjenljiva dobiti vrijednost nula. Kako smo već rekli, po izlasku iz bloka, promjenljiva se ne "uništava", već ona ostaje "živa" sa vrijednošću koju je imala prilikom izlaska programa iz bloka. Ovaj mehanizam nam omogućava, na primjer, da istu promjenljivu (isti memorijski prostor) koristimo za različite pozive iste funkcije.

Primjer 7.10. Jedan od najjednostavnijih primjera upotrebe lokalne statičke promjenljive je brojanje broja poziva funkcije.

```
1  #include <stdio.h>
2  void f(){
3     static int brojac = 1;
4  
5     printf("%d. poziv funkcije.\n",brojac);
6     brojac++;
7  
8  }
9  int main(){
10    int i;
11    for(i = 0; i < 5; i++)
12       f();
13    return 0;
14 }
```

Mehanizam statičkih promjenljivih se može koristiti i za rješavanje problema gdje je korisno pamtiti prethodne rezultate poziva funkcija.

Primjer 7.11. Podsjetimo se Fibonačijevog niza i rekurzivne funkcije kojom je ona definisana. Ako nam treba funkcija koja redom računa elemente Fibonačijevog niza, možemo iskoristiti mehanizam statičkih promjenljivih. U dvije statičke promjenljive čuvamo vrijednosti dva prethodna elementa Fibonačijevog niza, koje koristimo u novom pozivu funkcije za računanje narednog elementa.

Elektronska verzija

```
#include <stdio.h>
long fibonacci(int i)
{
    static long f1 = 1, f2 = 1;
    long f;
    f = (i < 3) ? 1 : f1 + f2;
    f2 = f1;
    f1 = f;
    return f;
}
int main(){
   int i;
   for(i = 1; i < 10; i++)
   printf("%l: %d\n",fibonacci(i));

   return 0;
}
```

7.3 Organizacija programa u više datoteka

Uvedimo sada još jedan novi pristup koji uključuje organizaciju izvornog kôda u više datoteka. Uobičajeno je da se "veliki" programi, koji sadrže po nekoliko stotina, pa i hiljada redova izvornog kôda "razbijaju" na više manjih dijelova (datoteka), čime se sa jedne strane postiže preglednost i lakša manipulacija kôdom, dok se sa druge strane, što je i važnije, jedan isti izvorni kôd može koristiti u više različitih programa. Organizacija programa u više datoteka sa sobom nosi i nove izazove, koji su, prije svega, vezani za doseg (vidljivost) identifikatora unutar različitih datoteka.

Najprije se treba podsjetiti da se proces prevođenja izvornog kôda sastoji iz tri faze: preprocesiranja, glavne faze prevođenja (koja se često naziva i prevođenje) i povezivanja. Ovdje ćemo se najviše fokusirati na fazu povezivanja, za koju je odgovoran dio prevodioca koji se na engleskom, ali i na našem jeziku zove *linker*.

U fazi povezivanja zadatak linkera je da pronađe i identifikuje simbole koji se koriste u datoteci u kojoj nisu definisani. Praktično, linker "razrješava" pozive funkcija i adrese promjenljivih u svim modulima koji su uključeni u fazu povezivanja (uključujući i module standardne biblioteke) i to tako da "traži" odgovarajuće definicije funkcija i promjenljivih. Iz toga zaključujemo da svaki simbol, koji se koristi u nekoj datoteci, a nije definisan u toj datoteci, mora imati definiciju u nekoj drugoj datoteci i mora biti definisan tako, da je dostu-

Elektronska verzija

pan linkeru. Na primjer, ako se funkcija poziva u datoteci u kojoj nije definisana, onda ona u toj datoteci mora biti deklarisana (mora biti naveden prototip), a linker na osnovu tog prototipa u drugim datotekama traži odgovarajuću definiciju (koja odgovara tom prototipu). Svaki prototip mora imati tačno jednu definiciju koja mu odgovara. Slično je i sa promjenljivima, tj. ako se u datoteci koristi promjenljiva koja nije u njoj i definisana, onda linker na osnovu deklaracije promjenljive traži odgovarajuću definiciju te promjenljive, koja mora da postoji na tačno jednom mjestu. Identifikatore (funkcije i promjenljive) koje pripremamo da budu dostupni u drugim datotekama jednostavno zovemo *vanjski simboli*.

7.3.1 Spoljašnje promjenljive i globalne statičke promjenljive

Povežimo sada, koliko je to moguće, koncepte koje smo do sada pominjali i koji su u vezi za dosegom (lokalni i globalni identifikatori) i životnim vijekom (automatski i statički identifikatori), sa ovim konceptom povezivanja identifikatora između različitih datoteka. Ako radimo sa lokalnim promjenljivim, namjera nam je da one posluže svrsi u odgovarajućem bloku (na primjer u okviru tijela funkcije ili bloka iterativne naredbe), a one i jesu vidljive samo unutar tog bloka. Ako je riječ o automatskim lokalnim promjenljivim, tada njihov životni vijek traje do završetka bloka. Ukoliko lokalnim promjenljivim dodijelimo i kvalifikator `static`, one postaju statičke, tj. produžavamo im životni vijek do kraja programa, ali ne i doseg (one ostaju dostupne samo unutar bloka).

Sa druge strane, kada radimo sa globalnim promjenljivim, tada već imamo situaciju da su takve promjenljive dužeg životnog vijeka (traju do kraja izvršenja programa), a dostupne su i na nivou čitavog programa. Sada uvedimo i koncept *spoljašnje povezanosti*. Mehanizam programskog jezika C omogućava da globalne promjenljive, definisane prije svih blokova (a isto se odnosi i na funkcije), budu spoljašnji simboli, tj. simboli koji su dostupni linkeru i mogu se koristiti u drugim datotekama. Pored globalnih promjenljivih, funkcije koje su definisane iznad svih blokova su takođe spoljašnji simboli i one su, takođe, dostupne linkeru.

```
int a = 15;//globalna promjenljiva koja ima spoljasnju povezanost i
    dostupna je linkeru
int main(void)
{
...
}
```

Praktično, svaka globalna promjenljiva (kao i svaka funkcija definisana iznad

Elektronska verzija

svih blokova) je automatski i spoljašnji simbol i njeno ime je vidljivo linkeru. Ovdje pomenimo da smo u gornjem prikazu dodijelili i vrijednost promjenljivoj a, čime smo je stvarno definisali. U nastavku ovog poglavlja ćemo analizirati i situacije koje mogu nastati ukoliko se prilikom navođenja promjenljive i ne dodijeli inicijalna vrijednost.

Sa druge strane, ako bismo htjeli da promjenljiva (ili funkcija) ostane globalna u datoteci u kojoj je definisana, a da ne bude dostupna linkeru, onda ispred definicije promjenljive koristimo riječ `static`. Na primjer

```
static int a;//a ostaje globalna promjenljiva, ali sada ima unutrasnju
    povezanost i nije dostupna linkeru
int main(void)
{
...
}
```

Promjenljiva a jeste globalna, ali više nije dostupna linkeru i ne može se koristiti u drugim datotekama. Primijetimo da kvalifikator `static` u ovom slučaju ima drugačije značenje, nego kada se koristi kod lokalnih promjenljivih. Kod lokalnih promjenljivih kvalifikatorom `static` utičemo na životni vijek promjenljive, dok kod globalne promjenljive smanjujemo doseg i ograničavamo njenu dostupnost samo na datoteku u kojoj je definisana. Slično je i sa funkcijama. Kažemo da globalni objekat kome je pridružen i kvalifikator `static` ima *unutrašnju povezanost*, kojom se povezuju sva pojavljivanja istog identifikatora na nivou tačno jedne datoteke. Na ovaj način se postiže efekat da je dio programskog kôda "zatvoren" ili "enkapsuliran" (engl. encapsulated) u odnosu na druge dijelove programa, čime se povećava sigurnost kôda i smanjuje mogućnost pojave grešaka.

7.3.2 Pristup funkcijama iz drugih datoteka

Već smo rekli da je za funkciju, za koju hoćemo da bude dostupna iz drugih datoteka, važno da je definišemo tako da ona ima spoljašnju povezanost. To radimo tako što je definišemo iznad svih blokova i ne koristimo riječ `static`. U datoteci iz koje pozivamo funkciju, tu funkciju moramo deklarisati (navesti njen prototip). Prije navođenja prototipa može se (ali kod funkcija i ne mora) koristiti ključna riječ `extern`, koja ukazuje na to da je funkcija spoljašnji simbol i da je poznata linkeru, a da je definisana na nekom drugom mjestu (drugoj datoteci).S obzirom da su sva imena funkcija poznata automatski linkeru, riječ `extern` možemo i izostaviti.

Primjer 7.12. Jedan od najjednostavnijih primjera upotrebe funkcije koja je

Elektronska verzija

definisana u jednoj datoteci, a poziva se u drugoj bi mogao da izgleda ovako:

Prva datoteka

```c
/*datoteka1*/
#include <stdio.h>
void f()//napravimo neku jednostavnu funkciju
{
    printf("Funkcija f...\n");
}
```

Druga datoteka

```c
/*datoteka2*/
void f();//mogli smo pisati i extern void f(); ali ne bi bilo nikakvog
    dodatnog efekta

int main(){

    f();//pozivamo funkciju f

return 0;
}
```

Za potvrdu prethodno razmatranih koncepata spoljašnje i unutrašnje povezanosti, pomenimo sljedeće: da je funkcija `f` u prvoj datoteci bila definisana na sljedeći način:

```c
/*datoteka1*/
#include <stdio.h>
static void f()//ukljucujemo rijec static i f sad ima unutrasnju
    povezanost
{
    printf("Funkcija f...\n");
}
```

poziv funkcije (naredba `f();`) u `main` funkciji u drugoj datoteci ne bi bio ispravan, jer linker ne bi mogao da pronađe definiciju za funkciju `f`.

Da bi se izbjeglo moguće nepodudaranje prototipova i definicija funkcija, uobičajeno je da se sve deklaracije funkcija koje imaju spoljašnju povezanost smještaju u jednu datoteku zaglavlja, koja se kasnije uključuje u sve datoteke u kojima se planira korištenje tih funkcija. Osim što programer na taj način sebi olakšava rad i ne mora dalje da se brine da li je za funkcije koje koristi obezbijeđen pristup i njihovim definicijama, takođe postiže i efekat da se pojava

Elektronska verzija

greške da prototip funkcije nije usklađen sa definicijom može desiti samo na jednom mjestu (u toj datoteci zaglavlja).

7.3.3 Pristup promjenljivima iz drugih datoteka

Iako je koncept prilično sličan kao i u slučaju funkcija, prilikom pristupa promjenljivima iz drugih datoteka se, ipak, mora malo više voditi računa o deklaracijama u odnosu na definicije.

Najprije ponovimo da, ako se jedna promjenljiva planira koristiti u više datoteka, onda ona u tačno jednoj datoteci mora imati definiciju i mora imati vanjsku vidljivost, dok se u datotekama u kojima se samo koristi mora samo deklarisati. Slično, naravno, važi i kod funkcija. Međutim, za razliku od funkcija, kod kojih se deklaracija jasno razlikuje od definicije, kod promjenljivih to nekada nije sasvim jasno. Na primjer, ako napišemo

```
int a = 15;
```

jasno je da je promjenljiva `a` definisana (ne samo deklarisana), jer joj je, pored tipa i imena dodijeljen i memorijski prostor određene veličine (onoliko kolika je širina podatka tipa `int`) i u taj prostor je upisana vrijednost 15. Međutim, ako imamo globalnu promjenljivu `b` i sljedeći zapis

```
int b;
```

onda nije potpuno jasno da li je ovim zapisom izvršena i definicija, ili samo deklaracija. Programski jezik C omogućava da se vrši takozvano *načelno definisanje* (engl. tentative definition), koje postaje stvarno definisanje, ako se do kraja prevođenja ne pojavi definicija koja inicijalizuje vrijednost promjenljive.

Primjer 7.13. Posmatrajmo sljedeći primjer.

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

Elektronska verzija

Prvom naredbom `int a;` vrši se načelno definisanje promjenljive, što znači da se ostavlja mogućnost da se eventualno kasnije javi deklaracija koju prati i inicijalizacija. Slično je i sa drugom naredbom `int a;`. Zapravo, prevodilac "iza scene" kombinuje sve načelne definicije u jednu.

Primjer 7.14. U prethodnom primjeru, načelna definicija je, na kraju, postala i stvarna definicija i promjenljiva `a` je dobila vrijednost 0 (što se može vidjeti i ispisom vrijednosti promjenljive `a` na ekranu). Ako bismo sad imali sljedeći zapis

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

jasno je da načelno definisanje ne bi postalo i stvarno, jer je prevodilac naišao na inicijalizaciju, (`int a=3;`).

Primjer 7.15. U trećoj situaciji dobijamo grešku

```
1  #include <stdio.h>
2
3  int a = 9;
4  int a = 3; //pogresno, inicijalizacija je vec uradjena
5  int main()
6  {
7      printf("%d\n",a);
8
9      return 0;
10 }
```

jer druga naredba `int a=3;` nije druga deklaracija iste promjenljive, već pokušaj njene druge definicije, što nije dozvoljeno.

Da zaključimo, svaka deklaracija promjenljive u kojoj se vrši i inicijalizacija je nužno i njena definicija. Međutim, ako inicijalizacija nije pristuna, onda može doći do zabune koja je, od više deklaracija iste promjenljive ujedno i definicija.

Da se to ne bi dešavalo, koristimo mehanizam koji nam obezbjeđuje ključna riječ `extern`.

Elektronska verzija

Kvalifikator extern i promjenljive

Kada želimo da naglasimo da promjenljiva koju deklarišemo služi za spoljašnju povezanost i da je njena definicija na nekom drugom mjestu (u nekoj drugoj datoteci), koristimo kvalifikator `extern`. Podsjetimo se da smo ovaj kvalifikator već pominjali, kada smo razmatrali pristup funkcijama koje su definisane u nekoj drugoj datoteci. Sa druge strane, kako smo tamo napomenuli, kod funkcija je jasno da se prilikom navođenja samo deklaracije (prototipa) ona ne definiše, te se kvalifikator `extern` kod funkcija može izostaviti.

Primjer 7.16. Osnovni primjer upotrebe kvalifikatora `extern` ćemo prikazati u sljedećem listingu. Prva datoteka

```
/*datoteka1 */
#include <stdio.h>
int a = 10;


void f() {
    printf("Funkcija f: a=%d\n", a);
}
```

Druga datoteka

```
  /*datoteka2 */
#include <stdio.h>

extern int a;

void f();//ne moramo navoditi kao extern

int main() {
    printf("a u fajlu extern2: %d\n",a);//a je vidljivo i ovako
    f();//pozivamo funkciju koja je definisana u drugoj datoteci
    return 0;
}
```

Upotrebu riječi `extern` treba da prati samo deklaracija promjenljive, bez navođenja inicijalne vrijednosti. Slično je i sa promjenljivima kojima deklarišemo nizove. Ako želimo da promjenljivu nizovnog tipa samo deklarišemo, koristimo riječ `extern`, a dimenziju nije potrebno navoditi.

Primjer 7.17. Navedimo primjer kojim ćemo dodatno pojasniti uvedene koncepte.

Elektronska verzija

Prva datoteka

```
/*datoteka1 */
#include <stdio.h>
    int a =10;
    int b = 16;
    int c; //nacelna definicija, ali nece postati stvarna

    void f() {
        printf("Funkcija f: a=%d c = %d\n", a,c);
    }
    void g(){
        printf("Funkcija g...");
    }
```

Druga datoteka

```
 /*datoteka2 */
#include <stdio.h>

extern int a;//koristimo rijec extern da samo deklarisemo promjenljivu
    a
void f();//samo deklaracija, ali ne moramo navoditi kao extern jer je
    funkcija
//int b = 10; pogresno, ne mogu biti dvije definicije od b;
int c = 135;//dozvoljeno, c je samo ovdje definisano (u prvoj datoteci
    je samo nacelna definicija)


//void g(){ printf ("Funkcija g...\n"); }//nije dozvoljena jos jedna
    definicija od g
int main() {
    printf("U fajlu datoteka2.. a: %d c: %d\n",a,c);//a je vidljivo i
        ovako
    f(); //pozivamo funkciju koja je definisana u drugoj datoteci

    return 0;
}
```

Elektronska verzija

7.4 Pitanja i zadaci

1. Objasniti šta su lokalne promjenljive i koliki je njihov doseg. Ilustrovati primjerom upotrebu lokalne promjenljive.

2. Kojim standardom je dozvoljeno lokalno definisanje funkcije? Primjerom ilustrovati lokalnu definiciju funkcije.

3. Kada je korisno koristiti globalne promjenljive? Ilustrovati primjerom.

4. Dat je sljedeći program:

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

    int brojac = 10;
    char c = 'a';
    int i;

    for(i = 0; i < brojac; i++){
        int x = c + 1;
        printf("%d. izvrsenje naredbe x=%d\n",(i+1),x);
    }
    return 0;
}
```

Za svaku promjenljivu odrediti životni vijek trajanja. Odrediti vrijednost promjenljive `x` u svakoj iteraciji.

5. Šta je rezultat izvršenja narednog kôda? Zašto ima razlike između ispisa dobijenih pozivima funkcija `f` i `g`?

```
#include <stdio.h>
void f() {
    int a = 10;
    printf("f: a=%d ", a);
    a = a + 5;
}
void g() {
    static int a = 10;
    printf("g: a=%d ", a);
    a = a + 5;
}

int main() {
    f(); f(); f();
    g(); g(); g();
    return 0;
}
```

Elektronska verzija

6. Za sve promjenljive i funkcije iz narednog dijela kôda odrediti doseg i vijek trajanja.

```
int x;

int f(int y){
    int z;

    while(1){...
        int w;
    ...}

void g(){
    printf("Funkcija g...\n");
}

}
```

7. Ako se pri definisanju lokalne promjenljive ne dodijeli vrijednost, koju vrijednost uzima ta promjenljiva po *default*-u? Šta je *default*-na vrijednost globalne promjenljive, ako prije definicije nije navedena početna vrijednost?

8. Zašto je moguće izostaviti ključnu riječ **`extern`** pri definisanju prototipa funkcije, koja je definisana u nekoj drugoj datoteci?

9. Šta je načelno, a šta stvarno definisanje promjenljive? Samostalno konstruiši primjer, kojim možeš ilustrovati odgovor.

10. Koji kvalifikator koristimo kada želimo da naglasimo da promjenljiva koju deklarišemo služi za spoljašnju povezanost i da je njena definicija na nekom drugom mjestu?

Elektronska verzija

8 Pokazivači

Mehanizam pokazivača je izuzetno moćan programerski alat u programskom jeziku C. Pomoću pokazivača se mnogi zadaci rješavaju na lakši način, unapređuje se efikasnost programa i omogućava rad sa praktično neograničenom količinom podataka. Na primjer, upotrebom pokazivača omogućavamo izmjenu vrijednosti promjenljivih, koje su argumenti funkcije. Takođe, pokazivači se koriste i prilikom dinamičke alokacije memorije, što podrazumijeva pisanje programa koji omogućavaju upravljanje proizvoljnom količinom memorije, bez potrebe da se u toku pisanja programa unaprijed zna koliko memorije je potrebno.

Pokazivači (engl. pointers) su upravo dobili ime po tome što "pokazuju" na određena mjesta u memoriji. Kada govorimo o pristupu podacima koji su zapisani u memoriji, do sada smo uglavnom podrazumijevali rad sa promjenljivima. Da ponovimo, pod promjenljivima podrazumijevamo nazive memorijskih mjesta, pomoću kojih možemo pročitati odgovarajući podatak i promijeniti ga. Sa druge strane, pokazivači sadrže informacije gdje se nalazi odgovarajući podatak. U programskom jeziku C pomoću pokazivača, tj. informacije gdje se u memoriji nalazi neki podatak, tom podatku, takođe, možemo pristupiti i eventualno ga i promijeniti.

Drugim riječima, ako znamo mjesto (adresu) gdje je neka promjenljiva smještena u memoriji, pomoću te adrese možemo doći do tog podatka i raspolagati sa njim. Kakva je korist od toga? Na primjer, ako želimo da neki "veliki" podatak proslijedimo u funkciju, mnogo je lakše proslijediti samo informaciju gdje se taj podatak nalazi, nego funkciji slati cijeli taj podatak, što podrazumijeva trošenje dodatnog procesorskog vremena i memorije. Pored toga, ako se u toku izvršenja programa javi potreba da se raspolaže dodatnom memorijom, koja nije rezervisana, tj. "zakupljena" prilikom pisanja programa, programer raspolaže mehanizmom koji mu omogućava da, u određenom trenutku, zatraži dodatnu memoriju od sistema. U tom slučaju sistem vraća informaciju gdje se nalazi ta memorija koja se daje programu na raspolaganje, tj. program od sistema dobija adresu gdje se ta memorija nalazi, koja se upravo čuva u pokazivaču.

Pokazivači su, dakle, promjenljive koje sadrže memorijske adrese drugih podataka. Kako u programskom jeziku C svaki podatak ima svoj tip, tako se i za svaki pokazivač mora znati na koji tip podatka on pokazuje, tj. koji tip

Elektronska verzija

podatka je smješten u memorijskom prostoru čiju adresu taj pokazivač sadrži. Tako razlikujemo pokazivače na tip **int**, tip **double**, tip **char** itd. Treba, ipak, napomenuti da postoji i tzv. generički pokazivač (**void** pokazivač), kome nije pridružen nijedan tip podatka.

Kada kažemo da su pokazivači promjenljive, podrazumijevamo da oni mogu mijenjati vrijednost. Stoga, pokazivače zovemo i pokazivačkim promjenljivima, kako bismo ih razlikovali od "običnih" promjenljivih. S obzirom na to da su vrijednosti pokazivača adrese, promjena vrijednosti pokazivača znači mijenjanje adrese na koju pokazivač pokazuje. Dakle, pokazivač u jednom trenutku može da pokazuje na jednu memorijsku lokaciju, a u nekom drugom trenutku na drugu, uz poštovanje ograničenja, da sve te memorijske lokacije moraju biti istog tipa, kog je i sam pokazivač.

8.1 Sintaksa pokazivača

Pokazivač na neki tip se deklariše na sljedeći način:

```
tip *ime;
```

gdje je `ime` ime pokazivača (pokazivačke promjenljive), a **tip** tip podatka na koji pokazivač pokazuje. Iz navedenog vidimo da se deklaracija pokazivača, od deklaracije obične promjenljive, razlikuje u asterisk znaku (zvjezdici) *. Na primjer, deklaracijom

```
int *pok;
```

deklarisana je pokazivačka promjenljiva `pok` koja pokazuje na tip **int**, tj. ona može da dobije vrijednost adrese nekog cijelog broja.

Ako želimo da u istom redu deklarišemo više pokazivačkih promjenljivih, onda uz svaku od njih moramo pisati zvjezdicu. Na primjer, deklaracijom

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

samo će promjenljiva `pok1` biti pokazivač na podatak tipa **int**, dok će promjenljiva `pok2` biti obična promjenljiva tipa **int**.

Deklaracijom

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

obje promjenljive su deklarisane kao pokazivači.

Da bi pokazivač zaista pokazivao na neki podatak koji je smješen u nekoj

Elektronska verzija

promjenljivoj, potrebno je pročitati adresu te promjenljive. Čitanje adrese promjenljive, tj. njene lokacije u memoriji se postiže pomoću operatora &, koji se stavlja ispred naziva promjenljive. Ovaj operator (operator &) zovemo jednostavno – operator za čitanje adrese. Sljedećim primjerom ilustrujmo upotrebu ovog operatora:

```
int n = 15; //deklarisemo promjenljivu n kojoj dodijelimo vrijednost 15

int *pn = &n; //operatorom & procitamo adresu promjenljive n i
    vrijednost te adrese dodijelimo pokazivackoj promjenljivoj koja se
    zove pn
```

Pristup sadržaju memorijskog prostora na koji pokazuje neki pokazivač je omogućen pomoću tzv. operatora za dereferenciranje pokazivača *. Riječ dereferenciranje u nazivu ovog operatora je pomalo nezgrapna (čak i u engleskom jeziku). Stoga nije pogrešno koristiti jednostavniji izraz za ovaj operator – operator za pristup sadržaju na koji pokazuje pokazivač.

Iskoristimo prethodni primjer da ilustrujemo upotrebu ovog operatora. Ako pokazivač `pn` ima vrijednost adrese promjenljive `n`, a vrijednost promjenljive `n` je 15, tada promjenljiva `k` u izrazu

```
int k = *pn + 10;
```

dobija vrijednost 25, jer smo pomoću `*pn` pročitali vrijednost sadržaja memorijskog mjesta na koje pokazuje `pn` (a ta vrijednost je jednaka 15), te smo toj vrijednosti još dodali broj 10. Dalje, ako primjer nastavimo sljedećim izrazom

```
*pn = 23;
```

to znači da smo vrijednost sadržaja memorijskog mjesta na koje pokazuje `pn` promijenili na 23. Kako je to memorijsko mjesto, ustvari, mjesto gdje je sačuvana promjenljiva `n`, to znači da ta promjenljiva `n` sada ima vrijednost 23.

Iz prethodna dva izraza vidimo da operator za pristup sadržaja može da se koristi u izrazu i sa lijeve i sa desne strane operatora dodjele (operatora =), što nije slučaj i sa operatorom za čitanje adrese (operatorom &), koji može da stoji samo sa desne strane operatora =. Tako je izraz

```
&k = neka vrijednost;
```

sintaksno neispravan.

Operatori za čitanje adrese i operator za pristup sadržaju su unarni operatori koji imaju isti prioritet kao i drugi unarni operatori. Oni su većeg prioriteta od

Elektronska verzija

aritmetičkih operatora. Ako bismo nastavili prethodni primjer sa pokazivačem `pn` na promjenljivu `n` (promjenljiva `n` sada ima vrijednost 23), u izrazu

```
int m = 3 * (*pn + 2);
```

promjenljiva `m` dobija vrijednost 75.

Primjer 8.1. Evo jednog kompletno urađenog programa kojim ilustrujemo upotrebu pokazivača.

```
1  #include <stdio.h>
2
3  int main(){
4     int a = 10, b = 15;
5     int *pa = &a, *pb = &b;
6     printf("Promjenljiva a: %d i adresa od a: %p\n",a,&a);
7     //sad ispisujemo iste vrijednosti pomocu pa
8     printf("Sadrzaj od pa: %d i vrijednost od pa: %p\n",*pa,pa);
9
10    //promijenimo vrijednost od a preko pokazivaca
11
12    *pa = 25;
13    printf("Promjenljiva a: %d \n",a);
14    //ista vrijednost je i
15    printf("Sadrzaj od pa: %d \n",*pa);
16    //sada podesimo da pa pokazuje na b i ponovimo prethodna dva ispisa
17    pa=pb;
18    printf("Promjenljiva a: %d \n",a);
19    //vise nije ista vrijednost
20    printf("Sadrzaj od pa: %d \n",*pa);
21    return 0;
22 }
```

Pošto tačne vrijednosti memorijskih adresa zavise od samog računara i operativnog sistema, prikaz memorijskih adresa na ekranu će se, najvjerovatnije, razlikovati u zavisnosti od same konfiguracije računara. Vrijednosti adresa se na ekranu ispisuju kao heksadecimalni brojevi. Evo jednog prikaza ispisa na ekranu, koji je rezultat pokretanja programa na jednom računaru.

