Programski jezik C

predavanja

ak. g. 2003/04.

M. Jurak

Sadržaj

1 Uvod **8**
- 1.1 Programski jezici . . . 8
- 1.2 Osnove pisanja programa u Unix okruženju . . . 10
  - 1.2.1 Editor teksta . . . 10
  - 1.2.2 Compiler . . . 10
  - 1.2.3 man . . . 11
  - 1.2.4 Debugger . . . 11
  - 1.2.5 Linker . . . 12
  - 1.2.6 make . . . 12
  - 1.2.7 Programske biblioteke . . . 12

2 Osnove programskog jezika C **13**
- 2.1 Prvi program . . . 13
- 2.2 Varijable. `while` petlja . . . 16
- 2.3 `for` petlja . . . 20
- 2.4 `if` naredba . . . 22
- 2.5 Funkcije . . . 26
  - 2.5.1 Funkcija `main` . . . 30
  - 2.5.2 Prototip funkcije . . . 30
- 2.6 Polja . . . 31
  - 2.6.1 `<math.h>` . . . 36
- 2.7 Pokazivači . . . 37
- 2.8 Polja znakova . . . 41

3 Konstante i varijable **46**
- 3.1 Znakovi . . . 46
- 3.2 Komentari . . . 46
- 3.3 Identifikatori i ključne riječi . . . 47
- 3.4 Osnovni tipovi podataka . . . 49
  - 3.4.1 Cjelobrojni podaci . . . 49
  - 3.4.2 Znakovni podaci . . . 52

3.4.3 Logički podaci 53
3.4.4 Realni podaci 53
3.4.5 Kompleksni podaci 55
3.5 Varijable i deklaracije 55
3.5.1 Polja 56
3.5.2 Pokazivači 57
3.6 Konstante 59
3.7 Inicijalizacija varijabli 63
3.8 Enumeracije 64

4 Operatori i izrazi 66
4.1 Aritmetički operatori 66
4.1.1 Konverzije 67
4.1.2 Prioriteti i asocijativnost 68
4.2 Unarni operatori 69
4.2.1 Inkrementiranje, dekrementiranje 70
4.2.2 `sizeof` operator 71
4.3 Relacijski i logički operatori 72
4.4 Operatori pridruživanja 74
4.5 Uvjetni operator `:` `?` 75
4.6 Prioriteti i redoslijed izračunavanja 76

5 Ulaz i izlaz podataka 78
5.1 Funkcije `getchar` i `putchar` 78
5.2 Funkcije iz datoteke `<ctype.h>` 80
5.3 Funkcije `gets` i `puts` 81
5.4 Funkcija `scanf` 82
5.4.1 Učitavanje cijelih brojeva 83
5.4.2 Učitavanje realnih brojeva 85
5.4.3 Drugi znakovi u kontrolnom nizu 85
5.4.4 Učitavanje znakovnih nizova 86
5.4.5 Prefiks `*` 87
5.4.6 Maksimalna širina polja 88
5.4.7 Povratna vrijednost 89
5.5 Funkcija `printf` 90
5.5.1 Ispis cijelih brojeva 91
5.5.2 Ispis realnih brojeva 92
5.5.3 Širina i preciznost 93
5.5.4 Ispis znakovnih nizova 94
5.5.5 Zastavice 95

6 Kontrola toka programa **96**
6.1 Izrazi i naredbe 96
6.1.1 Izrazi 96
6.1.2 Naredbe 97
6.1.3 Složene naredbe 98
6.1.4 Popratne pojave i sekvencijske točke 98
6.2 Naredbe uvjetnog grananja 100
6.2.1 `if` naredba 100
6.2.2 `if-else` naredba 101
6.2.3 Primjer: Djelitelji broja 102
6.2.4 Višestruka `if-else` naredba 104
6.2.5 Primjer. Višestruki izbor 106
6.2.6 Sparivanje `if` i `else` dijela 106
6.3 `switch` naredba 107
6.4 `while` petlja 110
6.5 `for` petlja 111
6.5.1 Operator zarez 113
6.5.2 Datoteka zaglavlja `<stdlib.h>` 113
6.6 `do - while` petlja 114
6.7 Naredbe `break` i `continue` 115
6.8 `goto` naredba 116

7 Funkcije **119**
7.1 Definicija funkcije 119
7.2 Deklaracija funkcije 123
7.3 Prijenos argumenata 126
7.4 Inline funkcije 129
7.5 Rekurzivne funkcije 130
7.6 Funkcije s varijabilnim brojem argumenata 132

8 Preprocesorske naredbe **134**
8.1 Naredba `#include` 134
8.2 Naredba `#define` 135
8.3 Parametrizirana `#define` naredba 136
8.4 Uvjetno uključivanje 138
8.5 Predefinirani makroi 141
8.6 `assert` 142

9 Struktura programa **144**
9.1 Doseg varijable 144
9.1.1 Lokalne varijable 144

9.1.2 Globalne varijable 145
9.1.3 Argumenti funkcijskog prototipa 148
9.1.4 Lokalne varijable i standard C99 148
9.1.5 Funkcijski doseg 149
9.2 Vijek trajanja varijable 149
9.2.1 Automatske varijable 149
9.2.2 Identifikatori memorijske klase 150
9.2.3 `auto` 150
9.2.4 `register` 150
9.2.5 Statičke varijable 151
9.2.6 Statičke lokalne varijable 151
9.3 Vanjski simboli 152
9.3.1 Funkcije 153
9.3.2 Globalne varijable 155
9.3.3 Vanjska imena 156

10 Polja 157
10.1 Definicija i inicijalizacija polja 157
10.2 Polja znakova 160
10.3 Funkcije za rad sa stringovima 161
10.4 `sscanf(), sprintf()` 164
10.5 Polje kao argument funkcije 165
10.6 Višedimenzionalna polja 166
10.7 Polja varijabilne duljine 170

11 Pokazivači 173
11.1 Deklaracija pokazivača 173
11.2 Pokazivači i funkcije 175
11.3 Operacije nad pokazivačima 177
11.3.1 Povečavanje i smanjivanje 177
11.3.2 Pokazivači i cijeli brojevi 179
11.3.3 Uspoređivanje pokazivača 179
11.3.4 Oduzimanje pokazivača 179
11.3.5 Primjer 180
11.3.6 Generički pokazivač 181
11.4 Pokazivači i jednodimenzionalna polja 183
11.5 Pokazivači i `const` 185
11.6 Polja pokazivača 186
11.7 Pokazivači i višedimenzionalna polja 187
11.7.1 Matrica kao pokazivač na pokazivač 188
11.8 Dinamička alokacija memorije 189

11.9 Pokazivač na funkciju 191
11.10 Argumenti komandne linije 192
11.11 Složene deklaracije 194

12 Strukture 196
12.1 Deklaracija strukture 196
12.1.1 Varijable tipa strukture 196
12.1.2 Inicijalizacija strukture 197
12.1.3 Struktura unutar strukture 198
12.2 Rad sa strukturama 198
12.2.1 Operator točka 198
12.2.2 Struktura i funkcije 200
12.3 Strukture i pokazivači 200
12.3.1 Operator strelica (->) 200
12.3.2 Složeni izrazi 202
12.4 Samoreferentne strukture 203
12.4.1 Jednostruko povezana lista 203
12.5 typedef 209
12.6 Unija 211

13 Datoteke 213
13.1 Vrste datoteka 213
13.1.1 Tekstualne i binarne datoteke 214
13.1.2 Razine ulaza/izlaza 214
13.1.3 Standardne datoteke 215
13.2 Otvaranje i zatvaranje datoteke 216
13.2.1 Standardne datoteke 217
13.3 Funkcije za čitanje i pisanje 218
13.3.1 Čitanje i pisanje znak po znak 218
13.3.2 Čitanje i pisanje liniju po liniju 220
13.3.3 Prepoznavanje greške 221
13.3.4 Formatirani ulaz/izlaz 222
13.3.5 Binarni ulaz/izlaz 222
13.3.6 Direktan pristup podacima 224

14 Operacije nad bitovima 227
14.1 Operatori 227
14.2 Polja bitova 231

A **234**
A.1 ANSI zaglavlja 234
A.2 ASCII znakovi 235
A.3 Matematičke funkcije 236

Poglavlje 1

Uvod

1.1 Programski jezici

Mikroprocesor i drugi logički sklopovi računala imaju svoj vlastiti programski jezik koji se naziva strojni jezik, a sastoji se od nizova binarnih riječi koje predstavljaju instrukcije logičkim sklopovima i podatke koje treba obraditi. Program napisan u strojnom jeziku nazivamo izvršni program ili izvršni kôd budući da ga računalo može neposredno izvršiti. Strojni jezik je određen arhitekturom računala, a definira ga proizvođač hardwarea. Izvršni program je strojno zavisan, što znači da se kôd napisan na jednom računalu može izvršavati jedino na računalima istog tipa.

Pisanje instrukcija u binarnom kôdu posve je nepraktično pa su razvijeni simbolički jezici u kojima su binarne instrukcije zamijenjene mnemoničkim oznakama. Programer unosi program napisan u mnemoničkim oznakama u tekstualnu datoteku pomoću editora teksta i zatim poziva program koji mnemoničke oznake prevodi u binarne instrukcije strojnog jezika. Program koji vrši konverziju naziva se asembler (eng. *assembler*) a sam se programski jezik naziva asemblerski jezik ili jednostavno asembler. Program napisan u asemblerskom jeziku nazivamo izvorni program (eng. *source code*). Pisanje programa time postaje dvostepeni proces koji čine pisanje izvornog programa i prevođenje izvornog programa u izvršni program. Programer se tako oslobađa mukotrponog pisanja binarnih instrukcija te se dobiva do određene mjere strojna neovisnost izvornog programa.

Sljedeći primjer pokazuje dio izvršnog kôda (strojni jezik) i njemu ekvivalentan izvorni, asemblerski kôd. Rad se o asembleru za Intelov mikroprocesor 8086 ([5]).

Strojni jezik	Ekvivalentan	asemblerski kôd
00011110	PUSH	DS
00101011	SUB	AX,AX
11000000		
10111000	PUSH	AX
10111000	MOV	AX,MYDATA
00000001		
10001110		
11011000	MOV	DS,AX

Pisanje programa u asemblerskom jeziku daje programeru potpunu kontrolu nad svim komponentama računala. Programer stoga mora poznavati arhitekturu računala za koje piše program te kontrolirati sve operacije koje računalo izvršava. Programiranje i najjednostavnijih zadaća rezultira velikim i složenim programima pa se programiranje u asembleru koristi se samo za specifične zadaće vezane uz manipulacije s hardwareom. Izvorni program napisan u asembleru nije prenosiv između računala različite arhitekture. Zbog svih tih razloga za većinu programerskih zadaća koriste se viši programski jezici.

Viši programski jezici (C, Pascal, FORTRAN, C++, Java, Perl, Python, ...) razvijeni su kako bi se prevladali nedostaci asemblerskog jezika. Oni oslobađaju programera potrebe poznavanja arhitekture računala, omogućavaju prenosivost programa između računala različitih arhitektura te brže i jednostavnije programiranje. Programi napisani u višem programskom jeziku moraju prije izvođenja proći postupak prevođenja u izvršni kôd što je zadatak prevodioca (eng. *compiler*) za dani jezik. Naredbe višeg programskog jezika prilagođene su tipu podataka s kojima programski jezik manipulira i operacijama koje nad podacima treba vršiti. To je osnovna razlika naspram asemblera koji je prilagođen načinu funkcioniranja mikroprocesora i drugih logičkih sklopova. Zadatak je prevodioca da program napisan u višem programskom jeziku prevede u kôd koji se može izvršavati na zadanom računalu. Na taj se način program napisan u nekom višem programskom jeziku može izvršavati na svakom računalu koje ima prevodilac za taj jezik.

Programski jezik C viši je programski jezik opće namjene. Razvio ga je *Dennis Ritchie* sedamdestih godina prošlog stoljeća u *Bell Telephone Laboratories, Inc.* Opis jezika dan je u knjizi Brian W. Kernighan, Dennis M. Ritchie: The C Programming Language, Prentice-Hall, 1978. Tijekom sedamdesetih i osamdesetih godina jezik se brzo širio te je *American National Standard Institute* (ANSI) pristupio njegovoj standardizaciji koja je dovršena 1989. godine. Novi standard uveo je značajne izmjene u jezik koji se stoga, za

razliku od prvotne verzije, često naziva ANSI-C. Novi je standard opisan u knjizi Brian W. Kernighan, Dennis M. Ritchie: The C Programming Language, 2nd ed., Prentice-Hall, 1988. Danas gotovo svi moderni C-prevodioci implementiraju ANSI-C verziju jezika. ANSI standard usvojila je i Međunarodna organizacija za standarde (*International Organisation for Standardization*) 1990. godine (ISO C). Konačan ANSI/ISO standard ćemo jednostavno nazivati C90 standard. Godine 1999. ISO je prihvatio novi C standard koji uvodi manje dopune u C90 standard, a koji ćemo zvati C99 standard.

1.2 Osnove pisanja programa u Unix okruženju

Postupak programiranja u programskom jeziku C, pod operacijskim sustavima UNIX i LINUX, sastoji se od tri koraka: prvo se programski kôd smjesti u tekstualnu datoteku koja ima ekstenziju `.c` (npr. `prvi.c`). Zatim se napisani program transformira u izvršni kôd pozivom programa prevodioca. Kao rezultat dobiva se izvršna datoteka čijim se pozivom izvršava napisani program. Budući da se u pisanju programa redovito javljaju greške, treći korak u postupku programiranja je nalaženje i ispravljanje grešaka.

1.2.1 Editor teksta

Da bismo program upisali u tekstualnu datoteku potreban nam je editor teksta. Svaki UNIX sustav ima editor koji se naziva `vi` (*visual editor*) i koji se može koristiti i bez grafičkog sučelja. Pored `vi` editora moguće je koristiti svaki drugi editor koji ne umeće znakove formatiranja u datoteku.

1.2.2 Compiler

Nakon što je program napisan i pohranjen u datoteku, recimo da se zove `prvi.c`, potrebno je pozvati C-prevodilac. Pod UNIX-om njegovo je ime najčešće cc (ili gcc ako se radi o GNU prevodiocu). Tako bismo na komandnoj liniji otipkali

```
cc prvi.c
```

i ukoliko je program napisan bez greške prevodilac će kreirati izvršni program i dati mu ime `a.out`; u slučaju neispravnosti programa prevođenje će biti zaustavljeno i prevodilac će dati poruke o pronađenim greškama. Nakon što je izvršni program kreiran pod UNIX-om je dovoljno otipkati

```
./a.out
```

i program će biti izvršen.

1.2.3 man

Prevodilac cc ima brojne opcije koje su opisane u njegovoj man stranici (man je skraćeno od manual). Dovoljno je na komandnoj liniji utipkati

```
man cc
```

i opis komande `cc` bit će izlistan. Na isti način možemo dobiti opis bilo koje naredbe operacijskog sustava te niza programa i alata. Dovoljno je otipkati

```
man naslov
```

gdje je `naslov` naredba ili program koji nas zanima. Jednako tako man će nam dati informacije o funkcijama iz standardne biblioteka jezika C. Na primjer, otipkajte

```
man scanf
```

i dobit ćete opis funkcije za učitavanje podataka `scanf`.

1.2.4 Debugger

Ukoliko program upisan u datoteku `prvi.c` nije ispravno napisan to se može manifestirati na tri načina. Prvi, najpovoljniji, je onaj u kome prevodilac javi poruke o greškama i zaustavi proces prevođenja u izvršni kôd. Druga mogućnost je da proces prevođenja bude uspješan i da `cc` kreira izvršnu datoteku `a.out`, no izvršavanje programa završava s porukom operacijskog sustava o neispravnom izvršnom programu. Treća mogućnost, često najnepovoljnija, je ona u kojoj je izvršni program ispravan ali ne radi ono što je programer zamislio. U svakom slučaju, kod pojave grešaka potrebno je greške pronaći, korigirati i ponoviti postupak prevođenja u izvršni kôd. Budući da se greške redovito javljaju, programiranje je iterativan proces u kojem je ispravljanje grešaka treći (i najvažniji) korak.

Prilikom pronalaženja grešaka mogu nam pomoći dva alata. Jedan je lint (ukucajte `man lint` na komandnoj liniji) koji provjerava sintaksu napisanog programa i daje detaljniju dijagnostiku od poruka o greškama koje daje sam prevodilac. Drugi alat je debugger (`dbx` ili `gdb`), program pomoću kojega naš kôd možemo izvršavati liniju po liniju, pratiti vrijednosti varijabli, zaustavljati ga na pojedinim mjestima itd. Debugger je najefikasnije sredstvo za otkrivanje razloga neispravnog funkcioniranja kôda.

1.2.5 Linker

Složeniji C programi sastoje se od više datoteka s ekstenzijama `.c` i `.h`. Datoteke s ekstenzijom `.c` sadrže dijelove C programa odnosno izvorni kôd (eng. *source code*). Datoteke s ekstenzijom `.h` nazivaju se datoteke zaglavlja (eng. *header files*) i sadrže informacije potrebne prevodiocu za prevođenje izvornog kôda u strojni jezik. To su tekstualne datoteke koje programer kreira zajedno s `.c` datotekama.

Svaku pojedinu `.c` datoteku prevodilac prevodi u strojni jezik no time se još ne dobiva izvršni program jer njega čine tek sve `.c` datoteke zajedno. Prevodilac za svaku `.c` datoteku generira strojni kôd i smješta ga u datoteku istog imena ali s ekstenzijom `.o`. Dobiveni se kôd naziva objektni kôd ili objektni program. Zadatak je linkera[1] kreirati izvršni program povezivanjem svih objektnih programa u jednu cjelinu. Prevodilac će nakon generiranja objektnog kôda automatski pozvati linker tako da njegovo neposredno pozivanje nije potrebno. Ukoliko povezivanje objektnog kôda nije bilo uspješno linker će dati poruke o greškama.

1.2.6 make

Pri prevođenju u izvršni kôd programa koji je razbijen u više datoteka potrebno je svaku pojedinu `.c` datoteku prevesti u objektni kôd, a zatim sve objektne datoteke povezati, eventualno i s vanjskim bibliotekama. Pri tome nam pomaže programski alat koji se naziva `make`. On provjerava datume zadnje izmjene pojedinih datoteka i osigurava da se prevedu u objektni kôd samo one datoteke koje su mijenjane nakon posljednjeg generiranja izvršnog programa.

1.2.7 Programske biblioteke

Svaki složeniji C-program koristi niz funkcija kao što su `printf`, `exit`, `fgets` itd. koje su pohranjene u standardnim bibliotekama funkcija. Potpun popis funkcija dostupnih programeru razlikuje se od računala do računala no jedan broj tih funkcija prisutan je na svakom sustavu koji ima C prevodilac. U ovoj skripti bit će opisan dio najčešće korištenih funkcija iz standardne biblioteke. Potpuniji popis može se naći u [4] ili u priručnicima za C-prevodilac na danom računalu.

[1] Koristi se još i naziv punjač.

Poglavlje 2

Osnove programskog jezika C

U ovom poglavlju dajemo pregled programskog jezika C uvodeći osnovne elemente jezika kroz jednostavne primjere. Svi ovdje uvedeni pojmovi bit će detaljnije analizirani u narednim poglavljima.

2.1 Prvi program

Jedan jednostavan ali potpun C program izgleda ovako:

```
#include <stdio.h>

int main(void)
{
   printf("Dobar dan.\n");
   return 0;
}
```

Program treba spremiti u datoteku s ekstenzijom `.c` kako bi se znalo da se radi o C-programu. Nazovimo stoga datoteku `prvi.c`. Program zatim *kompiliramo* naredbom

```
cc prvi.c
```

nakon čega prevodilac kreira izvršni program i daje mu ime `a.out`. Da bismo izvršili program dovoljno je otipkati

```
./a.out
```

Rezultat izvršavanja programa je ispis poruke

```
Dobar dan.
```

Svi C-programi sastoje se od funkcija i varijabli. Funkcije sadrže instrukcije koje određuju koje će operacije biti izvršene, a varijable služe memoriranju podataka.

Izvršavanje programa počine izvršavanjem funkcije `main` koja mora biti prisutna u svakom programu. Funkcija `main` svoju zadaću obavlja općenito pozivanjem drugih funkcija. Tako se u našem programu poziva funkcija `printf` iz standardne biblioteke, koja je zadužena za ispis podataka na ekranu. Da bismo mogli koristiti funkcije iz standardne biblioteke zadužene za ulaz i izlaz podataka program započinjemo naredbom

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

Njom se od prevodioca traži da uključi (`include`) u program datoteku `stdio.h` koja sadrži informacije nužne za korištenje funkcije `printf` i mnogih drugih. Datoteke s ekstenzijom `.h` nazivaju se datoteke zaglavlja (eng. *header files*) i njihovo stavljanje u oštre zagrade `< >` informira prevodilac da se radi o standardnim datotekama zaglavlja, koje se nalaze na unaprijed određenim mjestima u datotečnom sustavu.

Sljedeća linija predstavlja deklaraciju funkcije `main`:

```
int main(void)
```

Funkcija može uzimati jedan ili više argumenata i obično vraća neku vrijednost. Deklaracijom se uvodi ime funkcije, broj i tip argumenata koje uzima i tip povratne vrijednosti. U našem primjeru ime funkcije je `main` i ona ne uzima niti jedan argument. To je deklarirano tako što je u oble zagrade stavljena ključna riječ `void` (*eng. void=prazan*). Povratna vrijednost je tipa `int`, što je oznaka za cijeli broj.[1]

Iza oblih zagrada dolaze vitičaste zagrade koje omeđuju tijelo funkcije. Tijelo funkcije je sastavljeno od deklaracija varijabli i izvršnih naredbi. Sve deklaracije varijabli dolaze prije prve izvršne naredbe. U našem primjeru nemamo deklaracija varijabli. Tijelo sadrži samo dvije naredbe: poziv funkcije `printf` i `return` naredbu:

```
{
   printf("Dobar dan.\n");
   return 0;
}
```

Svaka naredba završava znakom točka-zarez (;). To omogućava da se više naredbi stavi na istu liniju. Mogli smo, na primjer, pisati

[1] `int`=integer.

```
#include <stdio.h>

int main(void){
   printf("Dobar dan.\n"); return 0;
}
```

Funkcija se poziva tako da se navede njezino ime iza koga idu oble zagrade s listom argumenata koji se funkciji predaju. Funkcija `printf` dobiva samo jedan argument: to je niz znakova `"Dobar dan.\n"`. Navodnici služe za ograničavanje konstantnih znakovnih nizova. Sve što se nalazi između " i " predstavlja niz znakova koji čini jednu cjelinu. Pored običnih znakova između navodnika mogu se naći i specijalni znakovi koji počinju znakom \ (eng. *backslash*). Tako `\n` označava prijelaz u novi red. Funkcija `printf` nakon završenog ispisa ne prelazi automatski u novi red nego je potrebno ubaciti znak `\n` tamo gdje takav prijelaz želimo. Program smo stoga mogli napisati u obliku

```
#include <stdio.h>

int main(void)
{
   printf("Dobar ");
   printf("dan.");
   printf("\n");
   return 0;
}
```

i dobili bismo posve isti ispis.