```
Promjenljiva a: 10 i adresa od a: 000000000062FE3C
Sadrzaj od pa: 10 i vrijednost od pa: 000000000062FE3C
Promjenljiva a: 25
Sadrzaj od pa: 25
Promjenljiva a: 25
```

Elektronska verzija

```
Sadrzaj od pa: 15
```

Pomenimo da postoji i praksa naslijeđena iz nekih drugih programskih jezika, da se pokazivačkim tipovima, pomoću ključne riječi **`typedef`** dodjeljuju nova imena. Tako bi se, na primjer, umjesto uobičajene deklaracije pokazivačke promjenljive na tip **`int`**, prvo mogao definisati odgovarajući tip podatka

```
typedef int * Pint;
```

pa bi se pokazivači na podatke tipa **`int`** mogli deklarisati pomoću

```
Pint p1, p2;
```

8.2 Pokazivači kao argumenti funkcija

Kao što smo napomenuli na samom početku ovog poglavlja, pokazivači mogu da budu i argumenti funkcija. U tom slučaju funkcija može da promijeni vrijednost promjenljivih, koje su argumenti funkcije, što kod upotrebe "običnih" promjenljivih kao argumenata funkcije nije slučaj.

Podsjetimo se funkcije

```
void uvecaj(int x){

    x++;

printf("Vrijednost x unutar funkcije: \%d\n",x);
}
```

i programa koji je koristi

```
1  int main(){
2
3      int x = 10;
4
5      uvecaj(x);
6
7      printf("Vrijednost x nakon u glavnom programu nakon poziva
           funkcije: \%d\n",x);
8      return 0;
9  }
```

u kome se argument funkcije lokalno uvećava za jedan, dok po završetku

Elektronska verzija

funkcije to uvećanje nema efekta na promjenljivu `x`, na koju je funkcija pozvana. Ponovimo da je razlog za ovo činjenica da se argument u funkciju prenosi po vrijednosti. Vrijednost stvarnog argumenta (promjenljive `x` iz glavnog dijela programa) se kopira u lokalnu promjenljivu funkcije (koja se sticajem okolnosti isto zove `x`), te se sve izmjene u okviru funkcije vrše na toj lokalno kreiranoj promjenljivoj. Po završetku funkcije, promjenljiva `x` definisana u glavnom dijelu programa ostaje nepromijenjena.

Slično se dešava sa sljedećom funkcijom:

```
void razmjena(int x, int y) {
    int t = x;
    x = y;
    y = t;
}
```

i programa koji je koristi

```
1  int main(){
2
3      int x = 10, y = 15;
4      razmjena(x,y);
5
6      printf("Vrijednosti x i y nakon u glavnom programu nakon poziva
           funkcije: \%d \%d \n",x,y);
7      return 0;
8  }
```

jer se promjenljive `x` i `y` u funkciju prenose po vrijednosti, te se razmjena vrši u lokalno kreiranim promjenljivima, dok se memorijski prostor u kome su smještene promjenljive `x` i `y` iz glavnog dijela program ne dira. Za posljedicu imamo da ovakva funkcija **razmjena** ne utiče na vrijednosti promjenljivih `x` i `y` iz glavnog dijela programa.

Međutim, ako bismo funkciju za razmjenu vrijednosti napisali na sljedeći način

```
void razmjena(int *px, int *py) {
    int t = *px;
    *px = *py;
    *py = t;
}
```

omogućili bismo da argumenti funkcije budu adrese promjenljivih čije vrijednosti trebamo razmijeniti, dok se u samoj funkciji, na osnovu informacija o

Elektronska verzija

adresama podataka pristupa odgovarajućim sadržajima, čije se vrijednosti, uz pomoć pomoćne promjenljive `t` mijenjaju. U glavnom dijelu programa, treba voditi računa i o tome da ovakvu funkciju treba pozvati sa odgovarajućim argumentima, koji više nisu obični cijeli brojevi, već adrese:

```
int main(){

    int x = 10, y = 15;

    razmjena(&x,&y);

    printf("Vrijednosti x i y nakon u glavnom programu nakon poziva
        funkcije: \%d \%d \n",x,y);
    return 0;
}
```

Za posljedicu primjene ovakve funkcije `razmjena` imamo da su cjelobrojne promjenljive `x` i `y` razmijenile vrijednost.

Na primjeru funkcije `uvecaj`, za koju želimo da vrijednost argumenta uveća za jedan, imali bismo ovakvu situaciju

```
void uvecaj(int* x){ //argument funkcije je pokazivac

    (*x)++; //vrijednost sadrzaja na koji pokazuje x se uvecava za 1

printf("Vrijednost x unutar funkcije: \%d\n",x);
}
```

i programa koji je koristi

```
int main(){

    int x = 10;

    uvecaj(&x); //vodimo racuna da je tip argumenta pokazivac na
        podatak tipa int

    printf("Vrijednost x nakon u glavnom programu nakon poziva
        funkcije: \%d\n",x);
    return 0;
}
```

I po završetku funkcije promjenljiva `x` u glavnom dijelu programa ima vrijednost 11.

Elektronska verzija

8.3 Pokazivačka aritmetika

Pored osnovnih operacija koje su vezane za pokazivače - čitanja adrese (za šta koristimo operator &) i pristup sadržaju (pomoću operatora *), na pokazivačima je moguće vršiti i neke aritmetičke operacije. Pokazivačima je moguće dodavati i od njih oduzimati cijele brojeve. Od pokazivača je moguće i oduzimati vrijednosti koje su takođe pokazivačkog tipa. Takođe, dopušteni su i operatori `++`, `--`, `+=` i `-=`.

Dodavanje cijelog broja n pokazivaču `p` za rezultat ima pokazivač koji pokazuje na onaj memorijski blok koji je udaljen n blokova od bloka na koji pokazuje pokazivač `p`. Pod jednim blokom podrazumijevamo memorijski prostor koji je potreban za čuvanje jednog podatka odgovarajućeg tipa. Širina bloka zavisi od tipa podatka. Na primjer, ako je `p` pokazivač na promjenljivu tipa **`int`**, onda će `p+1` pokazivati na sljedeću promjenljivu tipa **`int`** u memoriji. Dakle, dodavanje jedinice na pokazivač ne povećava adresu za 1, već za onoliko koliko je potrebno da nova vrijednost pokazuje na sljedeću promjenljivu istog tipa u memoriji.

Oduzimanjem jednog pokazivača od drugog dobija se cijeli broj, koji govori koliko je blokova prvi objekat, na koji se pokazuje, udaljen od drugog bloka, na koji se pokazuje.

Primjer 8.2. Započnimo zadatak na sljedeći način. Definisaćemo 5 običnih cjelobrojnih promjenljivih i dodijeliti neke početne vrijednost. Iako to nije garantovano, očekujemo da će njima biti dodijeljene susjedne memorijske adrese. Upotrebom pokazivača to možemo lako i provjeriti.

```
...
int i5=5, i4=4, i3=3, i2=2, i1=1;
int *ip1, *ip2, *ip3, *ip4, *ip5;
ip1 = &i1;
ip2 = &i2;
ip3 = &i3;
ip4 = &i4;
ip5 = &i5;

printf("Adresa od i1: %p\n",ip1);
printf("Adresa od i2: %p\n",ip2);
printf("Adresa od i3: %p\n",ip3);
printf("Adresa od i4: %p\n",ip4);
printf("Adresa od i5: %p\n",ip5);
...
```

Elektronska verzija

Razlog za navođenje promjenljivih `i1` do `i5` ovakvim redoslijedom je taj što se lokalne promjenljive alociraju na steku (o tome je već bilo riječi kada smo govorili o dosegu promjenljivih), pa promjenljive, koje se kasnije alociraju, dobijaju niže adrese. Podsjetimo se da će se, pošto tačne vrijednost memorijskih adresa zavise od samog računara i operativnog sistema, prikaz memorijskih adresa na ekranu najvjerovatnije razlikovati u zavisnosti od same konfiguracije računara. Bez obzira na to, prikažimo jedan ispis, koji je dobijen prilikom testiranja gornjeg dijela programa na jednom računaru.

```
Adresa od i1: 000000000062FE08
Adresa od i2: 000000000062FE0C
Adresa od i3: 000000000062FE10
Adresa od i4: 000000000062FE14
Adresa od i5: 000000000062FE18
```

Kao što vidimo, vrijednosti adresa se na ekranu ispisuju kao heksadecimalni brojevi, koji se između sebe razlikuju za 4. Na primjer, broj 62FE0C je za 4 veći od 62FE08 (cifra C odgovara cifri 12 u heksadecimalnom zapisu). Praktično, na ekranu su ispisane adrese prvog od četiri bajta za svaku od promjenljivih `i1` do `i5`. Podsjetimo se da su podaci tipa **`int`** širine 4 bajta. Zaključujemo da su, u ovom slučaju, promjenljive alocirane na susjednim adresama, koje se razlikuju za 4.

Primjer 8.3. Primijenimo sada neke jednostavne aritmetičke operacije na pokazivače.

```
...
//definisemo promjenljive na isti nacin
int i5 = 5, i4 = 4, i3 = 3, i2 = 2, i1 = 1;

int *ip = &i1;

printf("Sadrzaj pokazivaca ip=%d\n",*ip);//ovo je kao i1
printf("Adresa pokazivaca ip=%x\n",ip);

ip += 4;
printf("Nakon pomjeranja za 4 mjesta desno:\n");
printf("Sadrzaj pokazivaca ip=%d\n",*ip);//ovo je sad kao i5
printf("Adresa pokazivaca ip=%x\n",ip);

ip -= 2;
printf("Nakon pomjeranja za 2 mjesta lijevo:\n");
printf("Sadrzaj pokazivaca ip=%d\n",*ip);//ovo je sad kao i3
```

Elektronska verzija

```
printf("Adresa pokazivaca ip=%x\n",ip);

*ip = *ip+3;//sad ostajemo na istom mjestu ali mijenjamo sadrzaj
printf("Nakon promjene sadrzaja pokazivaca na trenutnoj poziciji:\n");
printf("Sadrzaj pokazivaca ip=%d\n",*ip);
printf("Adresa pokazivaca ip=%x\n",ip);

ip++;
printf("Nakon upotrebe operatora ++:\n");
printf("Sadrzaj pokazivaca ip=%d\n",*ip);//ovo je sad kao i4
printf("Adresa pokazivaca ip=%x\n",ip);

int i = &i5 - ip;
printf("Rastojanje izmedju adrese i5 i trenutog pokazivaca ip=%d\n",i);
...
```

Prikaz na ekranu zavisi od konfiguracije, a na jednom računaru izgleda ovako.

```
Sadrzaj pokazivaca ip=1
Adresa pokaivaca ip=62fe08
Nakon pomjeranja za 4 mjesta desno:
Sadrzaj pokazivaca ip=5
Adresa pokaivaca ip=62fe18
Nakon pomjeranja za 2 mjesta lijevo:
Sadrzaj pokazivaca ip=3
Adresa pokaivaca ip=62fe10
Nakon promjene sadrzaja pokazivaca na trenutnoj poziciji:
Sadrzaj pokazivaca ip=6
Adresa pokaivaca ip=62fe10
Nakon upotrebe operatora ++:
Sadrzaj pokazivaca ip=4
Adresa pokaivaca ip=62fe14
Rastojanje izmedju i5 i trenutog sadrzaja pokazivaca i=1
```

Pokazivaču možemo pridružiti vrijednost 0, čime zapravo pokazivaču dodjeljujemo nultu vrijednost (engl. `NULL`). Na taj način, signaliziramo da pokazivačka promjenljiva ne sadrži validnu adresu, tj. pokazivač koji ima vrijednost 0 ne pokazuje ni na šta smisleno. Takođe, nije moguće pročitati sadržaj pokazivača koji ima vrijednost 0, tj. takav pokušaj bi doveo do greške tokom izvršenja programa. Da bi se povećala čitljivost programskog koda, u programskom jeziku C postoji i simbolička konstanta `NULL`, koju, takođe, možemo da koristimo, ako želimo da pokazivaču dodijelimo nultu vrijednost. Simbolička konstanta `NULL` je definisana kao makro u zaglavlju `<stdio.h>`.

Elektronska verzija

8.4 Konstantni tipovi i pokazivači

Pored običnih konstanti, možemo da definišemo i pokazivače koji pokazuju na konstantnu vrijednost, ali i konstantne pokazivače.

Ukoliko imamo običnu konstantu, onda nije moguće definisati običan pokazivač koji pokazuje na tu konstantu vrijednost. Na primjer, u slučaju ovakvog koda

```
const int maksimum = 1000;
int * pokazivac = &maksimum; //greska
```

prevodilac će javiti grešku, jer bi se definisanjem ovakvog pokazivača omogućila izmjena vrijednosti konstante maksimum, što nije dozvoljeno.

Umjesto posljedenjeg reda, potrebno je definisati pokazivač na konstantu vrijednost, što se u ovom primjeru radi na sljedeći način:

```
const int * pokazivac = &maksimum;
```

Definisanjem ovakvog pokazivača na konstantu vrijednost, sada nije dozvoljena izmjena sadržaja na koji pokazivač pokazuje. Na primjer, naredba

```
 * pokazivac = 2000; //greska
```

nije dozvoljena, jer je pokazivac tipa `const int`.

Pored pokazivača koji pokazuju na konstante vrijednosti, moguće je definisati i konstantne pokazivače, čime se onemogućava da jednom dodijeljena vrijednost bude promijenjena. Na primjer

```
 int n = 15;
 int m = 20;
 int * const pok = &n;
```

Pošto je pokazivač `pok` definisan kao konstantan pokazivač, sljedeća naredba nije dopuštena:

```
pok = &m;
```

Moguće je definisati i konstantne pokazivače na konstantne objekte, što bi izgledalo ovako:

```
const int maksimum = 1000;
const int const *pokazivac = &maksimum;
```

Elektronska verzija

Pokazivači na konstantne vrijednosti igraju važnu ulogu u situacijama kada su pokazivači argumenti funkcija. Već smo ranije razmatrali situacije, kada je u okviru funkcije potrebno promijeniti vrijednost njenih argumenata. Tada smo naveli da je to moguće uraditi, ukoliko funkcija uzima za argumente pokazivače na objekte. Sa druge strane, prenos argumenata u funkciju preko pokazivača se često radi i iz drugih razloga, a najprije zbog povečanja efikasnosti programa izbjegavanjem prenošenja velikih objekata po vrijednosti. Ako je potrebno naglasiti (i dodatno se osigurati) da funkcija ne treba da mijenja podatak koji je njen argument (a koji u funkciju ulazi kao pokazivač), onda se kao tip argumenta navodi upravo pokazivač na konstantan tip.

Primjer 8.4. Ako želimo da spriječimo izmjenu vrijednosti argumenta unutar funkcije, to možemo uraditi kao u funkciji u sljedećem primjeru.

```
...
void f1(const int* p){
   printf("Funkcija f1: %d\n",*p);
   *p = 40;//pogresno!!! nije dozvoljeno mijenjati sadrzaj koji je na
       adresi p

}
...
```

8.5 Nizovi i pokazivači

Nizovi i pokazivači su u veoma uskoj vezi. Sama priroda ove dvije strukture, ali i mehanizmi koji su implementirani u programskom jeziku C omogućavaju da se u mnogim situacijama upotreba nizova poistovjećuje sa upotrebom pokazivača. Tako se sintaksa definisana na nizovima (kao što je, na primjer, operator uglasta zagrada, kojim se pristupa elementima niza) može koristiti i u radu sa pokazivačima. Slično, sintaksa na pokazivačima (kao što je, na primjer, upotreba operatora za čitanje sadržaja) se može koristiti i na nizovima. Ipak, iako ima dosta sličnosti, između nizova i pokazivača postoje i razlike o kojima treba voditi računa.

Za potrebe niza koji se definiše, (uzmimo najjednostavniju definiciju niza kao primjer)

```
int niz[10];
```

Elektronska verzija

u memoriji se na uzastopnim memorijskim lokacijama rezerviše odgovarajući prostor za smještanje elemenata (u našem slučaju prostor za smještanje 10 cijelih brojeva). Prilikom prevođenja programa, imenu niza (u našem slučaju imenu `niz`) će biti dodijeljena vrijednost adrese prvog elementa niza. Iz ovoga već možemo zaključiti da je promjenljiva kojom označavamo niz, praktično, veoma slična pokazivaču na prvi element niza. Niz, ipak, nije običan pokazivač. Pokazivač je promjenljiva koja može da mijenja svoju vrijednost (adresu na koju pokazuje), dok je nizu uvijek dodijeljena vrijednost adrese prvog elementa i ta vrijednost ne može da se mijenja. Zapravo, imena nizova nisu l-vrijednosti (imena nizova uvijek pokazuju na prvi element niza), dok pokazivači jesu.

Sa druge strane, u svim slučajevima, osim kad je riječ o operatoru **`sizeof`** i operatoru &, ime niza se implicitno konvertuje u pokazivač odgovarajućeg tipa, što omogućava "miješanje", tj. kombinovanje pokazivačke i nizovne sintakse. Na našem primjeru, ime `niz` će implicitno biti konvertovano u pokazivač tipa **`int *`**, koji pokazuje na prvi element niza i na ovu promjenljivu možemo da primjenjujemo sintaksu pokazivača. Vrijednosti `niz` će odgovarati vrijednost adrese početnog elementa, vrijednosti `niz+1` vrijednost adrese elementu niza sa indeksom 1 (drugi element po redu), itd. U opštem slučaju, adresi elementa niza `niz[i]` odgovara vrijednost pokazivača `(niz+i)`, a vrijednosti `i`-tog elementa `niz[i]` odgovara vrijednost `*(niz+i)`.

Prilikom pristupa elementima niza pomoću odgovarajućeg pokazivača se ne vrši nikakva kontrola granica, te je pomoću pokazivača moguće "skakanje" po memorijskim prostorima koji se, uslovno rečeno, nalaze i lijevo i desno, u odnosu na prostor zauzet za same elemente niza. Tako, prilikom prevođenja programa, prevodilac neće prijaviti grešku ako se pomoću pokazivača pokuša pristupiti elementu niza sa negativnim indeksom, ili indeksom koji je veći ili jednak dimenziji niza (jednostavno će se smatrati da se pristupa adresama koje su niže od adrese početnog elementa, odnosno, koje su više od adrese posljednjeg elementa). Tako je u slučaju našeg niza sintaksno dozvoljeno pristupati (i pokušati mijenjati vrijednosti) podacima koji su na adresama, na primjer, `niz-1`, `niz-10`, `niz+10`, `niz+100` itd. Sa druge strane, ovakav pristup je pogrešan, jer u najvećem broju slučajeva može da prouzrokuje nekontrolisano ponašanje programa i pojavu grešaka.

Kao što se pravila jezika, koja su definisana za pokazivače, mogu primijeniti na nizove, slično se i nizovna sintaksa može koristiti na pokazivačima. Tako se, na primjer, uglasta zagrada, koja služi za pristup elementima niza može primijeniti i na pokazivač. Ako je, na primjer, pokazivač `p` definisan na sljedeći način

```
int * p;
```

Elektronska verzija

onda bi izraz `p[0]` značio isto što i `*(p+0)`, što je ustvari isto što i `*p`, odnosno na taj način bi se čitao sadržaj memorijske lokacije na koju pokazuje `p`. Izraz `p[5]` bi predstavljao onaj cijeli broj koji se nalazi na memorijskoj lokaciji koja je za 5 širina **`int`**-a udaljena od lokacije na koju pokazuje `p`. Isti taj broj bi se dobio i izrazom `*(p+5)`.

Naglasimo još jednom, a ovdje ćemo to i dodatno ilustrovati primjerom, da nizovi i pokazivači nisu potpuno ravnopravni. Prilikom definisanja, promjenljiva tipa niz dobija vrijednost memorijske lokacije prvog elementa i ta vrijednost se više ne može mijenjati. Za razliku od niza, pokazivač može da promijeni vrijednost. Praktično, mi možemo da radimo sljedeće:

```
int niz[10];
int * pok = niz;
pok++; //promijenili smo vrijednost adrese na koju pokazuje p, sada
    ovaj pokazivac pokazuje na element niza sa indeksom 1
```

ali ne i sljedeće

```
niz++; //pogresno, jer ne mozemo da promijenimo vrijednost adrese na
    koju pokazuje niz. On uvijek pokazuje na prvi element.
```

Stoga, zaključujemo da su nizovi praktično ekvivalentni konstantnim pokazivačima, tj. pokazivačima koji uvijek pokazuju na istu adresu koja im je inicijalno dodijeljena. U našem primjeru, to bi bilo ovako:

```
int niz[10];
int * const pok = niz;
pok++; //sada je ovo pogresno, ne moze se mijenjati adresa na koju
    pokazuje pok
```

Još jedna razlika između nizova i pokazivača se može uočiti primjenom operatora **`sizeof`**. Posmatrajmo sljedeći primjer:

```
int niz[10] = {0,1,2,3,4,5,6,7,8,9};
//odstampajmo velicinu niza
printf("Velicina niza: \%d\n", sizeof(niz)); //bice odstampana
    vrijednost 40, uz pretpostavku da je velicina int-a 4
//pravimo konstantan pokazivac na niz
int* const pok = niz;
printf("Size: \%d\n", sizeof(ptr)); //bice odstampana vrijednost 4, uz
    istu pretpostavku da je velicina int-a 4
```

Sličnost između pokazivača i nizova koristimo prilikom poziva funkcije, kada

Elektronska verzija

funkcija kao argument očekuje niz. Na primjer, ako imamo funkciju `f` koja za argument očekuje cjelobrojan niz, onda su sljedeća tri zapisa ekvivalentna:

```
void f(int * niz);
void f(int niz[]);
void f(int niz[10]);
```

U sva tri slučaja prenosi se samo informacija o prvom elementu niza (informacija o dimenziji niza se ne prenosi čak ni u trećem slučaju).

8.6 Pokazivači i niske

Sada, kada smo savladali osnovne tehnike koje se odnose na rad sa pokazivačima, možemo se vratiti razmatranju niski karaktera, jer su one u direktnoj vezi sa pokazivačima, konkretno, sa pokazivačima na podatak tipa `char`.

Da bismo kreirali nisku karaktera koju kasnije možemo koristiti, koristimo pokazivač na podatak tipa `char` i inicijalizujemo njegovu vrijednost na znakove te niske, tako što praktično inicijalizujemo vrijednost pokazivača na prvi karakter te niske. Na primjer

```
char * rijec = "Ovo je jedna niska";
```

Prisjetimo se važne činjenice da se svaka niska završava terminalnom nulom `'\0'`, što znači da i naša niska `rijec` iz gornjeg primjera u memoriji zauzima jedno mjesto više u odnosu na broj karaktera. Pošto znamo da se niska završava terminalnom nulom, tu činjenicu možemo koristiti u raznim situacijama, posebno onda, kada je potrebno "proći" kroz sve karaktere niske. Evo nekoliko primjera.

Primjer 8.5. Napisati funkciju koja računa dužinu niske karaktera i program koji je promjenjuje.

```
#include <stdio.h>
int duzina(char *s){
   char *p = s;
   int rezultat = 0;
   while(*p++ != '\0')
      rezultat++;

   return rezultat;
}
int main(){
```

Elektronska verzija

```
11      char *niska = "tabla";
12
13      printf("Duzina je %d.\n",duzina(niska));
14
15      return 0;
16  }
```

Primjer 8.6. Napisati funkciju koja za argument uzima nisku i karakter, a kao rezultat vraća indeks prvog pojavljivanja karaktera u toj niski. Ako se karakter ne pojavljuje u niski, funkcija vraća -1.

```
1   #include <stdio.h>
2   int pronadji_karakter(char *s, char c) {
3     int i = 0;
4     for (; s[i]; i++)
5       if (s[i] == c)
6         return i;
7     return -1;
8   }
9
10  int main(){
11
12      char NIZ[100];
13      char *t;
14
15      printf("Unesi rijec: ");
16      scanf("%s", &NIZ);
17      char k;
18      printf("Unesi karakter:\n");
19      scanf("\n%c",&k);
20
21      t = NIZ;
22
23      if(pronadji_karakter(t,k) == -1)
24          printf("Karakter %c se ne pojavljuje u nisci %s.\n",k,t);
25      else
26          printf("Karakter %c se pojavljuje u nisci %s na poziciji
                %d.\n.",k,t,pronadji_karakter(t,k));
27      return 0;
28  }
```

Primjer 8.7. Napisati funkciju `okreni(char* )` koja kao argument uzima nisku i okreće obrće njene karaktere, tako što mijenjaju mjesta prvi i posljednji

Elektronska verzija

karakter, pa drugi i pretposljednji itd.

```
#include <stdio.h>

void okreni(char *p){

    char pom;
    int i = 0, j = 0, br = 0;
    //prvo odredimo duzinu niske
    while (*p != '\0'){
        br++;
        p++;
    }

    p = p-br;//vratimo p na pocetak niske

    for(i = 0, j = br - 1; i < br / 2; i++, j--){
        pom = *(p + i);
        *(p + i) = *(p + j);
        *(p + j) = pom;
    }
}

main(){

    char NIZ[100];
    char *t;

    printf("Unesi neku rijec: ");
    scanf("%s", &NIZ);

    t = NIZ;
    okreni(t);
    printf("%s", t);

    return 0;
}
```

Elektronska verzija

8.6.1 Zaglavlje string.h

Na prethodnim primjerima smo vidjeli da, ukoliko dobro poznajemo i razumijemo način reprezentacije niski i ako vladamo osnovnim vještinama programiranja, lako možemo realizovati neke od prilično “univerzalnih” funkcija

na niskama, kao što su računanje dužine niske, obrtanje karaktera niske itd. Sa druge stane, u programskom jeziku C su sve ove, ali i mnoge druge funkcije implementirane i definisane u datoteci zaglavlja `string.h`. Ovdje navedimo neke od njih.

- `strcat(char s1[], char s2[])` – funkcija koja spaja dvije niske u jednu, tako što na karaktere niske `s1` dopiše karaktere niske `s2`.
- `strncat(char s1[], char s2[], int n)` – funkcija koja spaja nisku `s1` sa prvih `n` karaktera niske `s2`.
- `strcpy(char s1[], char s2[])` – funkcija koja kopira sadržaj niske `s2` u nisku `s1`.
- `strncpy(char s1[], char s2[], int n)` – funkcija koja kopira prvih `n` karaktera niske `s2` u nisku `s1`.
- `strlen(char *s)` – funkcija koja računa dužinu niske `s`, odnosno kao rezultat vraća informaciju od koliko karaktera se sastoji niska `s`.
- `strcmp(char *s1, char *s2)` – funkcija koje leksički poredi dvije niske i kao rezultat vraća: 0 – ako su niske iste; pozitivan broj ako je niska `s1` leksički veća od niske `s2`; negativan broj, ako je niska `s1` leksički manja od niske `s2`.
- `strcmpi(char *s1, char *s2)` – funkcija koja radi isto kao i funkcija `strcmp`, ali ne pravi razliku između malih i velikih slova.
- `strncmp(char *s1, char *s2, int n)` – funkcija koja radi isto kao i funkcija `strcmp`, ali poredi samo prvih `n` karaktera.
- `strchr(char *s, char c)` – funkcija koja vraća pokazivač na prvo pojavljivanje karaktera `c` u niski `s`. Ako se karakter `c` ne pojavljuje u niski `s` rezultat funkcije je `NULL`.
- `strrch(char *s, char c)` – funkcija koja vraća pokazivač na posljednje pojavljivanje karaktera `c` u niski `s`.
- `strstr(char *s1, char *s2)` – funkcija koja vraća pokazivač na prvo pojavljivanje niske `s2` u niski `s1`. Ako se niska `s2` ne pojavljuje u niski `s1`, onda se vraća `NULL`.
- `strdup(char *s)` – funkcija koja kao rezultat vraća nisku koja je kopija niske `s`.

Elektronska verzija

- `strlwr(char *s)` – funkcija koja sve karaktere niske `s` koji su velika slova konvertuje u mala slova i vraća opet kao rezultat nisku `s`.
- `strupr(char *s)` – funkcija koja sve karaktere niske `s` koji su mala slova konvertuje u velika slova i vraća opet kao rezultat nisku `s`.
- `strrev(char *s)` – funkcija koja okreće sadržaj niske `s` i opet vraća `s` kao rezultat.
- `strset(char *s, char c)` – funkcija koja sve karaktere niske `s` postavlja na karakter `c`.
- `strnset(char *s, char c, int n)` – funkcija koja prvih `n` karaktera niske `s` postavlja na karakter `c`.
- `strtok(char *s1, char *s2)` – funkcija koja izdvaja prvi token iz niske `s1` koji ne sadrži karaktere iz niske `s2`. Ako ne nađe, vraća `NULL`. Ako pronađe, postavlja oznaku za kraj niske na kraju prvog takvog tokena i vraća adresu njenog prvog znaka.
- `strspn(char *s1, char *s2)` – funkcija koja kao rezultat vraća broj karaktera iz početnog dijela niske `s1`, koji se nalaze u niski `s2`.
- `strcspn(char *s1, char *s2)` – funkcija koja kao rezultat vraća broj karaktera iz početnog dijela niske `s1`, a koji se ne nalaze u niski `s2`.
- `strpbrk(char *s1, char *s2)` – funkcija koja vraća pokazivač na prvi karakter niske `s1` koji je sadržan u niski `s2`.

Ilustrujmo primjerima upotrebu nekih od ovih funkcija.

Primjer 8.8. Prikažimo dio programa koji koristi funkcije `strcpy`, `strcat` i `strncat`.

```
...
//najprije zauzmimo dovoljno prostora da mozemo nesmetano da spajamo
    niske
char s1[40];
char s2[40];
char* s3 = "Programski " ;
char* s4= "jezik C" ;
//prekopiramo s3 u s1 i u s2
strcpy(s1,s3);
```

Elektronska verzija

```
strcpy(s2,s3);
//pozovimo funkcije za spajanje niski
strcat(s1,s4);
strncat(s2,s4,5);
printf ( "Nakon primjene funkcije strcat s1= %s\n",s1);
printf ( "Nakon primjene funkcije strncat s2= %s\n",s2);
...
```

Na ekranu ćemo dobiti

```
Nakon primjene funkcije strcat s1= Programski jezik C
Nakon primjene funkcije strncat s2= Programski jezik
```

Primjer 8.9. Ilustrujmo upotrebu funkcija za poređenje dvije niske. Treba napomenuti da nije tačno propisano koja će se negativna (odnosno pozitivna) vrijednost vratiti kao rezultat, ukoliko je prva niska leksički manja (odnosno veća) od druge. U zavisnosti od prevodioca i same funkcije za poređenje (a imamo nekoliko takvih funkcija), najčešće se javljaju dvije varijante: vraća se vrijednost -1 (odnosno 1), ili se vraća broj koji je jednak razlici rednih brojeva u ASCII kôdu prvog para odgovarajućih karaktera dvije niske koji su različiti.

Najprije analizirajmo funkciju `strcmp`.

```
...
char *s1 = "abcd";
char *s2 = "abcm";

int i = strcmp(s1,"abcd");
printf("Rezultat poredjenja dvije iste niske =%d\n",i);

int j = strcmp(s1,s2);
printf("Rezultat poredjenja ako je prva niska leksicki manja od
    druge=%d\n",j);

int k = strcmp(s2,s1);
printf("Rezultat poredjenja ako je prva niska leksicki veca od
    druge=%d\n",k);
...
```

Na ekranu se, u jednoj verziji prevodioca dobija sljedeći prikaz

```
Rezultat poredjenja dvije iste niske =0
Rezultat poredjenja ako je prva niska leksicki manja od druge=-1
Rezultat poredjenja ako je prva niska leksicki veca od druge=1
```

Elektronska verzija

Sada posmatrajmo i funkciju `strcmpi`.

```
...
char *s1 = "abcd";
char *s2 = "Abcg";
int i = strcmp(s1,s2);
printf("Rezultat poredjenja funkcijom strcmp=%d\n",i); //dobija se 1
    jer 'a'=97>65='A'

int j = strcmpi(s1,s2);
printf("Rezultat poredjenja funkcijom strcmpi=%d\n",j); //dobija se
    negativna vrijednost jer 'd'<'g'
...
```

i prikaz na ekranu po pokretanju programa.

```
Rezultat poredjenja funkcijom strcmp=1
Rezultat poredjenja funkcijom strcmpi=-3
```

Primijetimo da je funkcija `strcmpi` vratila vrijednost -3, što je zapravo razlika rednih brojeva karaktera `'d'` i `'g'` u ASCII kôdu. Bez obzira na to o kom se prevodiocu radi, ta vrijednost mora biti manja od nule.

Uradimo sada još dva malo složenija zadatka.

Primjer 8.10. Napisati program koji broji pojavljivanje date riječi u datom tekstu.

```
1  #include <stdio.h>
2  #include <string.h>
3  int main ()
4  {
5      char s[] = "Mi ucimo programski jezik C. C je vrlo popularan
           jezik.";
6      char r[] = "jezik";
7      char *p;
8      int brojac = 0;
9
10     p = strtok(s," .");//druga niska se sastoji od razmaka i tacke
11     printf("%s\n",p);
12     while (p != NULL)
13     {
14
15         if(strcmp(p,r) == 0)
16             brojac++;
```

Elektronska verzija

```
17          p = strtok(NULL, " .");
18      }
19      printf("Rijec %s se pojavljuje u tekstu %d puta \n",r,brojac);
20      return 0;
21  }
```

Primjer 8.11. Napisati program koji ispisuje sve indekse na kojima se pojavljuje dati karakter u datoj nisci. Takođe, ispisati i sva pojavljivanja.

```
1   #include <stdio.h>
2   #include <string.h>
3   int main ()
4   {
5       char s[] = "Mi ucimo programski jezik C. C je vrlo popularan
            jezik.";
6       char *p;
7       int k = 1;
8       p = strchr (s,'i');
9       while (p != NULL)
10      {
11          printf("Karakter i je pronadjen na poziciji %d\n",p-s+1);
12          printf("%d. pojavljivanje karaktera i je : \n",k);
13          printf("\" %s \" \n",p);
14
15          p = strchr(p+1,'i');
16          k++;
17      }
18    return 0;
19  }
```

8.7 Pokazivači i višedimenzionalni nizovi

Analogija između pokazivača i nizova može se uspostaviti i u slučaju višedimenzionalnih nizova.

Kao što je slučaj i sa jednodimenzionalnim nizovima, ime niza se konvertuje u pokazivač na odgovarajući tip, osim u slučajevima kada je u pitanju argument operatora **sizeof** ili operatora &. Tako, ako imamo proizvoljan višedimenzionalan niz

```
tip niz[dim1][dim2]...[dimN];
```

Elektronska verzija

ime niza `niz` će se konvertovati u odgovarajući pokazivač tipa
`tip (*)[dim2]...[dimN]`.
Na primjer, pretpostavimo da imamo dvodimenzionalni niz

```
int matrica[5][6];
```

Ako bismo željeli da ovu matricu pošaljemo kao argument neke fukcije, zapravo bi se smatralo da je argument funkcije pokazivač na jednodimenzionalni niz cijelih brojeva, koji je dimenzije 6.

Podsjetimo se i da će ovakvom deklaracijom u memoriji biti zauzeto 30 prostora za smještanje elemenata matrice i da se ti prostori u memoriji nalaze na susjednim memorijskim lokacijama: prvo idu redom elementi prve vrste, pa elementi druge vrste, itd. Praktično, deklaracijom matrice na navedeni način imamo situaciju da je svaki od izraza `matrica[0]`, `matrica[1]`,`matrica[2]`, `matrica[3]` i `matrica[4]` niz od po 6 elemenata. Pored toga, pošto nizove, praktično, možemo posmatrati i kao pokazivače na prvi element tog niza, zaključujemo i da izraz `matrica[0]` sadrži adresu prvog elementa prve vrste (tj. elementa `matrica[0][0]`), izraz `matrica[1]` sadrži adresu prvog elementa druge vrste (elementa `matrica[1][0]`), itd. Pošto su `matrica[0]`, `matrica[1]`... jednodimenzionalni cjelobrojni nizovi, oni se mogu konvertovati i u pokazivače na tip `int`. Primjerom ilustrujmo kako bi se na nekoliko različitih načina moglo pristupiti elementima matrice:

```
printf("%d\n",**matrica);

printf("%d\n",*(matrica[2]));

printf("%d\n",*(*(matrica+1)));

printf("%d\n",*(matrica[4])+2);

printf("%d\n",*(*(matrica+1)+3));

printf("%d\n",(*(matrica+2))[4]);
```

Kao što vidimo, matrice (a slično i nizove još većih dimenzija) možemo posmatrati kao “nizove nizova”, ili kao pokazivače na nizove.

Međutim, višedimenzionalni nizovi i nizovi pokazivača nisu isto. Posmatrajmo

```
int matrica[5][6];
int *pokazivac[5];
```

Elektronska verzija

Promjenljiva `matrica` je pravi dvodimenzionalni niz, čijom je deklaracijom zauzet memorijski prostor od 30 brojeva tipa **`int`**. Sa druge strane, drugom deklaracijom smo definisali samo niz pokazivača (kojih ukupno ima 5), ali nije određen (alociran) memorijski prostor kojim taj niz pokazivača raspolaže. Da bi se niz pokazivača mogao praktično koristiti, mora se alocirati odgovarajući memorijski prostor (ili navođenjem inicijalizatora, ili dinamičkom alokacijom, o kojoj će biti govora kasnije). Pored toga, prilikom takve alokacije, svi pokazivači (iz tog niza pokazivača) ne moraju pokazivati na niz iste dužine (kod matrice imamo tačno 5 nizova jednake dužine koja iznosi 6). Takođe, za razliku od matrice, gdje se elementi alociraju na susjednim memorijskim lokacijama, alokacijom memorijskih prostora za pojedinačne nizove pokazivača, to ne mora biti slučaj.

Primjer 8.12. Pretpostavimo da je potrebno obezbijediti čuvanje niza stringova, koji mogu biti različite dužine. To možemo uraditi na dva načina, prvi je da definišemo dvodimenzionalni niz, a drugo da napravimo niz pokazivača na karaktere. Na konkretnom primjeru, to bi izgledalo ovako.

```
char studenti1[][20] = {"Milan", "Sara", "Milica", "Pero", "Slobodan",
    "Tatjana"};
char * studenti2[] = {"Milan", "Sara", "Milica", "Pero", "Slobodan",
    "Tatjana"};
```

8.8 Argumenti komandne linije

Moguće je, a u nekim situacijama je i neizostavno, da se programu proslijede određeni parametri, koje program prihvata prilikom samog pokretanja. Ovi parametri se prosljeđuju preko tzv. argumenata komandne linije, a prihvata ih `main` funkcija. Do sada smo u udžbeniku radili sa `main` funkcijom koja ne uzima argumente (uvijek smo pisali `main()`). Međutim, ako želimo da programu "proslijedimo" neke parametre prilikom samog pokretanja, potrebno je da tome "prilagodimo" i funkciju `main`. Funkcija `main` može da prihvati dva parametra: prvi je broj argumenata koji se prosljeđuju programu, dok je drugi niz argumenata koji se prosljeđuju. Prvi argument je tipa **`int`** i obično se označava sa `argc`, a drugi je niz pokazivača na **`char`**, obično nazvan `argv`. Tako ovaj oblik funkcije `main` izgleda:

```
int main(int argc, char *argv[])
{ .... }
```

Elektronska verzija

i prihvatanje argumenata prilikom pokretanja programa izgleda ovako: u promjenljivu `argc` se smješta informacija o broju argumenata koji su proslijeđeni funkciji `main` pri pokretanju programa, uvećan za 1. Ako nema argumenata komandne linije, tada je `argc` jednako 1. Argumenti koji se prosljeđuju u program se šalju kao niske, koje su elementi niza `argv`, s tim što je početni element ovog niza (`argv[0]`) uvijek naziv datoteke/programa. Tako je `argv[1]` prvi argument koji se prosljeđuje, `argv[2]` drugi, itd. Argumenti se u program šalju tako što se odvajaju razmacima.

Primjer 8.13. Napisati program koji ispisuje zbir numeričkih argumenata komandne linije.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    if(argc < 2){
        printf("Greska nedostaje argument komandne linije:\n");
        return -1;
    }


    int suma = 0;
    int i = 1;
    while(i < argc){
        suma += atoi(argv[i]);
        i++;
    }
    printf("Suma numerickih argumenata=%d\n",suma);

    return 0;
}
```

Primijetimo da smo u okviru ovog zadatka koristili funkciju `atoi`, koja konvertuje podatak tipa `char*` u cio broj.

Primjer 8.14. Napisati program koji na osnovu realnog broja n koji se zadaje kao argument komandne linije ispisuje realne brojeve iz intervala $[-n, n]$ sa korakom 0.5.

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

int main(int argc, char *argv[])
```

Elektronska verzija

```
5  {
6      if(argc != 2){
7          printf("Greska nedostaje argument komandne linije:\n");
8          return -1;
9      }
10
11     float f = atof(argv[1]);
12     float i = (-1) * f;
13     for(; i <= f; i += 0.5)
14         printf("%f\t",i);
15     printf("\n");
16
17     return 0;
18 }
```

Funkcija `atof` konvertuje podatak tipa `char*` u realan broj.

8.9 Pokazivači na funkcije

Iako početniku to može izgledati pomalo čudno, ili nepotrebno, u nekim situacijama je zgodno imati mehanizam kojim se funkcija može proslijediti kao argument drugoj funkciji. To se u programskom jeziku C ne može uraditi direktno, ali može pomoću pokazivača na funkciju. Nakon što objasnimo način kako se definišu pokazivači na funkcije, nekim primjerima ćemo ilustrovati situacije kada je to praktično.

Pokazivač na funkciju se deklariše na sljedeći način:

```
tipRezultata (*ime)(tip1 arg1, tip2 arg2, ..., tipN argN);
```

gdje je `ime` pokazivač na funkciju koja uzima `N` argumenata (iz deklaracije se vidi da se argumenti zapisuju u standardnom obliku – prvo naziv tipa, pa onda naziv formalnog argumenta). Funkcija na koju pokazuje pokazivač vraća vrijednost `tipRezultata`. Zagrade su obavezne. Na primjer, deklaracijom

```
int (*pokfun)(char c, float x);
```

je deklarisan pokazivač `pokfun` koji pokazuje na funkciju koja uzima dva argumenta (jedan je tipa `char`, drugi je tipa `float`) i koja za rezultat vraća cijeli broj tipa `int`. Ako imamo deklarisanu neku funkciju, na primjer, funkciju `f`, koja za argument uzima podatke tipa `char` i `float`, a za rezultat vraća `int`

```
int f(char, float);
```

Elektronska verzija

tada je moguće dodijeliti

```
pokfun = &f;
```

Odnosno, pokazivač na funkciju `pokfun` sada dobija vrijednost adrese funkcije `f`. Funkcija `f` se sada može pozvati sa, na primjer

```
(*pokfun)('c', 2.25);
```

Zagrade oko izraza `*pokfun` su obavezne zbog prioriteta operatora. Kao što vidimo, dereferenciranjem pokazivača na funkciju dobijamo funkciju, koja se, onda, poziva na uobičajen način.

Upotreba pokazivača na funkcije dobija pravi smisao kada želimo da funkcija bude argument neke druge funkcije. U tom slučaju, koristimo pokazivač na funkciju i njega šaljemo kao argument. Na primjer, funkcija

```
void g(int (*)(char, float));
```

za argument uzima pokazivač na funkciju koja za argumente uzima **`char`** i **`float`**, a kao rezultat vraća int. Funkcija `g` bi zapravo mogla da uzme funkciju `f` (naravno preko pokazivača) kao argument, na sljedeći način:

```
g(pokfun);
```

Složene deklaracije pokazivača na funkcije, kao i funkcija koje uzimaju takve pokazivače kao argumente, djelimično možemo pojednostaviti upotrebom ključne riječi **`typedef`**. U prethodnom primjeru smo najprije mogli uvesti novi tip podatka

```
typedef int (*PF)(char, float);
```

pa bi onda deklaracija funkcije `g` izgledala

```
void g(PF f);
```

Primjer 8.15. U ovom primjeru kompletiramo zadatak koji koristi prethodno uvedene koncepte.

```
1 #include <stdio.h>
2 typedef int (*PF)(char, float);//uvedemo novo ime za tip pokazivaca na
    funkciju
3 void g(PF f); //deklaracija od g
4 int f1(char c, float f){
```


Elektronska verzija

```
5     //proizvoljno definisemo funkciju na neki nacin
6     return (int)f + c;
7  }
8  int main(){
9    g(&f1);
10   return 0;
11
12 }
13
14 void g(PF f) //definicija od g
15 {
16   char c = 'a';//podesimo stvarne argumente za funkciju f
17
18   float broj = 5.2;
19
20   printf("%d\n", (*f)(c,broj));
21 }
```

Možemo kreirati i nizove pokazivača na funkcije. Kao što je to slučaj i sa drugim nizovima, svi elementi (pokazivači na funkcije) ovakvih nizova moraju biti istog tipa. Takođe, moguće je izvršiti i inicijalizaciju ovakvih nizova. Sintaksa kreiranja ovakvih nizova bi izgledala ovako:

```
int (*niz[5]) (int, int) = {&f1, &f2, &f3, &f4, &f5};
```

čime bismo deklarisali niz od 5 pokazivača na 5 funkcija koje su redom `f1`, `f2`, `f3`, `f4` i `f5`. Sve ove funkcije moraju biti takve da za argument uzimaju dva cijela broja i kao rezultat vraćaju cijeli broj. Pomoću promjenljive `niz` možemo pozivati ove funkcije, naravno, poštujući odgovarajuća sintaksna pravila. Na primjer, ako funkciju `f2` želimo da pozovemo na argumente 5 i 6, to bismo uradili ovako:

```
(*niz[1])(5,6);
```

Primjer 8.16. Algoritme za sortiranje nizova ćemo detaljno razmatrati u Poglavlju 11. U međuvremenu, iskoristimo priliku da uvedemo tzv. "ugrađeni" algoritam za sortiranje podataka, koji je zasnovan na poznatom algoritmu 'brzog sorta" (engl. quicksort). Funkcija `qsort` za argumente uzima niz koji se sortira, broj elemenata, veličinu memorije koju niz zauzima, kao i pokazivač na funkciju koja definiše kriterijum za sortiranje (ta funkcija se naziva i komparator). U ovom primjeru ćemo napisati program koji u rastućem poretku sortira elemente cjelobrojnog niza, koji se unosi sa tastature.

Elektronska verzija

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  int uporedi_vrijednost(const void *a, const void *b){
6      return *((int *)a) - *((int *)b);
7  }
8
9  int main()
10 {
11     int n;
12     printf("Unesite broj elemenata niza:\n");
13     scanf("%d",&n);
14
15     int a[n];
16     int i;
17
18     for(i = 0; i < n; i++){
19         printf("Unesite %d. element niza:\n",(i+1));
20         scanf("%d",&a[i]);
21     }
22
23     int s = sizeof(int);
24
25     qsort(a,n,s,&uporedi_vrijednost);
26
27     printf("Sortiran niz po brojnoj vrijednosti:\n");
28     for(i = 0; i < n; i++)
29         printf("%d\n",a[i]);
30     return 0;
31 }
```

Da komparator može da bude i neka druga funkcija, možemo vidjeti iz narednog primjera.

Primjer 8.17. Sortirati niz koji se unosi sa tastature, a kriterijum za sortiranje je broj različitih cifara.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  int broj_razlicitih_cifara(int n){
6      int niz[10];
```

Elektronska verzija

```
7     int i;
8 
9     for(i = 0; i < 10; i++)
10        niz[i] = 0;
11 
12    while(n > 0){
13        int cifra = n%10;
14        if(niz[cifra] > 0){
15            n /= 10;
16            continue;
17        }
18        else{
19            niz[cifra]++;
20            n/=10;
21        }
22    }
23    int broj_razlicitih=0;
24    for(i = 0; i < 10; i++)
25        broj_razlicitih += niz[i];
26 
27    return broj_razlicitih;
28 
29 }
30 
31 int uporedi_broj_cifara(const void *a, const void *b){
32    int ak = *(int *)a;
33    int bk = *(int *)b;
34    return (broj_razlicitih_cifara(ak) - broj_razlicitih_cifara(bk));
35 }
36 
37 int main()
38 {
39    int n;
40    printf("Unesite broj elemenata niza:\n");
41    scanf("%d",&n);
42 
43    int a[n];
44    int i;
45 
46    for(i = 0; i < n; i++){
47        printf("Unesite %d. element niza:\n",(i+1));
48        scanf("%d",&a[i]);
49    }
50 
51    int s = sizeof(int);
```

Elektronska verzija

```
52
53     qsort(a,n,s,&uporedi_broj_cifara);
54
55     printf("Sortiran niz po broju razlicitih cifara:\n");
56     for(i = 0; i < n; i++)
57         printf("%d\n",a[i]);
58     return 0;
59 }
```

8.10 Pitanja i zadaci

1. Objasni šta su pokazivači i kakva je razlika između "običnih" i pokazivačkih promjenljivih.

2. Kolika je vrijednost promjenljive `x`?

```
int a = 100;
int *pa = &a;

int b = 250;
int *pb = &b;

int c, d;
c = 12, d = 13, *pa = 1;

int x = a * b - c + d;
```

3. Ako je funkcija `f` zadata sljedećim kôdom

```
void f(int *p1, int *p2, int a){
    *p1 = *p2 + a;
    a = (*p1) + (*p2);
}
```

   i ako su u glavnom dijelu programa deklarisane sljedeće promjenljive

```
int x = 110, y = 22, z = 33;
int *px = &x, *py = &y;
```

   a) Koji od navedenih poziva funkcije su ispravni?

      - `f(px,py,z);`

Elektronska verzija

- `f(*px,*py,z);`
- `f(px,py,&z);`
- `f(x,y,z);`
- `f(&x,&y,z);`
- `f(x,*y,*z);`

b) Nakon ispravnog poziva funkcije, kolike su vrijednosti promjenljivih `x`, `y` i `z`?

4. Šta se ispisuje sljedećim programom? Obrazloži odgovor.

```
#include <stdio.h>
void f(int *p){
    p += 5;
    p--;
    p--;
}

int main(){
    int niz[] = {10, 20, 30, 40, 50, 60, 70};
    int *q = niz;

    f(q);
    printf("%d\n",*q);
}
```

5. Ako je poznato da su `pa` i `pb` pokazivači na promjenljive tipa **`int`** i da su njihove vrijednosti `62fe20` i `62fe04`, koliko memorijskih blokova se nalazi između njih?

6. Objasniti razliku između pokazivača i konstantnog pokazivača. Ilustrovati obrazloženje odgovarajućim primjerom.

7. Ako je `x` niz dužine n, `i` cjelobrojna promjenljiva iz intervala $[0, n]$, da li je `x[i]` ekvivalento nekom od sljedećih zapisa, ako jeste kojem i zašto, a ako nije zašto nije?

   a) `x[0] + i`

   b) `*x + i`

   c) `*(x + i)`

   d) `&x[i]`

Elektronska verzija

e) `&(x + i)`

8. Uporediti sličnosti i razlike između nizova i pokazivača. Ilustrovati primjerom.

9. Kakva je razlika između zapisa
   - `double *a(double, int);` i
   - `double (*b)(double, int);`

10. Napisati funkciju `f` koja za argument uzima nisku, a kao rezultat za svako veliko slovo koje se pojavi u toj niski ispisuje njegov broj pojavljivanja.

11. Napisati funkciju koja za argument uzima nisku sastavljenu samo od cifara, a kao rezultat vraća brojnu vrijednost te niske.

12. Napisati i testirati definicije sljedećih funkcija iz zaglavlja `string.h`
    a) `strcpy`
    b) `strcmp`
    c) `strrch`
    d) `strupr`
    e) `strnset`

13. Ako se rastojanje između dvije niske definiše kao prva pozicija na kojoj sadrže različite karaktere, napisati program koji za datu nisku `s` među n niski, koje se unose sa tastature, pronalazi onu koja je na najmanjem rastojanju od niske `s`. Broj n se takođe unosi sa tastature.

14. Primjerom ilustrovati razliku između funkcija `strcpy` i `strncpy`.

15. Napisati program koji broji broj pojavljivanja date rečenice u datom tekstu. Rečenica može da se završi tačkom, upitnikom ili uzvičnikom.

16. Napisati funkciju koja za argument uzima nisku, a kao rezultat vraća 1 ako se niska završava samoglasnikom, u suprotnom funkcija vraća 0. Testirati funkciju u glavnom dijelu programa.

17. Napisati funkciju koja u niski `s` određuje dužinu najduže riječi koja počinje i završava se istim simbolom i vraća pokazivač na početak te riječi.

18. Napisati funkciju kojom se ispisuju sve riječi prisutne u datoj niski `s`, ako se neka riječ pojavljuje više puta ispisati svako njeno pojavljivanje. Riječi ispisivati po jednu u redu.

Elektronska verzija

19. Napisati funkciju koja za argument uzima nisku koja se sastoji samo od slova, a kao rezultat vraća broj različitih slova koja se nalaze u niski. Pri tome se ne pravi razlika između malih i velikih slova, npr. `'a'` i `'A'` se smatraju istim slovom. Testirati funkciju u glavnom dijelu programa.

20. Napisati funkciju `void modifikacija` koja za argument uzima nisku i mijenja je tako sto svaki samoglasnik zamijeni karakterom zvjezdica `'*'`, na primjer riječ Beograd nakon primjene funkcije postaje B**gr*d. Testirati funkciju u glavnom dijelu programa.

21. Napisati program koji za dvije niske, koje se unose kao argumenti komandne linije, određuje koliko se uzastopnih karaktera prve niske nalazi u drugoj niski, počev od početka.

22. Napisati program koji prebrojava koliko među zadatim argumentima komandne linije ima istih.

23. Koristći funkciju `qsort`, sortirati niz cijelih brojeva, a kriterijum za sortiranje je zbir cifara broja.

24. Napisati program koji koristi trapeznu formulu za integraciju funkcije i koji se može primijeniti za integraciju različitih funkcija (definisati funkciju za integraljenje koja, između ostalih argumenata uzima i pokazivač na funkciju koju treba integraliti).

Elektronska verzija

9 Strukture i unije

Do sada smo se u udžbeniku susretali sa prostim tipovima podataka (brojevima i karakterima), te nizovima takvih podataka (cjelobrojnim nizovima, niskama karaktera i sl.). Iako nizovi u svim programskim jezicima predstavljaju moćan alat za rješavanje velikog broja problema, značajno ograničenje predstavlja činjenica da su svi elementi niza uvijek istog tipa. U praksi se često javljaju situacije gdje se jedan složeniji podatak predstavlja podacima različitih tipova. Na primjer, podatak kojim se opisuje Student bi bio složen podatak, koji se sastoji od nekoliko članova, koji su različitog tipa. Tako bi se ime i prezime predstavili niskama karaktera, broj indeksa bi mogao biti cijeli broj, godina studija, takođe, cijeli broj itd. Osnovni podaci o knjizi se mogu predstaviti na sljedeći način: naziv knjige, ime i prezime autora bi bili niske karaktera, dok bi, recimo, broj strana ili godina izdavanja knjige bili podaci tipa `int`.

Za prestavljanje složenih podataka, čiji su članovi podaci različitih tipova, u programskom jeziku C koristimo strukture.

9.1 Strukture

Strukture su složeni tipovi podataka u kojima se grupišu podaci koji ne moraju biti istog tipa, a koji su povezani na neki logičan način. Definisanjem strukture uvodi se novi tip podataka, koji se dalje može koristiti za definisanje promjenljivih tog novog tipa.

Struktura se deklariše na sljedeći način:

```
struct imeStrukture {
    tip1 ime1;
    tip2 ime2;
    .... ....
    tipN imeN;
};
```

Kao što vidimo, za deklarisanje strukture koristi se ključna riječ `struct`, nakon koje slijedi ime strukture. U okviru vitičastih zagrada se definiše spisak članova (polja) strukture. Za svakog člana strukture se navodi njegov tip i ime, a te deklaracije se odvajaju znakom `;`.

Elektronska verzija

Na primjer, ako bismo htjeli da definišemo jednostavnu strukturu kojom bi se mogli opisati osnovni podaci o studentu (ime, prezime i broj indeksa), onda bismo to mogli uraditi na sljedeći način:

```
struct Student {
    char ime[50];
    char prezime[50];
    int brojIndeksa;
};
```

Deklaracijom strukture se samo definiše novi tip podatka, ali se ne rezerviše memorijski prostor. Ako želimo da napravimo dvije promjenljive `s1` i `s2`, koje bi bile tipa ovako definisane strukture, koristimo sljedeću sintaksu:

```
struct Student s1,s2;
```

Promjenljive tipa strukture možemo definisati i odmah unutar deklaracije strukture, što bi u opštem slučaju izgledalo ovako:

```
struct imeStrukture {
    tip1 ime1;
    tip2 ime2;
    .... ....
    tipN imen;
} promjenljiva1, promjenljiva2,...;
```

ili u konkretnom primjeru sa studentima:

```
struct Student {
    char ime[50];
    char prezime[50];
    int brojIndeksa;
} s1,s2;
```

U slučaju da se promjenljive neke strukture uvode samo na jednom mjestu, onda čak i ne moramo davati ime strukturi, tj. možemo pisati

```
struct {
    tip1 ime1;
    tip2 ime2;
    .... ....
    tipN imeN;
} promjenljiva1, promjenljiva2,...;
```

Elektronska verzija

Moguće je dodijeliti i početne vrijednosti pojedinačnim članovima promjenljive, što bi u primjeru prethodne strukture i promjenljive `s3` izgledalo ovako:

```
struct Student s3 = {"Pero", "Peric", 12345};
```

U opštem slučaju, vrijednosti u listi odgovaraju članovima promjenljive u poretku koji odgovara njihovoj deklaraciji.

Kako bismo izbjegli riječ **`struct`** prilikom deklarisanja promjenljivih koje su tipa neke strukture, prisjetimo se ključne riječi **`typedef`**, koju možemo da koristimo i kada radimo sa strukturama. Tako bismo našu strukturu Student mogli deklarisati na ovaj način:

```
typedef struct {
    char ime[50];
    char prezime[50];
    int brojIndeksa;
} Student;
```

te bi onda promjenljive tipa Student deklarisali kao

```
Student s1,s2;
...
```

Pristup pojedinačnim članovima strukture se vrši pomoću operatora tačka . U primjeru našeg studenta `s3`, broju indeksa bi se pristupilo pomoću
`s3.brojIndeksa`.

Primjer 9.1. Primjerom ilustrujmo najjednostavniju upotrebu struktura.