Izvršavanje funkcije završava naredbom `return`. Kad funkcija vraća neku vrijednost u pozivni program ta se vrijednost navodi u `return` naredbi. Funkcija `main` vraća nulu pa je stoga zadnja naredba `return 0;`

Funkcija `main` mora biti prisutna u svakom programu. Kada na komandnoj liniji otipkamo

```
./a.out
```

operacijski sustav poziva funkciju `main`. Zatim se redom izvršavaju naredbe unutar funkcije `main` sve dok se ne dođe do `return` naredbe. Ona operacijskom sustavu vraća cjelobrojnu vrijednost koja ima značenje izlaznog statusa. Nula se interpretira kao uspješni završetak programa, a svaka druga vrijednost signalizira završetak uslijed greške.

2.2 Varijable. while petlja

Napišimo program koji ispisuje prvih deset faktorijela. Brojeve 1!, 2!, 3!,...,10! želimo ispisati u obliku tablice[2]

1	1
2	2
3	6
4	24
...	...

```
#include <stdio.h>

/*  Program koji ispisuje
    prvih 10 faktorijela.  */

int main(void)
{
    int n,fakt;

    n=1;
    fakt=1;
    while(n<=10)
    {
        fakt=fakt*n;
        printf(" %d  %d\n",n,fakt);
        n=n+1;
    }
    return 0;
}
```

Prva linija programa uključuje u program standardnu datoteku zaglavlja `<stdio.h>` koja nam je nužna jer koristimo funkciju `printf`.

Dvije linije

```
/*  Program koji ispisuje
    prvih 10 faktorijela.  */
```

predstavljaju komentar unutar programa. Prevodilac ignorira dio programa između znakova `/*` i `*/`. Tekst koji se nalazi između tih znakova ima ulogu komentara koji treba olakšati razumijevanje programa.

Prva linija unutar funkcije `main`

[2] $n!$ je broj definiran formulom $n! = 1 \cdot 2 \cdot 3 \cdots n$

```
int n,fakt;
```

predstavlja deklaraciju varijabli `n` i `fakt`.

U C-u je potrebno deklarirati sve varijable koje se u programu koriste. Preciznije, unutar svake funkcije (ovdje unutar funkcije `main`) sve varijable koje funkcija koristi treba deklarirati prije prve izvršne naredbe.

Deklaracija se sastoji od tipa varijable koji slijede imena varijabli odvojena zarezima. Deklaracija kao i svaka druga naredba u programu završava znakom točka-zarez. U deklaraciji

```
int n,fakt;
```

imena varijabli koje se deklariraju su `n` i `fakt`, a njihov tip je `int`. Tip `int` predstavlja cjelobrojnu varijablu koja može biti pozitivna i negativna. Granice u kojima se može kretati varijabla tipa `int` ovise o računalu i danas je to najčešće od -214748347 do 214748348. Pored varijabli tipa `int` postoje još dvije vrste cjelobrojnih varijabli. To su varijable tipa `short` i varijable tipa `long`. Razlika je u tome da `short` pokriva manji raspon cijelih brojeva i zauzima manje memorijskog prostora, dok `long` pokriva veći raspon i treba više memorijskog prostora. Varijable realnog tipa (tzv. brojevi s pokretnim zarezom) pojavljuju se u dvije forme: brojevi jednostruke preciznosti `float` i brojevi dvostruke preciznosti `double`. Konačno, postoje i varijable tipa `char` u kojima se pamte znakovi.

Tipovi varijabli:

`char`	jedan znak,
`short`	"kratki" cijeli broj,
`int`	cijeli broj,
`long`	"dugi" cijeli broj,
`float`	realan broj jednostruke preciznosti,
`double`	realan broj dvostruke preciznosti.

Pored ovih tipova podataka koje nazivamo osnovnim, postoje još i složeni tipovi podataka kao što su strukture, polja, unije te pokazivači kao posebni osnovni tip podatka.

Prva izvršna naredba u programu je naredba pridruživanja

```
n=1;
```

Znak jednakosti je operator pridruživanja. On vrijednost konstante ili varijable na desnoj strani pridružuje varijabli na lijevoj strani. Nakon te naredbe vrijednost varijable `n` postaje jednaka 1.

Naredba

```
while(n<=10)
{
   ....
}
```

je `while` petlja. Ona se sastoji od ključne riječi `while`, testa `n<=10` i tijela petlje; tijelo petlje je skup naredbi koje se nalaze unutar vitišastih zagrada. `while`-petlja funkcionira na sljedeći način: prvo se izračunava izraz u zagradama: `n<=10`. Ukoliko je je on istinit (`n` je manji ili jednak 10) onda se izvršava tijelo petlje (sve naredbe unutar vitičastih zagrada). Zatim se ponovo testira izraz `n<=10` i izvršava tijelo petlje ako je izraz istinit. Izvršavanje petlje prekida se kada izraz `n<=10` postane neistinit. Program se tada nastavlja prvom naredbom koja slijedi iza tijela `while` petlje.

Tijelo petlje može se sastojati od jedne ili više naredbi. Ako se sastoji od samo jedne naredbe nije potrebno koristiti vitičaste zarade, iako nihova upotreba nije pogrešna. Kada se tijelo sastoji od više naredbi one moraju biti omeđene vitičastim zagradama.

Manje ili jednako (`<=`) je jedan od relacijskih operatora. Ostali su:

operator	primjer	značenje
`<=`	`a <= b`	manje ili jednako,
`>=`	`a >= b`	veće ili jednako,
`<`	`a < b`	strogo manje,
`>`	`a > b`	strogo veće,
`==`	`a == b`	jednako,
`!=`	`a != b`	nejednako.

Važno je razlikovati operator pridruživanja (`=`) i relacijski operator jednakosti (`==`).

U tijeli `while` petlje imamo tri naredbe:

```
fakt=fakt*n;
printf(" %d  %d\n",n,fakt);
n=n+1;
```

U prvoj i trećoj naredbi pojavljuju se operacije množenja i zbrajanja. Osnovne aritmetičke operacije dane su sljedećoj tabeli:

	operator	primjer
zbrajanje	`+`	`a+b`
oduzimanje	`-`	`a-b`
množenje	`*`	`a*b`
dijeljenje	`/`	`a/b`

U naredbi `fakt=fakt*n` varijable `fakt` i `n` su pomnožene i rezultat je pridružen varijabli `fakt`. Varijabla `fakt` pri tome mjenja vrijednost, dok varijabla `n` ostaje nepromijenjena. U naredbi `n=n+1` varijabli `n` povećavamo vrijednost za 1. Operator pridruživanje (`=`) djeluje tako da se prvo izračuna izraz na desnoj strani (`n+1`) i zatim se ta vrijednost pridruži varijabli na lijevoj strani (varijabla `n`). Ukupan efekt je povećanje vrijednosti varijable `n` za 1.

Operacija dijeljenja s cjelobrojnim operandima daje cjelobrojni rezultat. To se radi tako da se rezultatu dijeljenja *odsijeku* decimale kako bi se dobio cijeli broj (npr., `2/3 = 0`). Ako je bar jedan operand realan broj dijeljenje predstavlja uobičajeno dijeljenje realnih brojeva. Na primjer,

```
6/4     == 1
6.0/4.0 == 1.5
6/4.0   == 1.5
6.0/4   == 1.5
```

Ako konstanta nema decimalnu točku, onda je cjelobrojna (tipa `int`), inače je tipa `double`.

Naredba

```
printf(" %d  %d\n",n,fakt);
```

je poziv funkcije `printf`. Ona ispisuje brojeve `n` i `fakt` na ekranu. Funkcija je pozvana s tri argumenta. Prvi je konstantan niz znakova `" %d %d\n"`, a drugi i treći su varijable `n` i `fakt`. Prvi se argument naziva *kontrolni znakovni niz* i on sadrži *znakove konverzije* koji se sastoje od jednog slova kome prethodi znak `%`. Njihova uloga je sljedeća: Prvi znak konverzije `%d` označava da na mjestu na kome se on nalazi treba ispisati prvi argument koji slijedi nakon kontrolnog znakovnog niza. Slovo `d` osnačava da ga treba ispisati kao cijeli decimalni broj. Drugi znak `%d` označava da na tom mjestu treba ispisati drugi argument koji slijedi nakon kontrolnog znakovnog niza, kao cijeli decimalni broj.

Broj argumenata koji se daju `printf` funkciji može biti proizvoljan. Želimo li ispisati realan broj (`float` ili `double`) trebamo koristiti `%f` umjesto `%d`. Svi ostali znakovi u kontrolnom znakovnom nizu bit će ispisani onako kako su uneseni. U ovom primjeru imamo dvije bjeline i znak za prijelaz u novi red. U prvom programu je funkcija `printf` bila pozvana samo s jednim argumentom, kontrolnim znakovnim nizom. Pri takvom pozivu on ne sadrži znakove konverzije već samo tekst koji treba ispisati.

2.3 `for` petlja

Treba napisati program koji za proizvoljni prirodni broj `n` računa sumu

$$\sum_{k=1}^{n} \frac{1}{k(k+1)}$$

i ispisuje rezultat.

```
#include <stdio.h>

int main(void)
{
    int n,i;
    double suma;

    printf("Unesite broj n: n= ");
    scanf(" %d",&n);
    suma=0.0;

    for(i=1;i<=n;i=i+1)
    {
        suma=suma+1.0/(i*(i+1));
    }
    printf("Suma prvih %d clanova = %f\n",n,suma);
    return 0;
}
```

Program započinje deklaracijama varijabli. Varijable `n` i `i` su cjelobrojne, dok je `suma` realna varijabla tipa `double` (dvostruke preciznosi).

Naredba

```
printf("Unesite broj n: n= ");
```

ispisuje znakovni niz `"Unesite broj n:  n= "` (bez prijelaza u novi red). Pozivom funkciji `scanf` čita se cijeli broj sa standardnog ulaza:

```
scanf(" %d",&n);
```

Program u ovom trenutku čeka da korisnik upiše jedan cijeli broj.

Funkcija `scanf` pripada standardnoj biblioteci i deklarirana je u standardnoj datoteci zaglavlja `<stdio.h>`. Po strukturi je slična funkciji `printf`.

Njen prvi argument je niz znakova koji sadrži znakove konverzije. U našem slučaju to je `%d` koji informira `scanf` da mora pročitat jedan broj tipa `int` i smjestiti ga u varijablu koja je sljedeći argument funkcije (varijabla `n`).

Stvarni argument funkcije `scanf` je memorijska adresa varijable `n`, a ne sama varijabla `n`. Adresa varijable dobiva se pomoću adresnog operatora `&` (`&n` je adresa varijable `n`). Stoga svi argumenti funkcije `scanf`, osim prvog, moraju imati znak `&` ispred sebe.

Naredba

```
for(i=1;i<=n;i=i+1)
{
    ....
}
```

je `for` petlja. Ona djeluje na sljedeći način. Prvo se varijabla `i` inicijalizira tako što joj se pridruži vrijednost 1 (`i=1`). Inicijalizacija se izvrši samo jednom. Zatim se testira izraz

```
i<=n;
```

Ako je rezultat testa istinit izvršavaju se naredbe iz tijela petlje (naredbe u vitičastim zagradama). Nakon toga se izvršava naredba

```
i=i+1;
```

i kontrola programa se vraća na testiranje istinitosti izraza `i<=n`. Petlja završava kad izraz koji se testira postane lažan. Program se tada nastavlja prvom naredbom iza tijela petlje.

`for`-petlja iz našeg primjera može se zapisati pomoću `while`-petlje na ovaj način:

```
i=1;
while(i<=n)
{
   suma=suma+1.0/(i*(i+1));
   i=i+1;
}
```

Naredbe oblika `i=i+1` i `i=i-1` kojima se brojač povećava, odnosno smanjuje za jedan sreću se vrlo često pa za njih postoji kraća notacija:

`n=n+1` je ekvivalentno s `n++`,
`n=n-1` je ekvivalentno s `n--`.

Operator ++ naziva se operator inkrementiranja i on povećava vrijednost varijable za 1; Operator -- koji smanjuje vrijednost varijable za 1 naziva se operator dekrementiranja. Ti se operatori redovito koriste u `for` petljama pa bismo gornju petlju pisali u obliku:

```
for(i=1;i<=n;i++)
     suma=suma+1.0/(i*(i+1));
```

Operatore inkrementiranja i dekrementiranja možemo primijeniti na cjelobrojne i realne varijable. Uočimo još da smo vitičaste zagrade oko tijela `for`-petlje mogli ispustiti jer se tijelo sastoji samo od jedne naredbe. To vrijedi za sve vrste petlji.

Svaka petlja sadrži u sebi jedan ili više brojača kojim se kontrolira odvijanje petlje. Brojač je potrebno na početku inicijalizirati i zatim ga u svakom prolazu petlje povećavati (ili smanjivati). Pelja se zaustavlja testiranjem brojača. U `while` petlji jedino testiranje brojača ulazi u zagrade nakon ključne riječi `while`. Inicijalizaciju moramo staviti izvan petlje, a promjenu brojača negdje unutar petlje. U `for` petlji sva tri elementa dolaze u zagradama iza ključne riječi `for`, u obliku

```
for(inicijalizacija brojaca; test; promjena brojaca)
{
   .........
}
```

Inicijalizacija, test i promjena brojača odvajaju se znakom točka-zarez (`;`).

Napomena. U C-u postoji još `do-while` petlja o koju ćemo objasniti u sekciji 6.6.

2.4 `if` naredba

Treba napisati program koji riješava kvadratnu jednadžbu: $ax^2+bx+c=0$. Rješenja su općenito kompleksna i zadana su formulom

$$z_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Koristit ćemo notaciju $z_1 = x_1 + iy_1$, $z_2 = x_2 + iy_2$.

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

```
#include <math.h>

/* Resavanje kvadratne jednadzbe.
      a x^2 + b x + c = 0         */

int main(void) {
  double a, b, c, /* Koeficijenti jednadzbe        */
         d,       /* Diskriminanta                 */
         x1, x2,  /* Realni dijelovi korijena.     */
         y1, y2;  /* Imaginarni dijelovi korijena. */

  printf("Upisite koeficijente  kvadratne jednadzbe: a, b, c: ");
  scanf ("%lf%lf%lf", &a, &b, &c);

  y1=0.0;
  y2=0.0;
  if(a != 0.0) {
     d=b*b-4*a*c;
     if (d > 0) {
        x1=(- b + sqrt (d))/(2 * a);
        x2=(- b - sqrt (d))/(2 * a);
     } else if (d == 0) {
        x1=- b/(2 * a);
        x2=x1;
     } else{
        x1=-b/(2 * a);        x2 = x1;
        y1=sqrt(-d)/(2 * a);  y2 = - y1;
     }
  }
  else {
     printf("Jednadzba nije kvadratna.\n");
     exit(-1);
  }
  printf("z1=%f + i*(%f), z2=%f + i*(%f)\n",x1, y1, x2, y2);
  return 0;
}
```

U programu imamo nekoliko novih elemenata. Prvo, imamo tri `include` naredbe. Datoteku `stdio.h` uključujemo radi funcija `printf` i `scanf`; datoteka `stdlib.h` uključujemo radi funkcije `exit`, a datoteku `math.h` radi funkcije `sqrt` (`sqrt(x)`$=\sqrt{x}$). U datoteci zaglavlja `stdlib.h` deklarirano

je nekoliko funkcija koje se često koriste tako da se ta datoteka uključije vrlo često u programe. Datoteku `math.h` potrebno je uključiti kada program koristi bilo koju matematičku funkciju.

U prvoj naredbi unutar funkcije `main` deklariramo osam varijabli tipa `double: a, b, c, d, x1, x2, y1, y2`. Varijable smo razbili u četiri grupe i svaku smo grupu napisali u zasebnom redu da bismo lakše komentirali njihovo značenje. Takav način pisanja je moguć stoga što prevodilac tretira znak za prijelaz u novi red isto kao i bjelinu. Prijelaz u novi red možemo stoga ubaciti svugdje gdje može doći bjelina.

Naredbama

```
printf("Upisite koeficijente  kvadratne jednadzbe: a, b, c: ");
scanf ("%lf%lf%lf", &a, &b, &c);
```

učitavaju se koeficijenti kvadratne jednadžbe. Uočimo da je znak konverzije kojim se učitava varijabla tipa `double` jednak `%lf`. Ako želimo učitati varijablu tipa `float` (realni broj u jednostrukoj preciznosti) trebamo koristiti znak konverzije `%f`. [3] Nasuprot tome, kod ispisivanja varijabli tipa `float` i `double` funkcija `printf` koristi isti znak konverzije `%f`.

Nakon inicijalizacije varijabli `y1` i `y2` nulom izvršava se `if` naredba:

```
if(a != 0.0) {
   ......
}
else{
   ......
}
```

Naredba `if` je naredba uvjetnog grananja. Općenito ima oblik

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

Ona funkcionira na ovaj način: prvo se testira `uvjet` i ako je zadovoljen izvršava se `naredba1`, a ako nije izvršava se `naredba2`. U našem slučaju uvjet je `a != 0.0` (da li je `a` različito od nule?). Naredbe koje se izvršavaju, `naredba1` i `naredba2`, mogu biti pojedinačne naredbe ili grupe naredbi omeđene vitičastim zagradama, kao u našem primjeru. Dio naredbe `else` ne mora biti prisutan, tako da `if` naredba može imati i oblik

[3] `%f` možemo čitati kao *float*, a `%lf` kao *long float* = *double*.

```
if(uvjet)
  naredba1;
```

pri čemu se `naredba1` izvršava ako je `uvjet` ispunjen, dok se u protivnom prijelazi na sljedeću instrukciju.

Testiranjem uvjeta `a != 0.0` provjeravamo je li jednažba kvadratna ili nije. Ako nije izvršava se `else` dio naredbe uvjetnog grananja u kojem se ispisuje poruka da jednadžba nije kvadratna i poziva se funkcija `exit` s argumentom `-1`. Zadaća funkcija `exit` je zaustaviti izvršavanje programa i predati operacijskom sustavu svoj argument (`-1` u našem primjeru) koji se interpretira kao kod greske. Pod operacijskim sustavima Unix i Linux konvencija je da kod 0 označava ispravan završetak programa dok svaka druga cjelobrojna vrijednost signalizira zaustavljanje usljed greške.

Ako je uvjet `a != 0.0` ispunjen, izvrsava se dio koda

```
d=b*b-4*a*c;
if (d > 0) {
   x1=(- b + sqrt (d))/(2 * a);
   x2=(- b - sqrt (d))/(2 * a);
} else if (d == 0) {
   x1=- b/(2 * a);
   x2=x1;
} else{
   x1=-b/(2 * a);        x2 = x1;
   y1=sqrt(-d)/(2 * a);  y2 = - y1;
}
```

Prvo se računa diskriminanta jednadžbe `d`, a onda se ulazi u jednu višestruku naredbu uvjetnog granjnja u kojoj se ispituje je li diskriminanta, pozitivna, jednaka nuli ili negativna. Proizvoljan broj `if` naredbi možemo ugnijezditi tako da dobijemo višestruku `if` naredbu. Na primjer, pomoću dvije `if` naredbe dobivamo naredbu oblika

```
if(uvjet1){
   naredba1;
} else if (uvjet2){
   naredba2;
} else{
   naredba3;
}

naredba4;
```

U ovoj se konstrukciji prvo testira `uvjet1`. Ako je on ispunjen izvrsava se blok naredbi `naredba1` i izvršavanje programa se nastavlja s prvom naredbom koja slijedi višestruku naredbu uvjetnog granjnja (`naredba4`). Ako `uvjet1` nije ispunjen testira se `uvjet2`. Ako je on ispunjen izvršava se blok naredbi `naredba2` i nakon toga `naredba4`. Ako niti `uvjet2` nije ispunjen izvršava se `else` blok.

U slučaju `d > 0.0` imamo dva realna korijena, dok u slučaju `d == 0.0` imamo jedan dvostruki realni korijen. Konačno, ako je diskriminanta negativna imamo konjugirano kompleksne korijene. Uočimo da smo u tom dijelu koda dvije naredbe pisali u jednoj liniji. C nam dozvoljava pisati bilo koji broj naredbi u istoj liniji, premda zbog preglednosti najčešće pišemo svaku naredbu u svojoj liniji.

Konačno, u zadnjoj naredbi

```
printf("z1=%f + i*(%f), z2=%f + i*(%f)\n",x1, y1, x2, y2);
```

ispisujemo rezultat.

Centralni dio programa čini jedna `if-else` naredba u koju je smještena druga, složena `if` naredba. Pogledajmo još jednom kako se program izvršava, na primjer u situaciji kada je `a != 0` i `d > 0`. Program će ući u `if` dio vanjske `if-else` naredbe, izračunat će `d` i zatim će ući u `if` dio unutarnje složene `if` naredbe. Kada se izračunaju `x1` i `x2` program izlazi iz složene `if` narebe i nastavlja s *prvom sljedećom* naredbom. Kako takve naredbe nema, izvršavanje `if` bloka prve `if-else` naredbe je završeno i program se nastavlja s prvom naredbom iza nje, a to je poziv funkcije `printf`.

U dobro pisanom programu njegova logička struktura mora biti vidljiva iz njegove forme. Posebnu pažnju treba posvetiti ugnježdenim naredbama uvjetnog granjanja te petljama. Kod `if-else` naredbe potrebno je uvijek uvući tijelo `if` dijela i tijelo `else` dijela da se naglasi logička struktura. Posve isto se postupa i sa petljama.

Napomena. Kod kompilacije programa koji koristi matematičke funkcije moramo prevodiocu dati `-lm` opciju. Na primjer, ako je program spremljen u datoteku `kvadratna_jed.c`, onda ga komiliramo linijom

```
cc kvadratna_jed.c -lm
```

□

Napomena. C ima i `switch` naredbu koja također služi uvjetnom granjanju (vidi sekciju 6.3). □

2.5 Funkcije

Treba napisati funkciju koja računa binomni koeficijent

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdots (n-k+1)}{1 \cdot 2 \cdots k}. \tag{2.1}$$

Funkcija treba uzeti dva cjelobrojna parametra `n` i `k` te vratiti broj $\binom{n}{k}$. U glavnom programu treba zatim ispisati tzv. Pascalov trokut prikazan niže. U liniji s indeksom `n` ispisani su brojevi $\binom{n}{k}$, za sve vrijednosti `k` od 0 do `n`.

```
 n = 0                     1
 n = 1                   1   1
 n = 2                 1   2   1
 n = 3               1   3   3   1
 n = 4             1   4   6   4   1
 n = 5           1   5   10  10   5   1
 n = 6         1   6   15  20  15   6   1
 n = 7       1   7   21  35  35   21  7   1
 n = 8     1   8   28  56  70  56   28  8   1
 n = 9   1   9   36  84  126 126 84   36  9   1
```

Radi jednostavnosti ispisat ćemo Pacalov trokut poravnan na lijevoj strani.

```
#include <stdio.h>

/* Funkcija binom() racuna binomni koeficijent. */

long binom(int n, int k)
{
   long rezult=1;
   int    i;

   if(n == k) return 1;
   if(k == 0) return 1;

   for(i=n;i>n-k;i--)
      rezult=rezult*i;

   for(i=1;i<=k;++i)
      rezult=rezult/i;

   return rezult;
}
```

```
int main(void)
{
   long bnm;
   int n, k;

   for(n=0;n<10;n++){
      for(k=0;k<=n;k++){
         bnm=binom(n,k);
         printf("%ld ",bnm);
      }
      printf("\n");
   }
   return 0;
}
```

Funkcija `binom` definirana je ispred funkcije `main`. Iz definicije se vidi da uzima dva argumenta tipa `int` i vraća rezultat tipa `long`.

Definicija funkcije općenito ima oblik

```
tip ime_funkcije(tip_1 arg_1, tip_2 arg_2, ...)
{
     deklaracije varijabli
     narebe
}
```

gdje je `tip` tip podatka koji funkcija vraća kao rezultat svog djelovanja. Unutar zagrada nalaze se deklaracije argumenata funkcije: `tip_1 arg_1, tip_2 arg_2, ...`. Za svaki argument navodi se njegov tip i ime. Tijelo funkcije je omeđeno vitičastim zagradama i unutar njega prvo dolaze deklaracije varijabli, a zatim izvršne naredbe.

U dijelu koda

```
long binom(int n, int k)
{
   long rezult=1;
   int    i;

   if(n == k) return 1;
   if(k == 0) return 1;

   for(i=n;i>n-k;i--)
```

```
      rezult=rezult*i;

   for(i=2;i<=k;++i)
      rezult=rezult/i;

   return rezult;
}
```

definirana je funkcija `binom`. Njezini se argumenti zovu `n` i `k` i tipa su `int`. Funkcija vraća vrijednost tipa `long`. Unutar funkcije deklarirane su dvije varijable: `rezult` te `i`. To su tzv. lokalne varijable koje postoje samo za vrijeme izvršavanja funkcije. U deklaraciji

```
   long rezult=1;
```

varijabla `rezult` je inicijalizirana s 1. Na taj način, pri deklaraciji mogu se inicijalizirati i ostali tipovi varijabli.

Svaka funkcija koja vraća neku vrijednost mora sadržavati barem jednu `return` naredbu. Nailaskom na `return` zaustavlja se izvođenje funkcije i u pozivni program se vraća ona vrijednost koja stoji uz `return`. U našem primjeru ako je `n == k` funkcija se zaustavlja i vraća 1, jer je $\binom{n}{n} = 1$ za svako n. Isto tako, ako je `k == 0` ponovo se vraća 1, jer vrijedi $\binom{n}{0} = 1$ za svako n. U tim smo naredbama koristili `if` naredbe bez pripadnog `else` dijela.

Ukoliko prethodna dva uvjeta nisu zadovoljena, izvršavanje funkcije će doći do `for` petlje

```
   for(i=n;i>n-k;i--)
      rezult=rezult*i;
```

Ona računa brojnik `rezult`$=n(n-1)\cdots(n-k+1)$ u formuli (2.1). Uočimo da je brojač `i` inicijaliziran s `n` i da se u svakom prolasku kroz petlju smanjuje za jedan dok ne postane jednak `n-k`. Tad se izlazi iz petlje. Kako unutar petlje imamo samo jednu naredbu, nismo koristili vitičaste zagrade. U sljedećoj `for` petlji brojnik dijelimo brojevima 2,3,..., k. Time dolazimo do izraza koji smo trebali izračunati.

U glavnom programu (tj. u funkciji `main`)

```
int main(void)
{
   long bnm;
   int n, k;
```

```
   for(n=0;n<10;n++){
      for(k=0;k<=n;k++){
         bnm=binom(n,k);
         printf("%ld ",bnm);
      }
      printf("\n");
   }
   return 0;
}
```

deklarirane su varijable `bnm` (tipa `long`), `n` i `k` (tipa `int`). Funkcija `binom` se poziva unutar dvostruke `for` petlje u naredbi

```
         bnm=binom(n,k);
```

Varijable `n` i `k` deklarirane u `main` su stvarni argumenti funkcije `binom`. Argumenti koji su deklarirani u definiciji funkcije `binom` nazivaju se formalni argumenti. Njima smo dali ista imena `n, k` kao i stvarnim argumentima, ali to nije nužno budući da se radi o posve različitim varijablama. Varijable `n` i `k` deklarirane u funkciji `main` postoje za vrijeme izvršavanja funkcije `main` dok varijable `n` i `k` deklarirane u funkciji `binom` postoje samo za vrijeme izvršavanja funkcije `binom` i nestaju nakon izlaska iz funkcije `binom`. Prilikom poziva funkcije vrijednosti stvarnih argumenata se kopiraju u formalne argumente. Nakon što funkcija `binom` izvrši `return` naredbu, vrijednost koja stoji uz `return` se kopira u varijablu `bnm` koja stoji na lijevoj strani jednakosti. Program se zatim nastavlja ispisivanjem izračunate vrijednosti `bnm`. Kako se radi o varijabli tipa `long`, za ispis u `printf` naredbi koristimo znak konverzije `%ld`.

2.5.1 Funkcija `main`

Svaki program mora sadržavati funkciju `main` budući da izvršavanje programa započinje prvom izvršnom naredbom u `main`-u. Nju poziva operacijski sustav kad korisnik pokrene program. Nakon završetka `main` vraća cjelobrojnu vrijednost operacijskom sustavu koja ima značenje izlaznog statusa. Nula se interpretira kao uspješni završetak programa, a svaka vrijednost različita od nule signalizira završetak uslijed greške.

2.5.2 Prototip funkcije

Funkcija mora biti deklarirana prije mjesta na kome se poziva kako bi prevodilac znao broj i tip argumenata funkcije i tip rezultata koji vraća. Stoga

je prirodno definiciju funkcije staviti prije funkcije `main`, kao što smo učinili u prethodnom primjeru. Ako u programu imamo puno funkcija takav način pisanja može biti nepregledan jer funkcija `main` dolazi na kraju datoteke. C nam stoga dozvoljava da definiciju funkcije stavimo iza funkcije `main` (odn. iza mjesta na kome se poziva) ako prije funkcije `main` postavimo prototip funkcije.

Prototip funkcije dobiva se tako da se u definiciji funkcije ispusti čitavo tijelo funkcije. Prototip kao i sve druge naredbe završava s točka-zarezom. Općenito, dakle, prototip ima oblik

```
tip ime_funkcije(tip_1 arg_1, tip_2 arg_2, ...);
```

Prototip za funkciju `binom` bi bio

```
long binom(int n, int k);
```

On predstavlja samo deklaraciju funkcije koja kaže da je `binom` ime funkcije koja uzima dva argumenta tipa `int` i vraća argument tipa `long`. Ako je takva deklaracija postavljena prije funkcije `main`, onda se definicija funkcije može premjestiti iza funkcije `main`. Na osnovu prototipa prevodilac može provjeriti da li je funkcija ispravno pozvana.

Istaknimo da definicija funkcije mora biti u skladu s prototipom funkcije – tip funkcije te broj i tip argumenata moraju se podudarati.

2.6 Polja

Treba napisati program koja uzima dva trodimenzionalna vektora i ispituje njihovu okomitost. Matematički pojmovi vektora i matrica implementiraju se u C-u pomoću *polja*. Polje je niz indeksiranih varijabli istog tipa, smještenih u memoriji na uzastopnim lokacijama. Indeks polja uvijek kreće od nule.

Da bismo ispitali okomitost vektora iskoristit ćemo formulu za kosinus kuta između dva vektora:

$$\cos\phi = \frac{\vec{a}\cdot\vec{b}}{\|\vec{a}\|\|\vec{b}\|},$$

gdje je

$$\vec{a}\cdot\vec{b} = a_1b_1 + a_2b_2 + a_3b_3$$

skalarni produkt vektora, a

$$\|\vec{a}\| = \sqrt{a_1^2 + a_2^2 + a_3^2}$$

norma vektora. Vektori su okomiti ukoliko je kosinus kuta među njima jednak nuli. Budući da se u računu kosinusa pojavljuju greške zaokruživanja, uzimat ćemo da su vektori okomiti čim je kosinus njihovog kuta manji od nekog malog broja ε, u našem primjeru 10^{-10}.

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

double epsilon=1.0E-10;
double kosinus_kuta(double x[], double y[]);

int main(void)
{
    double a[3], b[3];
    double cos_phi;
    int i;

    printf("Unesite vektor a.\n");
    for(i=0;i<3;++i){
        printf("a[%d]= ",i+1);
        scanf(" %lf",&a[i]);
    }
    printf("Unesite vektor b.\n");
    for(i=0;i<3;++i){
        printf("b[%d]= ",i+1);
        scanf(" %lf",&b[i]);
    }

    cos_phi= kosinus_kuta(a, b);

    if(fabs(cos_phi) < epsilon){
        printf("Vektori su okomiti.\n");
        printf("Kosinus kuta = %f\n", cos_phi);
    }
    else{
        printf("Vektori nisu okomiti.\n");
        printf("Kosinus kuta = %f\n", cos_phi);
    }

    return 0;
}
```

```
double norma(double x[]) {
    int i;
    double suma;

    suma=0.0;
    for(i=0;i<3;++i) suma = suma + x[i]*x[i];

    return sqrt(suma);
}

double produkt(double x[], double y[]) {
    int i;
    double suma;

    suma=0.0;
    for(i=0;i<3;++i) suma = suma + x[i]*y[i];

    return suma;
}

double kosinus_kuta(double x[], double y[]) {
    double cos_phi;

    cos_phi=produkt(x,y);
    cos_phi=cos_phi/(norma(x)*norma(y));

    return cos_phi;
}
```

Program se sastoji od funkcije `main`, funkcije `norma` koja računa normu vektora, funkcije `produkt` koja računa skalarni produkt dva vektora i funkcije `kosinus_kuta` koja računa kosinus kuta pozivajući funkcije `produkt` i `norma`. Ispred funkcije `main` naveden je prototip funkcije `kosinus_kuta`

```
double kosinus_kuta(double x[], double y[]);
```

To je nužno jer je funkcija definirana iza `main` funkcije, u kojoj se poziva. S druge strane, funkcija `kosinus_kuta` poziva funkcije `produkt` i `norma`, no kako su one definirane ispred nje, prototip nije potreban.

Funkcija `main` započine deklaracijom polja `a` i `b`:

```
    double a[3], b[3];
```

Uglate zagrade iza imena varijabli govore da je varijabla polje s tri elementa. Općenita, definicija polja je oblika

```
tip ime[max];
```

gdje je `tip` tip varijable, `max` broj elemenata polja, a `ime` ime polja. Indeks polja stavlja se unutar uglatih zagrada. Elementi polja `ime` su

```
ime[0], ime[1], ime[2], ... , ime[max-1]
```

U našem primjeru deklarirali smo polja `a` i `b`, oba sa po tri element tipa `double`. Elementi tih polja su `a[0], a[1], a[2]` i `b[0], b[1], b[2]`.

U dijelu koda

```
printf("Unesite vektor a.\n");
for(i=0;i<3;++i){
    printf("a[%d]= ",i+1);
    scanf(" %lf",&a[i]);
}
printf("Unesite vektor b.\n");
for(i=0;i<3;++i){
    printf("b[%d]= ",i+1);
    scanf(" %lf",&b[i]);
}
```

učitavaju se dva vektora, element po element. Pri ispisu

```
    printf("a[%d]= ",i+1);
```

pomaknuli smo indeks `i` za jedan kako bi korisnik imao dojam da komponente vektora imaju indekse 1, 2 i 3. Stvarni indeksi u C-u uvijek počinju s nulom.

Elemente polja indeksiramo cjelobrojnom varijablom. Izraz `a[i]` je `i+1`-vi element poja (jer prvi ima indeks `i=0`). On predstavlja jednu varijablu tipa `double` i može se javiti u svim izrazima u kojima se može pojaviti varijabla tipa `double`. Tako, na primjer, izraz `&a[i]` daje adresu `i+1`-vog elementa poja `a`.

U liniji

```
cos_phi= kosinus_kuta(a, b);
```

poziva se funkcija `kosinus_kuta`. Njena definicija je dana iza `main`-a:

```
double kosinus_kuta(double x[], double y[]) {
    double cos_phi;

    cos_phi=produkt(x,y);
    cos_phi=cos_phi/(norma(x)*norma(y));

    return cos_phi;
}
```

Funkcija uzima dva argumenta tipa polja. Ti su argumanti deklarirani s uglatim zagradam koje označavaju da se radi o poljima. Dimenzije polja nisu navedene, jer to u deklaraciji argumenata funkcije nije potrebno.

Kod poziva funkcije formalni argumenti `x` i `y` zamijenjuju se imenima polja `a` i `b`, definiranim u funkciji `main`. To su stvarni argumenti funkcije i sve operacije koja funkcija obavlja nad poljima obavljaju se nad stvarnim argumentima.

Treba dobro uočiti razliku između definicije varijable tipa polja i deklaracije argumenta tipa polja u nekoj funkciji. Kad se definira varijabla obavezno navodimo njezinu dimenziju, kako bi prevodilac mogao rezervirati dovoljno memorijskog prostora za nju. Definicije poput

```
double x[];   /* POGRESNO */
```

predstavljaju grešku. U deklaraciji argumenta tipa polja dimenziju ne treba navoditi jer za taj argument neće biti rezerviran memorijski prostor. Funkcija će vršiti operacije nad stvarnim argumentima. Pri tome je odgovornost programera da osigura da stvarni argument funkcije bude odgovarajuće dimenzije. Ipak, nije pogrešno deklarirati argumente tipa polja s njihovim dimenzijama. Tako smo mogli pisati

```
double kosinus_kuta(double x[3], double y[3])
{
   ......
}
```

U implementaciji funkcija `norma` i `produkt` koristili smo varijablu istog imena: `suma`. Svaka varijabla definirana unutar tijela funkcije je lokalna varijabla. Ona postoji samo za vrijeme izvršavanja tijela funkcije. U nekoj drugoj funkciji lokalnoj varijabli možemo dati isto ime jer samim time što su definirane u različitim funkcijama, te su varijable posve različite. Isto vrijedi i za argumente funkcije, tako da različite funkcije mogu imati argumente istog imena.

U završnom dijelu funkcije `main`

```
if(fabs(cos_phi) < epsilon){
    printf("Vektori su okomiti.\n");
    printf("Kosinus kuta = %f\n", cos_phi);
}
else{
    printf("Vektori nisu okomiti.\n");
    printf("Kosinus kuta = %f\n", cos_phi);
}
```

ispitujemo da li je apsolutna vrijednost kosinusa manja od epsilona i ispisujemo odgovarajuću poruku. Uočimo da je varijabla `epsilon` definirana ispred funkcije `main` i tamo je inicijalizirana s 10^{-10}.

Varijabla definirana ispred funkcije `main` je **globalna varijabla**. Njeno područje djelovanja je od mjesta definicije do kraja datoteke u kojoj je definirana. Takvim se varijablama može pristupiti u svim funkcijama definiranim u datoteci. Tako varijabli `epsilon` možemo pristupiti i u `main` i u svim ostalim funkcijama.

Napomena. Realne konstante s eksponentom pišu se pomoću slova `E` koje razdvaja mantisu i eksponent. Na primjer, $5,23 \times 10^4$ je `5.23E4` itd. □

2.6.1 `<math.h>`

Funkcije poput `printf` i `scanf` nisu dio programskog jezika C već pripadaju standardiziranim bibliotekama funkcija kojima je opskrbljen svaki C prevodilac. Prototipovi funkcija iz standardnih biblioteka nalaze se u sistemskim datotekama zaglavlja koje je potrebno uključiti u svaki program koji ih koristi. Ako koristimo matematičke funkcije iz standardne biblioteke, onda moramo uključiti zaglavlje `<math.h>` i pri kompilaciji na kraj komandne linije staviti `-lm` opciju.

Biblioteka matematičkih funkcija sadrži brojne funkcije. Između ostalih `sin`, `cos`, `tan` (tg), `asin` (arcsin), `acos` (arc cos), `atan` (arc tg), `exp` (e^x), `log` (ln), `log10` (log) itd. Sve one uzimaju jedan argument tipa `double` i vraćaju rezultat tipa `double`.

U C-u nema operatora potenciranja pa zato postoji funkcija `pow`:

```
double pow(double,double)
```

koja računa `pow`$(x, y) = x^y$. Ona daje grešku ako je $x = 0$ i $y \leq 0$ te $x < 0$ i y nije cijeli broj.

Treba razlikovati funkciju `abs` koja uzima argument tipa `int` i vraća apsolutnu vrijednost broja (tipa `int`) od funkcije `fabs` koja uzima `double` i vraća apsolutnu vrijednost (tipa `double`).

2.7 Pokazivači

Svaka varijabla deklarirana u C programu ima svoju adresu koju je moguće dohvatiti putem adresnog operatora &. Na primjer,

```
#include <stdio.h>

int main(void) {
    double x=5;

    printf("x = %f, adresa od x= %p\n",x,&x);
    return 0;
}
```

Program će ovisno o računalu na kojem se izvodi ispisati nešto kao

```
x=5.000000, adresa od x= 0065FDF0
```

Adresa varijable `x` ispisana je heksadecimalno; znak konverzije `%p` služi za ispis adresa.

Adresu varijable nekog tipa možemo zapamtiti u varijabli koji je tipa pokazivač na dani tip. Na primjer, program

```
#include <stdio.h>

int main(void) {
    double x=5;
    double *px;

    px=&x;
    printf("x = %f, adresa od x= %p\n",x,px);
    return 0;
}
```

daje posve isti rezultat kao i prethodni program. Deklaracija

```
    double *px;
```

definira varijablu `px` kao pokazivač na tip `double`. Zvjezdica (`*`) prije imena varijable govori da se ne radi o varijabli tipa `double` nego o pokazivaču na tip `double`. Pokazivač na tip `double` može sadržavati samo adrese varijabli tipa `double`. U naredbi

```
px=&x;
```

u varijablu `px` smješta se adresa varijable `x`.

Vrijednost na koju pokazivač pokazuje možemo dohvatiti putem operatora dereferenciranja `*`. Na primjer,

```
double x=5,y;  <-- varijable tipa double
double *px;    <-- pokazivac na tip double

px=&x;         <-- px sada ima adresu varijable x
y=*px;         <-- y prima vrijednost varijable x (=5)
```

Na primjer, prethodni program bismo mogli zapisati u obliku

```
#include <stdio.h>

int main(void) {
    double x=5;
    double *px;

    px=&x;
    printf("x = %f, adresa od x= %p\n",*px,px);
    return 0;
}
```

Uočimo da `*` ima različito značenje u deklaraciji i u izvršnoj naredbi. U deklaraciji varijable ona ukazuje da je varijabla tipa pokazivač na dani tip. U izvršnoj naredbi ona predstavlja operator dereferenciranja.

Skalarni argumenti prenose se funkciji "po vrijednosti", što znači kopiranjem. U primjeru,

```
double x,y;
.....
x=2.0;
y=sin(x);
......
```

prevodilac će kopirati vrijednost varijable `x` u formalni argument funkcije `sin`, koja će na osnovu te vrijednosti izračunati sinus. Na taj način funkcija ne može (namjerno ili nenamjerno) promijeniti vrijednost varijable `x` koja joj je dana kao argument. To je vrlo korisno svojstvo koje u ovom slučaju osigurava da računanje vrijednosti `sin(x)` neće promijeniti vrijednost argumenta `x`. Ipak, u nekim slučajevima želimo da funkcija promjeni vrijednost

svog argumenta, kao kod funkcije `scanf`. U takvim slučajevima moramo se poslužiti pokazivačima. Funkciji dajemo kao argument pokazivač koji sadrži adresu varijable koju treba promijeniti. Ona neće moći promijeniti vrijednost pokazivača, ali će putem operatora dereferenciranja moći promijeniti vrijednost na koju pokazivač pokazuje.

Pogledajmo sljedeći primjer. Treba napisati funkciju koja uzima niz znakova i u njemu nalazi prvi samoglasnik. Funkcija treba vratiti nađeni samoglasnik i mjesto u nizu na kojem je nađen. Ukoliko u danom nizu nema samoglasnika, umjesto mjesta u nizu vraća se -1.

U ovom primjeru funkcija mora vratiti dvije vrijednosti. Kroz `return` naredbu može se vratiti samo jedna vrijednost, tako da se druga mora vratiti kroz argument funkcije. (Funkcija ne može vratiti polje.) To možemo realizirati na sljedeći način:

```
#include <stdio.h>

int trazi(char linija[], int n, char *psamoglasnik)
{
    int i;
    char c;

    for(i=0; i<n; i++){
       c=linija[i];
       if(c == 'a' || c == 'e' || c == 'i'
                   || c == 'o' || c == 'u')
        {
            *psamoglasnik=c;
             return i;
        }
    }
    return -1;
}
```

Funkcija `trazi` uzima niz znakova `linija`, cijeli broj `n` koji daje broj znakova u nizu `linija` i pokazivač na varijablu tipa `char`, koji smo nazvali `psamoglasnik` (pointer na samoglasnik). Nađeni samoglasnik će biti vraćen kroz argument `psamoglasnik`, a mjesto na koje je nađen vraća se kroz `return` naredbu.

Varijabla tipa `char` služi za pamćenje pojedinačnih znakova. Kada treba memorirati više znakova koristimo se poljima tipa `char`. Znakovne konstante formiraju se tako da se znak stavi u jednostruke navodnike. Na primjer, `'a'`, `'/'`, `'?'` itd.

U `for` petlji prolazimo kroz čitav niz znakova. Za svaki znak u `if` naredbi ispitujemo da li je samoglasnik. Uočimo da smo pet testova spojili operatorom logičko ILI.

Svi logički operatori dani su u ovoj tabeli:

Operator	Značenje
`&&`	logičko I
`||`	logičko ILI
`!`	logička negacija (unarno)

Ako je samoglasnik nađen, u naredbi

```
*psamoglasnik=c;
```

se znak `c=linija[i]` kopira na lokaciju na koju pokazuje `psamoglasnik`. Nakon toga se u pozivni program vraća položaj znaka koji je dan varijablom `i`. Ukolikoje petlja `for` izvršena a da samoglasnik nije nađen, vraća se -1, a lokacija na koju pokazuje `psamoglasnik` se ne mijenja. Poziv funkcije `trazi` mogao bi izgledati ovako:

```
int main(void)
{
    char ime[]={'P','r','o','g','r','a','m','s','k','i',' ',
        'j','e','z','i','k',' ','C','.'};
    char znak;
    int no;

    no=trazi(ime,19,&znak);
    if(no != -1){
    printf("Prvi samoglasnik = %c\n",znak);
    printf("Nalazi se na mjestu %d\n",no);
    }
    else    printf("Nema samoglasnika.\n");

    return 0;
}
```

Varijabla u koju smještamo nađeni samoglasnik je `znak`. Njena se adresa predaje funkciji `trazi`.

U deklaraciji polja `ime` susrećemo se s inicijalizacijom polja. Svako se polje prilikom deklaracije može inicijalizirati tako da se nakon znaka jednakosti u vitičastim zagradama navedu svi elementi polja, odvojeni zarezom. Dimenziju polja ne moramo nužno navesti jer će ju prevodilac sam izračunati na osnovu inicijalizacijskog niza.

2.8 Polja znakova

Napišimo program koji učitava niz znakova limitiran bjelinama, računa broj numeričkih znakova u nizu i ispisuje učitani niz u obrnutom poretku.