```
#include <stdio.h>
#include <string.h>
typedef struct {
    char ime[50];
    char prezime[50];
    int brojIndeksa;
} Student;

int main(){

   printf("Velicina strukture: %d\n",sizeof(Student));

   Student s1 = {"Pero", "Peric", 12345};
   printf("Ime prvog studenta: %s\n",s1.ime);
   printf("Prezime prvog studenta: %s\n",s1.prezime);
```

Elektronska verzija

```
16      printf("Broj indeksa: %d\n",s1.brojIndeksa);
17
18      Student s2;
19      strcpy(s2.ime,"Bojana");
20      strcpy(s2.prezime,s1.prezime);
21      s2.brojIndeksa = s1.brojIndeksa + 1;
22
23      printf("Ime drugog studenta: %s\n",s2.ime);
24      printf("Prezime drugog studenta: %s\n",s2.prezime);
25      printf("Broj indeksa: %d\n",s2.brojIndeksa);
26      return 0;
27  }
```

Kao rezultat pokretanja programa, na ekranu bismo mogli dobiti sljedeći ispis:

```
Velicina strukture: 104
Ime prvog studenta: Pero
Prezime prvog studenta: Peric
Broj indeksa: 12345
Ime drugog studenta: Bojana
Prezime drugog studenta: Peric
Broj indeksa: 12346
```

Kao što vidimo, ukupna veličina strukture je 104, jer struktura sadrži dva niza karaktera veličine po 50 bajtova (svaki karakter zauzima po jedan bajt) i jedan cio broj (4 bajta). Pojedinačnim članovima pristupamo pomoću tačke, a dalje sa podacima radimo kao i ranije sa običnim promjenljivima.

Strukture mogu biti i ugniježdene, tj. unutar jedne strukture možemo deklarisati članove koji su tipa neka druga struktura.

Ako bismo, na primjer, htjeli da napravimo strukturu ParStudenata, koja sadrži podatke o dva studenta, onda bi to izgledalo ovako:

```
struct ParStudenata{
struct Student a;
struct Student b;
}
```

Ako smo uveli novi tip podatka `Student`, možemo da izbjegnemo riječ **`struct`**, a, isto tako, i za par studenata možemo da uvedemo novi tip podatka. Uz te pretpostavke, strukturu bismo napravili na sljedeći način

```
typedef struct{
Student a;
Student b;
```

Elektronska verzija

```
}ParStudenata;
```

9.1.1 Operacije na strukturama

Na strukturama su definisane neke operacije koje važe i na prostim tipovima podataka. Već smo pomenuli da se može vršiti inicijalizacija vrijednosti pojedinačnih članova strukture, tako što se, nakon znaka za operator dodjele (operatora =), u vitičastim zagradama navode vrijednosti za pojedinačne članove, redom kako su deklarisani u strukturi.

Moguće je i jednoj promjenljivoj dodijeliti vrijednost druge promjenljive (ako su obe promjenljive istog tipa strukture). Na našem primjeru strukture Student, to bi moglo da izgleda ovako:

```
Student s3 = {"Pero","Peric",12345};
Student s4 = s3;
```

Nije moguće koristiti operator == direktno na strukturama, tj. dvije strukture nije moguće porediti direktno, već treba porediti član po član.

Primjer 9.2. Ako bismo imali dva studenta `s1` i `s2`, kao u Primjeru 9.1, onda bi poređenje, da li su ta dva studenta jednaka, bilo realizovano na sljedeći način:

```
...
  if(!strcmp(s1.ime,s2.ime) && !strcmp(s1.prezime,s2.prezime) &&
      s1.brojIndeksa == s2.brojIndeksa)
    printf("Podaci o dva studenta su isti\n");
  else
    printf("Podaci o dva studenta nisu isti\n");
...
```

Pomenimo i da operator tačka (.), pomoću koje se pristupa članu strukture spada u grupu operatora sa najvišim prioritetom i ima lijevu asocijativnost. Pošto ovaj operator ima najviši prioritet, tada bi, na primjer, izraz

```
++s1.brojIndeksa;
```

bio ekvivalentan izrazu

```
++(s1.brojIndeksa);
```

Ako bismo napravili jedan par studenata, koristeći tipove koje smo do sada uveli, onda bismo u sljedećem kôdu

Elektronska verzija

```
...
   ParStudenata par1;
   par1.a = s1;
   par1.b = s2;
   printf("Ime prvog studenta u paru: %s\n",par1.a.ime);
...
```

zapisom `par1.a.ime` pristupili imenu studenta `a`, koji je prvi student u paru studenata `par1`. Operator `.` djeluje sa lijeva na desno, te je ovaj zapis ekvivalentan zapisu `(par1.a).ime`.

9.1.2 Pokazivači na strukture

Definisanje pokazivača na strukturu se vrši na analogan način kao i kod drugih tipova podataka. Kada imamo pokazivač na strukturu, onda se za pristup pojedinim članovima strukture ne pristupa pomoću operatora tačka (.), već pomoću operatora koji podsjeća na strelicu, a piše se pomoću dva karaktera: karaktera minus i karaktera veće `->`. U opštem slučaju, to bi izgledalo ovako:

```
struct imeStrukture {
    tip1 ime1;
    tip2 ime2;
    .... ....
    tipN imeN;
};
...
struct imeStrukture s1;//promjenljiva za koju se alocira memorijski
    prostor
struct imeStrukture * pok = &s1;
...
pok -> ime1...
...
pok -> ime2...
...
```

Treba napomenuti da se samo deklarisanjem pokazivača na strukturu ne alocira memorijski prostor na koji pokazuje pokazivač. Podsjetimo se da slično važi i za pokazivače na druge (uključujući i proste) tipove. Alokacija memorije na koju pokazuje pokazivač na strukturu ćemo pomenuti kada budemo razmatrali dinamičku alokaciju memorije, u Poglavlju 12.

Primjer 9.3. Za sada, iskoristimo našu strukturu Student i posmatrajmo sljedeći programski kôd i komentare unutar njega.

Elektronska verzija

```
#include <stdio.h>
#include <string.h>
typedef struct{
    char ime[50];
    char prezime[50];
    int brojIndeksa;
}Student;

int main(){

   Student* pok;
   Student s1 = {"Pero", "Peric", 12345};

   // pok->brojIndeksa = 100; pogresno, nije alociran prostor
   // printf("Ime preko pokazivaca: %s\n",pok->ime); pogresno, nije
       alociran prostor

   pok  = &s1;//sada znamo na sta pok pokazuje
   pok->brojIndeksa = 100;// sada je u redu
   printf("Ime preko pokazivaca: %s\n",pok->ime);// i ovo je u redu

   printf("Broj indeksa od s1: %d\n",s1.brojIndeksa);//ispisuje se
       vrijednost 100, jer pok pokazuje na s1
   return 0;
}
```

Dakle, da bi pokazivač na strukturu mogao dalje da se koristi, najprije mu se mora dodijeliti vrijednost adrese neke strukture, za koju je već rezervisan memorijski prostor.

9.1.3 Strukture kao argumenti funkcija

Strukture mogu biti argumenti funkcija, a mogu biti i rezultati funkcija. Ako je struktura argument funkcije, tada se ona (kao i svaki drugi podatak) u funkciju prenosi po vrijednosti. S obzirom na činjenicu da strukture često sadrže velike podatke, prenos struktura u funkcije po vrijednosti, može biti prilično neefikasan. Zbog toga je mnogo praktičnije informaciju o strukturi u funkciju slati preko pokazivača.

Primjer 9.4. Iskoristimo ponovo našu strukturu Student da ilustrujemo način kako struktura može biti argument funkcije. U funkciji `ispis1` struktura se prenosi po vrijednosti, dok je u funkciji `ispis2` argument pokazivač na strukturu.

Elektronska verzija

```
#include <stdio.h>
#include <string.h>
typedef struct{
    char ime[50];
    char prezime[50];
    int brojIndeksa;
}Student;
void ispis1(Student s){
   printf("Ispis1...\n");
   printf("Ime studenta: %s\n",s.ime);
   printf("Prezime studenta: %s\n",s.prezime);
   printf("Broj indeksa: %d\n",s.brojIndeksa);
}
void ispis2(Student *s){
   printf("Ispis2...\n");
   printf("Ime studenta: %s\n",s->ime);//s je sada pokazivac
   printf("Prezime studenta: %s\n",s->prezime);
   printf("Broj indeksa: %d\n",s->brojIndeksa);
}
int main(){
   Student s1 = {"Pero", "Peric", 12345};
   Student s2;
   strcpy(s2.ime,"Bojana");
   strcpy(s2.prezime,s1.prezime);
   s2.brojIndeksa = s1.brojIndeksa + 1;
   ispis1(s1); //ovdje struktura ide po vrijednosti
   ispis2(&s2); //ovdje saljemo pokazivac na strukturu
   return 0;
}
```

9.1.4 Nizovi struktura

U programskom jeziku C dozvoljeno je kreiranje nizova struktura. Nizovi struktura se kreiraju na isti način kao i u slučajevima drugih tipova podataka. Takođe, principi kombinovanja pokazivačkog i nizovnog pristupa važe i kada je riječ o nizovima struktura. Da bismo ilustrovali rad sa nizovima struktura, uradimo sljedeći zadatak.

Primjer 9.5. Sa tastature se unosi broj `n`, a nakon toga i podaci o `n` gradova koji su predstavljeni tačkama u koordinatnom sistemu. Za svaki grad se unosi njegovo ime i njegove `x` i `y` koordinate. Odrediti onaj par gradova koji su na najvećoj udaljenosti.

Elektronska verzija

```c
#include <stdio.h>
#include <math.h>
#include <string.h>
typedef struct {
    char ime[50];
    double x;
    double y;
} Grad;
double udaljenost(Grad *g1, Grad *g2){
   return
       sqrt((g2->y-g1->y)*(g2->y-g1->y)+(g2->x-g1->x)*(g2->x-g1->x));

}
void ispisGrada(Grad * g){
   printf("Ispis grada\n");
   printf("Ime: %s\n",g->ime);
   printf("Koordinate: (%f,%f)\n",g->x,g->y);
}
void unos (Grad * niz, int n){
   int i;
   printf("Unos gradova...\n");
   for (i = 0; i < n; i++){
      printf("Ime %d. grada:",i);
      scanf("%s",&niz[i].ime);
      printf("Koordinate:");
      scanf("%lf %lf",&niz[i].x,&niz[i].y);
   }
}
void kopiraj(Grad *g1,Grad *g2){
   strcpy(g1->ime,g2->ime);
   g1->x = g2->x;
   g1->y = g2->y;
}
int main(){
   int n,i,j;
   double najvece;
   printf("Koliko gradova:");
   scanf("%d",&n);
   Grad niz[n];
   unos(niz,n);
   printf("Uneseni su gradovi...\n");
   for(i = 0; i < n; i++)
      ispisGrada(&niz[i]);
   Grad g1,g2;
```

elektronska verzija

```
44    kopiraj(&g1,&niz[0]);
45
46    kopiraj(&g2,&niz[1]);
47    najvece = udaljenost(&g1,&g2);
48    for(i = 0; i < n-1; i++)
49       for (j = i + 1; j < n; j++)
50          if(udaljenost(&niz[i],&niz[j])>najvece){
51             kopiraj(&g1,&niz[i]);
52             kopiraj(&g2,&niz[j]);
53             najvece = udaljenost(&niz[i],&niz[j]);
54          }
55    ispisGrada(&g1);
56    ispisGrada(&g2);
57    printf("Udaljenost: %f\n",najvece);
58    return 0;
59 }
```

Iako to nije bilo neophodno, u zadatku smo kao argumente funkcija koristili pokazivače na strukturu `Grad`, jer je to uobičajen pristup kada se radi sa strukturama. Funkcija `kopiraj` je u ovom zadatku jednostavna i mogla se čak i izbjeći. Međutim, u slučaju kada se radi o još složenijim strukturama (posebno ako strukture sadrže nizove kao elemente), dodjeljivanje vrijednosti koje sadrži jedna struktura drugoj strukturi često nije tako jednostavno. Stoga je preporuka da se u tim slučajevima koristi funkcija za kopiranje sadržaja jedne strukture u drugu, kao što je to ilustrovano u ovom primjeru.

9.2 Unije

Pored struktura, u programskom jeziku C postoje i unije, koje takođe obezbjeđuju mehanizam za objedinjavanje promjenljivih koje ne moraju biti istog tipa. Međutim, za razliku od struktura, kod kojih svaki element posjeduje svoju memorijsku lokaciju, kod unija svi elementi dijele istu memorijsku lokaciju, dok se u jednom trenutku može koristiti samo jedan od elementa. Na taj način se obezbjeđuje mogućnost da se isti memorijski prostor koristi za različite namjene, što dovodi do uštede memorije. Veličina memorije, koju zauzima unija, jednaka je veličini njenog najvećeg elementa.

Za unije važe skoro sva sintaksna pravila koja važe i za strukture, uz osnovnu razliku da se prilikom definisanja unija, umjesto riječi **`struct`**, koju smo koristili kod struktura, koristi riječ **`union`**.

Primjer 9.6. Ilustrujmo kako rade unije na sljedećem primjeru. Unija sadrži tri člana, jedan cijeli broj, jedan realan i jednu nisku.

Elektronska verzija

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

typedef union {
   int i;
   float f;
   char str[20];
} Podaci;

int main( ) {

   Podaci pod;

    pod.i = 10;
   printf( "pod.i : %d\n", pod.i);
    pod.f = 220.5;
    printf( "pod.f : %f\n", pod.f);
    strcpy( pod.str, "C programiranje");
    printf( "pod.str : %s\n", pod.str);

    printf("Velicina unije: %d\n",sizeof(Podaci));
    return 0;
}
```

Na ekranu dobijamo sljedeći ispis:

```
pod.i : 10
pod.f : 220.500000
pod.str : C programiranje
Velicina unije: 20
```

Odnosno, ako odmah po dodjeljivanju vrijednosti nekom članu, tom članu odmah i pristupimo (kao što smo mi uradili funkcijom za ispis), dobijamo očekivanu vrijednost, kojom možemo da raspolažemo.

Primjer 9.7. Posmatrajmo sad istu uniju, ali sa malo izmijenjenim redoslijedom naredbi dodjele vrijednosti i ispisa.

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

typedef union {
   int i;
   float f;
```

Elektronska verzija

```
7     char str[20];
8  } Podaci;
9
10 int main( ) {
11
12    Podaci pod;
13     //prvo dodijelimo sve tri vrijednosti
14    pod.i = 10;
15    pod.f = 220.5;
16    strcpy( pod.str, "C programiranje");
17     //pa onda ispisemo
18    printf( "pod.i : %d\n", pod.i);
19    printf( "pod.f : %f\n", pod.f);
20    printf( "pod.str : %s\n", pod.str);
21
22    printf("Velicina unije: %d\n",sizeof(Podaci));
23    return 0;
24 }
```

Posmatrajmo ispis na ekranu, koji bi mogao da izgleda kao na sljedećem prikazu. Očigledno, cjelobrojna i realna promjenljiva nisu jednake 10 odnosno 220.5.

```
pod.i : 1919950915
pod.f : 4756185880441909600000000000000.000000
pod.str : C programiranje
Velicina unije: 20
```

U ovom listingu smo redom prvo dodjeljivali vrijednosti članovima strukture, ali smo tek nakon toga funkcijom za ispis pokušali da pristupimo jednom po jednom članu. Zapravo, u redovima kôda 14, 15 i 16 se pisalo po istoj memoriji, jer je za sve članove unije upravo i predviđena ista memorija. Veličina te memorije je jednaka veličini memorije potrebnoj za čuvanje največeg elementa, što je u našem slučaju niz od 20 karaktera.

9.3 Pitanja i zadaci

1. Šta su strukture i za šta se koriste?
2. Da li je moguće porediti dvije strukture pomocu operatora ==?
3. Data je struktura

```
typedef struct{
    float cijena;
```

Elektronska verzija

```
    char naziv[50];
    char barKod[20];
} Artikl;
```

koja sadrži informacije o cijeni artikla, nazivu artikla i bar-kodu artikla. Napisati program koji formira dva podataka tipa Artikl i dodjeljuje im vrijednosti koje se unose sa tastature, a zatim izjednačava cijene ta dva artikla, tako što ih postavlja na manju od njih.

4. Definisati strukturu kojom se predstavlja vrijeme, odnosno strukturu koja sadrži tri cjelobrojna podataka koji redom predstavljaju sate, minute i sekunde. Napisati funkciju koja (a) sabira dva vremena; (b) oduzima od kasnijeg vremena ranije vrijeme; (c) ispisuje informacije o vremenu na ekran.

5. Proučiti standardnu biblioteku `time.h` i strukture koje su tamo definisane. Napisati program koji očitava sistemsko vrijeme i ispisuje ga na ekranu u formatu, koji je prilagođen korisniku programa.

6. Koristeći strukture i funkcije iz standardne biblioteke `time.h` obezbijediti mehanizam kojim se mjeri vrijeme izvršenja programa.

7. Definisati strukturu koja predstavlja Planinu. Struktura sadrži informacije o nazivu planine (podatak tipa `char[]`) i najvišem vrhu planine (podatak tipa `int`). Sa tastature se unosi broj n, a potom podaci o n planina. Napisati program koji određuje najvišu i najnižu planinu, kao i prosječnu visinu unesenih planina.

8. Da li struktura može biti rezultat funkcije? Ilustrovati primjerom.

9. Za šta se, kod struktura, koristi operator tačka ., a za šta operator strelica `->`?

10. Kako se vrši definisanje pokazivača na strukturu? Kada je korisno koristiti pokazivače na strukturu? Ilustrovati primjerom.

11. Uporediti strukture i unije. Primjerom ilustrovati razliku između strukture i unije.

12. Kolika je vrijednost promjenljive `s` nakon izvršenja narednog kôda?

```
typdef union{
    int a;
    int b;
```

Elektronska verzija

```
    int c;
} UredjenaTrojka;

int main(){
    UredjenaTrojka x;
    x.a = 2;
    x.b = 3;
    x.c = 4;

    int s = x.a + x.b + x.c;
    printf("s=%d\n",s);
    return 0;
}
```

Elektronska verzija

10 Datoteke

Do sada se u udžbeniku podrazumijevalo da se podaci čitaju sa standardnog ulaza (tastature), dok su se rezultati izvršenja programa, kao i druge poruke, ispisivali na standardni izlaz (ekran). Pored ovog, uobičajenog, pristupa (čitanje podataka sa tastature i pisanje podataka na ekran), programski jezik C (kao i većina drugih programskih jezika) omogućava i čitanje podataka iz ulaznih datoteka, kao i pisanje podataka u izlazne datoteke. Deklaracije koje su potrebne za rad sa datotekama nalaze u zaglavlju <stdio.h>.

Prema načinu interpretiranja sadržaja datoteke, razlikujemo tekstualne datoteke i binarne datoteke, a na osnovu toga razlikujemo i dva načina pristupa u radu sa njima. Tekstualne datoteke, kako i sam naziv kaže, sadrže tekst, tj. niz "vidljivih" karaktera, sa dodatkom oznake kraja reda i horizontalnog tabulatora. Uobičajeno je da se podaci u tekstualnim datotekama obrađuju red po red, tj. podaci se iz ulaznih tekstualnih datoteka čitaju, odnosno u izlaznu upisuju redom linija po linija. Binarne datoteke su datoteke koje se sastoje od bajtova svih vrijednosti od 0 do 255, a za obradu ovakvih datoteka u programskom jeziku C postoji tzv. "binarni način" rada sa datotekama. Svaki bajt (binarni zapis dužine 8 bitova) se čita i piše onakav kakav jeste, bez ikakve konverzije i interpretiranja. U ovom udžbeniku ćemo se ograničiti samo na rad sa tekstualnim datotekama.

10.1 Otvaranje i zatvaranje datoteka

Bez obzira na to, da li se radi o ulaznoj ili izlaznoj datoteci, najprije je potrebno povezati datoteku i program. Datoteka se povezuje pomoću pokazivača na strukturu tipa FILE, koja je deklarisana u okviru standardne input/output biblioteke <stdio.h>. Ova struktura sadrži informacije o datoteci (fajlu), kao što su informacija da li se datoteka koristi za pisanje ili za čitanje, pozicija bafera, trenutna pozicija karaktera u baferu, informaciju da li se došlo do kraja datoteke, da li je došlo do greške, itd. Povezivanje datoteke i programa se vrši pomoću funkcije fopen, koja ima sljedeću deklaraciju

```
FILE *fopen(const char *ime, const char *nacinRada);
```

Elektronska verzija

Iz deklaracije vidimo da funkcija za argument uzima nisku karaktera, koja predstavlja ime datoteke. Drugi argument, koji smo nazvali `nacinRada` je, takođe, niska karaktera, pomoću koje se određuje uloga i način rada sa posmatranom datotekom. Postoje tri načina rada:

- čitanje podataka, kada se niski `nacinRada` dodjeljuje vrijednost `"r"` (slovo r potiče od engleske riječi read - čitati). U slučaju da se pomoću funkcije `fopen` program pokušava povezati sa nepostojećom datotekom, dobija se greška i funkcija `fopen` vraća vrijednost `NULL`.

- pisanje podataka, kada se niski `nacinRada` dodjeljuje vrijednost `"w"` (slovo w potiče od engleske riječi write - pisati). Ako se otvara datoteka za pisanje koja ne postoji, onda se ona kreira (ako je to moguće). U slučaju da se datoteka koja već postoji otvara za pisanje, njen stari sadržaj se briše (novi sadržaj se upisuje preko starog).

- dopisivanje podataka, kada se niski `nacinRada` dodjeljuje vrijednost `"a"`, koje je prvo slovo engleske riječi append - dodavati, dopisivati. Slično kao i kod prethodnog slučaja, ako se otvara datoteka koja ne postoji, onda se ona kreira (ako je to moguće). Sa druge strane, ako je već postojala datoteka sa tim imenom, onda se njen stari sadržaj ne briše, već se novi sadržaj dodaje na kraj datoteke.

U slučaju da se prilikom povezivanja programa i datoteke javi greška, jer se pokušava pristup datoteci za koju program nema odgovarajuću dozvolu (na primjer zabranjena je izmjena sadržaja datoteke, a program pokušava da pripremi datoteku za pisanje), funkcija `fopen`, takođe, vraća vrijednost `NULL`.

Pored navedenih osnovnih načina rada sa datotekama, mogući su i neki drugi. Načini rada `"r+"`, `"w+"` i `"a+"` definišu da će rad sa datotekom uključivati i čitanje i pisanje (ili dopisivanje). Pri radu sa binarnim datotekama, na osnovni način rada se još dopisuje i slovo `"b"` (na primjer `"rb"`, `"wb"`, `"ab"`), što redom znači binarno čitanje, pisanje i dodavanje.

Na kraju programa, svaku datoteku koja je otvorena treba i zatvoriti funkcijom `fclose`, koja uzima kao argument pokazivač na datoteku. Funkcija `fclose` vraća nulu, ako je datoteka uspješno zatvorena, a u slučaju da zatvaranje nije uspjelo, vraća `EOF`. Prilikom poziva funkcije `fclose` prazne se baferi koji privremeno čuvaju sadržaj datoteka, što za posljedicu ima da se sav taj sadržaj fizički upisuje u odgovarajući memorijski prostor na disku. Pored toga, kreirana struktura tipa `FILE`, pomoću koje se povezivala datoteka i program se briše iz memorije.

Elektronska verzija

Primjer 10.1. U primjeru prikažimo najjednostavniji kôd kojim se program povezuje sa ulaznom i izlaznom datotekom. S obzirom na to da je povezivanje datoteka sa programom "rizična" operacija, jer je moguća pojava grešaka na koju programer često i ne može da utiče, uobičajeno je da se odmah nakon pokušaja povezivanja vrši provjera da li je povezivanje uspjelo ili ne. Neke od najčešćih situacija koje uzrokuju pojavu grešaka su nepostojanje datoteke sa datim imenom iz koje trebaju da se čitaju podaci, zabranjen pristup datoteci, izlaznoj datoteci je pridružen atribut *read–only* i sl.

```
#include <stdio.h>
#include <stdlib.h>//ukljucujemo i ovo zaglavlje zbog funkcije exit
int main(){
   FILE* f = fopen("ulaz.txt","r");
   FILE* g = fopen("izlaz.txt","w");
    if(f == NULL)
   {
        printf("Ulazna datoteka ne moze da se otvori.\n");
        exit(1);
   }
   if(g == NULL)
   {
        printf("Nije moguce pisanje u izlaznu datoteku.\n");
        exit(1);
   }

    ...
    //citamo podatke iz datoteke ulaz
    ...
    //pisemo podatke u datoteku izlaz
   ...
   fclose(f);
   fclose(g);
   return 0;
}
```

Prilikom pokretanja svakog programa u programskom jeziku C, otvaraju se tri toka podataka koji se nazivaju standardni ulaz, stadnardni izlaz i standardni izlaz za greške, za koje se koriste redom tri pokazivača: `stdin`, `stdout` i `stderr`, koji su konstantni pokazivači na `FILE` strukturu, definisani u datoteci `<stdio.h>`.

Elektronska verzija

10.2 Funkcije za čitanje i pisanje u i iz datoteke

Funkcije za čitanje i pisanje koje smo ranije koristili prilikom rada sa podacima koji se čitaju sa standardnog ulaza (tastature) i pišu na standardni izlaz (ekran), imaju svoje analogne verzije, kada se radi o čitanju iz ulaznih i pisanju u izlazne datoteke. Funkcijama za čitanje `getchar`, `gets` i `scanf` odgovaraju funkcije `getc`, `fgets` i `fscanf`, dok funkcijama za pisanje `putchar`, `puts` i `printf` odgovaraju `putc`, `fputs` i `fprintf`. Prvi argument funkcija, pomoću kojih se čita ili piše u izlaznu datoteku, je pokazivač na tu datoteku, tj. pokazivač tipa `FILE`, koji je prethodno povezan sa datotekom.

10.2.1 Čitanje i pisanje pojedinačnih karaktera

Za čitanje narednog karaktera iz ulazne datoteke koristimo funkcije `getc` i `fgetc`, koje ima sljedeću deklaraciju:

```
int getc(FILE *fp);
int fgetc(FILE *fp);
```

gdje je `fp` pokazivač na datoteku. I jedna i druga funkcija kao rezultat vraćaju brojevnu vrijednost karaktera, odnosno `EOF`, ako je došlo do kraja datoteke ili do greške.

Razlika između ove dvije funkcije je u tome što `getc` može biti implementirana kao makro naredba , dok `fgetc` ne može. Treba pomenuti i da je funkcija `getchar()`, koju smo ranije koristili za čitanje pojedinačnih karaktera sa standardnog ulaza upravo i implementirana kao `getc(stdin)`.

Primjer 10.2. Navedimo program koji pomoću funkcije `fgetc` čita karakter po karakter iz ulazne datoteke, broji ukupan broj karaketera i svaki karakter ispisuje na ekranu.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char *argv[])
4  {
5  char ch;
6  int brojac=0;
7     FILE* f = fopen("ulaz.txt","r");
8     if(f == NULL)
9     {
10         printf("Ulazna datoteka ne moze da se otvori.\n");
11         exit(1);
```

Elektronska verzija

```
12    }
13     ch=fgetc(f); //procitamo prvi znak da mozemo da udjemo u petlju
14     while(ch!=EOF) //izlazimo kad dodjemo do kraja datoteke
15     {
16       printf("Procitan je znak: %c\n",ch);
17       brojac++;
18       ch = fgetc(f);//citamo dalje
19     }
20
21     printf("Broj karaktera = %d\n",brojac);
22
23     fclose(f);
24
25    return 0;
26 }
```

Pisanje jednog po jednog karaktera u izlaznu datoteku se vrši pomoću funkcija `putc` i `fputc`, čija je sintaksa deklaracije:

```
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
```

Slično kao i kod prethodno pominjanih funkcija za čitanje, razlika između funkcija `putc` i `fputc` je u tome što se `putc` može implementirati kao makro naredba, dok to za funkciju `fputc` ne važi. Obje funkcije kao rezultat vraćaju znak koji je ispisan, ili `EOF`, ako je došlo do greške prilikom ispisa. Pomenimo i da je funkcija `putchar(c)`, pomoću koje se ispisuje pojedinačni karakter na standardni izlaz, zapravo, implementirana kao `putc(c,stdout)`.

Primjer 10.3. Uradimo jednostavan primjer, koji sada koristi i funkciju za čitanje iz ulazne datoteke i funkciju za pisanje u ulaznu datoteku. Iz ulazne datoteke čitamo karakter po karakter, mala slova upisujemo u datoteku “mala.txt”, velika slova u datoteku “velika.txt”, a ostale karaktere u datoteku “ostali.txt”.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char *argv[])
4  {
5  int ch;
6    FILE* f = fopen("ulaz.txt","r");
7    FILE* g1 = fopen("mala.txt","w");
8    FILE* g2 = fopen("velika.txt","w");
9    FILE* g3 = fopen("ostali.txt","w");
10   if(f == NULL)
```

Elektronska verzija

```
11      {
12        printf("Ulazna datoteka ne moze da se otvori.\n");
13        exit(1);
14      }
15      if(g1 == NULL || g2 == NULL || g3 == NULL)
16      {
17        printf("Nije moguce pisanje u neku od izlaznih datoteka.\n");
18        exit(1);
19      }
20
21       ch = fgetc(f); //procitamo prvi znak da mozemo da udjemo u petlju
22       while(ch != EOF) //izlazimo kad dodjemo do kraja datoteke
23       {
24         if(ch >= 97 && ch <= 122) //ako je malo
25             putc(ch,g1);
26         else
27           if(ch >=65 && ch <=90) //ako je veliko
28            putc(ch,g2);
29           else //ostali
30            putc(ch,g3);
31         ch = fgetc(f); //citamo dalje
32   }
33      fclose(f);
34      fclose(g1);
35      fclose(g2);
36      fclose(g3);
37
38      return 0;
39   }
```

10.2.2 Čitanje i pisanje linije teksta

Funkcija koja čita podatke iz ulazne datoteke liniju po liniju ima sljedeću deklaraciju:

```
char *fgets(char *buffer, int n, FILE *fp);
```

Prvi argument funkcije je pokazivač na memorijski prostor (engl. buffer), gdje će linija koja se čita biti privremeno smještena, dok drugi argument (argument `n`) definiše veličinu memorije, na koju prvi argument pokazuje. Ukupan broj karaktera koji može da bude pročitan je `n-1`. Ukoliko je ulazna linija duža od toga, ostatak će biti pročitan prilikom sljedećeg poziva funkcije `fgets`.

Ako je čitanje niske uspjelo, funkcija kao rezultat vraća pokazivač na prvi

Elektronska verzija

karakter niske, dok se u slučaju kraja datoteke ili greške vraća `NULL`. Niska, koja je rezultat čitanja, se završava karakterom `'\0'`. Treći argument funkcije `fp` je pokazivač na datoteku iz koje se čitaju podaci.

Primjer 10.4. Prikažimo jednostavan program koji čita linije teksta iz ulazne datoteke i ispisuje ih na ekranu.

```c
#include <stdio.h>
#include <stdlib.h>
int main () {
   FILE *f;
   char str[60];
   f = fopen("ulaz.txt","r");
   if(f == NULL)
   {
      printf("Ulazna datoteka ne moze da se otvori.\n");
      exit(1);
   }
   while( fgets (str, 60, f)!= NULL ){
      printf("%s",str);
   }
   fclose(f);

   return(0);
}
```

Ovdje treba ponovo spomenuti funkciju `gets`, koja služi za čitanje podataka sa standardnog ulaza. Funkcija `char *gets(char *buf)` ne uzima u obzir veličinu bafera, te se stoga može desiti da linija koja se čita bude veća od rezervisane memorije. Zbog toga je i za čitanje linije teksta sa standardnog ulaza umjesto funkcije `gets` bolje koristiti funkciju `fgets`, gdje je posljednji argument pokazivač na standardni ulaz: `fgetf(buffer,n,stdin)`. Dodatno, treba voditi računa da funkcija `fgets` učitava i oznaku za kraj reda (`'\n'`), dok funkcija `gets` oznaku za novi red ne učitava, a na njegovo mjesto stavlja oznaku za kraj niske `'\0'`.

Pisanje linija teksta u izlaznu datoteku se vrši funkcijom `fputs`, koja ima sintaksu

```c
int fputs(const char *str, FILE *fp);
```

Kao što se može i pretpostaviti iz deklaracije, ova funkcija ispisuje znakovni niz na koji pokazuje `str` u datoteku, na koju pokazuje `fp`. U slučaju uspješnog ispisa, funkcija vraća nenegativnu vrijednost, a ako ispis nije uspio, vraća vrijed-

Elektronska verzija

nost EOF. U izlaznu datoteku se ne upisuje ni terminalni karakter koji označava kraj niske, ni znak za prelazak u novi red.

Treba pomenuti i da funkcije `int puts(const char *str)` i `fputs(str,stdout)` u principu rade slične stvari (obje ispisuju nisku `str` na standardni izlaz), sa tom razlikom da prva funkcija (`puts`) dodaje znak za prelazak u novi red, a druga (`fputs`) ne.

Primjer 10.5. Ako bismo, umjesto na ekran, kao što smo to radili u prethodnom primjeru, sada linije teksta iz ulazne datoteke htjeli da ispišemo u izlaznu datoteku, onda bi `while` petlja izgledala ovako

```
..
  while( fgets (str, 60, f)!= NULL ) {
     fputs(str,g);
  }
..
```

uz pretpostavku da je promjenljiva `g` ispravno povezana sa izlaznom datotekom.

10.2.3 Prepoznavanje grešaka prilikom rada sa datotekama

Već smo pomenuli da funkcije za čitanje i pisanje u datoteke u slučaju pojave greške vraćaju vrijednost EOF. Stoga samo na osnovu takve vraćene vrijednosti nije moguće zaključiti da li je tokom rada zaista došlo do greške, ili se stiglo do kraja datoteke. Stoga je nekada pogodno koristiti i funkcije

```
int ferror(FILE *fp);
int feof(FILE *fp);
```

koje detektuju i razlikuju upravo ove dvije situacije. Funkcija `ferror` vraća broj različit od nule (možemo smatrati kao da funkcija vraća logičku vrijednost tačno) u slučaju da je došlo do greške, a inače vraća nulu (logička vrijednost netačno). Funkcija `feof` vraća broj različit od nule (vrijednost tačno) ako je došlo do kraja datoteke, a inače vraća nula (netačno). Ovom funkcijom, na praktičan način, možemo da obezbijedimo čitanje podataka iz datoteke, sve dok se ne dođe do kraja datoteke. Na primjer, ako je promjenljiva `f` vezana za ulaznu promjenljivu, tada pomoću `while` petlje možemo lako da kontrolišemo čitanje, na sljedeći način:

```
...
while(!feof(f))//izlazimo kad dodjemo do kraja datoteke
{
```

Elektronska verzija

```
    ch = fgetc(f);//citamo karakter
    ...
}
...
```

10.2.4 Funkcije formatiranog ulaza i izlaza

Slično kao i kod čitanja i pisanja sa standardnog ulaza, kod čitanja iz datoteka i pisanja u datoteke postoje funkcije za formatirani ulaz i izlaz. Sintaksa deklaracija ovih funkcija su:

```
int fscanf(FILE *fp, const char *format,...);
int fprintf(FILE *fp, const char *format,...);
```

Princip rada ove dvije funkcije je identičan principu rada odgovarajućih funkcija standardnog ulaza `scanf` i `printf`, s tim što funkcije `fscanf` i `fprintf` imaju dodatni argument (koji je prvi po redu) i odnosi se na dateteku iz koje se čita, odnosno u koju se piše. Funkcija `fscanf` vraća broj pročitanih karaktera, ako je čitanje uspjelo, a ako čitanje nije uspjelo, vraća vrijednost `EOF`. Funkcija `fprintf` vraća broj upisanih karaktera, ako je pisanje uspjelo, ili negativan broj (logičku vrijednost netačno), ako je došlo do greške. Treba još pomenuti i da je funkcija `scanf(...)` ekvivalentna funkciji

`fscanf(stdin,...)`, dok je funkcija `printf(...)` ekvivalentna funkciji `fprintf(stdout,...)`.

Primjer 10.6. Ilustrujmo primjerom upotrebu funkcija za formatirani ulaz i izlaz. U datoteci je u svakom redu upisana po jedna riječ i jedan broj (koji su odvojeni razmakom). U izlaznu datoteku upisati u odgovarajući red prvo broj, pa zatim i riječ.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char *argv[])
4  {
5     FILE* f = fopen("podaci.txt","r");
6     FILE* g = fopen("izlaz.txt","w");
7     char rijec[50];
8     int broj;
9     if(f == NULL)
10    {
11       printf("Ulazna datoteka ne moze da se otvori.\n");
12       exit(1);
```

Elektronska verzija

```
13    }
14    if(g == NULL)
15    {
16       printf("Nije moguce pisanje u izlaznu datoteku.\n");
17       exit(1);
18    }
19
20    while(!feof(f))//izlazimo kad dodjemo do kraja datoteke
21    {
22       fscanf(f,"%s",rijec); //procitamo rijec
23       fscanf(f,"%d",&broj); //procitamo broj
24       fprintf(g,"%d %s\n",broj,rijec); // u izlaznu datoteku upisemo
            prvo broj, pa rijec
25    }
26    fclose(f);
27    fclose(g);
28
29    return 0;
30 }
```

Elektronska verzija

10.3 Pitanja i zadaci

1. Koje vrste ulaznih i izlaznih datoteka, prema načinu interpretiranja podataka, postoje u programskom jeziku C?
2. Navesti i detaljno objasniti koji načini rada sa datotekama postoje.
3. U čemu je razlika između pisanja i dopisivanja u datoteku, koja već ima neki sadržaj?
4. Kakva je razlika između načina rada `r+` i `w+`?
5. Napisati program koji sa ekrana čita nisku i upisuje jedan po jedan karakter niske u tekstualnu datoteku `izlaz.txt`.
6. Uporediti funkcije:
   a) `gets` i `fgets`;
   b) `puts` i `fputs`.
7. U slučaju neuspjelog povezivanja sa datotekom, pozivom funkcije `exit(1)` smo prekidali izvršenje programa. U pouzdanoj literaturi pronaći objašnjenja dva standardna statusa `EXIT_SUCCESS` i `EXIT_FAILURE`. Ima li razlike između naredbe `exit(1)` i `exit(EXIT_FAILURE)`?

8. Napisati program koji čita jednu po jednu liniju iz tekstualne datoteke i određuje redni broj linije koja ima najviše velikih slova.

9. Napisati program koji čita tekstualnu datoteku, koja u svakom redu sadrži nekoliko cijelih brojeva međusobno odvojenih jednim razmakom. Program treba da u svaki red izlazne datoteke upiše sumu brojeva koji se nalaze u istom redu ulazne datoteke.

10. Napisati program koji poredi dvije tekstualne datoteke i kao rezultat vraća broj linija koje su iste u obje datoteke.

11. Navesti najčešće greške koje se mogu desiti pri radu sa datotekama. Objasni kada se one javljaju i zašto.

12. Napisati program koji čita dvije tekstualne datoteke i popunjava izlaznu tekstualnu datoteku tako što prvo prepiše prvu liniju prve datoteke, zatim prvu liniju druge datoteke, pa drugu liniju prve datoteke, zatim drugu liniju druge datoteke, itd. Ako jedna datoteka ima manje linija od druge, onda se, nakon što je završeno naizmjenično prepisivanje, ostatak sadržaja datoteke koja ima više linija dopiše u izlaznu datoteku.

Elektronska verzija

Elektronska verzija

11 Algoritmi za sortiranje nizova

Sortiranje podataka je važna operacija u mnogim procesima, te je, stoga, poznavanje algoritama za sortiranje i principa njihovog rada vrlo korisno za početnike. Mnogi zadaci koji podrazumijevaju rad sa složenim tipovima podataka se mogu lakše, brže i efikasnije riješiti kada se radi sa uređenim (sortiranim) strukturama. Na primjer, pretraživanje niza, izbacivanje duplikata ili upoređivanje sadržaja dvije strukture su neki tipični primjeri zadataka, koji se lakše i brže rješavaju ukoliko se te operacije izvršavaju na sortiranim strukturama.

Iako skoro svi prevodioci za većinu savremenih programskih jezika imaju svoje "ugrađene" algoritme za sortiranje, koji su pouzdani i efikasni, prilikom savladavanja osnovnih tehnika programiranja korisno je shvatiti i savladati principe rada nekoliko različitih i često korištenih algoritama za sortiranje. Kroz analizu tih algoritama, početnik lakše može razumjeti i usvojiti pojmove koji se odnose na efikasnost algoritma, neke važne principe programiranja (npr. rekurziju ili princip "podijeli pa vladaj"), šta je to najbolji, najgori i prosječan slučaj koji se može javiti u realnim situacijama, itd. Stoga se u većini uvodnih kurseva programiranja izučava nekoliko najpoznatijih algoritama za sortiranje, koji se između sebe razlikuju po principu rada, kompleksnosti, količini potrebne memorije za implementaciju, stabilnosti, prilagodljivosti, itd. Od svih algoritama za sortiranje, kojih u raznim varijantama ima nekoliko desetina, mi ćemo u ovom udžbeniku objasniti nekoliko najpoznatijih: sortiranje izborom (engl. selection sort), sortiranje umetanjem (engl. insertion sort), sortiranje pomoću "mjehurića" (engl. bubble sort), brzo sortiranje (engl. quick sort) i sortiranje spajanjem (engl. merge sort).

11.1 Osnovni pojmovi o sortiranju

Sortirati niz znači poredati njegove članove u nekom redoslijedu (poretku). Nizovi se mogu sortirati po različitim kriterijumima. Kada je riječ o brojevnim nizovima, sortiranje je najčešće zasnovano na standardnoj relaciji poretka - relaciji "manje ili jednako". Za standardno poređenje riječi koristimo tzv. leksikografski poredak, zasnovan na definisanom poretku slova u nekom alfabetu (na primjer, riječi u rječnicima su poredane u leksikografskom poretku). Kada je riječ o složenijim strukturama, kriterijum za sortiranje može biti i nešto

Elektronska verzija

drugo. Na primjer, strukturu koja sadrži niz podataka o studentima možemo sortirati po broju indeksa studenta, imenu, prezimenu, godini studija, itd. Zbog toga se kod sortiranja složenijih podataka uvodi termin "ključ" za sortiranje, pa kažemo da se podaci sortiraju prema "ključu". Takođe, podaci se mogu sortirati i po više kriterijuma. Na primjer, niz studenata možemo prvo sortirati po prezimenima, pa potom po imenima.

Nizove sortiramo u rastućem ili opadajućem poretku u odnosu na ključ koji je osnov za sortiranje. Da bismo bili malo precizniji, moramo uzeti u obzir da nizovi mogu sadržavati i neke elemente koji imaju istu vrijednost ključa. Zbog toga se za precizniju definiciju "sortiranog niza", umjesto termina rastući i opadajući poredak, koriste izrazi neopadajući i nerastući, koji uključuju i situacije kada ključevi nekih elemenata imaju istu vrijednost.

Radi lakšeg razumijevanja samog principa rada svakog od algoritama, u nastavku ćemo se ograničiti na sortiranje cjelobrojnih nizova u neopadajućem poretku, gdje je kriterijum za sortiranje zasnovan na standardnoj relaciji poretka. Kasnije, kroz primjere i zadatke, imaćemo priliku da sortiramo i druge podatke, po raznim kriterijumima.

Ako posmatramo niz od n brojeva $a_0, a_1, ..., a_{n-1}$ i ako je kriterijum za sortiranje standardni operator poređenja (operator $\leq$), tada za takav niz kažemo da je sortiran u neopadajućem poretku ako važi

$$a_0 \leq a_1 \leq ... \leq a_{n-1}.$$

Slično, za niz od n brojeva $a_0, a_1, ..., a_{n-1}$ kažemo da je sortiran u nerastućem poretku po standardnom kriterijumu za sortiranje, ako važi

$$a_0 \geq a_1 \geq ... \geq a_{n-1}.$$

Sa aspekta mjesta gdje se čuvaju podaci za vrijeme sortiranja, algoritme za sortiranje možemo podijeliti u dvije klase: one koji vrše tzv. *unutrašnje sortiranje* (engl. internal sorting) i one koji vrše *spoljašnje sortiranje* (engl. external sorting). Kod unutrašnjeg sortiranja kompletni podaci koji se sortiraju se nalaze u samoj radnoj memoriji tokom čitavog procesa sortiranja. Suprotno tome, kod spoljašnjeg sortiranja nisu uvijek svi podaci prisutni u memoriji, već se podaci po potrebi smještaju u radnu memoriju u dijelovima, dok se ostatak podataka nalazi na nekom drugim uređaju (na primjer, na disku). Spoljašnje sortiranje se koristi za sortiranje ogromnih količina podataka, koji se ne mogu smjestiti u glavnu memoriju. Od algoritama za sortiranje, koje ćemo ovdje spomenuti, većina ih radi po principu unutrašnjeg sortiranja, dok se jedino sortiranje spajanjem (engl. mergesort), uz dodatne modifikacije može smatrati algoritmom koji može da obavlja i spoljašnje sortiranje.

Elektronska verzija

Uz algoritme za sortiranje koje razmatramo u ovom poglavlju, pomenimo još jedan važan koncept, koji se odnosi na algoritme koji se izvršavaju "u mjestu" (engl. in place). Izvršavanje algoritma u mjestu podrazumijeva rad sa ulaznim podatkom (koji može da uključuje i transformaciju ulaznog podatka, što je kod sortiranja upravo i slučaj), tako da se ne uvode nove strukture podataka i ne koristi dodatni memorijski prostor. Doduše, dozvoljena je upotreba male količine dodatnog prostora, uglavnom za smještanje pomoćnih promjenljivih. Algoritmi koji rade "u mjestu" obično koriste princip da ulazni podaci (na primjer elementi niza) razmjenjuju mjesta (engl. swapping). Opet ćemo pomenuti da svi algoritmi za sortiranje, koje ćemo ovdje pomenuti, rade "u mjestu", osim sortiranja spajanjem, za koji nam je potreban još jedan dodatni niz, jednake dužine kao i niz koji se sortira. Treba pomenuti da postoje i implementacije sortiranja spajanjem koje rade "u mjestu", ali su one zbog prilično složene implementacije ovdje izostavljene.

Pomenimo i pojam stabilnosti algoritma za sortiranje. Za algoritam za sortiranje kažemo da je *stabilan*, ako je nakon sortiranja očuvan početni redoslijed elemenata koji imaju isti ključ. O stabilnosti algoritma za sortiranje posebno ima smisla govoriti, ako se radi o sortiranju složenih struktura, gdje se sortiranje vrši po više ključeva.

Primjer 11.1. Pretpostavimo da treba sortirati podatke o studentima koji su podijeljeni u dvije grupe (recimo A i B), ali tako da je lista sortirana najprije po imenima (po abecednom redoslijedu), pa zatim i po grupama. Neka je početna lista, u zapisu `(ime ,grupa)` data sa:

```
(Dejana, A)
(Ana, B)
(Kosta, A)
(Goran, B)
(Brankica, A)
```

Ako sortiramo listu najprije po imenima, dobićemo

```
(Ana, B)
(Brankica, A)
(Dejana, A)
(Goran, B)
(Kosta, A)
```

I ako bismo sada primijenili algoritam za sortiranje, gdje nam je ključ za sorti-

Elektronska verzija

ranje ime grupe i koji nije stabilan, mogli bismo dobiti, na primjer

```
(Brankica, A)
(Dejana, A)
(Kosta, A)
(Goran, B)
(Ana, B)
```

odnosno situaciju da je narušen redoslijed imena (u našem primjeru je sada Goran ispred Ane, što ne bi trebalo da bude). Sa druge strane, ako bismo na listu koja je sortirana po imenima primijenili stabilan algoritam za sortiranje, dobili bismo listu

```
(Brankica, A)
(Dejana, A)
(Kosta, A)
(Ana, B)
(Goran, B)
```

što se u startu i tražilo.

Od algoritama za sortiranje koje ćemo ovdje razmatrati većina ih je stabilna po samoj svojoj prirodi, uz izuzetak brzog sorta, gdje stabilnost zavisi od same implementacije.

11.1.1 O analizi vremenske složenosti algoritma

Analizom vremenske složenosti algoritma procjenjujemo količinu vremena, koje je potrebno da se algoritam izvrši. Potrebno vrijeme za izvršenje algoritma zavisi od *dimenzije problema*, odnosno količine ulaznih podataka koje je potrebno obraditi unutar algoritma, ali i od same konstrukcije algoritma. Vremenska složenost se obično procjenjuje određivanjem broja elementarnih operacija koje je potrebno izvršiti unutar algoritma, uz pretpostavku da svaka elementarna operacija za svoje izvršavanje zahtijeva fiksnu količinu vremena.

Pošto vrijeme izvršavanja algoritma može da zavisi i od prirode samih podataka, pri analizi vremenske kompleksnosti se obično razmatra "najgori slučaj" (engl. worst case), koji, zapravo, predstavlja maksimalnu količinu vremena koja je potrebna da algoritam završi obradu podataka date dimenzije. Pored najgoreg slučaja, analizira se i tzv. prosječan slučaj (engl. average case), kojim određujemo prosječno vrijeme izvršavanja algoritma u zavisnosti od zadate dimenzije problema. U oba slučaja, vremenska složenost se izražava funkcijom koja zavisi od veličine ulaza. U opštem slučaju je teško potpuno precizno odrediti kako izgleda ta funkcija, pa se obično fokusiramo na računanje asimptotske

Elektronska verzija

složenosti algoritma. Tako se vremenska složenost najčešće izražava u tzv. "veliko O" (engl. big O) notaciji. Detaljna razrada koncepata koji se odnose na analizu vremenske složenosti algoritama (uključujući i detaljno pojašnjenje "veliko O" notacije) izlazi iz okvira ovog udžbenika. Pojednostavljeno gledano, za određivanje vremenske složenosti algoritma potrebno je odrediti funkciju kojom procjenjujemo maksimalan broj elementarnih operacija koje je potrebno izvršiti i za zapisivanje te funkcije koristimo "veliko O" notaciju. Na primjer, ako broj elementarnih operacija algoritma linearno zavisi od broja ulaznih podataka, kojih ima n, tada kažemo da je vremenska složenost algoritma $O(n)$, ako broj elementarnih operacija algoritma zavisi kvadratno, onda je vremenska složenost $O(n^2)$. Jasno, što funkcija kojom opisujemo vremensku složenost brže raste, algoritam je sporiji. Tako su, na primjer, algoritmi sa vremenskom složenošću $O(n!)$ ili $O(2^n)$ izuzetno spori i praktično upotrebljivi samo za rješavanje problema malih dimenzija. Sa druge strane, algoritmi koji imaju logaritamsku složenost su brzi, čak i za ogromne dimenzije ulaznih podataka.

Kao što smo već pomenuli, sortiranje podataka je veoma česta operacija, koja se često primjenjuje i na podatke jako velikih dimenzija (reda veličine nekoliko miliona, pa i više). Stoga je i analiza vremenske složenosti algoritama za sortiranje izuzetno važna. U okviru udžbenika nećemo detaljno ulaziti u analizu vremenske složenosti navedenih algoritama, ali ćemo ukratko pomenuti da prva tri algoritma (sortiranje izborom, umetanjem i sortiranje mjehurićima) spadaju u grupu "sporijih" algoritama za sortiranje i imaju vremensku zavisnost $O(n^2)$, dok brzi sort i sortiranje spajanjem spadaju u "brže" algoritme. U prosječnom slučaju, njihova vremenska zavisnost je $O(n \log n)$. U najgorem slučaju vremenska zavisnost brzog sorta je $O(n^2)$, dok vremenska zavisnost sortrianja spajanja ostaje $O(n \log n)$, čak i u najgorem slučaju.

11.2 Sortiranje izborom

Sortiranje izborom je jedan od algoritama za sortiranje, koji su relativno jednostavni za razumijevanje i implementaciju. Osnovna ideja ovog algoritma je da se u svakom koraku među neuređenim elementima pronađe najmanji od njih i da ga postavi na odgovarajuću poziciju. Detaljnije, u prvom koraku algoritma prolazimo kroz čitav niz, pronalazimo najmanji element i postavljamo ga na početnu poziciju. U drugom koraku, među preostalim elementima (elementima iz neuređenog dijela niza) opet pronalazimo najmanji i postavljamo ga na drugu poziciju. Ovaj postupak ponavljamo, tj. u svakom narednom koraku pronalazimo najmanji element iz neuređenog dijela i postavljamo ga na odgovarajuću poziciju, sve dok ne "potrošimo" elemente iz neuređenog dijela niza.

Elektronska verzija

Najjednostavniji algoritam koji niz sortira ovim principom:

```
void selectionsort(int niz[], int n){
   int i,j;
   int temp;
   for(i = 0; i < n - 1; i++)
      for(j = i + 1; j < n; j++)
       if(niz[i] > niz[j]){
          temp = niz[i];
          niz[i] = niz[j];
          niz[j] = temp;
       }
}
```

Pažljivijom analizom ovog algoritma možemo primijetiti da je moguće da se u toku sortiranja javi i veći broj "nepotrebne" izmjene redoslijeda elemenata. Na primjer, ako posmatramo niz

`46 40 19 10 15 29 4  63`

već u prvom koraku (za vrijednosti `i=0` i `j=1`, doći će do zamjene brojeva 40 i 46:

`40 46 19 10 15 29 4  63`

ali već za narednu vrijednost `j=2` imamo da je element na indeksu 2 (to je broj 19) manji od početnog, te opet dolazi do zamjene

`19 46 40 10 15 29 4  63`

Opet, za `j=3` element na indeksu 3 (to je broj 10) je ponovo manji od početnog, pa će se zamjena opet izvršiti. Međutim, ni broj 10 neće ostati na prvom mjestu, jer je element niza na indeksu `j=6` (to je broj 4) manji od 10, pa će opet doći do zamjene. Na kraju prvog prolaska kroz niz (za vrijednost `i=0`) ćemo dobiti:

`4  46 40 19 15 29 10 63`

Vidimo da je prije postavljanja najmanjeg elementa na početnu poziciju u međuvremenu došlo do nekoliko nepotrebnih zamjena. Slična situacija bi se ponovo javila i u nastavku procesa sortiranja ovog niza.