```
#include <stdio.h>
#include <string.h>
#define MAX 128
void inv(char []);

int main(void) {
    char niz_znakova[MAX],c;
    int i,brojac=0;

    scanf("%s", niz_znakova);

    i=0;
    while((c=niz_znakova[i]) !='\0') {
       if(c>='0' && c<='9') ++brojac;
       i++;
    }

    printf("Broj ucitanih numerickih znakova = %d\n",brojac);
    inv(niz_znakova);
    printf("%s\n",niz_znakova);
    return 0;
}

void inv(char s[]) {
    int c,i,j;

    for(i=0,j=strlen(s)-1; i<j; i++,j--) {
       c=s[i]; s[i]=s[j]; s[j]=c;
    }
}
```

Program započinje uključivanjem standardnih datoteka zaglavlja `<stdio.h>` (zbog `printf` i `scanf`) te `<string.h>` (zbog funcije `srtlen`). Zatim

```
#define MAX 128
```

uvodi simboličku konstantu `MAX` koja ima vrijednost 128.

Naredba `#define` je preprocesorska naredba. Preprocesor je program koji prolazi kroz izvorni kod prije prevodioca i izvršava naredbe koje su mu namijenjene. Na primjer, `#include` je preprocesorska naredba kojom se u datoteku, na onom mjestu na kojem se nalazi, uključuje sadržaj datoteke navedene u `#include` liniji. `#define` je prepprocesorska naredba kojom se definira simboličko ime. U našem primjeru ime je `MAX`, a vrijednost koja mu se pridružuje je 128. Preprocesor nalazi u programu svako pojavljivanje simbola `MAX` i zamijenjuje ga sa 128. Na taj način umjesto konstanti možemo koristiti simbolička imena.

Linija

```
void inv(char []);
```

je prototip funkcije `inv` koja je definirana iza `main` funkcije. Funkcija je tipa `void` što znači da ne vraća nikakvu vrijednost u pozivni program. Ona će vršiti inverziju niza znakova.

Argument funkcije `inv` je niz znakova. Vidjeli smo da pri deklaraciji polja kao argumenta funkcije dimenziju polja ne moramo navesti. U ovom primjeru vidimo da u prototipu niti ime varijable nije važno, pa je stoga ispušteno. Važna je samo prisutnost uglatih zagrada koje signaliziraju da je argument *polje* tipa `char`.

Varijabla `niz_znakova` iz funkcije `main` je polje od `MAX` (=128) znakova tj. varijabli tipa `char`. Ti su znakovi smješteni jedan do drugog u memoriji i indeksirani su od 0 do 127.

Deklaracije običnih varijabli i polja možemo slobodno miješati (ako su istog tipa) te smo stoga mogli pisati

```
char niz_znakova[MAX],c;
```

umjesto

```
char niz_znakova[MAX];
char c;
```

U deklaraciji varijabli `i` i `brojac` varijabla `brojac` je inicijalizirana nulom.

```
int i,brojac=0;
```

Takva inicijalizacija kod same deklaracije varijable moguća je za sve vrste varijabli. Na primjer, varijablu tipa `char` možemo iniijalizirati znakom `a` na ovaj način:

```
char c='a';
```

Prva izvršna naredba je

```
scanf("%s", niz_znakova);
```

koja učitava niz znakova u varijablu `niz_znakova`. Znak konverzije `"%s"` označava da se učitava niz znakova.

Znakovni niz (ili string) predstavlja poseban tip podatka u C-u. Konstantan znakovni niz (konstantan string) je svaki niz znakova smješten u dvostruke navodnike, kao npr. "Dobar dan.\n". To je znakovni niz koji se sastoji od sljedećih znakova:

```
D o b a r   d a n . \n \0
```

Uočimo da je na kraj niza dodan *nul-znak* `'\0'` koji ima ulogu signaliziranja kraja niza znakova. Znakovni niz se time razlikuje od niza znakova. Niz znakova je niz bilo kakvih znakova, a znakovni niz (string) je niz znakova čiji je zadnji znak nul-znak.

U datoteci zaglavlja `<string.h>` deklariran je niz funkcija namjenjenih radu sa stringovima. Jedna od njih je i `strlen` koja vraća duljinu niza znakova, ne računajući nul-znak. tako je `strlen("Dobar dan.\n")=11`.

Deskriptor `"%s"` označava da funkcija `scanf` mora učitati niz znakova u varijablu `niz_znakova` i dodati \0 iza posljednjeg znaka. Učitavanje niza znakova sa ulaza prestaje s prvom bjelinom. Pod bjelinama se osim bjelina unesenih razmaknicom smatraju i znakovi uneseni tipkama Tab (tabulator) i Enter (prijelaz u novi red).

Uočimo da ispred varijable `niz_znakova` u pozivu funkciji `scanf` nema adresnog operatora `&`. To je općenito pravilo za nizove. Kada je niz stvarni argument funkcije ona ne dobiva kopiju čitavog niza već adresu prvog elementa u nizu. U funkciji se članovi niza mogu dohvatiti (i mijenjati) isto kao i u pozivnom programu pomoću indeksa.

U C-u izraz pridruživanja ima vrijednost koja je jednaka vrijednosti lijeve strane nakon pridruživanja. Stoga izrazi poput

```
c=niz_znakova[i]
```

mogu sudjelovati u složenijim izrazima. U `while` petlji

```
i=0;
while((c=niz_znakova[i]) !='\0') {
   if(c>='0' && c<='9') ++brojac;
   i++;
}
```

prvo se varijabli `c` pridružuje `niz_znakova[i]`, a zatim se vrijednost izraza pridruživanja (to je vrijednost varijable `c`) uspoređuje s znakom `'\0'`. Operator `!=` je logički operator uspoređivanja. Rezultat uspoređivanja

```
c!='\0'
```

je istina ako varijabla `c` ne sadrži nul-znak `'\0'`.

Gornja `while` petlja mogla je biti napisana i na sljedeći način:

```
i=0;
c=niz_znakova[0];
while(c !='\0') {
   if(c>='0' && c<='9') ++brojac;
   i++;
   c=niz_znakova[i];
}
```

Vidimo da korištenje izraza pridruživanja unutar složenijih izraza čini kôd kompaktnijim. Pri tome su bitne zagrade je operatori pridruživanja imaju najniži prioritet. Izraz

```
c=niz_znakova[i] !='\0'
```

ekvivalentan je izrazu

```
c=(niz_znakova[i] !='\0')
```

koji bi pridružio varijabli `c` vrijednost 0 (laž) ako `niz_znakova[i]` sadrži znak `'\0'` ili 1 (istina) ako `niz_znakova[i]` ne sadrži znak `'\0'`.

Relacijski izrazi kao što je to `c != '\0'` ili `i<=10` imaju logičku vrijednost istine ili laži koja se u C-u reprezentira cijelim brojem: 1 (i bilo koja vrijednost različita od nule) predstavlja istinu, a 0 laž. To je razlog zbog kojeg bi u prethodnom primjeru varijabla `c` dobila vrijednost 0 ili 1.

Naredbom

```
if(c>='0' && c<='9') ++brojac;
```

provjerava se da li je znakovna vrijednost smještena u varijablu `c` broj. Pri tome koristimo činjenicu da su brojevi u skupu znakova poredani u prirodnom poretku (isto vrijedi i za slova engleske abecede).

Po završetku `while` petlje iz funkcije `main` varijabla `brojac` sadrži broj numeričkih znakova u znakovnom nizu. Uočimo da smo pisali `++brojac` umjesto `brojac++`. Operatore inkrementiranja i dekrementiranja možemo pisati prije i poslije varijable. Iako među tim zapisima postoji razlika koju ćemo naučiti kasnije, oba ova oblika povećavaju varijablu za jedan.

Funkcija `inv` sastoji se od jedne `for` petlje

```
for(i=0,j=strlen(s)-1; i<j; i++,j--)
```

u kojoj se inicijaliziraju dvije varijable: `i` koja se pozicionira na početak stringa `i` `j` koja se pozicionira na kraj. Višetruke inicijalizacije odvajaju se zarezom. Slično, u dijelu u kome se mijenja brojač petlje mijenjaju je oba brojača: `i` se povećava, a `j` se smanjuje. Ove se dvije naredbe ponovo odvajaju zarezom. Petlja se izvršava sve dok je izraz `i<j` istinit. Unutar tijela petlje vrši se zamjena znakova

```
c=s[i]; s[i]=s[j]; s[j]=c;
```

koji su na pozicijama `i` i `j`. Pri toj zamjeni morali smo koristiti pomoćnu varijablu `c`. Evidentno je da će na izlazu iz `for` petlje poredak znakova u znakovnom nizu biti invertiran. Pri tome posljednji nul-znak nismo pomicali s njegovog mjesta.

Poglavlje 3

Konstante i varijable

3.1 Znakovi

C koristi sljedeći skup znakova: velika i mala slova engleske abecede A-Z i a-z, decimalne znamenke 0-9 te specijalne znakove

```
+  -  *  /  =  %  &  #
!  ?  ^  "  '  ~  \  |
<  >  (  )  [  ]  {  }
:  ;  .  ,  _  (bjelina)
```

Pod bjelinom se podrazumijeva, osim sam bjeline, horizontalni i vertikalni tabulator te znak za prijelaz u novi red. Mnoge verzije jezika C dozvoljavaju da i drugi znakovi budu uključeni unutar stringova i komentara.

3.2 Komentari

U program je potrebno umetati komentare radi lakšeg razumijevanja njegovog funkcioniranja. Minimalno je potrebno komentirati svaku funkciju u programu, objašnjavajući zadaću funkcije, njene argumente i vrijednost koju vraća.

Komentar započinje znakom `/*` i završava znakom `*/`. Tekst između tih graničnika ignorira se prilikom prevođenja programa u izvršni kôd. Komentar se može umetnuti bilo gdje u programu i može sadržavati više linija teksta.

```
/*      Ovo je
           komentar.  */
```

Prevodilac zamijenjuje svaki komentar jednom bjelinom. Zamjena se vrši prije prolaska preprocesora, tako da se mogu komentirati i preprocesorske naredbe.

Komentari ovog tipa mogu dovesti do gubitka kôda u situacijama u kojim je ispušten jedan graničnik. U primjeru

```
/*    Ovo je prvi komentar.
   x=72.0;
/*    Ovo je drugi komentar. */
```

prvi komentar smo zaboravili zatvoriti. Prevodilac će ignorirati text sve do prvog znaka `*/` na koji naiđe, što rezultira gubitkom naredbe `x=72.0;`. Jednako tako ne možemo komentirati dio programa koji već sadrži komentar. Na primjer, kôd

```
/*
   x=72.0;   /*   Inicijalizacija  */
   y=31.0;
*/
```

je neispravan jer komentar započet prvim `/*` graničnikom završava s prvim `*/` graničnikom, što znači u liniji

```
   x=72.0;   /*   Inicijalizacija  */
```

Prevodilac će javiti sintaktičku grešku jer zadnji `*/` graničnik nema odgovarajućeg `/*` graničnika.

Standard C99 omogućava korištenje drugog tipa komentara koji dolazi iz Jave i C++a. Komentar započinje znakom `//` i prostire se do kraja linije. Na primjer,

```
int n;  // brojac znakova
```

Prevodilac ignorira tekst od znaka `//` do kraja linije. Stariji prevodioci ne prihvaćaju ovaj način komentiranja.

Ako treba komentirati veliki dio programa koji u sebi sadrži brojne komentare, najbolje je koristiti preprocesor (vidi Poglavlje 8).

3.3 Identifikatori i ključne riječi

Identifikatori su imena koja se pridružuju različitim elementima programa kao što su varijable polja i funkcije. Sastoje se od slova i brojeva u bilo kojem poretku s ograničenjem da prvi znak mora biti slovo. Velika i mala slova se razlikuju i znak _ (*underscore*) smatra se slovom.

Duljina imena je proizvoljna premda je prema standardu C90 prevodilac dužan prepoznati samo prvih 31 znakova. U standardu C99 ta je granica dignuta na 63 znaka. Na imena vanjskih varijabli postoje veća ograničenja i ona ne bi trebala biti duža od 6 znakova (C90), odnosno 31 (C99).

Primjer 3.1 *Primjeri pravilno napisanih identifikatora:*

```
x      y13   sum_1   _temp
names  Pov1  table   TABLE
```

Primjer 3.2 *Primjeri nepravilno napisanih identifikatora:*

```
3dan    (prvi znak je broj)
"x"     (nedozvoljeni znak ")
ac-dc   (nedozvoljeni znak -)
```

Jedan broj riječi, tzv. ključne riječi, imaju posebno značenje u jeziku i stoga se ne mogu koristiti kao identifikatori. Ključne riječi prema C90 standardu su:

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

Standard C99 uveo je i sljedeće ključne riječi:

```
inline    _Bool     _Imaginary
restrict  _Complex
```

U nekim verzijama jezika postoje i sljedeće (nestandardne) ključne riječi:

```
ada    far      near
asm    fortran  pascal
entry  huge
```

Imena koja počinju sa znakom _ treba izbjegavati jer su rezervirana za varijable i funkcije iz standardne biblioteke.

3.4 Osnovni tipovi podataka

Osnovni tipovi podataka su

`char`, `int`, `float`, `double`.

Količina memorije koju pojedini tip zauzima ovisi o računalu. Mi navodimo tipične vrijednosti.

`int`: cjelobrojni podatak. Tipično zauzima 4 okteta.

`char`: znakovni podatak. Sadržava jedan znak iz sustava znakova koji računalo koristi. U memoriji tipično zauzima jedan oktet.

`float`: broj s pokretnim zarezom u jednostrukoj preciznosti. U memoriji tipično zauzima četiri okteta.

`double`: broj s pokretnim zarezom u dvostrukoj preciznosti. U memoriji tipično zauzima osam okteta.

3.4.1 Cjelobrojni podaci

Cjelobrojni tipovi podataka služe pamćenju cijelih brojeva. Svaki tip pokriva određen raspon cijelih brojeva ovisno o količini memorije koju zauzima i načinu kodiranja. Na primjer, ako varijabla određenog integralnog tipa zauzima n bitova memorije i način kodiranja je 2-komplement, onda je pokriveni raspon od -2^{n-1} do $2^{n-1} - 1$.

Cjelobrojni tip `int` može se modificirati pomoću kvalifikatora `short` i `long`. Tako dobivamo nove cjelobrojne tipove:

`short int` ili kraće `short`
`long int` ili kraće `long`

`short`: "kratki" cjelobrojni podatak. U memoriji zauzuma manje memorijskog prostora od podatka tipa `int` pa može prikazati manji raspon cijelih brojeva.

`long`: "dugi" cjelobrojni podatak. U memoriji zauzuma više memorijskog prostora od podatka tipa `int` pa može prikazati veći raspon cijelih brojeva.

C propisuje samo minimalnu preciznost pojedinih cjelobrojnih tipova. Tako tip `int` mora imati širinu najmanje 16 bitova, dok tip `long` mora imati minimalno 32 bita.

Cijeli brojevi tipa `int`, `short` i `long` pokrivaju područje pozitivnih i negativnih brojeva. Primjenom kvalifikatora `unsigned` dobivamo tipove podataka

unsigned int ili kraće unsigned
unsigned short int ili kraće unsigned short
unsigned long int ili kraće unsigned long

koji zauzimaju isti memorijski prostor kao osnovni tipovi podataka (int, short, long), a mogu reprezentirati samo pozitivne cijele brojeve (uključivši nulu). Oni pokrivaju stoga približno dvostruko veći raspon pozitivnih cijelih brojeva od osnovnih tipova.

Tip int je prirodan cjelobrojni podatak za dano računalo. Za njegovo pamćenje koristi se jedna računalna riječ, što je u današnje vrijeme uglavnom 32 bita. Korištenje širih cjelobrojnih tipova može stoga imati za posljedicu sporije izvršavanje programa.

Koje su minimalne i maksimalne vrijednosti cjelobrojnih varijabli pojedinih tipova? Odgovor je zavisan o računalu. Stoga postoji sistemska datoteka zaglavlja <limits.h>[1] u kojoj su granične vrijednosti definirane. Na nekom računalu dio sadržaja datoteke <limits.h> mogao bi biti sljedeći:[2]

```
#define SHRT_MIN      (-32768)
#define SHRT_MAX      32767
#define INT_MIN       (-2147483648)
#define INT_MAX       2147483647
#define LONG_MIN      (-9223372036854775808L)
#define LONG_MAX      9223372036854775807L
#define USHRT_MAX     65535
#define UINT_MAX      4294967295U
#define ULONG_MAX     18446744073709551615UL
```

(Oznake L,U i UL su objašnjene u sekciji 3.6.) Ovdje su

- SHRT_MIN, SHRT_MAX: Minimalna i maksimalna vrijednost za tip short.
- INT_MIN, INT_MAX: Minimalna i maksimalna vrijednost za tip int.
- LONG_MIN, LONG_MAX: Minimalna i maksimalna vrijednost za tip long.
- USHRT_MAX: Maksimalna vrijednost za tip unsigned short.
- UINT_MAX: Maksimalna vrijednost za tip unsigned.
- ULONG_MAX: Maksimalna vrijednost za tip unsigned long.

[1] Na Unix sustavima ta se datoteka najčešće nalazi u /usr/include/ direktoriju.
[2] Zašto su negativne vrijednosti u zagradama objašnjeno je u sekciji 8.

Ovdje imamo primjer računala na kojem tip `short` zauzima 2 okteta, tip `int` 4 okteat, a tip `long` 8 okteta. Stoga imamo sljedeće raspone:

- `unsigned short`: 0,1,..., $2^{16} - 1 = 65535$,
- `short`: $-2^{15} = -32768$,..., -1,0,1,..., $2^{15} - 1 = 32767$,
- `unsigned`: 0,1,..., $2^{32} - 1 = 4294967295$,
- `int`: $-2^{31} = -2147483648$,..., -1,0,1,..., $2^{31} - 1 = 2147483647$,
- `unsigned long`: 0,1,..., $2^{64} - 1 = 18446744073709551615$,
- `long`: -2^{63},..., -1,0,1,..., $2^{63} - 1$.

Na vašem računalu ovi rasponi mogu biti drugačiji. Sljedeći program ispisuje gornje vrijednosti:

```
#include <stdio.h>
#include <limits.h>
int main()
{
  short          s_max  = SHRT_MAX;
  short          s_min  = SHRT_MIN;
  unsigned short us_max = USHRT_MAX;
  int            i_max  = INT_MAX;
  int            i_min  = INT_MIN;
  unsigned       ui_max = UINT_MAX;
  long           l_max  = LONG_MAX;
  long           l_min  = LONG_MIN;
  unsigned long  x_max  = ULONG_MAX;

  printf("SHRT_MAX = %hd\n",s_max);
  printf("SHRT_MIN = %hd\n",s_min);
  printf("USHRT_MAX = %hu\n",us_max);
  printf("INT_MAX = %d\n",i_max);
  printf("INT_MIN = %d\n",i_min);
  printf("UINT_MAX = %u\n",ui_max);
  printf("LONG_MAX = %ld\n",l_max);
  printf("LONG_MIN = %ld\n",l_min);
  printf("ULONG_MAX = %lu\n",x_max);
  return 0;
}
```

Uočite kontrolne znakove s kojima se ispisuju pojedini integralni tipovi.

Što se dešava ako pokušamo generirati cijeli broj veći od najvećeg ili manji od najmanjeg dopuštenog? Da bismo to vidjeli dodajmo prethodnom programu sljedeće naredbe:

```
printf("SHRT_MAX+1 = %hd\n",s_max+1);
printf("SHRT_MIN-1 = %hd\n",s_min-1);
printf("USHRT_MAX+1 = %hu\n",us_max+1);
printf("INT_MAX+1 = %d\n",i_max+1);
printf("INT_MIN-1 = %d\n",i_min-1);
printf("UINT_MAX+1 = %u\n",ui_max+1);
printf("LONG_MAX+1 = %ld\n",l_max+1);
printf("LONG_MIN-1 = %ld\n",l_min-1);
printf("ULONG_MAX+1 = %lu\n",x_max+1);
```

Rezultat koji se dobiva je sljedeći: kada se najvećem broju bez predznaka doda jedan dobiva se nula; kad se doda dva dobiva se jedan itd. Brojevi bez predznaka su dakle "poredani u krug", odnosno implementirana je aritmetika modulo 2^n, gdje je n širina tipa (u bitovima).

Kod brojeva s predznakom imamo slično ponašanje. Dodavanje jedinice najvećem pozitivnom broju daje najveći negativni broj, a oduzimanje jedinice od najvećeg negativnog broja daje najveći pozitivni broj. I ovdje su brojevi "poredani u krug", ali ponašanje ovisi o načinu kodiranja negativnih brojeva (2-komplement u našem slučaju).

Standard C99 uvodi dodatne tipove cijelih brojeva `long long` i `unsigned long long`. Tip `long long` mora biti implementiran s minimalno 64 bita.

3.4.2 Znakovni podaci

Podatak tipa `char` namijenjen je pamćenju individualnih znakova. Znakovi se u računalu kodiraju i sve manipulacije sa znakovima obavljaju se putem njihovih numeričkih kodova. Zbog toga je tip `char` ustvari cjelobrojni tip podataka vrlo malog raspona.

Postoje različiti načini kodiranja znakova, Jedan od najstarijih je tzv. ASCII kôd.[3] ASCII kôd ima numerički raspon od 0 do 127 i dovoljno je 7 bitova za njegovo pamćenje. Npr. slovo A (veliko) ima ASCII kôd 65, B ima kôd 66 itd. Budući da ASCII kôd ne sadrži slova iz europskih jezika razvijeni su razni drugi osam-bitni kodovi od kojih treba spomenuti ISO Latin 1 (za zapadno europske jezike), ISO Latin 2 (za srednjoeuropske jezike) itd.

[3]ASCII je kratica od "American Standard Code for Information Interchange".

U C-u je `char` cjelobrojni tip podatka s vrlo malim rasponom. Najčešće je to jedan oktet što je dovoljno za sve osam-bitne kodove. Na tip `char` možemo primijeniti kvalifikatore `signed` i `unsigned` kao i na ostale cjelobrojne tipove. Ako se za `char` koristi jedan oktet, onda tip `unsigned char` poprima pozitivne cjelobrojne vrijednosti između 0 i 255, dok tip `signed char` poprima pozitivne i negativne cjelobrojne vrijednosti između -127 (ili -128) i 127. Pravilo je da se kvalifikatori `signed` i `unsigned` ne koriste s tipom `char` ako on služi za manipulaciju sa znakovima, već samo ako ga koristimo kao cjelobrojni tip malog raspona. Širina tipa `char` te njegova minimalna i maksimalna vrijednost dani su u datoteci `<limits.h>`.

U datoteci zaglavlja `<stddef.h>` definiran je cjelobrojni tip `wchar_t`, tzv. široki znakovni tip, koji je namijenjen kodiranju znakova iz različitih jezika.

3.4.3 Logički podaci

U ANSI C-u (standard C90) ne postoji poseban logički tip. Svaki cjelobrojni tip može preuzeti ulogu logičkog tipa tako što se vrijednost različita od nule interpretira kao istina, a nula kao laž. Standard C99 uvodi poseban logički tip `_Bool` koji prima samo vrijednosti 0 i 1 ("laž" i "istina"). I nadalje se cjelobrojne varijable mogu koristiti za prezentaciju logičkih vrijednosti, ali upotreba novog tipa može učiniti program jasnijim. Ključna riječ `_Bool` je izabrana da se izbjegnu konflikti s aplikacijama koje su implementirale svoj tip `bool`. Korisnik može uključiti datoteku zaglavlja `<stdbool.h>` u kojoj se definira `bool` kao sinonim za `_Bool` te se uvode simbolička imena `true` i `false`.

3.4.4 Realni podaci

Podaci tipa `float`, `double` i `long double` su realni brojevi koji zbog specifičnog načina pamćenja u računalu imaju naziv **brojevi s pokretnim zarezom** (eng. *floating point numbers*). Oni su analogni cjelobrojnim tipovima `short`, `int` i `long`. Svaki sljedeći tip pokriva veći raspon realnih brojeva i ima višu preciznost (veći broj značajnih znamenaka). Uobičajeno je brojeve tipa `float`, `double` i `long double` nazivati brojevima jednostruke, dvostruke i četverostruke preciznosti. (Na pojedinim računalima `double` može biti isto što i `long double`, što se lako provjerava pomoću `sizeof` operatora [vidi sekciju 4.2.2].)

Napomena. Mada su brojevi s pokretnim zarezom konačan podskup skupa realnih brojeva, radi kratkoće ćemo ih jednostavno zvati realnim brojevima (ili realnim varijablama, konstantama, podacima). □

Jezik ne propisuje širinu pojedinih tipova brojeva s pokretnim zarezom već to ovisi o računalu na kojem se program izvršava. S druge strane, svojstva sustava brojeva s pokretnim zarezom propisana su IEEE standardom kog

se danas pridržava većina proizvođača hardwarea. Stoga će svojstva tipa `float` i tipa `double` na većini strojeva biti ona propisana IEEE standardom (standard IEEE ne propisuje strogo brojeve četverostruke preciznosti).

Ovdje nećemo ulaziti u diskusiji brojeva s pokretnim zarezom. Navedimo samo granice u kojima se ti brojevi kreću (prema IEEE standardu): U jednostrukoj preciznosti (`float`) imamo

$$1.17549 \cdot 10^{-38} \approx 2^{-126} \leq |x| \leq (1 - 2^{-24}) \cdot 2^{128} \approx 3.40282 \cdot 10^{38},$$

dok u dvostrukoj (`double`) vrijedi

$$2.225 \cdot 10^{-308} \approx 2^{-1022} \leq |x| \leq (1 - 2^{-53}) \cdot 2^{1024} \approx 1.798 \cdot 10^{308}.$$

Kao i u slučaju cijelih brojeva vrijednosti ovisne o hardwareu stavljene su u standardnu datoteku zaglavlja. U slučaju brojeva s pokretnim zarezom to je datoteka `<float.h>`. Ona sadrži niz simboličkih konstanti koje daju različite informacije o realnim tipovima podataka. Na nekom računalu dio sadržaja datoteke `<float.h>` mogao bi biti sljedeći:

```
#define FLT_EPSILON     1.192092896E-07F
#define FLT_MIN         1.175494351E-38F
#define FLT_MIN_10_EXP  (-37)
#define FLT_MAX         3.402823466E+38F
#define FLT_MAX_10_EXP  (+38)

#define DBL_EPSILON     2.2204460492503131E-16
#define DBL_MIN         2.2250738585072014E-308
#define DBL_MIN_10_EXP  (-307)
#define DBL_MAX         1.7976931348623157E+308
#define DBL_MAX_10_EXP  (+308)
```

Konstante koje počinju s `FLT` odnose se na tip `float`, dok se one s `DBL` odnose na tip `double`. Ovdje redom imamo: strojni epsilon, najmanji (normalizirani) broj s pokretnim zarezom, najmanji dekadski eksponent, najveći (normalizirani) broj s pokretnim zarezom i najveći dekadski eksponent. Strojni epsilon je udaljenost između broja 1.0 i prvog većeg broja s pokretnim zarezom. On je bitan utoliko što pri aproksimaciji realnog broja najbližim brojem s pokretnim zarezom[4] činimo grešku koja ne prelazi jednu polovicu strojnog epsilona. On je stoga dobra mjera preciznosti brojevnog sustava.

Sustav brojeva s pokretnim zarezom sadrži simboličke veličine `+Inf` i `-Inf` koje predstavljaju plus i minus beskonačno. Njih dobivamo kada računskim

[4]Ukoliko je realan broj u području koje pokrivaju brojevi s pokretnim zarezom.

operacijama izađemo iz područja koje pokrivaju brojevi s pokretnim zarezom (eng. overflow). Operacija koja matematički nije dobro definirana (u proširenom skupu realnih brojeva) daje kao rezultat vrijednost NaN (eng. Not a Number). Malu ilustraciju tih svojstava predstavlja sljedeći program:

```
#include <stdio.h>
#include <float.h>
#include <math.h>
int main(void)
{
  double x_max=DBL_MAX;
  double x_min=DBL_MIN;

  printf("DBL_MAX = %e\n",x_max);
  printf("DBL_MAX = %e\n",x_max);
  printf("\n");
  printf("DBL_MAX*1.1 = %e\n",x_max*1.1);
  printf("DBL_MIN/0 = %e\n",x_min/0);
  printf("sqrt(-1) = %e\n",sqrt(-1.0));
  return 0;
}
```

Uočimo da smo morali uključiti datoteku zaglavlja `<math.h>` radi funkcije `sqrt(x)`=$\sqrt{x}$. Kompilirati treba s opcijom `-lm`.

3.4.5 Kompleksni podaci

Kompleksni podaci su dodani jeziku tek u standardu C99. Standard uvodi tri kompleksna tipa `float _Complex`, `double _Complex` i `long double _Complex`. Na primjer, jedna `float _Complex` varijabla sastoji se od dvije `float` varijable koje reprezentiraju realni i kompleksni dio broja. K tome se uvode još i tri imaginarna tipa: `float _Imaginary`, `double _Imaginary` i `long double _Imaginary`. Uključivanje zaglavlja `<complex.h>` supstituira ime `complex` za `_Complex`, `imaginary` za `_Imaginary` te `I` za $\sqrt{-1}$.

3.5 Varijable i deklaracije

Varijabla je objekt koji zauzima dio memorijskog prostora i koji se može dohvatiti putem svog imena. Varijabla ima tri atributa o kojim ovisi kako će se podatak pohranjen u memoriji interpretirati: to su tip , memorijska klasa varijable i doseg. Memorijska klasa varijable je najčešće određena implicitno položajem varijable u programu i stoga ćemo ju za sada zanemariti (vidi sekciju 9.2 gdje će biti više riječi od memorijskoj klasi i dosegu).

Prije upotrebe varijablu je potrebno deklarirati. Deklaracija određuje ime i tip varijable i ima sljedeći oblik:

```
tip ime;
```

gdje je `tip` varijable, a `ime` njeno ime. Na primjer, u deklaracijama

```
int a,b;
unsigned c;
char d;
```

`a` i `b` su cjelobrojne varijable, `c` je cjelobrojna varijabla bez predznaka, a `d` je znakovna varijabla. Varijable istog tipa moguće je deklarirati u istom retku kao u primjeru

```
short a,b,c;
```

ili svaku zasebno

```
short a;
short b;
short c;
```

Varijable se deklariraju na početku funkcije, prije prve izvršne naredbe. Moguće je varijablu deklarirati i izvan svih funkcija, obično na početku datoteke i tada takve varijable nazivamo vanjskim varijablama (vidi sekciju 9.1). Kada je varijabla deklarirana unutar neke funkcije (npr. `main` ili neke druge) onda se na mjestu deklaracije rezervira memorijski prostor za varijablu. Deklaracija varijable je tada ujedno i njena definicija. Kod vanjskih varijabli pojam deklaracije i definicije varijable mogu se razlikovati kao što ćemo vidjeti u sekciji 9.1. Varijable definirane unutar neke funkcije i izvan svih funkcija razlikuju se po svom dosegu. Doseg varijable je dio programa u kojem je varijabla vidljiva, odnosno, dio programa u kojem joj se može pristupiti putem njenog imena. Doseg varijable definirane u nekoj funkciji je tijelo te funkcije. Iz drugih funkcija toj se varijabli ne može pristupiti. Nasuprot tome, varijabli deklariranoj izvan svih funkcija (vanjskoj varijabli) može se pristupiti iz svake funkcije (vidi sekciju 9.1).

3.5.1 Polja

Polje je niz varijabli istog tipa indeksiranih cjelobrojnim indeksom koji se kreće u rasponu od `0` do `n-1`, gdje je `n` broj elemenata polja. Ako je npr. `vektor` polje od 10 elementa nekog tipa, onda su elementi polja `vektor[0]`, `vektor[1]`, `vektor[2]`,..., `vektor[9]`. Za indeksiranje polja koristimo uglate zagrade i početna vrijednost indeksa je uvijek nula.

Deklaracija polja ima oblik:

```
tip ime[dimenzija];
```

gdje je `tip` tip podatka, `ime` ime polja, a `dimenzija` broj elemenata polja. Na primjer, polje `vektor` od 10 realnih brojeva jednostruke preciznosti deklarira se naredbom

```
float  vektor[10];
```

Deklaracija polja može biti dana zajedno s deklaracijama drugih varijabli kao u primjeru

```
float  x,vektor[10],y;
```

u kome su `x` i `y` varijable tipa `float`, a `vektor` polje od 10 elemenata tipa `float`.

3.5.2 Pokazivači

Svakoj varijabli u programu pridružena je određena memorijska lokacija koju možemo dohvatiti pomoću adresnog operatora `&`. Ako je `v` neka varijabla, onda je `&v` njena adresa u memoriji. Adrese se pohranjuju i s njima se manipulira pomoću posebnih varijabli koje se nazivaju pokazivači (pointeri). Ukoliko želimo memorirati adresu npr. cjelobrojne varijable, moramo deklarirati pokazivač na cjelobrojnu varijablu. Sintaksa deklaracije je

```
tip_p *ime;
```

gdje je `ime`, ime pokazivača, a `*` u deklaraciji označava da identifikator `ime` nije varijabla tipa `tip_p` već pokazivač na varijablu tipa `tip_p`. Na primjer, u kôdu

```
int v;
int *pv;
.....
pv=&v;
```

deklarirana je cjelobrojna varijabla `v` i pokazivač na cjelobrojnu varijablu, `pv`. U liniji kôda

```
pv=&v;
```

u pokazivač `pv` spremljena je adresa varijable `v`.

Deklaracije varijabli nekog tipa i pokazivača na isti tip mogu se pisati u jednom retku kao u primjeru

```
float u,*pu;
```

gdje je deklarirana varijabla `u` tipa `float` i pokazivač `pu` na `float`.

Sadržaj memorijske lokacije može se dohvatiti pomoću pokazivača. Ako je `pv` varijabla tipa pokazivač na neki tip podatka, recimo `char`, onda je `*pv` znak spremljen na tu lokaciju. Tako možemo imati

```
char *pv;
char v;
.....
v='a';
*pv='b';
printf("Na adresi %p smjesten je znak %c\n",&v,v);
printf("Na adresi %p smjesten je znak %c\n",pv,*pv);
.....
```

Vidimo da `*` ima dvije različite uloge: u deklaraciji varijable ona označava da je varijabla pokazivačkog tipa, a u izvršnim naredbama ona predstavlja operator dereferenciranja koji primijenjen na pokazivač daje vrijednost na koju pokazivač pokazuje. Sintaksa deklaracije pokazivača je u skladu s primjenom `*` kao operatora dereferenciranja tako što deklaracija tipa

```
char *pv;
```

sugerira da je `pv` nakon dereferenciranja (`*pv`) tipa `char`.

Napomena. Prethodni primjer već ukazuje na opasnost koja se javlja upotrebom pokazivača. Pomoću pokazivača moguće je unutar programa pristupiti svakoj memorijskoj lokaciji koju je operacijski sustav dodijelio programu. Tu se krije mogućnost (namjerne ili nenamjerne) korupcije memorije. U gornjem primjeru nije pokazana linija kôda u kojoj je pokazivač inicijaliziran. Pretpostavimo stoga da smo ga zaboravili inicijalizirati prije dereferenciranja

```
*pv='b';
```

Na koju memorijsku lokaciju će biti upisan znak `'b'`? Precizan odgovor ovisi o memorijskoj klasi pokazivača (vidi sekciju 9.2), no u slučaju da `pv` sadrži neku slučajnu vrijednost naš će program pokušati pisati na slučajno odabranu adresu. Posljedice toga mogu biti najrazličitije, od prekida programa do pogrešnog funkcioniranja. Radi se, u svakom slučaju, o tipu greške koju je teško otkriti. □

3.6 Konstante

C poznaje cjelobrojne, realne i znakovne konstante te konstantne znakovne nizove.

Cjelobrojne konstante mogu biti zapisane u tri brojevna sustava: decimalnom (baza 10), oktalnom (baza 8) i heksadecimalnom (baza 16).

Decimalne cjelobrojne konstante sastavljene su od decimalnih znamenki 0 do 9. Ako konstanta sadrži više od jedne znamenke prva ne smije biti jednaka nuli. Primjeri su

```
0     1     234     -456     99999
```

Decimalna točka nije dozvoljena, tako da 10.0 nije cjelobrojna konstanta.

Oktalne cjelobrojne konstante dobivaju se kombinacijom decimalnih znamenki 0 do 7. Prva znamenka mora uvijek biti 0 što signalizira da se radi o broju zapisanom u oktalnom sustavu. Primjeri su

```
0     01     -0651     077777
```

Decimalna točka nije dozvoljena.

Heksadecimalne cjelobrojne konstante započinju s 0x ili 0X i sastoji se od kombinacije znamenki 0 do 9 i slova a do f (mala ili velika). Interpretacija slova je sljedeća:

$$\begin{aligned} \texttt{a} &= 10 \\ \texttt{b} &= 11 \\ \texttt{c} &= 12 \\ \texttt{d} &= 13 \\ \texttt{e} &= 14 \\ \texttt{f} &= 15 \end{aligned}$$

Primjeri su

```
0x0     0x1     -0x7FFF     0xabcd
```

Decimalna točka nije dozvoljena.

Sve numeričke konstante moraju biti unutar granica određenih tipom podatka koji predstavljaju. Cjelobrojne konstante moraju stoga biti unutar granica određenih tipom `int`. Konstante tipa `unsigned`, `long` i `unsigned long` mogu poprimiti vrijednosti veće od običnih cjelobrojnih konstanti i stoga moraju biti posebno označene.

Konstanta tipa `long` formira se tako da se na kraj cjelobrojne konstante doda slovo `L` (veliko ili malo); konstanta tipa `unsigned` formira se dodavanjem slova `U` (veliko ili malo). Konstanta tipa `unsigned long` formira se dodavanjem slova `UL` (veliko ili malo). Primjeri su

Konstanta	Tip
500000U	`unsigned` (decimalna)
123456789L	`long` (decimalna)
123456789ul	`unsigned long` (decimalna)
0123456l	`long` (oktalna)
0X50000U	`unsigned` (heksadecimalna)
-5000ULL	`long long` (decimalno)
50000ULL	`unsigned long long` (decimalno)

`U` i `L` mogu se kombinirati u bilo kojem poretku. Znakovi konverzije za `printf` funkciju vide se u sljedećem primjeru:

```
unsigned            a=20U;
long                b=2000L;
unsigned long       c=200000LU;
long long           d=20000000LL;
unsigned long long  e=2000000000ULL;

printf("unsigned           = %u\n",a);
printf("long               = %Ld\n",b);
printf("unsigned long      = %lu\n",c);
printf("unsigned long      = %lo\n",c);
printf("long long          = %lld\n",d);
printf("unsigned long long = %llu\n",e);
```

Znakovne konstante. Znakovna konstanta je jedan znak u jednostrukim navodnicima. Npr.

```
'A'     'x'     '5'     '?'     ' '
```

(uočimo da je i bjelina znak). Znakovne konstante imaju svoje cjelobrojne vrijednosti koje su određene načinom kodiranja znakova. Korištenjem konstanti poput `'a'`, `'3'` postiže se neovisnost o načinu kodiranja znakova.

Jedan broj posebnih znakova može se reprezentirati u C-u pomoću dva znaka: prvi je obrnuta kosa crta \(eng. backslash), a drugi je neko od slova ili znakova. Često korišteni posebni znakovi su

Kod	Značenje
\b	povratak za jedno mjesto unazad (backspace)
\f	nova stranica (form feed)
\n	novi red (new line)
\r	povratak na početak linije (carriage return)
\t	horizontalni tabulator
\v	vertikalni tabulator
\0	nul znak (null character)
\?	upitnik
\"	navodnik
\'	jednostruki navodnik
\\	obrnuta kosa crta (backslash)

Na primjer,

```
'\n'    '\b'    '\\'
```

Znakovne konstante mogu biti zadane svojim kodom koji se piše u obliku `\ooo`, gdje je `ooo` troznamenkasti oktalni broj. Na primjer `'\170'` je znak koji ima kod 170 oktalno, odnosno 120 decimalno (x u ASCII znakovima). Slično, kodovi mogu biti zadani i heksadecimalno u obliku `\xoo`, gdje `oo` dvoznamenkasti heksadecimalni broj.

Realna konstanta je broj zapisan u dekadskom sustavu s decimalnom točkom. Npr.

```
0.    1.    -0.2    5000.
```

Realne se konstante mogu pisati i s eksponentom i tada decimalna točka nije potrebna. Eksponent mora biti cijeli broj kome prethodi slove e (malo ili veliko). Na primjer, broj 3×10^5 može biti napisan na sljedeće načine:

```
300000.    3e5    3E+5    3.0e+5    .3e6    30E4
```

Uočimo da ne smije biti razmaka između mantise i eksponenta (npr. `3.0 e+5` nije ispravno napisana konstanta).

Tip svih realnih konstanti je `double`. Ukoliko se želi konstanta tipa `float` na kraj joj se dodaje `f` ili `F`. Slično, realna konstanta koja nakraju ima `l` ili `L` je tipa `long double`:

```
0.00fe-3f    1.345E+8F    -0.2e3l    5000.0L
```

Kontrolni znakovi u `printf` funkciji mogu se vidjeti u sljedećem primjeru:

```
float       x=-1.0F;
double      y=2.0;
long double z=-0.2e8l;
long double w=-0.2e-8L;

printf("float       = %f\n",x);
printf("double      = %f\n",y);
printf("long double = %Lf\n",z);
printf("long double = %Le\n",w);
```

Kompleksne konstante se formiraju kao kombinacija realnih konstanti i imaginarne jedinice `_Complex_I` (ili `I`), definirane u `<complex.h>`.

Konstantni znakovni nizovi (konstantni *stringovi*) su nizovi znakova navedeni unutar (dvostrukih) navodnika. Na primjer,

```
"Zagreb"    "01/07/2001"    "Linija 1\nLinija 2\nLinija3"
```

Specijalni znakovi kao `\n` i `\t` mogu biti uključeni u konstantne znakovne nizove. Svakom znakovnom nizu automatski se na kraj dodaje nul-znak (`\0`) kao oznaka kraja znakovnog niza. Na taj način algoritmi koji rade sa znakovnim nizovima ne trebaju poznavati broj znakova već prepoznaju kraj niza pomoću nul znaka `\0`. To ima za posljedicu da su `'a'` i `"a"` dva različita objekta. Prvo je jedan znak koji sadrži slovo `a`, a drugo je znakovni niz koji sadrži slovo `a`, dakle niz od dva znaka: `a` i `\0`. Prazan niz znakova "" sadrži samo nul-znak.

Ako je konstantan znakovni niz suviše dugačak da bi stao u jednu liniju, onda imamo dvije mogućnosti. Možemo ga produžiti u sljedeću liniju ako ispred prijelaza u novi red stavimo znak \. Prevodilac će tada ignorirati kombinaciju znaka \ i znaka za prijelaz u novi red. Druga mogućnost je iskoristiti svojstvi automatskog nadovezivanja konstantnih znakovnih nizova: ako se dva takva niza nalaze jedan pored drugog, onda će ih prevodilac automatski nadovezati. U sljedećem primjeru oba stringa su jednako inicijalizirana:

```
 char s1[]="Vrlo dugacak \
niz znakova";
 char s2[]="Vrlo dugacak "
           "niz znakova";
```

Simboličke konstante su imena koja preprocesor (vidi Poglavlje 8) substituira nizom znakova. Definiraju se na početku programa a sintaksa im je

```
#define ime  tekst
```

gdje je `ime` ime simboličke konstante, a `tekst` niz znakova koji će biti substituiran na svakom mjestu u programu na kojem se pojavljuje `ime`. Obično se na taj način definiraju konstante kao npr.

```
#define PI  3.141593

#define TRUE  1
#define FALSE 0
```

Ipak, u tu svrhu bolje je koristiti `const` varijable (sekcija 3.7).

3.7 Inicijalizacija varijabli

Varijable se mogu inicijalizirati u trenutku deklaracije. Sintaksa je

```
tip varijabla = konstantni izraz;
```

Na primjer,

```
int a=7,b;
unsigned c=2345;
char d='\t';
```

gdje su inicijalizirane varijable `a`, `c` i `d`, dok `b` nije inicijalizirana.

Sve deklaracije varijabli mogu nositi kvalifikator `const` koji indicira da varijabla neće nikad biti mijenjana, tj. da se radi o konstanti. Npr.

```
const double e=2.71828182845905;
```

Prevodilac neće dozvoliti izmjenu varijable `e` u preostalom dijelu programa.

Polja znakova mogu se inicijalizirati konstantnim nizovima znakova. Tako je deklaracijom

```
char tekst[]="Inicijalizacija";
```

definiran niz znakova `tekst` koji ima 16 znakova, 15 za riječ `Inicijalizacija` i jedan za nul-znak koji dolazi na kraju niza znakova. Uočimo da nismo morali eksplicitno navesti dimenziju polja `tekst`. Ona će biti izračunata iz stringa `"Inicijalizacija"`. Alternativno smo mogli napisati

```
char tekst[16]="Inicijalizacija";
```

Općenito, polja se mogu inicijalizirati navođenjem vrijednosti elemenata polja unutar vitičastih zagrada:

```
double x[]={1.2,3.4,-6.1};
```

Dimenzija polja bit će izračunata na osnovu broja konstanti unutar zagrada. Nakon ove deklaracije imat ćemo

```
x[0]=1.2, x[1]=3.4, x[2]=-6.1
```

3.8 Enumeracije

Enumeracije predstavljaju alternativu uvođenju simboličkih konstanti putem preprocesorske direktive `#define`. Pomoću enumeracije deklariramo simbolička imena koja primaju cjelobrojne vrijednosti i čija je osnovna svrha povećanje čitljivosti programa. Na primjer, nakon deklaracije

```
enum {FALSE, TRUE};
```

imena `FALSE` i `TRUE` možemo koristiti u programu kao cjelobrojne konstante. `FALSE` ima vrijednost 0, a `TRUE` 1. To ekvivalentno preprocesorskim naredbama

```
#define  FALSE 0
#define  TRUE  1
```

Princip je da se svakom imenu deklariranom u enumeraciji pridruži jedan cjeli broj: prvom se pridruži nula, drugom 1 itd. Tako smo doobili `FALSE`=0, `TRUE`=1.

Enumeraciji možemo dati ime i tada možemo deklarirati varijable tipa enumeracije. To su varijable koje primaju samo konačno mnogo vrijednosti navedenih u enumeraciji. Prethodnu enumeraciju smo mogli deklarirati kao

```
enum boolean {FALSE, TRUE};
```

Varijable `x` i `y` tipa `boolean` definiraju se naredbom

```
enum boolean x,y;
```

Možemo ih koristiti na sljedeći način:

```
x=FALSE;
........
if(x==TRUE) y=FALSE;
```

itd. Varijable tipa `enum` pridonose razumljivosti programa. Koristimo ih za one varijable koje primaju konačno mnogo vrijednosti i za koje je poželjno uvesti simbolička imena.

Općenito se enumeracija deklarira naredbom

```
enum ime {clan_1, clan_2,....,clan_n};
```

gdje je `ime` ime koje dajemo enumeraciji, a `clan_1, clan_2,....,clan_n` predstavljaju identifikatore koji se mogu pridružiti varijabli tipa `enum ime`. Identifikatori moraju biti međusobno različiti i različiti od drugih identifikatora u dosegu enumeracije. Identifikatorima se automatski pridružuju cjelobrojne vrijednosti

```
clan_1=0
clan_2=1
clan_3=2
.......
clan_n=n-1
```

Varijabla tipa enumeracije deklarira se naredbom

```
enum ime var_1, var_2,....,var_m;
```

Deklaracije enumeracije i varijabli tog tipa mogu se spojiti:

```
enum ime {clan_1, clan_2,....,clan_n}
                    var_1, var_2,....,var_m;
```

Vrijednosti koje se dodijeljuju identifikatorima mogu se modificirati kao u sljedećem primjeru

```
enum ESC {novi_red='\n', tabulator='\t'};
```

ili

```
enum boje {plavo=-1,zuto,crveno,zeleno=0,ljubicasto,bijelo};
```

Time dobivamo

```
plavo=-1
zuto=0
crveno=1
zeleno=0
ljubicasto=1
bijelo=2
```

Vidimo da će se u ovom slučaju neke vrijednosti ponoviti.

Poglavlje 4

Operatori i izrazi

U ovom se poglavlju bavimo operatorima i izrazima koji se formiraju pomoću operatora i operanada. Izostavit ćemo operatore vezane uz strukture (Poglavlje 12), pokazivače (Poglavlje 11) te operatore koji djeluju na bitovima (Poglavlje 14). Pažnju ćemo posvetiti načinu izračunavanja izraza.

4.1 Aritmetički operatori

Programski jezik C poznaje pet aritmetičkih operatora:

Operator	Značenje
`+`	zbrajanje
`-`	oduzimanje
`*`	množenje
`/`	dijeljenje
`%`	modulo

- Aritmetički operatori djeluju na numeričkim operandima tj. cjelobrojnim, realnim i znakovnim operandima (znakovna konstanta ima cjelobrojni tip).

- Operacije dijeljenja u slučaju cjelobrojnih operanada ima značenje *cjelobrojnog dijeljenje*. Rezultatu dijeljenja odsjecaju se decimalne znamenke kako bi se dobio cjelobrojni rezultat. Ukoliko je rezultat dijeljenja negativan, onda prema standardu C90 način zaokruživanja rezultata ovisi o implementaciji. Najčešće je to odsjecanje, tj. zaokruživanje prema nuli. Standard C99 je propisao da se i u slučaju negativnih brojeva vrši odsjecanje, tj. zaokruživanje prema nuli. Ako bar jedan od operanada nije cjelobrojan, onda je rezultat dijeljenja realan broj. Nazivnik pri dijeljenju mora biti različit od nule.

- Operacija modulo (%) djeluje na cjelobrojnim operandima i kao rezultat daje ostatak pri cjelobrojnom dijeljenju operanada. Na primjer, ako je $x = 10$ i $y = 3$, onda je $x/y = 3$ (cjelobrojno dijeljenje) te $x\%y = 1$. Uvijek mora vrijediti da je $(x/y) * y + (x\%y)$ jednako x, tako da ponašanje operatora `%` na negativnim operandima ovisi o ponašanju operatora `/`.

- Operator potenciranja u C-u ne postoji. Umjesto njega koristi se funkcija `pow` iz matematičke biblioteke.

Primjer. Uzmimo da je na primjer $x = 13$ i $y = -3$. Tada je prema standardu C99 $x/y = -4$ (odsjecanje) te $x\%y = 1$ (predznak prvog operanda). Stoga je $(x/y) * y + (x\%y) = 13 = x$.

4.1.1 Konverzije

Kada u aritmetičkom izrazu sudjeluju operandi različitih tipova onda prije izračunavanja izraza dolazi konverzije operanada u zajednički tip prema određenim pravilima. Općenito operandi će biti konvertirani u najprecizniji tip prisutan među operandima i rezultat aritmetičkog izraza bit će tog tipa. Na primjer, ako je `x` varijabla tipa `double`, a `n` varijabla tipa `int` u izrazu `x+n`, prije zbrajanja doći će do konverzije varijable `n` u tip `double` i vrijednost izraza bit će tipa `double`.

Vrijede sljedeća pravila:

- `short` i `char` (s predznakom ili bez) automatski se konvertiraju u `int` prije svake aritmetičke operacije. U slučaju da je `short` isto što i `int`, onda će `unsigned short` biti širi od `int`, pa će operandi biti konvertirani u `unsigned int`.

- U svakoj operaciji koja uključuje operanda različitih tipova prije izvršenja operacije vrši se konverzija operanada u najširi tip.

- Tipovi su prema širini poredani na sljedeći način (od najšireg prema najužem): `long double`, `double`, `float`, `unsigned long long`, `long long`, `unsigned long`, `long`, `unsigned int` i `int`. Jedina iznimka je kad su `long` i `int` isti. Tada je `unsigned int` širi od `long`. Uži tipovi se ne pojavljuju jer se oni automatski konvertiraju u `int` ili `unsigned int` prema prvom pravili.

Prilikom ovih konverzija ne dolazi do gubitka informacije osim eventualno u slučaju kada se cjelobrojni tip konvertira u realni.

Konverzija se dešava i kod operacije pridruživanja (vidi sekciju 4.4) ako su operandi s lijeve i desne strane različitog tipa. Operand na desnoj strani konvertira se u tip operanda na lijevoj strani. Pri tome može doći do gubitka informacije ako se konvertira širi tip u uži. Na primjer, ako je `x` varijabla tipa `float` i `n` varijabla tipa `int` prilikom pridruživanja `i=x` doći će do odsjecanja decimala u broju `x`.

Konverzija se dešava prilikom prenošenja argumenata funkciji (vidi sekciju 7.3). Naime, argumenti su izrazi i prije njihovog izračunavanja dolazi do konverzije ukoliko je ona potrebna. Pri tome vrijede različita pravila ovisno o tome da li funkcija ima prototip ili ne.

- Ako funkcija nema prototipa, onda se svaki argument tipa `char` i `short` transformira u `int`, a `float` u `double`.
- Ukoliko funkcija ima prototip, onda se svi argumenti pri pozivu konvertiraju (ako je to potrebno) u tipove deklarirane u prototipu (vidi sekciju 7.2).

Vrijednost nekog izraza može se eksplicitno konvertirati u zadani tip pomoću operatora eksplicitne konverzije `(tip)` (*eng. cast operator*). Sintaksa konverzije je

```
(tip) izraz;
```

`izraz` će biti izračunat i njegova vrijednost konvertirana u tip unutar zagrada. U primjeru

```
double x;
float y;
........
x= (double) y;
```

`y` je eksplicitno konvertiran u tip `double` prije pridruživanja. Eksplicitne konverzije mogu biti nužne u slučajevima kao što je sljedeći:

```
int i;
double x;
((int) (i+x))%2;
```

4.1.2 Prioriteti i asocijativnost

Svi su operatori hijerhijski grupirani prema svom prioritetu. Operacija iz grupe s višim prioritetom izvršit će se prije operacija s nižim prioritetom. Upotrebom zagrada moguće je promijeniti redoslijed vršenja operacija.

Operacije `*`, `/` i `%` spadaju u jednu prioritetnu grupu, dok operacije `+` i `-` spadaju u nižu prioritetnu grupu (vidi sekciju 4.6).

Tako na primjer u izraz

```
2+4/2
```

dijeljenje će biti izvršeno prije zbrajanja, pa je vrijednost izraza 4, a ne 3 što bi bilo da se prvo izvrši zbrajanje. Ukoliko želimo zbrajane izvršiti prije dijeljenja treba pisati

```
(2+4)/2
```

što daje 3.

Zagrade koristimo kako bismo grupirali operande oko operatora. Ukoliko ih nema, onda se grupiranje vrši oko operatora najvišeg prioriteta. Kada više operatora ima isti, najviši, prioritet, onda način izračunavanja izraza ovisi o asocijativnosti. Asocijativnost može biti slijeva na desno ili zdesna na lijevo. Ako imamo, na primjer, dva operatora istog prioriteta, čija je asocijativnost slijeva na desno, onda će se operandi prvo grupirati oko lijevog operanda. Na primjer,

```
a - b + c    je ekvivalntno s  (a - b) + c
```

4.2 Unarni operatori

C ima jedan broj operatora koji djeluju na jednom operandu koje stoga nazivamo unarni operatori.

Konstante i varijable mogu imati ispred sebe minus kao u primjerima `-345, -(x+y)`, gdje minus daje negativan predznak konstanti i mijenja predznak varijable. Takav minus nazivamo unarni minus za razlike od binarnog minusa koji predstavlja operaciju oduzimanja.

Operator konverzije tipa

```
(tip)  izraz
```

kojeg smo sreli u sekciji 4.1 također je unarni operator.

Svi unarni operatori spadaju u istu prioritetnu grupu i imaju viši prioritet od aritmetičkih operatora. U izrazu

```
-x+y
```

će se stoga prvo izvršiti unarni minus, a zatim zbrajanje.

Asocijativnost unarnih operatora je zdesna na lijevo što će biti važno kad uvedemo preostale unarne oparatore `!`, `~`, `*` i `&`.

4.2.1 Inkrementiranje, dekrementiranje

Operator inkrementiranja ++ povećava varijablu za jedan. Tako je izraz

```
x++;
```

ekvivalentan izrazu

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

Operator dekrementiranja -- smanjuje vrijednost varijable za 1, odnosno

```
x--;
```

je ekvivalentno izrazu

```
x=x-1;
```

Izraz `x++` i `x--` mogu se pojaviti kao elementi složenijih izraza i zbog toga imaju dvije forme. Možemo pisati `x++` ili `++x` i isto tako `x--` ili `--x`. U oba slučaja varijabla `x` se povećava (umanjuje) za 1. Razlika između prefiks i postfiks notacije pojavljuje se u složenim izrazima.

- U prefiks notaciji (`++x, --x`) varijabla će biti promijenjena prije no što će njena vrijednost biti iskorišten u složenom izrazu;
- U postfiks notaciji (`x++, x--`) varijabla će biti promijenjena nakon što će njena vrijednost biti iskorišten u složenom izrazu.

Na primjer, ako je `x=3`, onda izraz

```
y=++x;
```

daje `y=4, x=4`, a izraz

```
y=x++;
```

daje `y=3, x=4`.

Argumenti funkcije su izrazi koji se izračunavaju prije no što se njihova vrijednost prenese u funkciju. Prefiks i postfiks notacija operatora inkrementiranja/dekrementiranja stoga i ovdje ima isto značenje. U primjeru

```
i=7;
printf("i= %d\n", --i);
printf("i= %d\n", i--);
printf("i= %d\n", i);
```

prva `printf` naredba ispisat će 6, druga će također ispisati 6, a treća će ispisati 5. Razlog je to što se u drugoj `printf` naredbi vrijednost varijable `i` prvo prenese funkciji, a zatim smanji za 1.

4.2.2 `sizeof` operator

`sizeof` operator vraća veličinu svog operanda u bajtovima. U C-u se jedan bajt definira kao broj bitova potreban za pamćenje podatka tipa `char`. To je na večini računala jedan oktet (osam bitova).

Operand može biti izraz ili tip podatka. Na primjer, kôd

```
int i;
float x;
printf("Velicina tipa int   = %d\n", sizeof(i));
printf("Velicina tipa float = %d\n", sizeof(x));
```

bi na nekim sustavima ispisao

```
Velicina tipa int   = 4
Velicina tipa float = 4
```

Isti efekt postižemo ako `sizeof` primijenimo na tip podatka:

```
printf("Velicina tipa int   = %d\n", sizeof(int));
printf("Velicina tipa float = %d\n", sizeof(float));
```

Kod složenijih podataka dobivamo ukupan broj okteta koji podatak zauzima. Ako imamo

```
char tekst[]="Dalmacija";
```

naredba

```
printf("Broj znakova u varijabli tekst =%d\n",
                                   sizeof(tekst));
```

daje

```
Broj znakova u varijabli tekst =10
```

Operator `sizeof` vraća cjelobrojnu vrijednost bez predznaka koja ovisi o implementaciji. Taj je tip definiran u datoteci zaglavlja `<stddef.h>` i zove se `size_t`.

4.3 Relacijski i logički operatori

Četiri relacijska operatora su

Operator	Značenje
`<`	strogo manje
`<=`	manje ili jednako
`>`	strogo veće
`>=`	veće ili jednako

Oni imaju isti prioritet, niži od prioriteta aritmetičkih i unarnih operatora, a asocijativnost im je slijeva na desno.

Operatori jednakosti su

Operator	Značenje
`==`	jednako
`!=`	različito

Oni spadaju u zasebnu prioritetnu grupu s manjim prioritetom od relacijskih operatora i asocijativnost im je slijeva na desno.

Pomoću ovih šest operatora formiraju se logički izrazi. Njihova vrijednost je istina ili laž. Kako u ANSI C-u (C90) ne postoji poseban logički tip podatka, logički izrazi su tipa `int`. (Standard C99 je uveo tip `_Bool`). Logički izraz koji je istinit dobiva vrijednost 1, a logički izraz koji je lažan prima vrijednost 0. Na primjer:

Izraz	Istinitost	Vrijednost
`i<j`	istinito	1
`(i+j)>=k`	neistinito	0
`i==2`	neistinito	0
`k!=i`	istinito	1

Budući da je prioritet logičkih operatora niži od prioriteta aritmetičkih operatora izraz poput

```
i >= 'A'-'a'+1
```

ekvivalentan je s

```
i >= ('A'-'a'+1)
```

Složeniji logički izrazi formiraju se pomoću logičkih operatora:

Operator	Značenje
`&&`	logičko I
`||`	logičko ILI
`!`	logička negacija (unarno)

Operandi logičkih operatora su logičke vrijednosti (najčešće logički izrazi) s tim da se svaka cjelobrojna vrijednost različita od nule interpretira kao istina, a nula se interpretira kao laž. Vrijednost složenog logičkog izraza bit će 0 (laž) ili 1 (istina).

Svaki od ova dva operatora ima svoju prioritetnu grupu i prioritet im je niži od prioriteta relacijskih operatora i operatora jednakosti. Asocijativnost im je slijeva na desno.

Primjer. Ako je izraz `i>1` istinit, izraz `c=='t'` istinit, a izraz `j<6` lažan, onda imamo:

Izraz	Istinitost	Vrijednost
`i>1 || j<6`	istinito	1
`i>1 && j<6`	neistinito	0
`!(i>1)`	neistinito	0
`i>1 || (j<6 && c!='t')`	istinito	1

Logički izrazi koji se sastoje od individualnih logičkih izraza povezanih logičkim operatorima `&&` i `||` izračunavaju se slijeva na desno. Izraz se prestaje izračunavati kad je njegova vrijednost poznata. To je važnu u sljedećim situacijama: pretpostavimo da je `x` polje tipa `char` dimenzije `n`. Tada u izrazu

```
for(i=0; i<n && x[i] !='a'; ++i) {
....
}
```

neće doći do ispitivanja `x[i] !='a'` ako je `i=n` (granica polja `x` neće biti prijeđena). S druge strane, kôd

```
for(i=0; x[i] !='a' && i<n; ++i) {
....
}
```

bi ispitivao da li je `x[n]` različito od `'a'`. To može dovesti do nepredvidivog ponašanja programa.

Logička negacija koristi se često u `if` naredbama. Konstrukcija

```
if(!istina)
```

ekvivalentan je s

```
if(istina==0)
```

i budući da je kraća (za dva znaka) često se koristi.

4.4 Operatori pridruživanja

Osnovni operator pridruživanja je =. Naredba pridruživanja je oblika

```
varijabla = izraz;
```

Izraz na desnoj strani će biti izračunat i njegova vrijednost se zatim pridružuje varijabli na lijevoj strani. Na primjer,

```
i=2;
a=3.17;
c='m';
```

Pridruživanje `varijabla = izraz` je ustvari izraz i stoga može biti dio kompleksnijeg izraza. Preciznije, izraz `varijabla = izraz` ima vrijednost varijable na lijevoj strani nakon što se izvrši pridruživanje. Stoga je moguće koristiti pridruživanje unutar drugih naredbi kao u primjeru

```
while((a=x[i])!=0){
      ....
      ++i;
}
```

gdje se u testu `while` narebe prvo `x[i]` pridruži varijabli `a`, a zatim se testira da li je dobiveni izraz, a to je vrijednost varijable `a`, različit od nule.

Ovakva mogućnost može voditi na teško uočljive greške kao u slučaju ako napišemo

```
if(varijabla = izraz) .....;
```

umjesto

```
if(varijabla == izraz) .....;
```

U prvoj `if` naredbi prvo će se vrijednost izraza pridružiti varijabli, a zatim će se tijelo `if` naredbe izvršiti ako je varijabla različita od nule. U drugoj `if` naredbi vrijednost varijable se ne mijenja već se samo uspoređuje s vrijednošću koju daje izraz. Tijelo `if` naredbe se izvršava ako su te dvije vrijednosti jednake.

Operatore pridruživanja moguće je ulančati i pisati

```
varijabla1 = varijabla2 = varijabla3 = ... = izraz;
```

Niz pridruživanja ide zdesna na lijevo. To znači da se `izraz` jednom izračuna i zatim se redom pridružuje varijablama .. `varijabla3, varijabla2, varijabla1`. Na primjer,

```
x = y = cos(3.22);
```

je ekvivalentno s

```
x = (y = cos(3.22));
```

odnosno

```
y = cos(3.22);
x=y;
```

Složeni operatori pridruživanja su

```
+=   -=   *=   /=   %=
```

Općenito izraz oblika

```
izraz1 op= izraz2;
```

gdje je `op` jedna od operacija `+, -, *, /, %`, ekvivalentan je s

```
izraz1 = izraz1 op izraz2;
```

Ovi operatori spadaju u istu prioritetnu grupu s operatorom `=` i izračunavaju se zdesna na lijevo. Prioritet im je manji od prioriteta aritmetičkih i logičkih operatora.

Primjeri:

Izraz	Ekvivalentan izraz
`i +=5`	`i=i+5`
`i -=j`	`i=i-j`
`i *=j+1`	`i=i*(j+1)`
`i /=4`	`i=i/4`
`i %=2`	`i=i%2`

4.5 Uvjetni operator : ?

Uvjetni izraz je izraz oblika

```
izraz1 ? izraz2 : izraz3;
```

U njemu se prvo izračunava `izraz1`. Ako je on istinit (različit od nule) onda se izračunava `izraz2` i on postaje vrijednost čitavog uvjetnog izraza; ako je `izraz1` lažan (jednak nuli) onda se izračunava `izraz3` i on postaje vrijednost čitavog uvjetnog izraza. Na primjer,

```
double a,b;
  ........
(a<b) ? a : b;
```

je izraz koji daje manju vrijednost od brojeva `a` i `b`. Vrijednost uvjetnog izraza može se pridružiti nekoj varijabli, npr.