Da bi se izbjeglo nepotrebno premještanje elemenata, umjesto direktnih zamjena `i`-tog i `j`-tog elementa, može se koristiti princip da se samo pamti indeks trenutno najmanjeg elementa u ostatku niza, tj. indeks elementa koji je kandidat za premještanje na odgovarajuću poziciju. Zamjena se vrši tek na kraju, kada se provjere svi elementi do kraja. Modifikovan algoritam izgleda ovako

```
void selectionsort2(int niz[], int n)
{
   int i, j, min_idx;
```

Elektronska verzija

```
    int temp;
    for (i = 0; i < n-1; i++)
    {
       min_idx = i;//u startu pretpostavimo da je trenutni ujedno i
          najmanji
       for (j = i + 1; j < n; j++)
         if (niz[j] < niz[min_idx])
           min_idx = j; //ako pronadjemo da je j-ti manji od trenutnog
              najmanjeg, samo zapamtimo gdje smo ga nasli

       //na kraju zamijenimo i-ti i najmanji
       temp = niz[i];
       niz[i] = niz[min_idx];
       niz[min_idx] = temp;
    }
}
```

11.3 Sortiranje umetanjem

Kod ovog sortiranja niz posmatramo na način da se on sastoji iz dva dijela, od kojih je prvi dio sortiran, a drugi ne. U svakom koraku algoritma se uzima po jedan element iz nesortiranog dijela (najčešće je to prvi element po redu iz nesortiranog dijela), te se on "umeće" u prvi (sortirani) dio, na ono mjesto tako da sortirani dio i dalje ostane sortiran. Kao posljedicu ovakvog pristupa imamo da se u svakom koraku broj sortiranih elemenata (prvi dio niza) povećava za jedan, a broj nesortiranih (drugi dio niza) smanjuje za jedan. Proces se završva kada se "potroše" svi elementi iz nesortiranog dijela.

Da bismo objasnili kako ćemo ovaj princip implementirati, pretpostavimo da je u `i`-tom koraku, gdje je `i>0`, prvih `i` elemenata niza sortirano (to su elementi na indeksima `0,1,...,i-1`. Tada pokušavamo da element niza na poziciji `i` ubacimo u sortirani dio, tako da on ostane sortiran. Praktično, `i`-ti element pokušavamo da umetnemo između neka dva elementa, ili da ga eventualno postavimo skroz na početak, ukoliko je baš manji od svih elemenata u lijevom sortiranom dijelu. Kada to uspijemo, sortirani dio će sadržavati ukupno `i+1` sortiranih elemenata.

Algoritam koji sortira niz po ovom principu bi izgledao ovako:

```
void insertsort(int niz[], int n)
{
    int i, j;
    int temp;
```

Elektronska verzija

```
    for(i = 1; i < n; i++){
        temp = niz[i]; //sacuvamo i-ti element u temp jer cemo i-tu
            poziciju koristiti za pomjeranje elemenata
        for(j = i - 1; j >= 0; j--)
            if(niz[j] > temp)
                niz[j + 1] = niz[j]; //temp ide dalje ulijevo, a vece
                    elemente pomjeramo za jedno mjesto udesno
            else break; //niz[j] nije vece od temp, pa ne trebamo dalje
                da se pomjeramo
        niz[j + 1] = temp; //postavljamo temp (sto je prvobitno bilo
            niz[i] na odgovarajucu poziciju
}
}
```

11.4 Sortiranje pomoću mjehurića

Ovaj sort je dobio ime po engleskoj riječi *bubble*, što znači mjehurić. Da bismo lakše objasnili princip rada ovog sorta, posmatrajmo da su elementi niza postavljeni vertikalno, jedan na drugi. Kod mjehurića važi pravilo (koje se sada i ovdje primjenjuje) da lakši mjehurići teže da isplivaju na površinu, dok oni teži tonu nadole. Kada ovaj princip primijenimo na brojeve, možemo smatrati da se manji brojevi (predstavljeni "lakšim" mjehurićima) "penju" ka gore, dok veći brojevi (teži mjehurići) tonu u dubinu. Dalje, mjehurić može da se pomjera nagore, ako se nađe ispod mjehurića koji je "teži" od njega, tj., u tom slučaju, ta dva mjehurića mijenjaju mjesta. U svakoj iteraciji algoritma posmatraju se parovi elemenata niza (parovi mjehurića), počevši odozdo (sa kraja niza). Donji mjehurić se poredi sa onim iznad njega, i u slučaju da je lakši, njih dvojica mijenjaju mjesta. Ovaj proces uzastopnog poređenja parova elemenata se nastavlja, sve dok se ne dođe do već sortiranog dijela niza. Kao posljedicu ovog pristupa imamo to da se u svakoj iteraciji najlakši mjehurić među preostalima postavlja na odgovarajuću poziciju, te se broj nesortiranih elemenata u svakom koraku smanjuje za jedan.

Implementacija *bubblesort*-a je relativno jednostavna. Spoljašnja petlja kontroliše da se proces podizanja mjehurića odozdo ka gore ponovi potreban broj puta, dok unutrašnja petlja koristi za poređenje i eventualne izmjene elemenata.

```
void bubblesort(int niz[], int n)
{
    int i, j;
    int temp;
    for(i = 1; i < n; i++){
```

Elektronska verzija

```
        for(j = n - 1; j >= i; j--)
            if(niz[j] < niz[j - 1])//ako je donji laksi od gornjeg
            {
                temp = niz[j];
                niz[j] = niz[j-1];
                niz[j-1] = temp;

            }
}
}
```

11.5 Brzi sort

Brzi sort (engl. quick sort), kako i sama riječ kaže je dobio na osnovu činjenice da je brz, odnosno u prosječnom slučaju je brži od prethodno navedenih algoritama za sortiranje. Osnovna ideja ovog sorta je sljedeća: izabere se jedan element niza koji se proglasi pivotom. Dalje se čitav niz dijeli na dva dijela po sljedećem principu: svi elementi koji su manji od pivota se premještaju u jedan dio (recimo "lijevi" dio), dok se svi drugi elementi (uključujući i pivota) premještaju u u drugi dio ("desni" dio). Tako dobijamo situaciju da je svaki element iz lijevog dijela manji od bilo kog elementa iz desnog dijela. Nakon toga, princip sortiranja se (najčešće rekurzivno) poziva na lijevi dio i na desni dio. Tako se sada među elementima lijevog dijela bira novi pivot, a svi elementi koji su manji od njega se premještaju u lijevu stranu (lijevi dio lijevog dijela), dok ostali idu u desnu (u desni dio lijevog dijela). Slično se postupa i sa desnim dijelom, a postupak se rekurzivno ponavlja sve dok se ne dođe do dijelova niza koji su dovoljno mali i koji su trivijalno već sortirani (na primjer, nizovi bez elemenata ili nizovi sa samo jednim elementom).

Treba napomenuti da se izbor pivota može vršiti na proizvoljan način: za pivota možemo birati prvi ili posljednji element iz dijela niza na koji se sortiranje (rekurzivno) poziva, može se birati "srednji element", tj. element sa indeksom koji je na sredini u odnosu na početni i krajnji element dijela niza koji se sortira. Takođe, može se obezbijediti sistem na osnovu kog se pivot među svim potencijalnim kandidatima bira na slučajan način.

U funkciji za sortiranje niza pomoću brzog sorta, koju prikazujem,o koristimo pristup da se pivot bira kao početni element dijela niza koji se sortira. Da bi se pravilno mogla primijeniti rekurzija, funkcija za argument, pored samog niza uzima i još dva parametra koji predstavljaju donji i gornji indeks dijela niza koji se sortira u jednom datom pozivu funkcije. Element na donjem indeksu ulazi

Elektronska verzija

u razmatranje, dok je posljednji element koji se razmatra u pozivu funkcije na indeksu `gornji -1`.

```
void quicksort(int niz[] ,int donji, int gornji){
   int a, b;
   int pivot, temp;
   if(donji < gornji){
      a = donji; //pamtimo oba indeksa jer ce nam trebati kasnije
      b = gornji; //a i b koristimo za kretanje po nizu
      pivot = niz[a]; //pocetni element dijela niza koji se sortira se
          bira kao pivot
      while(1){
         while(niz[++a] < pivot && a < gornji); //pronadjemo indeks
             elementa koji je veci od posmatranog
         while(niz[--b] >= pivot && b > donji); //pronadjemo indeks
             elementa koji je manji od posmatranog
         if(a < b){ //ako se indeksi nisu sreli
            temp = niz[a];
            niz[a] = niz[b];
            niz[b] = temp;
         }
         else
      break; //prekidamo while petlju kada se indeksi sretnu

      }
      temp = niz[donji]; //razmijenimo jos pivota sa onim na kome smo
          se sreli
      niz[donji] = niz[b];
      niz[b] = temp;
   quicksort(niz ,donji, b);//rekurzivno pozovemo funkciju na lijevi
       dio
   quicksort(niz, b+1, gornji);//rekurzivno pozovemo funkciju na desni
       dio

    }
}
```

Da bi funkcija mogla rekurzivno da poziva samu sebe, pored niza ona uzima i još dva argumenta koji predstavljaju donji i gornji indeks. Stoga, moramo voditi računa i kako pozivamo funkciju, da bi ona sortirala čitav niz. Ako sortiramo niz `niz`, koji je dužine `duzina`, onda bi poziv funkcije za sortiranje tog niza bio

```
brzisort(a,0,duzina);
```

Elektronska verzija

11.6 Sortiranje spajanjem

Naziv ovog algoritma za sortiranje u našem jeziku (sortiranje spajanjem) potiče od engleskog naziva *mergesort* (engl. merge znači spojiti, sjediniti). Osnovna ideja ovog algoritma je da se niz prvo podijeli na dva jednaka dijela (dva podniza iste dužine), da se svaki od tih podnizova (rekurzivno) sortira i da se nakon toga ta dva (sortirana) podniza ponovo spoje u jedan, sortiran niz. Ako niz koji se dijeli ima paran broj elemenata, tada se on može podijeliti na dva podniza jednakih dužina, a ako je broj elemenata niza neparan, onda jedan podniz ima jedan element više od drugog. Niz se na dva dijela može podijeliti na proizvoljan način, a najčešće se koristi jedan od sljedeća dva pristupa: u prvom pristupu prva polovina niza čini jedan podniz, a druga polovina drugi, dok u drugom elementi na parnim indeksima čine jedan podniz, dok su u drugom podnizu elementi koji su na neparnim indeksima. Mogući su, naravno, i drugi pristupi za podjelu niza na dva dijela.

U standardnim implementacijama sortiranja spajanjem se najčešće koristi i dodatni memorijski prostor, u koji se privremeno smještaju elementi iz dva kraća niza, koje je potrebno spojiti u jedan. Postoje i implementacije u kojima je upotreba pomoćnog niza izbjegnuta. Mi ćemo se u našem algoritmu držati prvog pristupa i za spajanje elemenata dva podniza u jedan koristiti privremene pomoćne podnizove.

Za implementaciju sortiranja spajanjem biće nam potrebne dvije funkcije.

U funkciji `merge` ćemo obezbijediti sistem spajanja dva sortirana dijela niza u jedan segment, koji je čitav sortiran. Princip koji se koristi u ovoj funkciji je sljedeći. Neka su `L` i `D` (ovakve nazive uvodimo da nas asociraju na lijevi i desni niz) sortirani podnizovi koje trebamo da spojimo. Poredimo prvi element iz lijevog niza sa prvim elementom iz desnog niza. Ako je lijevi manji, onda se on upisuje u veliki niz na polaznu poziciju i taj element se dalje preskače u lijevom nizu. Ako je desni bio veći, onda on ide u veliki niz i njega dalje preskačemo u desnom nizu. Ovaj pristup ponavljamo tako što ponovo poredimo element iz lijevog niza (koji je prvi po redu za razmatranje) sa prvim elementom iz desnog niza koji je po redu za razmatranje. Manjeg od ta dva ubacujemo u veliki niz, a u manjem ga nizu (lijevom ili desnom, u zavisnosti od toga u kome se nalazio) preskačemo. Ovaj postupak ponavljamo sve dok "ne potrošimo" elemente iz jednog od nizova (lijevog ili desnog) i, nakon toga, u veliki niz prepisujemo ostatak elemenata iz niza u kome je ostalo još elemenata.

U funkciji `mergesort` dijelimo niz na dva dijela (lijevi i desni) i rekurzivno pozivamo funkciju za lijevi i desni dio. Nakon toga, pozivamo funkciju `merge` koja ponovo spaja lijevi i desni dio.

Elektronska verzija

```
void merge(int niz[], int l, int s, int d)
{
    int i, j, k;
    //prvo odredimo tacne duzine dva podniza
    int n1 = s - l + 1;
    int n2 = d - s;

   //napravimo podnizove
    int L[n1], D[n2];

    /* kopiramo elemente polaznog niza u lijevi i desni */
    for (i = 0; i < n1; i++)
        L[i] = niz[l + i];
    for (j = 0; j < n2; j++)
        D[j] = niz[s + 1+ j];

    /* spajamo nizove i pisemo po velikom nizu*/
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= D[j])//manji je u lijevom
        {
            niz[k] = L[i];
            i++;
        }
        else//manji je u desnom
        {
            niz[k] = D[j];
            j++;
        }
        k++;
    }

    /* Kopiramo ostatak iz lijevog niza, ako je sta ostalo */
    while (i < n1)
    {
        niz[k] = L[i];
        i++;
        k++;
    }

    /* Kopiramo ostatak iz desnog niza, ako je sta ostalo */
    while (j < n2)
```

Elektronska verzija

```
    {
        niz[k] = D[j];
        j++;
        k++;
    }
}


void mergesort(int niz[], int l, int d)
{
    if (l < d)
    {

        int s = l + (d - l) / 2; // ovo je matematicki isto kao i
            (l+d)/2, ali vodimo racuna da ne prekoracimo opseg int-a

        // rekurzivno sortiramo lijevi i desni dio
        mergesort(niz, l, s);
        mergesort(niz, s+1, d);

        merge(niz, l, s, d);
    }
}
```

Slčno kao i kod brzog sortiranja, da bi se omogućili rekurzivni pozivi, funkcija, pored niza, uzima i još dva argumenta, koji prestavljaju donji i gornji indeks. Ako sortiramo niz `niz`, koji je dužine `duzina`, onda bi poziv funkcije `mergesort` bio

```
mergesort(a,0,duzina);
```

11.7 Pitanja i zadaci

1. Objasniti razliku između klasa algoritama unutrašnjeg i spoljašnjeg sortiranja.
2. Šta se podrazumijeva pod tim da se algoritam izvršava "u mjestu"? Koji od algoritama, koji su razmatrani u ovom poglavlju, nije algoritam koji se izvršava "u mjestu" i zašto?
3. Račun za kupljeno voće sastoji od sljedećih informacija: naziv, cijena, klasa (P=prva, D=druga). Neka je dat sljedeći račun

Elektronska verzija

jabuke 1.3 P

kruske 2.8 D

ananas 3.4 P

banane 1.3 P

jagode 1.3 P

jagode 0.9 D

jabuke 0.9 D

jabuke 0.7 D

kruske 3.5 P

banane 1.0 D

Ako se vrše sortiranja redom po sljedećim kriterijumima: klasa (po abecednom redu), ime (po abecednom redu) i cijena (od manje ka većoj), napisati moguće izvršenje jednog stabilnog i jednog nestabilnog algoritma za sortiranje. Prikazati redoslijed stavki na računu u svim fazama sortiranja.

4. Uporediti vremensku složenost algoritama za sortiranje izborom i brzim sortom, u najboljem i najgorem slučaju.

5. U kom slučaju sortiranje umetanjem zahtijeva najveći broj iteracija? Obrazloži odgovor.

6. U kom slučaju brzi sort ima vremensku zavisnost $O(n^2)$? Objasniti i obrazložiti odgovor.

7. U Sekciji 11.2 dat je primjer primjene najjednostavnije verzije algoritma sortiranja izborom na zadati niz, zatim je navedena modifikovana verzija istog algoritma. Primjeniti modifikovanu verziju na isti niz, a zatim uporediti izvršenja i ukupan broj zamjena mjesta elemenata.

8. Zadat je niz

   10 5 6 1 13 7 5

   ilustrovati potrebne korake za sortiranje datog niza algoritmom pomoću mjehurića, a zatim i algoritmom koji sortira spajanjem. Uporediti dobijena izvršenja.

9. Napisati funkciju koja ispituje da li je niz koji uzima za argument sortiran u neopadajućem poretku. Funkcija kao rezultat vraća 1 ako niz jeste sortiran u neopadajućem poretku, u suprotnom 0. Testirati funkciju u glavnom dijelu programa.

Elektronska verzija

10. Napisati program koji sortira niz realnih brojeva u nerastućem poretku.

11. Jedan broj se smatra manjim od drugog ako ima manje različitih djelilaca. Napisati program koji sortira niz cijelih brojeva u nerastućem poretku, po navedenom kriterijumu.

12. Napisati program koji sortira niz karaktera, pri čemu se jedan karakter smatra manjim, ako ima manji ASCII kôd.

13. Napisati program koji sortira niz brojeva po sljedećem kriterijumu: jedan broj je manji od drugog ako mu je manja cifra najveće težine, npr. 1009 je manji od 981 jer je $1 < 9$.

14. Pronaći neki od algoritama za sortiranje, koji nije naveden u ovom poglavlju, objasniti njegov sistem rada, implementirati ga u programskom jeziku C, testirati i procijeniti njegovu vremensku složenost.

15. Da bi se uporedila vremenska složenost posmatranih algoritama za sortiranje, na nekoliko nizova različitih dužina treba primijeniti algoritme za sortiranje. Za svaki niz i svaki algoritam treba mjeriti vrijeme izvršenja programa. Detaljnije:

    a) Potrebno je najprije obezbijediti mehanizam za mjerenje vremena izvršetka programa. Jedan od načina kako se to može uraditi je upotreba funkcija biblioteke `time.h` za očitavanje sistemskog vremena.

    b) Potrebno je obezbijediti neki "brzi" način popunjavanja elemenata niza. Jedan od načina je da se nizovi popunjavaju na slučajan način. Za to je potrebno proučiti funkcije standardne biblioteke `rand()` and `srand()`.

    c) Potrebno je implementirati sve algoritme za sortiranje, uključujući i ugrađen brzi sort (funkcija `qsort`).

    d) Prilikom testiranja, mogu se praviti nizovi različitih dimenzija: na primjer, dimenzije 1000, 10000, 50000, 100000 i više. Nizove popunjavati cijelim brojevima na slučajan način. Za svaki niz pustiti svaki od algoritama za sortiranje i evidentirati vrijeme izvršavanja.

    e) Treba voditi računa da će vrijeme izvršenja algoritama rasti sa porastom dimenzije niza. Potrebno je procijeniti (na osnovu prethodnih rezultata) da li će algoritmi završiti rad u nekom razumnom vremenu (npr. do nekoliko sati). Zbog toga testiranje ne treba vršiti na računarima koji imaju problema sa hlađenjem.

Elektronska verzija

Elektronska verzija

12 Dinamička alokacija memorije

U realnim situacijama, sa kojima se programer sreće u svakodnevnom radu, često nije moguće unaprijed predvidjeti koliko podataka će u okviru programa biti obrađeno, te, samim tim, nije moguće predvidjeti ni tačnu količinu memorije koja je neophodna za smještanje tih podataka. Nekada je moguće, više ili manje precizno, procijeniti neku gornju granicu (na primjer, znamo da program radi sa cijelim brojevima, kojih neće biti više od hiljadu), ali ni to nije potpuno zadovoljavajuće: moguće je da gornja granica bude prevelika, te se, samim tim, nepotrebno zauzima više memorije, nego što je to potrebno, dok se u drugim slučajevima, tek nakon što program bude napisan, preveden i pokrenut, može ispostaviti da je količina planirane memorije nedovoljna. U nekim slučajevima ne možemo ni približno (na primjer, ni na red veličine) prije početka izvršenja programa pretpostaviti sa koliko podataka će se raspolagati (na primjer, ako čitamo podatke iz ulazne datoteke za koju unaprijed ne znamo kolike je veličine, ili ako u realnom vremenu podatke čitamo sa nekog ulaznog uređaja).

Ovakve probleme najčešće rješavamo dinamičkom alokacijom memorije, kojom se omogućava zauzimanje potrebne količine memorije u fazi izvršavanja programa. Dinamička alokacija memorije podrazumijeva da programer u programu predvidi situaciju kada je memorija potrebna i da obezbijedi odgovarajuće instrukcije, na osnovu kojih će program u toku svog izvršenja od operativnog sistema "zatražiti" potrebnu količinu memorije. Operativni sistem (ukoliko je to moguće) daje programu određeni dio memorije na raspolaganje, vraća mu informaciju o mjestu gdje se ta memorija nalazi i omogućava programu da je koristi. Nakon što program iskoristi dinamički alociranu memoriju, koju je dobio na raspolaganje u toku izvršavanja programa, dužan je da je oslobodi, kako bi ona bila ponovo na raspolaganju operativnom sistemu. O procesu oslobađanja memorije se, takođe, brine programer, tako što koristi odgovarajuće funkcije za to.

Kako se dinamička alokacija memorije odnosi na poseban dio memorijskog prostora kojim raspolaže sam program, ovdje je prilika da napravimo kratak pregled organizacije memorije koja je na raspolaganju svakom programu.

Elektronska verzija

12.1 Organizacija memorije C programa

Organizacija memorije koju program koristi se može razlikovati u zavisnosti od operativnog sistema, a često i od samog programa. Prilikom pokretanja programa, operativni sistem programu dodjeljuje memoriju koja se koristi za čuvanje samog programskog kôda i za smještanje podataka. Dodijeljena memorija je tako logički podijeljena u dva dijela: segment kôda i segment za smještanje podataka. Moderni sistemi obično koriste jedinstven dio memorije za smještanje kôda, dok je dio memorije za smještanje podataka izdijeljen na više segmenata.

12.1.1 Segment za smještanje programskog kôda

Segment kôda (koji se na engleskom jeziku zove code segment ili text segment) sadrži mašinski kôd prevedenog programa, koji uključuje mašinski kôd svih funkcija programa. Ovaj segment je obično "read only", čime se osigurava da u toku izvršenja programa ne može doći do izmjena samog kôda. U slučaju da je pokrenuto više instanci istog programa, neki operativni sistemi memoriju organizuju da sve instance dijele isti segment kôda (tj. u memoriji postoji samo jedan primjerak kôda), a onda se za svaku instancu programa posebno čuva informacija do koje je naredbe stiglo izvršavanje.

12.1.2 Segment za smještanje podataka

Segment za smještanje podataka je obično podijeljen na nekoliko dijelova (manjih segmenata), koji služe za čuvanje različitih vrsta objekata, u zavisnosti od životnog vijeka objekta, dosega ili inicijalizacije. Ti dijelovi su

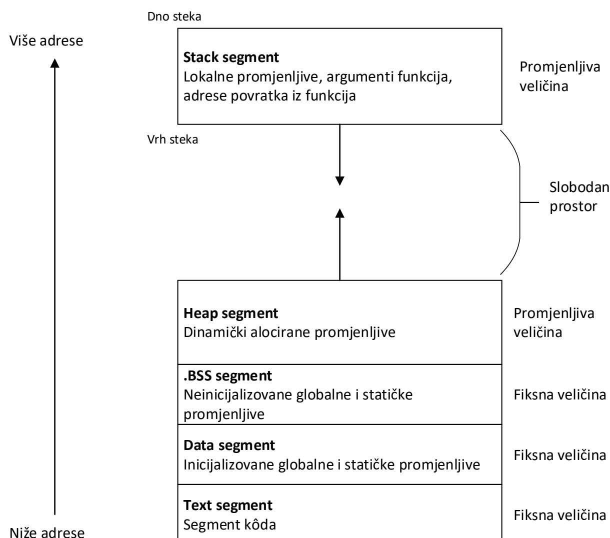
- Segment podataka za inicijalizovane podatke
- Segment podataka za neinicijalizovane podatke
- Stek segment
- Hip segment

Na Slici 12.1 je predstavljen prikaz organizacije memorije jednog C programa.

Segment podataka za inicijalizovane podatke

Segment podataka za inicijalizovane podatke (engl. Initialized Data ili jednostavno Data Segment) sadrži sve globalne i statičke promjenljive konstante i spoljašne promjenljive (koje su deklarisane pomoću riječi `extern`), koje su prethodno inicijalizovane.

Elektronska verzija



Slika 12.1: Prikaz organizacije memorije C programa

Segment podataka za neinicijalizovane podatke

Segment podataka za neinicijalizovane podatke, za koji se na engleskom jeziku koristi i izraz .BSS segment (potiče od izraza Block Started by Symbol) je segment u kome se smještaju neinicijalizovane globalne, statičke i spoljašnje promjenljive (deklarisane uz pomoć riječi **`extern`**). Podsjetimo se da, ako globalna ili statička promjenljiva nisu inicijalizovane, tada im se, prilikom pokretanja programa, dodjeljuje vrijednost 0, ili vrijednost `NULL`, ako je riječ o pokazivaču.

Stek segment

U stek segmentu ili steku poziva (od engleskog call stack) se smještaju podaci koji su vezani za izvršenje funkcija: lokalne promjenljive (koje i kreiramo unutar blokova, koji su dijelovi funkcija), argumenti funkcije, međurezultati i adresu povratka iz funkcije (mjesto odakle se, po povratku iz funkcije, nastavlja izvršenje programa). Kao što je poznato, stek je organizovan po tzv. LIFO (engl. last in - first out) principu, tj. podatak se na stek može dodati samo na vrh steka (standardna funkcija koja dodaje elemente na stek ima engleski naziv *push*) ili se podatak može "skinuti" za vrha steka (funkcijom *pop*).

Elektronska verzija

Hip segment

Hip segment (engl. heap segment) je dio memorije koji je rezerisan za dinamičku alokaciju promjenljivih. U nastavku ovog poglavlja ćemo detaljno analizirati funkcije pomoću kojih se dinamički alocira memorija.

12.2 Funkcije za dinamičku alokaciju memorije

Kao što je rečeno u prethodnoj sekciji, memorija koja se dinamički alocira pripada dijelu memorije koji se zove hip. U programskom jeziku C raspolažemo sa sljedećim funkcijama za dinamičku alokaciju memorije i njeno oslobađanje:

```
void *malloc(size_t n);
void *calloc(size_t n, size_t size);
void *realloc(void *ptr, size_t n);
void free(void *ptr);
```

Ove funkcije su definisane u standardnoj biblioteci, u zaglavlju `<stdlib.h>`. Primijetimo da su argumenti prve tri funkcije tipa `size_t`. Podsjetimo se da je `size_t` tip kojim se predstavljaju neoznačeni cijeli brojevi i koji se najčešće koristi prilikom rada sa funkcijama koje manipulišu količinom memorije. Prilikom poziva funkcija za alokaciju memorije na mjesto ovog argumenta se mogu slati i "obični" cjelobrojni podaci, ili se "obični" cjelobrojni podaci kombinuju sa podacima tipa `size_t`, u okviru aritmetičkih operatora.

12.2.1 Funkcija malloc

Kao što se vidi iz prototipa

```
void *malloc(size_t n);
```

funkcija `malloc` za argument uzima cijeli pozitivan broj `n` koji predstavlja broj bajtova koje treba alocirati. Za posljedicu uspješnog izvršenja funkcije imamo pojavu da je rezervisan memorijski blok koji sadrži `n` bajtova, dok je kao rezultat funkcije vraćen pokazivač na taj blok. Ako zahtjev za rezervisanjem memorije nije mogao biti ispunjen, tada je rezultat funkcije `NULL` pokazivač. Treba primijetiti da funkcija kao rezultat vraća generički pokazivač tipa `void*`, koji se prije upotrebe treba konvertovati u odgovarajući tip pokazivača.

Sljedećim primjerom ilustrujemo tipičan primjer upotrebe funkcije `malloc`.

```
int *p;
......
```

Elektronska verzija

```
p = (int *) malloc(100 * sizeof(int));
if(p == NULL) {
    printf("Greska: alokacija memorije nije uspjela!\n");
    exit(1);
}
```

Naredbom `p=(int *) malloc(100*sizeof(int));` traži se alokacija memorijskog prostora koji je jednak veličini stotinu cijelih brojeva tipa `int`. Primijetimo da smo operatorom `sizeof(int)` izbjegli rizik da prejudiciramo širinu `int`-a, već smo odgovornost za tačnu informaciju o širini prepustili operatoru `sizeof`. S obzirom na to da alokacija memorije uvijek sa sobom nosi određeni rizik, uobičajena je praksa da se neposredno nakon poziva funkcije `malloc` (a isti je slučaj i sa druge dvije funkcije za alokaciju), ispituje da li je alokacija uspješno obavljena.

12.2.2 Funkcija calloc

Iz sintakse prototipa funkcije `calloc`

```
void *calloc(size_t n, size_t size);
```

vidimo da funkcija uzima dva argumenta: pozivom funkcije `calloc` namjerava se alocirati `n` objekata veličine `size`. Slično kao i kod funkcije `malloc` i funkcija `calloc` kao rezultat vraća (generički) pokazivač na početak alocirane memorije u slučaju uspješne alokacije, odnosno, vrijednost `NULL`, ako alokacija nije uspjela.

Prilikom alokacije memorije funkcijom `calloc`, memorijski prostor se podešava tako da on sadrži sve nule (svaki bit u toj memoriji se postavlja na nulu). Ovo nije slučaj i sa funkcijom `malloc`. Ako iskoristimo sličan primjer koji smo koristili i kod funkcije `malloc` i dodamo jednu `for` petlju za ispis elemenata, lako možemo provjeriti da je alokacijom u ovom slučaju alociran memorijski prostor za 100 cijelih brojeva, od kojih su svi jednaki nuli.

```
int *p, i;
......
p = (int *) calloc(100,sizeof(int));
if(p == NULL)
{
    printf("Greska: alokacija memorije nije uspjela!\n");
    exit(1);
}
//ispis sadrzaja memorije na ekran
for (i = 0; i < 100; i++){
```

Elektronska verzija

```
    printf("\%d:\%d\n",i,*(p + i));//ispisuju se sve nule
}
```

Opravdano je postaviti pitanje da li su pozivi funkcije `malloc(100*sizeof(int));` i funkcije `calloc(100,sizeof(int));` ekvivalentni, pošto se i u jednom i u drugim slučaju alocira prostor za 100 cijelih brojeva. Kao što smo već pomenuli, funkcijom `calloc` sav memorijski prostor postavlja na nulu, dok kod poziva funkcije `malloc` to nije slučaj.

Druga, ne toliko očigledna razlika, je ta što operacija množenja broja objekata (označimo ga sa `n`) sa veličinom objekta (označimo ga sa `m`) kod funkcije `malloc(n*m)` može da bude problematična, ukoliko je proizvod (`mn`) toliko veliki da dolazi do prekoračenja opsega tipa `size_t`. U slučaju takvog prekoračenja, alokacija bi najvjerovatnije dala neželjen rezultat. Sa druge strane, ukoliko bi prilikom poziva funkcije `calloc(n,m)` veličine `n` i `m` bile takve da se ne može alocirati memorija tražene veličine, izvršenje funkcije `calloc` ne bi uspjelo, te bi se kao rezultat vratila vrijednost `NULL`, što je jasan indikator neuspjele alokacije.

12.2.3 Funkcija realloc

Kako već i sam prefiks "re" u funkciji `realloc` i ukazuje

```
void *realloc(void *ptr, size_t n);
```

ova funkcija se koristi za ponovnu alokaciju memorije, tako što za argument uzima pokazivač na memoriju, koja je alocirana funkcijama `malloc`, `calloc` ili nekim prethodnim pozivom funkcije `realloc` i mijenja njenu veličinu, čuvajući sadržaj te memorije. Nova veličina memorije (u bajtovima) koja se alocira je data u drugom argumentu. Slično kao i prethodne dvije funkcije i funkcija `realloc` kao rezultat vraća pokazivač na alocirani blok u slučaju uspješne alokacije, odnosno, vrijednost `NULL`, ako alokacija nije uspjela. Treba napomenuti da, u slučaju neuspješne nove alokacije, stari memorijski blok ostaje nepromijenjen. Ako se prilikom realokacije stari sadržaj prebacuje na novu lokaciju, onda će, nakon obavljene realokacije, stari memorijski prostor biti dealociran, a rezultat funkcije `realloc` će biti pokazivač na novu memorijsku lokaciju.

Funkcija `realloc` može da se koristi samo za dinamički alociranu memoriju. Razmatrajmo sljedeći primjer.

Primjer 12.1. Navešćemo prvo neispravan, a potom i ispravan kôd, kojim ilustrujemo upotrebu funkcije `realloc`.

Elektronska verzija

```
...
   int niz[2], i;
   int *p1 = niz; //ovo je dozvoljeno
   int *p2;

   niz[0] = 10;
   niz[1] = 20;

   // pogresno, jer se realocira prostor koji nije dinamicki alociran
   p2 = (int *)realloc(p1, sizeof(int) * 3);
...
```

U sljedećem listingu realociramo prostor koji jeste dinamicki alociran.

```
...
   int *p1= (int *)malloc(sizeof(int) * 2);
   int i;
   int *p2;

   *p1 = 10;
   *(p1 + 1) = 20;

   p2 = (int *)realloc(p1, sizeof(int) * 3);//ovo je sad u redu
   *(p2 + 2) = 30;
...
}
```

12.2.4 Funkcija free

Posljednja funkcija koja je veoma važna u radu sa dinamičkom alokacijom memorije je funkcija `free`, kojom se oslobađa memorijski prostor alociran nekom od prethodno razmatranih funkcija. Sintaksa upotrebe ove funkcije je jednostavna

```
free(p);
```

gdje je `p` pokazivač na alocirani blok memorije. Neophodno je da `p` pokazuje na blok memorije koji je alociran pozivima funkcija za alokaciju. Pokušaj oslobađanja memorije koja nije alocirana na ovaj način dovodi do nedefinisanog ponašanja i načešće uzrokuje grešku u toku izvršenja programa. Pored toga, treba voditi računa da se jedna te ista memorija ne oslobađa dva (ili više) puta, jer, i u tom slučaju, dolazi do nedefinisanog ponašanja. Ako je više segmenata memorije alocirano različitim pozivima funkcija za alokaciju, redoslijed

Elektronska verzija

oslobađanja (tj. pozivanja funkcije `free` ne mora da odgovara redoslijedu alociranja.

12.3 Dinamičke strukture podataka

Dinamička alokacija memorije nam omogućava da radimo sa strukturama promjenljive veličine. Za razliku od statičkih struktura podataka, koje su fiksne veličine, kod dinamičkih struktura se podrazumijeva da se veličina same strukture, a samim tim i količina podataka koji se u njoj smještaju, može mijenjati. Tako dolazimo do osnovne prednosti dinamičkih struktura, tj. činjenice da one najčešće koriste tačno onoliko memorijskog prostora koliko je u tom trenutku potrebno. Po potrebi, taj se prostor može smanjivati ili povećavati. Sa druge strane, dinamičke strukture su teže za implementaciju, jer je potrebno voditi računa i o mehanizmima pomoću kojih se upravlja memorijom (alokacija i oslobađanje memorijskog prostora, pristup određenim dijelovima memorije pomoću odgovarajućih pokazivača, povezivanje elemenata, itd.).

U nastavku če biti objašnjene neke od često korištenih dinamičkih strukura: dinamički nizovi, jednostruko i dvostruko povezana lista.

12.3.1 Dinamički nizovi

Upotrebom dinamičkih nizova pokušavamo da ispravimo nedostatak statičkih nizova, da se maksimalna dimenzija mora unaprijed ograničiti konstantom. Dinamički nizovi se prvo deklarišu kao pokazivači, a kasnije se funkcijom `malloc()` alocira prostor u memoriji za dati niz. Pri alociranju i dalje moramo imati ograničenje za maksimalnu dimenziju (pošto memorija mora da bude u jednom komadu – bloku), ali to više ne mora biti konstanta, već može biti proizvoljna promjenljiva ili izraz. Kao što je slučaj sa svom dinamički alociranom memorijom, na kraju upotrebe niza trebamo osloboditi dati prostor funkcijom `free`.

Kao i kod statičkih nizova, prilikom pristupa elementima ne vrši se kontrola granica, dok pristup memoriji van definisanog opsega može dovesti do neželjenog ponašanja programa, najčešće do greške prilikom izvršenja.

Dinamički alociran niz se jednostavno kreira funkcijom `malloc`. Evo primjera:

```
#include <stdlib.h>
. . .
int *a;
. . .
a = malloc(n * sizeof(int)); /* pravimo niz od n clanova tipa int */
. . .
```

Elektronska verzija

```
a[i] = 5;
. . .
free(a);
```

Treba napomenuti da promjenljiva `a` fizički predstavlja niz samo poslije alociranja (pomoću funkcije `malloc`), a prije oslobađanja (prije funkcije `free`). Izvan toga, ona je samo običan pokazivač.

Prikažimo sada jedan primjer u kome koristimo dinamički alociran niz.

Primjer 12.2. Iz datoteke `brojevi.txt` učitavaju se realni brojevi u dvostrukoj preciznosti do kraja ulaza. Sortirati ih u rastućem poretku i odštampati u datoteku `sortiran.txt`. Sortiranje implementirati u posebnoj funkciji.

Rješenje ćemo realizovati sa dva čitanja datoteke. U prvom čitanju ćemo prebrojati koliko ima redova, a u drugom ćemo učitati podatke u dinamički niz. Za pripremu datoteke za ponovno čitanje, koristićemo funkciju `rewind`.

Elektronska verzija

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void sort(double *a, int n)
4  {
5      int i, j;
6      double pom;
7      for(i = 0; i < n - 1; i++)
8          for(j = i + 1; j < n; j++)
9              if(a[i] > a[j])
10             {
11                 pom = a[i];
12                 a[i] = a[j];
13                 a[j] = pom;
14             }
15 }
16 void main()
17 { double *x, pom;
18     int n, i;
19     FILE *ul, *izl;
20     n = 0;
21     ul = fopen("brojevi.txt","rt");
22     izl = fopen("sortiran.txt","wt");
23     while(!feof(ul))
24     {
25         fscanf(ul,"%lf",&pom);
26         n++;
27     }
28     x = malloc(n * sizeof(double));
```

```
29      rewind(ul);
30      for(i = 0; i < n; i++)
31          fscanf(ul,"%lf",&x[i]);
32      sort(x,n);
33      for(i = 0; i < n; i++)
34          fprintf(izl,"%lf ",x[i]);
35      fclose(ul);
36      fclose(izl);
37      free(x);
38
39      return 0;
40  }
```

Uradimo još jedan važan zadatak, koji podrazumijeva da se dinamički niz formira unutar funkcije, a koristi se u glavnom dijelu programa.

Primjer 12.3. Napisati funkciju koja sa standardnog ulaza čita dimenziju niza, a zatim elemente tog niza. Napravljenu funkciju testirati u `main` funkciji.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   int* citaj(int *);
4   int main()
5   {
6     int i, n, *niz;
7
8     niz = citaj(&n);
9
10    printf("Niz:");
11    for (i = 0; i < n; i++)
12      printf(" %d", niz[i]);
13
14    free(niz);
15    return 0;
16  }
17  int* citaj(int *n)
18  {
19    int i, *niz;
20
21    do{
22      printf("n=");
23      scanf("%d", n);
24    } while (*n < 1);
25
```

Elektronska verzija

```
26    niz = (int *)malloc(*n * sizeof(int));
27
28    for (i = 0; i < *n; i++)
29    {
30      printf("%d. broj: ", i + 1);
31      scanf("%d", niz + i);
32    }
33    return niz; //vracamo pokazivac kao rezultat
34  }
```

12.3.2 Dinamički alocirane matrice

Pored jednodimenzionalnih nizova, dinamički se mogu alocirati i višedimenzionalni nizovi. Ovdje ćemo dati primjer kako se alociraju (i radi se radi sa) dinamički alociranim matricama.

Recimo da želimo da napravimo matricu `a` dimenzije `mxn` (sa `m` vrsta i `n` kolona). To možemo uraditi na dva načina: upotrebom niza pokazivača na vrste matrice, ili alokacijom odgovarajućeg dinamičkog jednodimenzionalnog niza.

Prvi način podrazumijeva da radimo sa "pokazivačem na pokazivač".

Najprije kreiramo takav pokazivač, koji ćemo, zapravo, koristiti za niz pokazivača na vrste matrice.

```
int m, n, i, j;
int **a;
...
a = (int **)malloc(m * sizeof(int *)); //pretpostavka je da je m u
    medjuvremenu dobilo neku vrijednost
```

Nakon toga, za svaku vrstu dinamički kreiramo odgovarajući niz elemenata, kojih ima onoliko koliko ima i kolona:

```
for(i = 0; i < m; i++)
    a[i] = (int *) malloc(n * sizeof(int));//pretpostavka je da je n u
        medjuvremenu dobilo neku vrijednost
```

Po završetku rada sa ovom strukturom, potrebno je, obrnutim redoslijedom u odnosu na kreiranje, da se ne bi izgubile informacije o adresama memorijskih lokacija, osloboditi zauzetu memoriju: prvo oslobodimo sve alocirane vrste, pa nakon toga i memoriju za niz vrsta.

```
for(i = 0; i < m; i++)
```

Elektronska verzija

```
    free(a[i]);
free(a);
```

Primjer kompletnog programskog kôda koji bi ilustrovao ovaj pristup

```
#include <stdio.h>
#include <stdlib.h>
void ispis(int** a, int m, int n){
    int i,j;
    printf("ispis\n");
    for(i = 0; i < m; i++){
      for (j = 0; j < n; j++)
          printf("%d\t",a[i][j]);
   printf("\n");
    }
}
int main()
{
    int n = 5, m = 6, i, j;
    int **a;
    a = (int **) malloc(m * sizeof(int *));
    for(i = 0; i < m; i++)
      a[i] = (int *) malloc(n * sizeof(int));
    for(i = 0; i < m; i++)
      for(j = 0; j < n; j++)
          a[i][j] = i * j - i;//inicijalizujemo elemente na neki
              proizvoljan nacin

    ispis(a,m,n);

    for(i = 0; i < m; i++)
    free(a[i]);
    free(a);

   return 0;
}
```

Drugi način, koji se čini jednostavnijim, ali je to opet stvar subjektivne procjene, je rad sa dinamičkim jednodimenzionalnim nizom. Za datu matricu `a` dimenzije $m \times n$ alociramo jednodimenzionalni niz dimenzije mn.

```
. . .
a = (int *)malloc(m * n * sizeof(int));
. . .
```

Elektronska verzija

Elementu matrice sa indeksima `i,j` pristupamo tako što "preskočimo" odgovarajući broj elemenata. Na primjer, ako `i,j` -tom elementu hoćemo da dodijelimo vrijednost 7, to možemo uraditi na neki od sljedećih načina

```
a[i * n + j] = 7;
```

ili

```
*(a + i * n + j) = 7
```

jer je `a` običan pokazivač na `int`.

Oslobađanje zauzetog prostora je jednostavno:

```
free(a);
```

Podsjetimo se da ovakav, pomalo konfuzan način pristupanja elementima niza možemo izbjeći upotrebom odgovarajućih makroa, na neki od sljedećih načina:

```
#define a(i,j) a[(i) * n + j]
```

ili

```
#define a(i,j) (*(a+ (i) * n + j))
```

odnosno

```
#define mat(a,i,j) (*(a + (i) * n + j))
```

odnosno

```
#define mat(a,i,j,n) (*(a + (i) * n + j))
```

Tada možemo skratiti i olakšati pisanje, pa bi se tada gornja dodjela pisala kao:

```
a(i, j) = 7;
```

odnosno

```
mat(a, i, j) = 7;
```

odnosno

```
mat(a, i, j, n) = 7;
```

Elektronska verzija

Primjer 12.4. Iskoristimo prvi od navedenih makroa, koji nam olakšava manipulaciju matricom.

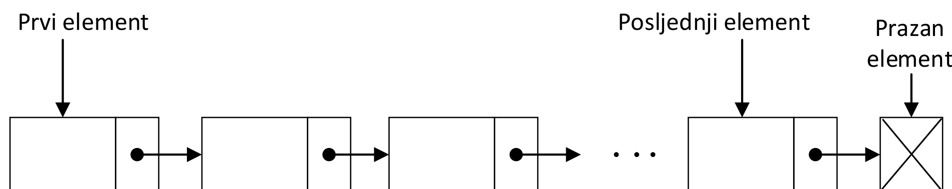
```
#define a(i,j) (*(a+ (i) * n + j))
#include <stdio.h>
#include <stdlib.h>
int main(){
   int i, j;
   int m, n;
   printf("Unesi dimenzije: ");
   scanf("%d %d",&m,&n);
   int * a = (int*)malloc(m * n * sizeof(int));
   time_t t;
   srand((unsigned) time(&t));

   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
         a(i,j) = rand() % 50;

   printf("Formirana je matrica\n");
   for (i = 0; i < m; i++){
      for (j = 0; j < n; j++)
         printf("%d\t",a(i,j));
         printf("\n");
   }
   free(a);
   return 0;

}
```

12.3.3 Liste kao dinamičke strukture

Lista je dinamička struktura podataka koja po organizaciji podsjeća na niz, jer su elementi liste linearno uređeni (elementi su poredani "jedan za drugim"). Zna se koji element je prvi element liste, koji element je sljedeći u odnosu na prvi, i uopšte, koji element slijedi za kojim elementom. Takođe, zna se i koji je posljednji element liste.

Ako se povezivanje elementa ostvaruje samo u jednom smjeru, tj. svaki element se povezuje samo sa svojim sljedbenikom, tada govorimo o jednostruko povezanoj listi. Ako od elementa liste postoje veze i ka njegovom prethodnom i ka njegovom sljedećem elementu, tada je riječ o dvostruko povezanoj listi. Lista koja nema "pravi" početak i kraj, već je ona zatvorena tako da pokazivač

Elektronska verzija



Slika 12.2: Grafički prikaz jednostruko povezane liste

na sljedeći element od posljednjeg elementa pokazuje na prvi element, zove se prsten ili kružna lista. U prstenu se od svakog elementa, krećući se kroz listu može doći opet do njega samog. Prsten može biti jednostruko ili dvostruko povezan.

Posljednji element u listi (ne prstenu) se prepoznaje tako što nema sljedećeg elementa u odnosu na njega, odnosno kažemo da je u tom slučaju sljedeći element jednak praznom elementu (pokazivač na sljedeći element od posljednjeg elementa je prazan element). Za prvi element liste važi da nema elementa kome je on sljedeći. Kod dvostruko povezane liste smatramo da je prethodni element od prvog elementa, takođe, prazan element (pokazivač na prethodni element od prvog elementa je prazan element).

Elementi liste se grafički najčešće predstavljaju kao pravougaonici, od kojih polazi strelica ka sljedećem pravougaoniku (sljedećem elementu liste) u slučaju jednostruko povezane liste, odnosno, ka prethodnom i sljedećem pravougaoniku (prethodnom i sljedećem elementu), u slučaju dvostruko povezane liste. Prazan element (pokazivač ka praznom elementu) se obično predstavlja znakom `X`, koji možemo ili ne moramo da stavimo u odgovarajući pravougaonik (Slika 12.2).

Da bismo mogli da pristupimo elementima jednostruko povezane liste (kažemo i da bismo mogli da se krećemo kroz listu), potrebno je da imamo informaciju (pokazivač) o tome gdje se nalazi prvi element liste. U slučaju dvostruko povezane liste, dovoljno je da imamo pokazivač na bilo koji element liste, a na početak se vraćamo sve dok u odnosu na trenutni element imamo i njegovog prethodnika. Radi veće efikasnosti nekih uobičajenih operacija na listi (na primjer ubacivanje novog elementa na kraj liste), često se pored informacije o tome gdje se nalazi početni element čuva i informacija o posljednjem elementu. Prvi element se još naziva i glava liste (od engleske riječi *head*), dok se ostatak liste (tj. podlista koja sadrži sve elemente, osim početnog) naziva i rep (od engleske riječi *tail*). Prazna lista je lista koja nema elemenata i ona se obično predstavlja praznim pokazivačem (`NULL`).

Za razliku od nizova, u listu možemo ubacivati nove elemente, a, takođe, možemo i izbacivati elemente iz liste. U nastavku ove sekcije objasnićemo naj-

Elektronska verzija

važnije korake u radu sa jednostruko i dvostruko povezanim listama. I kod jednostruko i kod dvostruko povezane liste, listu definišemo pomoću struktura.

Jednostruko povezane liste

U opštem slučaju, jednostruko povezana lista se realizuje tako što se za elemente liste definiše odgovarajuća struktura, kojom obezbijeđujemo da se u odgovarajućim promjenljivima (koje su elementi strukture) čuva sadržaj elementa liste, dok se veze između elemenata liste (veza od elementa ka sljedećem elementu) definišu pokazivačima. Praktično, pored sadržaja, svaki element liste sadrži i informaciju o tome gdje se nalazi njegov sljedbenik, koja se čuva u pokazivaču na sljedeći element liste.

U slučaju da radimo sa jednostruko povezanom listom, u kojoj se čuvaju cijeli brojevi, realizacija strukture koju koristimo za predstavljanje elemenata liste bi mogla da izgleda ovako:

```
typedef struct lista lista; //definisemo novi tip podatka koji se zove
    lista i koji je zapravo struct lista
struct lista{
    int broj;
    lista *sljedeci;
 };
```

Čitavom listom upravljamo pomoću pokazivača na listu, tj. pomoću pokazivača na prvi element liste. Od prvog elementa, pomoću pokazivača `sljedeci` možemo da dođemo do drugog, od drugog elementa (preko njegovog pokazivača `sljedeci`) do trećeg, itd.

U startu, lista je prazna, tj. pokazivač na listu je jednak `NULL`. Da bi se element ubacio u listu, prvo mora da se "napravi", tj. za element se mora alocirati odgovarajući memorijski prostor. Nakon što je element liste napravljen, pojedinačni elementi strukture (u našem slučaju elementi `broj i sljedeci`) dobijaju odgovarajuće vrijednosti, te se tako napravljen element ubacuje u listu. Element u listu možemo ubaciti na početak liste, na kraj, ili ga umetnuti između neka dva postojeća elementa liste.

Elemente iz liste možemo i brisati i tu takođe razlikujemo tri slučaja: brisanje prvog elementa, brisanje posljednjeg elementa ili brisanje nekog elementa iz sredine.

Kreiranje elementa se može realizovati pomoću funkcije, koja u našem slučaju cjelobrojne liste za argument uzima broj, a kao rezultat vrati pokazivač na taj element.

Elektronska verzija

```
lista* kreiraj(int n){
    lista * el = malloc(sizeof(lista));
    if(el == NULL) {
        printf("Kreiranje nije uspjelo\n");
        exit(1);
    }
    el->broj = n;
    el->sljedeci = NULL;
    return el;
}
```

Prikažimo sada kako bi izgledale funkcije za dodavanje elemenata u listu. Prvo navodimo funkciju za dodavanje elementa na početak liste.

```
lista* dodavanje_na_pocetak(lista * l, lista* el){
   if(l == NULL)
   {
      l = el;
      return l;
   }
   else
    {
      el->sljedeci = l;
      l = el;
      return l;
   }
}
```

Ako želimo da elemente dodajemo na kraj liste, a nemamo informaciju o posljednjem elementu, onda kao argumente u funkciju šaljemo informaciju o prvom elementu (to je argument `l`), pa se onda unutar petlje prvo sa početka trebamo pomjeriti do kraja i nakon toga možemo izvršiti dodavanje. Funkcija, koja bi radila po navedenom principu, bi mogla da izgleda ovako:

```
lista* dodavanje_na_kraj(lista *l, lista * el){
   if(l == NULL)
      {
         l = el;
         return l;
      }
   lista *p = l;
   while(p->sljedeci != NULL)
      p = p->sljedeci;
```

Elektronska verzija

```
  p->sljedeci = el;
  return l;
}
```

Primijetimo da funkcija kao rezultat vraća pokazivač na prvi element liste.

Ako za dodavanje na kraj koristimo i pokazivač na kraj liste, onda možemo koristiti funkciju koja za argumente uzima pokazivač na posljednji element (to je formalni argument q) i element koji se dodaje.

```
lista* dodavanje_na_kraj2(lista * q, lista * el){
  q->sljedeci = el;
  q = el;
  return q;
}
```

Ovako napisana funkcija vraća pokazivač na posljednji element. Primijetimo da se ova funkcija ne može koristiti, ukoliko je pokazivač na posljednji element prazan.

Funkcija koja dodaje element el iza elementa p (za koga pretpostavljamo da nije prazan), bi mogla da izgleda ovako:

```
lista * dodavanje_iza(lista * p, lista * el){
  el->sljedeci = p->sljedeci;
  p->sljedeci = el;
  return p;
}
```

Pored funkcija za dodavanje elemenata, važna je i funkcija za oslobađanje memorije. Prilikom oslobađanja memorije potrebno je obrisati sve elemente liste, jedan po jedan.

```
void brisanje(lista * l){

  lista* p;

  while(l != NULL){

    p = l; //pamtimo tekuci element
    l = l->sljedeci; //prebacimo se na sljedeci
    free(p); //brisemo tekuci

  }
}
```

Elektronska verzija

Primjer 12.5. Uradimo sada jedan malo duži kompletan zadatak. Pretpostavimo da su u datoteci, u svakom redu po jedan, upisani brojevi. Potrebno je brojeve učitati u jednostruko povezanu listu i nakon učitavanja izračunati njihov zbir. U zadatku ćemo, zbog kompletnosti, navesti i neke funkcije koje se neće koristiti.

```
#include <stdio.h>
#include<stdlib.h>
typedef struct lista lista;
struct lista{
    int broj;
    lista *sljedeci;
 };

lista* kreiraj(int n){

    lista * el = (lista*)malloc(sizeof(lista));
    if(el == NULL) {
        printf("Kreiranje nije uspjelo\n");
        exit(1);
    }
    el->broj = n;
    el->sljedeci = NULL;
    return el;
}
lista* dodavanje_na_pocetak(lista * l, lista* el){
   if(l == NULL)
      {
         l = el;
         return l;
      }
   else{
      el->sljedeci = l;
      l = el;
      return l;
   }

}
lista* dodavanje_na_kraj2(lista * q, lista * el){

   q->sljedeci = el;
   q = el;
   return q;
}
```

Elektronska verzija

```
39
40  void ispis(lista * l){
41     printf("Ispis liste...\n");
42     while(l != NULL){
43        printf("Element: %d\n",l->broj);
44        l = l->sljedeci;
45     }
46
47  }
48  lista * dodavanje_iza(lista * p, lista * el){
49
50     el->sljedeci = p->sljedeci;
51     p->sljedeci = el;
52     return p;
53
54
55  }
56  void brisanje(lista * l){
57     lista* p;
58     printf("Brisanje...\n");
59     while(l != NULL){
60        //printf("%d\n",l->broj);
61        p = l;
62        l = l->sljedeci;
63        free(p);
64     }
65  }
66  int suma(lista * l){
67
68     lista* p = l;
69     int rezultat = 0;
70     while(p != NULL){
71        rezultat += p->broj;
72        p = p->sljedeci;
73     }
74     return rezultat;
75  }
76  int main(){
77     lista* l = NULL, *zadnji = NULL;
78
79     FILE* f = fopen("lista1.txt","r");
80     FILE* g = fopen("izlaz.txt","w");
81     int broj;
82     if(f == NULL)
83     {
```

Elektronska verzija

```
84        printf("Ulazna datoteka ne moze da se otvori.\n");
85        exit(1);
86     }
87     if(g == NULL)
88     {
89        printf("Nije moguce pisanje u izlaznu datoteku.\n");
90        exit(1);
91     }
92
93     while(!feof(f))//izlazimo kad dodjemo do kraja datoteke
94     {
95        fscanf(f,"%d",&broj);
96        if(l == NULL)
97        {
98           l = kreiraj(broj);
99           zadnji = l;
100       }
101       else
102          zadnji = dodavanje_na_kraj2(zadnji,kreiraj(broj));
103    }
104    ispis(l);
105
106    printf("Suma elemenata %d\n",suma(l));
107    brisanje(l);
108    fclose(f);
109    fclose(g);
110
111    return 0;
112 }
```

U okviru ovog primjera smo naveli i funkciju za ispisivanje liste, koja se može koristiti i kod jednostruko, ali i kod dvostruko povezane liste.

Dvostruko povezane liste

Kao i u slučaju jednostruko povezane liste, dvostruko povezanu listu realizujemo tako što za elemente liste definiše odgovarajuća struktura kojom obezbijeđujemo da se u odgovarajućim promjenljivima (koje su elementi strukture) čuva sadržaj elementa dvostruko povezane liste, dok se veze između elemenata liste definišu pomoću pokazivača na prethodni i sljedeći element.