```
double a,b,min;
   ........
min=(a<b) ? a : b;
```

Uvjetni operator ima nizak prioritet tako da zagrade oko prvog, drugog i trećeg izraza najčešće nisu potrebne.

4.6 Prioriteti i redoslijed izračunavanja

U sljedećoj tabeli sumirani su prioriteti i asocijativnost svih operatora. Neke od tih operatora još nismo uveli.

<table>
<tr><th>Kategorija</th><th>Operatori</th><th>asocijativnost</th></tr>
<tr><td></td><td>() [] -> .</td><td>L → D</td></tr>
<tr><td>unarni op.</td><td>! ~ ++ -- - * & (type) sizeof</td><td>D → L</td></tr>
<tr><td>aritmetički op.</td><td>* / %</td><td>L → D</td></tr>
<tr><td>aritmetički op.</td><td>+ -</td><td>L → D</td></tr>
<tr><td>op. pomaka</td><td><< >></td><td>L → D</td></tr>
<tr><td>relacijski op.</td><td>< <= > >=</td><td>L → D</td></tr>
<tr><td>op. jednakosti</td><td>== !=</td><td>L → D</td></tr>
<tr><td>bit-po-bit I</td><td>&</td><td>L → D</td></tr>
<tr><td>bit-po-bit ex. ILI</td><td>^</td><td>L → D</td></tr>
<tr><td>bit-po-bit ILI</td><td>|</td><td>L → D</td></tr>
<tr><td>logičko I</td><td>&&</td><td>L → D</td></tr>
<tr><td>logičko ILI</td><td>||</td><td>L → D</td></tr>
<tr><td>uvjetni ? : op.</td><td>? :</td><td>D → L</td></tr>
<tr><td>op. pridruživanja</td><td>= += -= *= /= %=</td><td>D → L</td></tr>
<tr><td>zarez op.</td><td>,</td><td>L → D</td></tr>
</table>

Operatori istog prioriteta i asocijativnosti dani su u istoj liniji. Operatori u višoj liniji imaju viši prioritet od operatora u nižim linijama.

Primijetimo još da C ne propisuje redoslijed kojim se izračunavaju operandi nekog operatora osim u slučaju operatora `&&`, `||`, `? :` (i operatora "," kojeg još nismo uveli). Stoga u izrazu

```
x=f()+g();
```

C jezikom nije definirano koja će se funkcija prva izvršiti. Naredbe poput

```
printf("%d %d\n", ++n, n*n); /* greska */
```

mogu dati različite rezultate ovisno o stroju na kojem se izvršavaju.

Poglavlje 5

Ulaz i izlaz podataka

U ovom poglavlju uvodimo šest funkcija iz standardne ulazno-izlazne biblioteke: `getchar`, `putchar`, `gets`, `puts` `scanf` i `printf`. Program koji koristi neku od tih funkcija mora uključiti datoteku zaglavlja `<stdio.h>`.

5.1 Funkcije getchar i putchar

```
int getchar(void);
int putchar(int c);
```

Funkcija `getchar` čita jedan znak sa standardnog ulaza (tipično tipkovnice). Funkcija nema argumenata pa je sintaksa poziva:

```
c_var=getchar();
```

Funkcija `putchar` šalje jedan znak na standardni izlaz (tipično ekran). Ona uzima jedan argument (znak koji treba ispisati) i vraća cjelobrojnu vrijednost. Najčešće poziv funkcije ima oblik

```
putchar(c_var);
```

pri čemu se vraćena vrijednost ignorira.

Kada funkcija `getchar` naiđe na kraj ulaznih podataka vraća vrijednost `EOF` (skraćeno od eng. *End of File*). `EOF` je simbolička konstanta definirana u `<stdio.h>` koja signalizira kraj datoteke i kraj ulaznih podataka (ulaz je tretiran kao datoteka).

Konstanta `EOF` mora se razlikovati od znakova iz sustava znakova koje računalo koristi. Stoga funkcija `getchar` ne vraća vrijednost tipa `char` već vrijednost tipa `int` što daje dovoljno prostora za kodiranje konstante `EOF`.

Isto tako `putchar` uzima vrijednost tipa `int` i vraća vrijednost tipa `int`. Vraćena vrijednost je znak koji je ispisan ili `EOF` ako ispis znaka nije uspio.

Kao ilustraciju upotrebe funkcija `getchar` i `putchar` napišimo program koji kopira znak po znak ulaz na izlaz i pri tome sva slova pretvara u velika:

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

/* kopiranje ulaza na izlaz */

int main(void) {
    int c;

    c=getchar();
    while(c!=EOF) {
       putchar(toupper(c));
       c=getchar();
   }
   return 0;
}
```

Funkcija `toupper` je deklarirana u datoteci zaglavlja `<ctype.h>`. Ona uzima varijablu tipa `char` i ako se radi o slovu pretvara malo slovo u veliko. Sve druge znakove ostavlja na miru.

Uočimo da je za pamćenje znakovne varijable korišten tip `int` umjesto tipa `char`. U `while` petlji znakovi se čitaju sve dok se ne dođe do kraja ulaznih podataka (`EOF`). Pročitani znak se odmah ispisuje putem funkcije `putchar`, a vrijednost se koju `putchar` vraća ignorira. Funkciju `putchar` smo mogli pozvati i naredbom `i=putchar(c)` (`i` je cjelobrojna varijabla) koja bi vraćenu vrijednost smjestila u varijablu `i`.

Isti je program mogao biti napisan jednostavnije ubacivanjem poziva funkciji `getchar` u test `while` petlje:

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

/* kopiranje ulaza na izlaz */

int main(void) {
    int c;

    while((c=getchar())!=EOF)
```

```
        putchar(toupper(c));
}
```

Zagrade u testu `(c=getchar())!=EOF` su nužne jer bi `c=getchar()!=EOF`, zbog niskog prioriteta operatora pridruživanja, bilo interpretirano kao test `c=(getchar()!=EOF)`. Varijabla `c` bi tada dobila vrijednost 0 ili 1.

Ponašanje funkcije `getchar` ovisi o operacijskom sustavu na kojem se program izvršava. Operacijski sustav prikuplja podatke s tastature i šalje ih aplikacijskom programu najčešće liniju po liniju. Takav način rada omogućava operacijskom sustavu standardno editiranje ulazne linije (korištenje tipki *backspace* i *delete*) koje stoga ne mora biti ugrađeno u svaki program. Posljedica je toga da prvi poziv funkciji `getchar` blokira program sve dok se ne ukuca čitava ulazna linija i pritisne tipka RETURN (ENTER). Tada su svi učitani znakovi na raspolaganju funkciji `getchar`. Čitanje znaka odmah nakon što je otipkan ili onemogućavanje ispisa utipkanog znaka može se postići specifičnim tehnikama programiranja vezanim uz operacijski sustav koji se koristi (http://www.eskimo.com/ scs/C-faq/top.html).

Kada na ulazu nema podataka funkcija `getchar` čeka da se podaci utipkaju. Da bismo zaustavili čitanje moramo utipkati znak za "kraj datoteke". Pod UNIX-om to je znak `Ctr-D`, a pod DOSom `Ctr-Z`.

Funkciju `getchar` često koristimo za čitanje jednog znaka (npr. odgovora `y` ili `n`). U tim slučajevima moramo paziti da pročitamo sve znakove u liniji, uključivši znak za prijelaz u novi red, kako ne bi ostali za slijedeći poziv funkcije `getchar`. Na primjer, ovdje dajemo funkciju koja čita samo prvo slovo s ulaza:

```
int  get_first(void) {
   int ch;

   ch=getchar();
   while(getchar() != '\n')
       ;   // prazna naredba
   return ch;
}
```

5.2 Funkcije iz datoteke `<ctype.h>`

Datoteka zaglavlja `<ctype.h>` sadrži deklaracija niza funkcija koje služe testiranju znakova. Svaka od tih funkcija uzima jedan argument tipa `int` koji treba biti znak ili `EOF` i vraća vrijednost tipa `int` koja je različita od nule (istina) ako je uvjet ispunjen ili nula ako nije. Neke od funkcija su sljedeće:

`isalnum()`	Alfanumerički znak
`isalpha()`	Alfabetski znak
`isctrln()`	Kontrolni znak
`isdigit()`	Znamenka
`isgraph()`	Printabilni znak osim razmaka
`islower()`	Malo slovo
`isprint()`	Printabilni znak
`ispunct()`	Printabilni znak osim razmaka, slova i brojeva
`isspace()`	Razmak
`isupper()`	Veliko slovo

Pod razmakom smatramo: bjelinu, znak za novi red, znak *formfeed*, znak *carriage return*, tabulator i vertikalni tabulator (`' '`, `'\n'`, `'\f'`, `'\r'`, `'\t'`, `'\v'`).

Dvije funkcije omogućavaju konverziju velikih slova u mala i obratno. Ostale znakove ostavljaju na miru.

`int tolower(int c)`	Veliko slovo u malo
`int toupper(int c)`	Malo slovo u veliko

5.3 Funkcije gets i puts

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

Funkcije `gets` i `puts` služe čitanju i pisanju znakovnih nizova (stringova). Funkcija `gets` čita znakovni niz sa standardnog ulaza (tastature), a funkcija `puts` ispisuje znakovni niz na standardni izlaz (ekran).

Funkcija `gets` uzima kao argument znakovni niz u koji će biti učitan niz znakova s ulaza. Pri tome se znakovi s ulaza učitavaju sve dok se ne naiđe na kraj linije (`'\n'`) koji se zamijenjuje znakom `'\0'`. Funkcija vraća pokazivač na `char` koji pokazuje na učitani znakovni niz ili `NULL` ako se došlo do kraja ulaznih podataka ili se javila greška. Simbolička konstanta `NULL` definirana je `<stdio.h>` i njen iznos je 0. To je jedina cjelobrojna vrijednost koja se može pridružiti pokazivaču (vidi sekciju 11.3).

Funkcija `puts` uzima kao argument znakovni niz koji će biti ispisan na standardnom izlazu. Funkcija vraća broj ispisanih znakova ako je ispis uspio te `EOF` ako nije. Prije ispisa `puts` dodaje znak `'\n'` na kraju znakovnog niza.

Program koji kopira ulaz na izlaz liniju po liniju mogao bi biti napisan na ovaj način:

```
#include <stdio.h>

/* kopiranje ulaza na izlaz */
int main(void) {
    char red[128];

    while(gets(red)!=NULL)
           puts(red);
}
```

Unutar testa `while` petlje `gets` će pročitati ulaznu liniju i vratiti pokazivač različit od `NULL` ako ulaz nije prazan i ako nije došlo do greške. U tom slučaju izvršava se komanda `puts(red)` koja ispisuje učitanu liniju, a povratna vrijednost funkcije `puts` se zanemaruje.

Osnovni nedostatak funkcije `gets` je u tome što nije moguće odrediti maksimalni broj znakova koji će biti učitan. Ukoliko je broj znakova na ulazu veći od dimenzije polja koje je argument funkcije `gets` doći će do greške. Stoga je općenito bolje umjesto `gets` koristiti funkciju `fgets` (vidi sekciju 13.3).

5.4 Funkcija `scanf`

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

Funkcija `scanf` služi učitavanju podataka sa standardnog ulaza. Opća forma funkcije je

```
scanf(kontrolni_string, arg_1, arg_2, ... ,arg_n)
```

gdje je `kontrolni_string` konstantni znakovni niz koji sadrži informacije o vrijednostima koje se učitavaju u argumente `arg_1`,...,`arg_n`.

Kontrolni znakovni niz (string) je konstantan znakovni niz koji se sastoji od individualnih grupa znakova konverzije pri čemu je svakom argumentu pridružena jedna grupa. Svaka grupa znakova konverzije započinje znakom postotka (%) kojeg slijedi *znak konverzije* koji upućuje na tip podatka koji se učitava. Npr. `%c` ili `%d` itd.

Najčešće korišteni znakovi konverzije navedeni su u sljedećoj tablici:

znak konverzije	tip podatka koji se učitava
%c	jedan znak (char)
%d	decimalni cijeli broj (int)
%e,%f,%g	broj s pokretnim zarezom (float)
%h	kratak cijeli broj (short)
%i	decimalni, heksadecimalni ili oktalni cijeli broj (int)
%o	oktalni cijeli broj (int)
%u	cijeli broj bez predznaka (unsigned int)
%x	heksadecimalni cijeli broj (int)
%s	string (char *)
%p	pokazivač (void *)

Unutar kontrolnog niza znakova grupe kontrolnih znakova mogu se nastavljati jedna na drugu bez razmaka ili mogu biti odvojene bjelinama Bjeline će u ulaznim podacima biti učitane i ignorirane.

Argumenti funkcije scanf mogu biti samo pokazivači na varijable. Ukoliko podatak treba učitati u neku varijablu, onda scanf uzima kao argument adresu te varijable, a ne samu varijablu. To znači da pri pozivu funkcije scanf ispred imena varijable u koju scanf treba učitati vrijednost moramo staviti adresni operator &. Tako će naredba

```
int x;
......
scanf("%d",&x);
```

učitati cijeli broj s ulaza u varijablu x, dok naredba

```
int x;
......
scanf("%d",x);  /* pogresno */
```

predstavlja grešku.

Podaci koje scanf čita dolaze sa standardnog ulaza što je tipično tastatura. Ako se unosi više podataka oni moraju biti separirani bjelinama što u sebi uključuje i prijelaz u novi red (koji se računa kao bjelina). Numerički podaci na ulazu moraju imati isti oblik kao i numeričke konstante.

5.4.1 Učitavanje cijelih brojeva

Cijeli brojevi mogu biti uneseni kao decimalni (%d) ili kao oktalni i heksadecimalni (%i). Znak konverzije (%i) interpretira ulazni podatak kao oktalan broj ako mu prethodi nula ili kao heksadecimalan broj ako mu prethodi 0x ili 0X. Na primjer, uzmimo da kôd

```
int x,y,z;
......
scanf("%i %i %i",&x,&y,&z);
```

učitava sljedeći ulazni podatak:

```
 13  015    0Xd
```

onda će `scanf` u sve tri varijable (`x`, `y` i `z`) učitati vrijednost 13 (decimalno). Cijeli brojevi u oktalnom i heksadecimalnom zapisu mogu se upisivati i pomoću znakova konverzije `%o` i `%x`. Ti znakovi konverzije interpretiraju ulazne podatke kao oktalne odnosno heksadecimalne i stoga ne zahtijevaju da oktalna konstanta započinje nulom, a heksadecimalna s `0x` ili `0X`. Kôd

```
int x,y,z;
......
scanf("%d %o %x",&x,&y,&z);
```

ispravno će pročitati ulazne podatke

```
 13  15  d
```

i svim varijablama pridružiti vrijednost 13 (decimalno).

Podatak učitavamo u varijablu tipa `unsigned` sa znakom konverzije `%u`.

Znakovi konverzije `d`, `i`, `o`, `u`, `x` mogu dobiti prefiks `h` ako je argument pokazivač na `short` te prefiks `l` ako je argument pokazivač na `long`. Na primjer, kôd

```
int x;
short y;
long z;
......
scanf("%d %hd %ld",&x,&y,&z);
```

učitava tri decimalna cijela broja i konvertira ih u varijable tipa `int`, `short` i `long`. Uočite da `h` i `l` ne mogu stajati sami i da u kombinaciji uvijek prethode znaku konverzije (`d, i, o, u, x`).

Standard C99 uvodi prefiks `ll` za učitavanje varijabli tipa `long long` i `unsigned long long`.

5.4.2 Učitavanje realnih brojeva

Znakovi konverzije `e`, `f`, `g` služe za učitavnje varijable tipa `float`. Ukoliko se učitava vrijednost u varijablu tipa `double` treba koristiti prefiks `l` (`le`, `lf` ili `lg`). Na primjer,

```
float x;
double y;
......
scanf("%f %lg",&x,&y);
```

Prefiks `L` koristi se ako je argument pointer na `long double`.

znak konverzije	tip podatka koji se učitava
`%e,%f,%g`	broj tipa `float`
`%le,%lf,%lg`	broj tipa `double`
`%Le,%Lf,%Lg`	broj tipa `long double`

5.4.3 Drugi znakovi u kontrolnom nizu

Funkcija `scanf` dijeli niz znakova na ulazu u polja znakova odvojena bjelinama. Svako polje znakova intrepretira se prema odgovarajućem znaku konverzije i upisuje u varijablu na koju pokazuje odgovarajući argument funkcije. Svaki znak konverzije učitava jedno ulazno polje. U primjeru

```
scanf("%f%d",&x,&i);
```

znak konverzije `%f` učitava (i konvertira) prvo polje znakova. Pri tome se eventualne bjeline na početku preskaču. Prvo polje znakova završava bjelinom koju `%f` ne učitava. Drugi znak konverzije `%d` preskače sve bjeline koje odjeljuju prvo polje znakova od drugog polja znakova i učitava (i konvertira) drugo polje znakova.

Znakovi konverzije mogu biti odijeljeni bjelinama:

```
scanf("%f %d",&x,&i);
```

Svaka bjelina u kontrolnom znakovnom nizu ima za posljedicu preskakanje svih bjelina na ulazu do početka novog ulaznog polja. Stoga je pisanje znakova konverzije u kontrolnom znakovnom nizu razdvojeno bjelinam (kao u primjeru `"%f %d"`) ili nerazdvojeno (kao `"%f%d"`) posve ekvivalentno (to ne vrijedi za znakove konverzije `%c` i `[`).

U kontrolnom znakovnom nizu mogu se pojaviti i drugi znakovi osim bjelina i znakova konverzije. Njima moraju odgovarati posve isti znakovi na ulazu. Na primjer, ako realan i cijeli broj učitavamo naredbom

```
scanf("%f,%d",&x,&i);
```

onda ulazni podaci moraju biti oblika npr.