U primjeru, kroz koji ćemo ilustrovati rad sa dvostruko povezanom listom ćemo ponovo pretpostaviti da je sadržaj cijeli broj. Realizacije strukture za elemente dvostruko povezane liste bi mogla da izgleda ovako:

Elektronska verzija

```
typedef struct lista lista; //definisemo novi tip podatka koji se zove
    lista i koji je zapravo struct lista
struct lista{
    int broj;
    lista *sljedeci, *prethodni;
 };
```

Na sličan način, kao u slučaju jednostruko povezane liste, definišemo neke funkcije za kreiranje, dodavanje i brisanje elemenata.

```
lista* kreiraj(int n){

    lista *el = (lista*)malloc(sizeof(lista));
    if(el == NULL) {
        printf("Kreiranje nije uspjelo\n");
        exit(1);
    }
    el->broj = n;
    el->sljedeci = NULL;
    el->prethodni = NULL;//postavljamo i pokazivac na prethodni na NULL
    return el;
}
```

Funkcija za dodavanje na početak:

```
lista* dodavanje_na_pocetak(lista * l, lista* el){
   if(l == NULL)
     {
        l = el;
        return l;
     }
   else{
     el->sljedeci = l;
     l->prethodni = el;//uvezujemo i pokazivac na prethodni element
     l = el;
     return l;
   }
}
```

Funkcija za dodavanje na kraj, ako nema pokazivača na posljednji element.

```
lista* dodavanje_na_kraj(lista *l, lista * el){

   if(l == NULL)
```

Elektronska verzija

```
      {
         l = el;
         return l;
      }
   lista *p = l;
   while(p->sljedeci != NULL)
      p = p->sljedeci;
   p->sljedeci = el;
   el->prethodni = p;
   return l;
}
```

Funkcija za dodavanje elementa iza (poslije) datog elementa.

```
lista* dodavanje_iza(lista * p, lista * el){

   el->sljedeci = p->sljedeci;
   p->sljedeci = el;
   el->sljedeci->prethodni = el;
   el->prethodni = p;
   return p;
}
```

I ostale funkcije se definišu na sličan način. Uradimo ponovo jedan duži primjer.

Primjer 12.6. U ulaznoj datoteci su upisani brojevi, u svakom redu po jedan. Koristeći dvostruko povezanu listu, i izlaznu datoteku upisati te brojeve, obrnutim redoslijedom u odnosu na ulaz.

Zadatak možemo uraditi na nekoliko načina. Na primjer, brojeve možemo dodavati kao elemente liste na početak liste, pa bismo onda ispisom liste (počevši od početka), u izlaznoj datoteci upravo dobili obrnuti redoslijed. Drugi način, koji ćemo mi koristiti je da dodajemo brojeve na kraj, a da onda i ispis obavljamo sa kraja liste prema početku. Na taj način, krećući se i u jednom i u drugom smjeru, možemo provjeriti da li smo dobro podesili i pokazivače na prethodne elemente. Zbog kompletnosti, i u ovom primjeru, navodimo i druge funkcije koje se mogu koristiti pri radu sa dvostruko povezanim listama.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 typedef struct lista lista; //definisemo novi tip podatka koji se zove
     lista i koji je zapravo struct lista
4 struct lista{
```

Elektronska verzija

```
5       int broj;
6       lista *sljedeci, *prethodni;
7    };
8
9   lista* kreiraj(int n){
10
11      lista * el = (lista*)malloc(sizeof(lista));
12      if(el == NULL) {
13          printf("Kreiranje nije uspjelo\n");
14          exit(1);
15      }
16      el->broj = n;
17      el->sljedeci = NULL;
18      el->prethodni = NULL;
19      return el;
20  }
21  lista* dodavanje_na_pocetak(lista * l, lista* el){
22     if(l == NULL)
23        {
24           l = el;
25           return l;
26        }
27     else{
28        el->sljedeci = l;
29        l->prethodni = el;
30        l = el;
31        return l;
32     }
33
34  }
35  lista* dodavanje_na_kraj(lista *l, lista * el){
36
37     if(l == NULL)
38        {
39           l = el;
40           return l;
41        }
42     lista *p = l;
43     while(p->sljedeci != NULL)
44        p = p->sljedeci;
45     p->sljedeci = el;
46     el->prethodni = p;
47     return l;
48
49  }
```

Elektronska verzija

```
50 lista* dodavanje_na_kraj2(lista * q, lista * el){
51
52    q->sljedeci = el;
53    el->prethodni = q;
54    q = el;
55    return q;
56 }
57 void brisanje(lista * l){
58    lista* p;
59    printf("Brisanje...\n");
60    while(l != NULL){
61       p = l;
62       l = l->sljedeci;
63       free(p);
64
65    }
66 }
67
68 int main(){
69
70    lista* l = NULL, *zadnji = NULL;
71
72    FILE* f = fopen("lista1.txt","r");
73    FILE* g = fopen("izlaz.txt","w");
74    int broj;
75    if(f == NULL)
76    {
77       printf("Ulazna datoteka ne moze da se otvori.\n");
78       exit(1);
79    }
80    if(g == NULL)
81    {
82       printf("Nije moguce pisanje u izlaznu datoteku.\n");
83       exit(1);
84    }
85
86    while(!feof(f))//izlazimo kad dodjemo do kraja datoteke
87    {
88       fscanf(f,"%d",&broj);
89       if(l == NULL)
90       {
91          l = kreiraj(broj);
92          zadnji = l;
93       }
94       else
```

Elektronska verzija

```
95          zadnji = dodavanje_na_kraj2(zadnji,kreiraj(broj));
96      }
97       //krenemo od posljednjeg elementa, pa idemo unazad
98      lista * p = zadnji;
99      while(p != NULL){
100        fprintf(g,"%d\n",p->broj);
101        p = p->prethodni;
102     }
103     brisanje(l);
104     fclose(f);
105     fclose(g);
106
107     return 0;
108 }
```

12.4 Pitanja i zadaci

1. Ako je dat sljedeći dio kôda

```
extern int a = 100, b;
int x, y = 200;

void f(int s){
    ...
    int c = 25;
    int d;
    ...
}

int g(int n){
    ...
    int *p;
    ...
    p=(int*)calloc(n, sizof(int));
    ...
}
```

   za svaki od navednih podataka odrediti u kom segmentu za smještanje podataka je sačuvan.

2. Objasniti kako je organizovana raspodjela memorije u programskom jeziku C.

Elektronska verzija

3. U čemu je razlika između funkcija `malloc` i `calloc`?

4. Za šta se koristi funkcija `free`? Ilustrovati primjerom.

5. Objasniti razliku između statičkih i dinamičkih nizova. Ilustrovati primjerom.

6. Kako se mogu dinamički alocirati dvodimenzionalni nizovi? Uporediti mogućnosti i ilustrovati primjerom.

7. Definisati prsten listu.

8. Kako od datog elementa doći do njegovog prethodnika u listi, ako je lista
   a) kružna
   b) dvostruko povezana?

9. Kako prepoznati prvi, a kako posljednji element liste, ako je lista
   a) jednostruko
   b) dvostruko

   povezana?

10. Opisati redoslijed zauzimanja i oslobađanja memorije kod dinamički alociranih dvodimenzionalnih nizova.

11. Po čemu se razlikuju nizovi i liste?

12. Napisati rekurzivnu funkciju za sabiranje elemanata liste.

13. Napisati funkciju za sortiranje elemenata liste po principu sortiranja umetanjem.

14. Napisati prototip formiranja dvostruko povezane liste, koja sadrži karaktere.

15. Učitati realne brojeve iz datoteke u jednostruko povezanu listu, sortirati ih po vrijednosti razlomljenog dijela (npr. 123.45 je veće od 999.06 jer $0.45 > 0.6$) i, nakon sortiranja, ispisati ih u izlaznu datoteku.

16. U datoteci se nalaze cijeli brojevi, učitati ih u jednostruko povezanu listu, a zatim ispisati u izlaznu datoteku u obrnutom redoslijedu.

17. Iz datoteke učitati brojeve u dvostruko povezanu listu, a zatim odrediti i na ekranu ispisati pozicije onih elemenata liste koji su veći od svojih prethodnika, a manji od svojih sljedbenika.

Elektronska verzija

18. Vrati se na Primjer 12.3 i odgovori na sljedeća pitanja:
    a) Kog tipa je argument funkcije `citaj` i zašto?
    b) Čemu služi `do-while` petlja u funkciji `citaj`?
    c) Zašto je u trećem redu navedena deklaracija funkcije `citaj` i šta bi se desilo ako te deklaracije ne bi bilo?
    d) Šta funkcija `citaj` vraća kao rezultat?
    e) Zašto je u 19. redu kôda napisana naredba `scanf("%d", n);`, a ne `scanf("%d", &n);`?
    f) U kom dijelu memorije je alociran niz u 21. redu kôda?

Elektronska verzija

13 Preprocesorske naredbe

Preprocesorske naredbe su naredbe koje se izvršavaju prije procesa prevođenja programa. Njih izvršava zaseban dio prevodioca koji se zove preprocesor, koji na osnovu instrukcija koje su date u samim preprocesorskim naredbama (preprocesorskim direktivama) mijenja tekst izvornog kôda. Izmjene obično uključuju jednostavne operacije nad samim tekstom, dok se samo značenje naredbi koje će postati sastavni dio izvornog kôda ne analizira u toku preprocesiranja. Neke od operacija koje se obavljaju u fazi preprocesiranja su uklanjanje komentara ili spajanje linija razdvojenih u izvornom programu simbolom \. Pored toga, interpretiraju se i preprocesorske naredbe (direktive) koje se prepoznaju po tome što počinju znakom # i završavaju se znakom za kraj reda (ne znakom tačka-zarez). Opšti oblik preprocesorske naredbe izgleda ovako:

```
#naredba parametri
```

Neke od najčešće korištenih preprocesorskih naredbi, a sa nekima od njih smo se već i susretali, su `#include` `#define` `#undef` `#if` `#ifdef` `#ifndef` `#elif` `#else`. U nastavku ćemo objasniti čemu služe i kako se koriste ove naredbe.

13.1 Naredba #include

Naredbom `#include` uključujemo takozvane `header` fajlove, ili na našem jeziku datoteke zaglavlja. Osnovni primjer datoteka zaglavlja su zaglavlja standardne biblioteke, koje smo uključivali do sada u skoro svakom programu. Datoteke zaglavlja koristimo za smještanje deklaracija koje se koriste u više različitih datoteka izvornog kôda. U takve deklaracije ubrajamo prototipe funkcija, spoljašnje promjenljive, konstante, makroe ili tipove podataka definisane od strane korisnika. Smještanjem ovakvih deklaracija u datoteke zaglavlja izbjegavamo dupliranje kôda, a, samim tim, ubrzavamo proces razvoja programa i smanjujemo mogućnost pojave grešaka.

Datoteke zaglavlja se najčešće koriste za uključivanje:

- simboličkih konstanti (na primjer, u datoteci `limits.h` su deklarisane konstante koje specifikuju svojstva cijelih brojeva)

Elektronska verzija

- deklaracija funkcija (na primjer, funkcije za rad sa stringovima su deklarisane u datoteci `string.h`)
- deklaracija struktura (na primjer, u datoteci `time.h` je definisana struktura `tm`, koja se koristi za rad sa vremenom i datumom)
- deklaracija tipova (na primjer, u istoj datoteci `time.h` definisan je tip `time_t`, koji je pogodan za smještanje informacija o "kalendarskom vremenu")
- makroa funkcija – (na primjer, `getchar()` je obično definisan kao makro funkcija `getc(stdin)`)

Datoteke zaglavlja se po dogovoru završavaju ekstenzijom `.h`. Jedna datoteka zaglavlja može unutar sebe takođe sadržavati druge `#include` naredbe.

Kao što smo i imali priliku da vidimo u velikom broju prethodnih primjera, funkcije standardne biblioteke se uključuju tako što se naziv datoteke stavlja u okviru zagrada < ... >. U opštem slučaju

```
#include <ime_datoteke.h>
```

Uključivanje datoteke zaglavlja pomoću ovih zagrada ukazuje da se datoteka nalazi na nekom "standardnom" mjestu, zapravo u sistemskom `include` folderu, koji sadrži standardne datoteke zaglavlja. Lokacija ovog foldera zavisi od sistema i verzije C prevodioca koji se koristi.

Drugi način uključivanja datoteka zaglavlja je

```
#include "ime_datoteke.h"
```

koji ukazuje da se datoteka nalazi na nekom "lokalnom" mjestu, obično u istom folderu gdje je i datoteka u koju se uključuje. Generalno pravilo je da se navodnici koriste kada se uključuje datoteka koju je kreirao programer, dok se pomoću znakova < .. > najčešće uključuju datoteke koje su dio (obično standardne) biblioteke.

13.2 Naredba #define

Pomoću direktive `#define` jedan niz karaktera u izvornoj datoteci možemo da zamijenimo nekim drugim nizom karaktera. Forma ove preprocesorske direktive izgleda ovako

```
#define originalni_tekst novi_tekst
```

Elektronska verzija

Svako ime koje je definisano u nekoj **#define** naredbi nazivamo makro (engl. macro).

Preprocesor će od mjesta na kome se direktiva nalazi, pa sve do kraja datoteke svako pojavljivanje originalnog teksta zamijeniti novim tekstom. Ovo se ne odnosi na znakove unutar niski (znakove unutar dvostrukih navodnika).

Na primjer, na osnovu direktive

```
#define MAX_DIM 1000
```

će prije prevođenja tekst `MAX_DIM` u izvornom kôdu biti zamijenjen vrijednošću 1000, gdje god se on pojavljuje osim ako nije dio niske.

Tako će, na primjer, zapis

```
...
int niz[MAX_DIM];

...

printf("Vrijednost MAX_DIM: \%d", MAX_DIM);
...
```

nakon preprocesiranja, a prije prevođenja izgledati

```
...
int niz[1000];

...

printf("Vrijednost MAX_DIM: \%d", 1000);
...
```

Uvođenjem simboličkih konstanti pomoću preprocesorskih direktiva programer sebi može olakšati posao, u situacijama kada jednu istu konstantnu vrijednost treba koristiti na više mjesta u programu. Definisanjem te vrijednosti samo na jednom mjestu, smanjuje mogućnost pojave grešaka i olakšava posao ukoliko tu vrijednost treba promijeniti (u ovom slučaju potrebno je izvršiti izmjenu samo na jednom mjestu).

U nekim slučajevima, uvođenjem simboličkih konstanti možemo ubrzati razvoj izvornog kôda, tako što ćemo kucanje dugih izraza (posebno onih koji se često ponavljaju) izbjeći upotrebom odgovarajućeg makroa. Na primjer, sljedeću direktivu

```
#define OPSEG "Greska: Ulazni parametar je van opsega"
```

Elektronska verzija

bismo mogli koristiti u dijelu kôda

```
...
printf("Unesi pozitivan broj");

scanf("%d",&n);
if (n <= 0)
{
    printf(OPSEG);
    ...
}
...
```

Treba razlikovati simboličko ime, uvedeno preprocesorskom direktivom, od promjenljivih (uključujući i promjenljive koje imaju oznaku **const**). Simboličke konstante (kao što je u prethodnom primjeru MAX_DIM) ne zauzimaju prostor u memoriji tokom izvršenja programa, već se prije prevođenja mijenjaju zadatom vrijednošću. Originalni tekst, koji se mijenja makroom u okviru preprocesorske direktive, nije vidljiv prevodiocu.

Definicija imena koje je uvedeno naredbom **#define** može biti poništena pomoću **#undef**. Ova naredba se ne koristi često, a jedna od situacija kada ju je opravdano koristiti se dešava, kada nanovo želimo da definišemo neki makro.

13.2.1 Parametrizovana naredba #define

Osim za definisanje simboličkih konstanti, naredba **#define** se koristi i za definisanje takozvanih parametrizovanih makroa, tj. makroa koji raspolažu odgovarajućim parametrima. Za definisanje ovakvih makroa koristimo parametrizovanu direktivu **#define**, kod koje i simboličko, originalno ime, i novi tekst koji ga mijenja, sadrže parametre (argumente), koji se, prilikom poziva makroa, mijenjaju stvarnim argumentima. Opšti zapis parametrizovane naredbe **#define** je sljedeći

```
#define originalno_ime(argumenti) novi_tekst
```

Evo jednog primjera upotreba parametrizovanog makroa:

```
#define MAX(x,y) ((x)>(y) ? (x) : (y))
```

Ako se u kôdu, na primjer, pojavi naredba

```
c = MAX(a,b)
```

Elektronska verzija

preprocesor će je zamijeniti sa

```
c=((a)>(b) ? (a) : (b))
```

Iako na prvi pogled imaju dosta sličnosti, parametrizovani makro i funkcija se značajno razlikuju. Parametrizovani makro ne podrazumijeva funkcijski poziv i prenošenje argumenta, već se prilikom preprocesiranja samo odgovarajući tekst u izvornom kôdu mijenja tekstom samog makroa.

Parametrizovani makroi se u praksi često koriste. Evo još nekoliko primjera.

```
#define SQR(x) ((x)*(x))
#define SGN(x) (((x) < 0) ? -1 : 1)
#define ABS(x) (((x) < 0) ? -(x) : (x))
#define ISDIGIT(x) ((x) >= '0' && (x) <= '9')
#define NELEMS(array) (sizeof(array) / sizeof(array[0]))

#define a(i,j) a[(i) * n + j]
#define mat(a,i,j) (*(a+ (i) * n + j))
```

Podsjetimo se da smo dva posljednja parametrizovana makroa pominjali u Sekciji 12.3.2.

`SQR` računa kvadrat datog broja, `SGN` je poznata funkcija `signum(x)` koja vraća vrijednost -1, ako je argument negativan, a inače vraća 1. Funkcija `ABS` je apsolutna vrijednost. Funkcija `ISDIGIT` provjerava da li je argument cifra, dok funkcija `NELEMS` vraća informaciju o broju elementa u nizu. Podsjetimo se da smo dva posljednja parametrizovana makroa pominjali u Sekciji 12.3.2, pomoću kojih možemo pristupati elementima matrice, koja je dinamički alocirana kao jednodimenzionalni niz.

Pri radu sa parametrizovanim makroima, mora se biti izuzetno pažljiv, jer nedovoljno precizna definicija makroa često može da dovede do pogrešnih rezultata. Na primjer, ako bi se makro koji računa kvadrat datog broja definisao sa

```
#define SQR(x) (x*x)
```

što na prvi pogled izgleda sasvim logično, do problema bi se došlo, ako bi se taj makro pozvao, na primjer, u sljedećoj situaciji

```
...
int a = 5;
b = SQR(a + 1);
...
```

Elektronska verzija

Promjenljiva `b` bi se tada računala pomoću izraza `b = (a + 1 * a + 1)`, odnosno, `b = 2 * a + 1 = 11`, što naravno nije jednako kvadratu izraza `a+1`. Primijetimo da u ovom primjeru nije dovoljno makro definisati čak ni sa

```
#define SQR(x) (x)*(x)
```

jer bi, recimo, makro primijenjen na izraz `b/SQR(a)` rezultovao sa `b/(a)*(a)`, jer se zbog jednakih prioriteta dijeljenja i množenja izraz izvršava s lijeva na desno, pa je on ekvivalentan sa `(b/(a))*(a)`, a ne sa `b/((a)*(a))`. Stoga je opšta preporuka da se, prilikom definisanja makroa, veoma vodi računa o zagradama, kojima se precizno definiše redoslijed izvršenja operacija.

Sa druge strane, čak i ako je makro definisan na ispravan način, u nekim situacijama postoje ograničenja na koje argumente se može primijeniti. Na primjer, ako bismo ispravno napisan makro

```
#define SQR(x) ((x)*(x))
```

primijenili na argument `a++`, tj.

```
SQR(a++);
```

što bi se u fazi preprocesiranja zamijenilo sa

```
((a++) * (a++))
```

odnosno, vrijednost promjenljive `a` bi se uvećala dva puta (što vjerovatno nije planirano).

Ako u direktivi `#define`, u novom tekstu ispred imena parametra navedemo znak `#`, tada će odgovarajuća kombinacija karaktera biti zamijenjena vrijednošću parametra koji je sada naveden između dvostrukih navodnika. Znak `#` u ovom slučaju djeluje kao operator, koji makro parametar pretvara u nisku. Na primjer, makro definisan pomoću:

```
#define PSQR(X) printf("Kvadrat od broja" #X " je %d.\n",((X)*(X)));
```

bi uticao na sljedeći red

```
PSQR(y);
```

tako što bi se on preprocesirao u

```
printf("Kvadrat od broja" "y" " je %d.\n",((y)*(y)));
```

Elektronska verzija

U **#define** naredbi tekst zamjene se prostire od imena koje definišemo pa sve do kraja linije. U slučaju da je za definiciju makroa potrebno više linija teksta, koristimo kosu crtu \ na kraju svakog reda, osim posljednjeg.

Važno je napomenuti i da se prilikom definisanja makroa zagrade moraju "slijepiti" uz ime makroa, tj. između imena i otvorene zagrade ne smije biti razmaka. Na primjer, definicija makroa kod koga između riječi `MAX` i otvorene zagrade ima razmak:

```
#define MAX (x,y) ((x)>(y) ? (x) : (y))
```

ne bi bila ispravna, jer bi u ovom slučaju svako pojavljivanje riječi `MAX` bilo zamijenjeno tekstom `(x,y) ((x)>(y) ? (x) : (y))`.

13.3 Uslovno prevođenje

Uslovnim prevođenjem, koje se vrši u fazi preprocesiranja, možemo isključiti ili uključiti određene dijelove kôda. Postoji nekoliko preprocesorskih direktiva za uslovno prevođenje: **#if**, **#elif**, **#else**, **#ifdef**, **#ifndef** i **#endif**. Naredba **#if** se koristi u sljedećem obliku:

```
#if uslov
   blok naredbi
#endif
```

Ako je `uslov` ispunjen, blok naredbi će biti uključen u izvorni kôd, a inače neće. Uslov koji se pojavljuje u **#if** naredbi mora biti konstantan cjelobrojan izraz. Kao i u slučajevima do sada, nula se interpetira kao istinitosna vrijednost netačno, dok se vrijednost različita od nule smatra tačnom. Simbolička imena se, prije izračunavanja izraza, mijenjaju svojim vrijednostima. Ako se u uslovu pojavi simboličko ime koje prije toga nije definisano nekom **#define** naredbom, onda se ono mijenja nulom.

Složenije **if** naredbe se grade pomoću naredbi **#else** i **#elif**, koji ima značenje **else if**. U nekom opštem zapisu, ove bi se naredbe koristile ovako:

```
#if uslov1
   blok naredbi1
#elif uslov2
   blok naredbi2
#elif uslov3
   blok naredbi3
...
#else
```

Elektronska verzija

```
    blok naredbi4
#endif
```

U praksi se uslovno prevođenje koristi u nekom od sljedećih slučajeva:

- da bi se opcionalno uključio dio dodatnog kôda koji pomaže u otklanjanju grešaka,
- da bi se uključio/isključio dio kôda koji se odnosi na specifično okruženje (na primjer, ako poziv funkcija zavisi od platforme na kojoj platformi se pokreće program),
- da bi se spriječilo višestruko uključivanje datoteka zaglavlja.

Objasnimo ukratko svaki od slučajeva.

U programerskoj praksi nekada je potrebno proširiti izvorni kôd dodatnim naredbama koje daju informacije o toku izvršenja programa. Na primjer, u nekim kritičnim situacijama, kada nije potpuno jasno kako se program ponaša, korisno je na ekranu (ili na neki drugi izlaz) ispisati vrijednosti pojedinih promjenljivih, ili jasno prepoznati mjesto u programu do kog je izvršenje stiglo. Po uspješno završenom razvoju programa, taj dio kôda se može izbrisati, ili se on može učiniti neaktivnim upotrebom uslovnog prevođenja. Na primjer

```
#define DEBUG

...
#ifdef DEBUG
    printf("Pokazivac %p pokazuje na vrijednost %d", pd, *pd);
#endif
...
```

Ako je planirano da se program koristi na različitim platformama (na primjer, da se pokreće na različitim operativnim sistemima), moguće je da se neki dijelovi kôda moraju prilagođavati specifičnim uslovima, koje diktira sama platforma. Često korištena praksa u ovim situacijama je da se pomoću uslovnog prevođenja obezbijedi sistem uključivanja, odnosno isključivanja odgovarajućeg dijela kôda. Pojednostavljen primjer za ovu situaciju bi bio:

```

#ifdef __WIN32__ /* informacija da se radi o Windows operativnom
    sistemu */
.../* naredbe koje se izvrsavaju pod Windows-om */
#elif defined(__linux__) /* Linux operativni sistem */
...  /* naredbe koje se izvrsavaju pod Linux-om */
```

Elektronska verzija

```
#endif
```

Izraz `#defined(ime)` nam pomaže da dobijemo informaciju da li je `ime` definisano ili ne. Ako jeste, tada je vrijednost izraza 1, a ako nije, vrijednost izraza je 0.

Treći čest slučaj, kada je praktično da koristimo uslovno prevođenje se javlja kada želimo da se osiguramo da će datoteka zaglavlja u program biti uključena tačno jednom. Ukoliko se izvorni kôd programa nalazi u više datoteka (a to je veoma čest slučaj kod iole većih programa), moguće je da se u većem broju tih datoteka uključuje ista datoteka zaglavlja. Na taj način može doći do višestrukih definicija istih simboličkih vrijednosti, što najčešće dovodi do grešaka. Da bi se izbjegao ovaj problem višestrukog uključivanja, prije naredbe uključivanja se prvo vrši provjera, da li je ta datoteka već uključena ranije. Ako nije, onda se uključuje, a ako je bila uključena ranije, onda ne. Evo primjera, kako se to praktično realizuje:

```
#if !defined(__datoteka.h__)/*najprije ispitujemo da li je datoteka
    ranije ukljucena*/
#define __datoteka.h__ /*ako nije, evidentiramo da smo cemo je upravo
    sad ukljuciti */
/* ovdje dolazi datoteka.h */
#endif
```

Umjesto ispitivanja uslova pomoću naredbi `#if defined` i `#if !defined` možemo koristiti i `#ifdef` i `#ifndef`. Na primjer, umjesto kôda iz prethodnog primjera možemo koristiti i

```
#ifndef __datoteka.h__
#define __datoteka.h__
/* ovdje dolazi datoteka.h */
#endif
```

13.4 Pitanja i zadaci

1. Šta su datoteke zaglavlja i za šta se koriste?
2. U čemu je razlika između narednih uključivanja datoteke zaglavlja:
    - #include <dat.h>
    - #include "dat.h"

Elektronska verzija

3. Da li je makro funkcija ispravno definisana `#define kub(x) x * x * x`. Obrazloži odgovor.

4. Osmisliti primjer u kome je opravdana upotreba direktive `#undef`.

5. Pronaći odgovore na pitanje čemu služe sljedeći makroi:

    - `__DATE__`
    - `__TIME__`
    - `__FILE__`
    - `__LINE__`

Elektronska verzija

Literatura

B.W. Kernighan and D.M. Ritchie. The C Programming Language. Prentice-Hall, 2nd edition, 1988.

R. Sedgewick. Algorithms in C. Addison-Wesley, 3rd edition, 1998.

F. Marić, P, Janičić, PROGRAMIRANJE 1. Osnove programiranja kroz programski jezik C. Beograd. 2015

M. Jurak, Programski jezik C, predavanja ak. g. 2003/04.

M. Čabarkapa, C, Osnovi programiranja, Biblioteka Algoritam, 2000.

M. Čabarkapa, S. Matković, C/C++, Biblioteka Algoritam, Krug, Beograd, 2005.

D. Urošević, Algoritmi u programskom jeziku C, Mikro Knjiga, Beograd, 1996.

L. Kraus, Programski jezik C sa rešenim zadacima, Akademska Misao, Beograd, 2004.

M. Vujošević Janičić, J. Kovačević, D. Simić, A. Zečević, PROGRAMIRANJE 1, Zbirka zadataka, Matematički fakultet Beograd, 2017.

N. Mitić, S. Malkov, V. Nikić, Osnovi programiranja, Zbirka zadataka, Matematički fakultet Beograd, 2000.

Watt, David A. Programming language design concepts. John Wiley & Sons, 2004.

S.P. Harbison and G.L. Jr. Steele. C: A Reference Manual. Prentice-Hall, 4th edition, 1995.

D.E. Knuth. The Art of Computer Programming, volume 1: Fundamental Algorithms. Addison-Wesley, 3rd edition, 1998.

S Summit. C Programming FAQs: Frequently Asked Questions. Addison-Wesley, 1995.

Banahan, Mike, Declan Brady, and Mark Doran. The C book. New York: Addison-Wesley, 1988.

Elektronska verzija