```
1.456, 8
```

bez bjeline između prvog broja i zareza. Ako se želi dozvoliti bjelina prije zareza potrebno je koristiti naredbu

```
scanf("%f ,%d",&x,&i);
```

u kojoj bjelina nakon `%f` preskače sve eventualne bjeline na ulazu ispred zareza.

Znakovi u kontrolnom znakovnom nizu mogu npr. biti korisni kod učitavanja datuma u obliku `dd/mm/gg`. Tada možemo iskoristiti naredbu

```
scanf("%d/%d/%d",&dan,&mjesec,&godina);
```

5.4.4 Učitavanje znakovnih nizova

Znak konverzije `%s` učitava niz znakova; niz završava prvom bjelinom u ulaznom nizu znakova. Iza posljednjeg učitanog znaka automatski se dodaje nul-znak (\0). U primjeru

```
 char string[128];
 int x;
 ......
scanf("%s%d",string,&x);
```

funkcija `scanf` učitava jedan niz znakova i jedan cijeli broj. Budući da se svako polje kao argument funkcije interpretira kao pokazivač na prvi elemet polja, ispred varijable `string` ne stavlja se adresni operator. To je pravilo koje vrijedi za polje bilo kojeg tipa.

Znakom konverzije `%s` nije moguće učitati niz znakova koji sadrži u sebi bjeline jer bjeline služe za ograničavanje ulaznog polja. Za učitavanje nizova znakova koji uključuju i bjeline možemo koristiti uglate zagrade kao znak konverzije `%[...]`. Unutar uglatih zagrada upisuje se niz znakova. Funkcija `scanf` će učitati u pripadni argument najveći niz znakova sa ulazu koji se sastoji od znakova navedenih unutar uglatih zagrada. Učitavanje završava prvi znak na ulazu koji nije naveden u uglatim zagradama i na kraj učitanog niza dodaje se nul-znak (`\0`). Vodeće bjelina se ne preskaču.

Na primjer, naredba

```
char linija[128];
........
scanf(" %[ ABCDEFGHIJKLMNOPRSTUVWXYZ]", linija);
```

učitava najveći niz znakova sastavljen od velikih slova i razmaka. Argument `linija` mora naravno imati dovoljnu dimenziju da primi sve znakove i završni nul-znak `\0`. Uočimo da smo prije `%[` ostavili jedan razmak koji govori funkciji `scanf` da preskoči sve bjeline koje prethode znakovnom nizu. To je nužno ukoliko smo imali prethodni poziv `scanf` funkcije. Naime `scanf` uvijek ostavlja završni znak prijelaza u novi red u ulaznom nizu, tako da bi naredba

```
scanf("%[ ABCDEFGHIJKLMNOPRSTUVWXYZ]", linija);
```

pročitala prethodni znak prijelaza u novi red i budući da on nije u unutar uglatih zagrada završila bi čitanje ulaznih podataka i `linija` ne bi bila učitana.

Druga mogućnost s uglatim zagradama je koristiti sintaksu

```
scanf(" %[^niz znakova]", linija);
```

Sada se u odgovarajući argument učitava najveći mogući niz znakova sastavljen od svih znakova **osim** onih koji se nalaze u uglatim zagradama. Na primjer, učitati cijelu liniju bez znaka za prijelaz u novi red možemo pomoću naredbe

```
scanf(" %[^\n]", linija);
```

Na kraj učitanog niza znakova bit će dodan `\0`, a ispred `%[` mora biti ostavljeno prazno mjesto kako bi bile preskočene sve prethodne bjeline.

Znak konverzije `c` učitava jedan znak u varijablu bez obzira je li on bjelina ili ne. Stoga, ako je prvi znak konverzije `c` potrebno je ispred njega staviti jednu bjelinu kako ne bi pročitao znak za prijelaz u novi red koji je ostao nakon prethodnog poziva funkcije `scanf`.

Nadalje, kontrolni niz `" %c%c%c"` čita tri znaka. Počet će s prvim znakom koji nije bjelina (zbog bjeline ispred prvog `%c` znaka) i pročitat će tri uzastopna znaka bili oni bjeline ili ne. Ako se želi čitati samo znakove bez bjelina treba koristiti `" %c %c %c"` ili `%c` zamijeniti s `%1s`.

5.4.5 Prefiks *

Moguće je preskočiti neki podatak u listi i ne pridružiti ga odgovarajućoj varijabli. To se radi tako da se znaku konverzije doda prefiks *. Na primjer

```
scanf(" %s %*d %f", linija, &n, &x);
```

neće izvršiti pridruživanje drugog podatka varijabli `n`. Umjesto toga on će biti preskočen a treći podatak će normalno biti pridružen varijabli `x`.

5.4.6 Maksimalna širina polja

Uz svaki kontrolni znak moguće je zadati maksimalnu širinu ulaznog polja koje će se učitati tako da se ispred kontrolnog znaka stavi broj koji određuje širinu polja. Tako na primjer `%3d` učitava cijeli broj od najviše tri znamenke, a `%11c` učitava 11 znakova. Ukoliko podatak sadrži manje znakova od zadane maksimalne širine polja on se učita samo do prve bjeline. Ako pak podatak ima više znamenaka od maksimalne širine polja, višak znamenaka bit će učitan sljedećim konverzijskim znakom ili sljedećom `scanf` funkcijom. Na primjer, uzmimo naredbu

```
scanf(" %3d %3d %3d", &i, &j, &k);
```

Ukoliko na ulazu imamo

```
1 2 3
```

bit će učitano `i=1`, `j=2`, `k=3`. Ako na ulazu imamo

```
123 456     789
```

bit će učitano `i=123`, `j=456`, `k=789`. Uzmimo sada ulaz oblika

```
123456789
```

ponovo ćemo dobiti `i=123`, `j=456`, `k=789` budući da sljedeće ulazno polje započinje tamo gdje prethodno završava. Konačno pogledajmo ulaz oblika

```
1234 56 789
```

dobit ćemo `i=123`, `j=4` i `k=56`. Širina prvog ulaznog polja je tri znaka (`i=123`); drugo ulazno polje počine sa znakom `4` i završava prvom bjelinom (`j=4`). Treće ulazno polje posve je separirano bjelinama pa dobivamo `k=56`. Preostali znakovi ostat će na ulazu i mogu biti pročitani novim pozivom `scanf` funkcije (ili neke druge ulazne funkcije).

5.4.7 Povratna vrijednost

Funkcija `scanf` vraća broj uspješno učitanih podataka ili `EOF`. Tu činjenicu možemo iskoristiti za provjeru jesu li svi traženi podaci učitani. Uzmimo jedan primjer u kojem učitavamo i procesiramo samo pozitivne cijele brojeve. Kôd bi mogao izgledati ovako:

```
int n;
scanf("%d",&n);
while(n >= 0)
{
  // radi nesto s brojem
  scanf("%d",&n);  // ucitaj novi broj
}
```

Ovakav kôd ne može uspješno tretirati slučajeve u kojima korisnik učini grešku priklikom upisa (npr. upiše slovo).

Ponašanje programa možemo popraviti ako ispitujemo da li je funkcija `scanf` uspješno učitala broj. To možemo učiniti testom `scanf("%d",&n) == 1`. Nova verzija programa bi glasila

```
int n;
while(scanf("%d",&n) == 1 && n >= 0)
{
  // radi nesto s brojem
}
```

Uočimo da prvo ispitujemo je li broj dobro učitan. Ako nije, `n >= 0` neće biti ispitivano.

No program bi trebao kod neuspješnog upisa zatražiti od korisnika da ponovi upis. Da bismo to realizirali uočimo da funkcija `scanf` nakon neuspješnog pokušaja čitanja nekog podatka prekida čitanje i ostavlja taj podatak i sve iza njega u ulaznom spremniku. Stoga je potrebno prije novog poziva funkciji `scanf` isprazniti spremnik. To možemo učiniti pomoću funkcije `getchar`. Mi ćemo, štoviše, iskoristiti sadržaj spremnika da obavijestimo korisnika o grešci koju je napravio, Budući da kôd postaje kompleksniji možemo postupak čitanja broja zatvoriti u jednu funkciju koju ćemo nazvati `get_int`.

```
int get_int(void)
{
  int input;
  char ch;
  while(scanf("%d",&input) != 1)
```

```
  {
      while((ch = getchar())!='\n')
        putchar(ch);
      printf(" nije cijeli broj.\nMolim unesite ");
      printf("cijeli broj:  ");
  }
  return input;
}
```

Kada korisnik ne unese cijeli broj, nepročitani ulazni podatak će biti pročitan pomoću druge `while` petlje. Sadržaj će biti ispisan funkcijom `putchar` i korisnik će biti zamoljen da ponovi upis.

5.5 Funkcija printf

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

Funkcija `printf` služi za ispis podataka na standardnom izlazu. Opća forma funkcije je

```
printf(kontrolni_string, arg_1, arg_2, ... ,arg_n)
```

gdje je `kontrolni_string` konstantan znakovni niz koji sadrži informaciju o formatiranju ispisa argumenata `arg_1`,...,`arg_n`.

Kontrolni znakovni niz ima posve istu formu i funkciju kao kod funkcije `scanf`. Pojedine grupe znakova unutar kontrolnog znakovnog niza mogu se nastavljati jedna na drugu ili biti međusobno razmaknute bjelinama ili nekim drugim znakovima. Svi znakovi osim kontrolnih bit će ispisani onako kako su uneseni.

Najčešće korišteni znakovi konverzije dani su u sljedećoj tabeli:

znak konverzije	tip podatka koji se ispisuje
`%d,%i`	decimalni cijeli broj (`int`)
`%u`	cijeli broj bez predznaka (`unsigned int`)
`%o`	oktalni cijeli broj bez predznaka (`unsigned int`)
`%x`	heksadecimalni cijeli broj bez predznaka (`unsigned int`)
`%e,%f,%g`	broj s pokretnim zarezom (`double`)
`%c`	jedan znak (`char`)
`%s`	string (`char *`)
`%p`	pokazivač (`void *`)

Funkcija vraća broj ispisanih znakova ili EOF ako je došlo do greške.

Argumenti funkcije printf mogu biti konstante, varijable, izrazi ili polja. Na primjer, naredba

```
double x=2.0;
......
printf("x=%d, y=%f\n",x,sqrt(x));
```

će ispisati

```
x=2.000000, y=1.414214
```

Svi znakovi koji nisu znakovi konverzije ispisani su onako kako su uneseni u kontrolnom znakovnom nizu "x=%d, y=%f\n". Ako treba ispisati znak %, onda unutar kontrolnog znakovnog niza na tom mjestu treba staviti %%.

Funkcija printf ima varijabilan broj argumenata i stoga se pri pozivu funkcije vrši promocija argumenata: argumenti tipa char i short konvertiraju se u tip int, a argumenti tipa float u double. Ove konverzije dešavaju se prije predaje argumenata funkciji pa stoga pomoću znaka konverzije %f možemo ispisati i varijablu tipa float i tipa double, jer će u oba slučaja printf vidjeti samo varijablu tipa double. Slično pomoću %d možemo ispisati i varijablu tipa char i tipa short. Na primjer,

```
char c='w';
......
printf("c(int)=%d, c(char)=%c\n",c,c);
```

ispisat će

```
c(int)=119, c(char)=w
```

ukoliko računalo koristi ASCII skup znakova (119 je ASCII kod znaka "w").

5.5.1 Ispis cijelih brojeva

Pomoću znakova konverzije %o i %x cijeli brojevi se ispisuju u oktalnom i heksadecimalnom obliku bez vodeće nule odn. 0X. Na primjer,

```
short i=64;
......
printf("i(okt)=%o: i(hex)=%x: i(dec)=%d\n",i,i,i);
```

ispisuje

```
i(okt)=100: i(hex)=40: i(dec)=64
```

Izrazi tipa `long` ispisuju se pomoću prefiksa `l`. Na primjer, program

```
#include <stdio.h>
#include <limits.h>
long i=LONG_MAX;
int main(void){
    printf("i(okt)=%lo: i(hex)=%lx: i(dec)=%ld\n",i,i,i);
}
```

(ovisno o računalu na kojem se izvršava) može ispisati

```
i(okt)=17777777777: i(hex)=7fffffff: i(dec)=2147483647
```

Simbolička konstanta `LONG_MAX` definirana je u datoteci zaglavlje `<limits.h>` i predstavlja najveći broj tipa `long`.

5.5.2 Ispis realnih brojeva

Brojeve tipa `float` i `double` možemo ispisivati pomoću znakova konverzije `%f`, `%g` i `%e`. U konverziji tipa `f` broj se ispisuje bez eksponenta, a u konverziji tipa `e` s eksponentom. U konverziji tipa `g` način ispisa (s eksponentom ili bez) ovisi o vrijednosti koja se ispisuje. Naredba,

```
double x=12345.678;
.........
printf("x(f)=%f: x(e)=%e: x(g)=%g\n",x,x,x);
```

će ispisati

```
x(f)=12345.678000: x(e)=1.234568e+004: x(g)=12345.7
```

Znakovi konverzije `e`, `f`, `g` dobivaju prefiks `L` ako se ispisuje varijabla tipa `long double`.

znak konverzije	tip podatka koji se ispisuje
`%e,%f,%g`	broj tipa `float`
`%e,%f,%g`	broj tipa `double`
`%Le,%Lf,%Lg`	broj tipa `long double`

5.5.3 Širina i preciznost

Uz svaki kontrolni znak moguće je zadati minimalnu širinu ispisa tako da se ispred kontrolnog znaka stavi broj koji određuje širinu ispisa. Tako na primjer `%3d` ispisuje cijeli broj sa najmanje tri znamenke. Ukoliko podatak sadrži manje znakova od zadane minimalne širine polja, do pune širine bit će dopunjen vodećim bjelinama. Podatak koji ima više znamenaka od minimalne širine ispisa bit će ispisan sa svim potrebnim znamenkama. Tako, na primjer, sljedeća naredba

```
   double x=1.2;
   .........
   printf("%1g\n%3g\n%5g\n",x,x,x);
```

ispisuje

```
1.2
1.2
  1.2
```

Prva dva ispisa koriste 3 znaka dok treći koristi pet znakova tj. dvije vodeće bjeline.

Pored minimalne širine ispisa kod realnih brojeva moguće je odrediti i preciznost ispisa tj. broj decimala koje će biti ispisane. Sintaksa je sljedeća: `%a.bf` ili `%a.bg` ili `%a.be` gdje je `a` minimalna širina ispisa, a `b` preciznost. Na primjer `%7.3e` znači ispis u `e` formatu s najmanje 7 znamenaka, pri čemu će biti dano najviše 3 znamenke iza decimalne točke. Sljedeći program ispisuje broj π:

```
#include <stdio.h>
#include <math.h>
int main(void){
   double pi=4.0*atan(1.0);
   printf("%5f\n %5.5f\n %5.10f\n",pi,pi,pi);
}
```

Rezultat ispisa će biti

```
3.141593
3.14159
3.1415926536
```

Ispis bez specificirane preciznosti daje šet decimala, pa je prvi ispis zaokružen na šest decimala. Drugi i treći ispis su zaokruženi na pet i deset decimala budući da je preciznost dana eksplicitno u `printf` funkciji.

Širinu i preciznost ispisa moguće je odrediti dinamički tako da se na mjesto širine ili preciznosti umjesto broja stavi *. Cjelobrojna varijabla (ili izraz) na odgovarajućem mjestu u listi argumenata određuje širinu odn. preciznost. U primjeru

```
#include <stdio.h>
#include <math.h>
int main(void){
   double pi=4.0*atan(1.0);
   int i=10;
   printf("%*f\n %*.*f\n %5.*f\n",11,pi,16,14,pi,i,pi);
}
```

dobivamo ispis

```
   3.141593
3.14159265358979
3.1415926536
```

Prva varijabla ispisana je s 11 znamenaka i 6 decimala (budući da preciznost nije specificirana); druga je ispisana s 16 znamenaka i 14 decimala, a treća 10 decimala i 12 znamenaka.

5.5.4 Ispis znakovnih nizova

Konverzija tipa `%s` primjenjuje se na znakovne nizove (nizove znakova čiji je kraj signaliziran nul-znakom `\0`). Stoga se mogu ispisati i nizovi znakova koji sadrže bjeline. Na primjer,

```
  char naslov[]="Programski jezik C";
      ....
  printf("%s\n",naslov);
```

ispisat će

```
Programski jezik C
```

i prijeći u novi red.

Preciznost se može koristiti i kod `%s` konverzije. Tada znači maksimalni broj znakova koji će biti prikazan. Na primjer `%5.12s` specificira da će biti prikazano minimalno 5 znakova (dopunjenih bjelinama kao treba), a maksimalno 12 znakova. Ako je niz znakova duži od 12, višak znakova neće biti prikazan. Na primjer,

```
char naslov[]="Programski jezik C";
     ....
printf("%.16s\n",naslov);
```

ispisat će

```
Programski jezik
```

5.5.5 Zastavice

Svaka grupa znakova za konverziju može sadržavati i zastavicu. Zastavica je znak koji dolazi odmah nakon znaka `%`; moguće zastavice su: `-`, `+`, `0`, `' '` (bjelina) i `#`, a značenja su sljedeća:

- \- podatak će biti lijevo pozicioniran ako je manji od minimalne širine polja;
- \+ Znak + će biti napisan ispred pozitivnog broja;
- 0 Vodeće bjeline (ako su nužne) bit će zamijenjene nulama. Odnosi se samo na numeričke podatke koji su desno pozicionirani i koji su uži od minimalne širine ispisa;
- ' ' (bjelina) jedna bjelina će prethoditi svakom pozitivnom broju.
- `#` (uz `o` ili `x` konverziju) osigurava da će oktalni i heksadecimalni brojevi biti ispisani s vodećom 0 ili 0x;
- `#` (uz `e`, `f` ili `g` konverziju) osigurava da će decimalna točka biti ispisana i da će nule na krajnjoj desnoj poziciji broja biti ispisane.

Tako će, na primjer, naredba

```
int i=66;
 ....
printf(":%6d\n:%-6d\n:%06d\n%#x\n",i,i,i,i);
```

ispisati

```
:    66
:66
:000066
:0x42
```

Poglavlje 6

Kontrola toka programa

6.1 Izrazi i naredbe

6.1.1 Izrazi

Izrazom nazivamo svaku kombinaciju operatora i operanada koju jezik dozvoljava. Operatori mogu biti konstante i varijable ili njihove kombinacije. Primjeri izraza su

```
7.2
2+13-4
x=3*2
a*(b+c/d)%2
n++
x= ++n % 2
q>2
```

itd. Složeni izrazi mogu biti kombinacija više manjih izraza koje nazivamo *podizraima*. Na primjer, `c/d` je podizraz u četvrtom primjeru.

Svakom izrazu u programskom jeziku C pridružena je vrijednost koja se dobiva izvršavanjem svih operacija prisutnih u izrazu, u skladu s njihovim prioritetima.

Ovdje su primjeri nekih izraza i njihovih vrijednosti:

Izraz	**vrijednost**
`-3+5`	2
`x=2+17`	19
`5>3`	1
`y=7+(x=2+17)`	26

Ovdje treba uočiti da svaki logički izraz (npr. `5>3`) dobiva vrijednost 0 ukoliko nije istinit ili 1 ako je istinit. Drugo, izrazi pridruživanja primaju vrijednost koja se pridružuje lijevoj strani. Tako u primjeru `x=2+17` varijabla `x` prima vrijednost 17, što je ujedno i vrijednost čitavog izraza. To nam omogućava formiranje izraza poput `y=7+(x=2+17)`: u njemu se prvo izračunava podizraz `x=2+17` koji u varijablu `x` sprema vrijednost 17, a zatim se izračunava `y=7+17`, jer je vrijednost čitavog podizraza na desnoj strani 17. To nam svojstvo izraza omogućava i pisanje višestrukog pridruživanja u jednoj liniji kao npr. u

```
x=y=z=0.0
```

6.1.2 Naredbe

Program se sastoji od niza *naredbi*. Svaka naredba je potpuna instrukcija računalu. U C-u kraj naredbe označavamo znakom točka-zarez. Tako je

```
x=3.0
```

jedan izraz, dok je

```
x=3.0;
```

naredba.

C poznaje više vrsti naredbi. Svaki izraz kojeg slijedi točka-zarez predstavlja narednu. Tako su i

```
8;
4+4;
```

legalne naredbe premda ne čine ništa. Sljedeći program ilustrira različite tipove naredbi:

```
#include <stdio.h>
int main(void)      /* suma prvih 20 prirodnih brojeva */
{
   int brojac,suma;            /* deklaracijska naredba */

   brojac=1;                   /* naredba pridruzivanja */
   suma =0;                    /* isto                  */
   while(brojac++ < 21)        /* while                 */
     suma += brojac;           /* naredba               */
   printf("suma = %d\n",suma); /* funkcijska naredba    */
   return 0;
}
```

Deklaracijskim naredbama deklariramo varijable. Naredbe pridruživanja su izrazi pridruživanja završeni točka-zarezom. Funkcijska naredba predstavlja poziv funkcije. Njenim izvršavanjem izvršavaju se sve naredbe iz tijela funkcije. U našem slučaju poziva se funkcija `printf` za ispis sume.

Naredba `while` je primjer strukturirane naredbe. Ona se sastoji od tri dijela: ključne riječi `while`, testa `brojac++ < 21` i naredbe `suma += brojac;`. Ostale strukturirane naredbe su petlje `for`, `do-while`, `if`-nareba itd.

6.1.3 Složene naredbe

Vitičaste zagrade služe grupiranju deklaracija i naredbi u jednu cjelinu. Takva se cjelina naziva blok naredbi ili složena naredba. U strukturiranim naredbama jedna naredba se može uvijek zamijeniti blokom naredbi. Usporedimo dvije `while` petlje.

```
/* fragment 1 */
  while(brojac++ < 21)
     suma += brojac;
  printf("suma = %d\n",suma);

/* fragment 2 */
  while(brojac++ < 21)
  {
    suma += brojac;
    printf("suma = %d\n",suma);
  }
```

U prvom dijelu primjera `while` naredba se proteže od ključne riječi `while` do točke-zarez. Funkcija `printf` ostaje izvan petlje.

U drugom dijeli vitičaste zagrade osiguravaju da su obje naredbe dio `while` naredbe te se `printf` izvršava u svako prolazu kroz petlju. Sa stanovišta `while` petlje složena naredba tretira se isto kao i jednostavna naredba.

Svaka blok naredba može sadržavati deklaracije varijabli i izvršne naredbe. Prema standardu C90 sve deklaracije varijabli moraju prethoditi prvoj izvršnoj naredbi. Standard C99 dozvoljava slobodno preplitanje deklaracija i izvršnih naredbi.

6.1.4 Popratne pojave i sekvencijske točke

Ovdje želimo uvesti nešto C terminologije. Popratna pojava (eng. *side effect*) je svaka promjena nekog objekta koja se desi kao posljedica izračunavanja

nekog izraza. Na primjer, popratna pojava naredbe

```
x=50.0;
```

je postavljnje varijable `x` na vrijednost `50.0`. Takva se terminologija može činiti čudnom jer je očita intencija ove naredbe postaviti varijablu `x` na `50.0`, no sa stanovišta jezika osnovna intencija je izračunati izraz. Kao posljedica izračunavanja izraza varijabla `x` je postavljena na `50.0`.

Drugi primjer popratne pojave nalazimo u `while` naredbi iz prethodnog programa

```
    while(brojac++ < 21)
```

gdje kao popratnu pojavu izračunavanja izraza `brojac++ < 21` dobivamo povećanje varijable `brojac` za 1. Ovdje je termin popratna pojava posve primjeren.

Sada se nameće pitanje u kojem se trenutku izvršavaju popratne pojave. Mjesto u programu na kojem se popratne pojave moraju izvršiti prije nastavka izvršavanja programa naziva se **sekvencijska točka** (eng. *sequence point*).

Točka-zarez označava sekvencijsku točku. Prije prijelaza na novu naredbu sve popratne pojave prethodne naredbe moraju biti izvršene.

Isto tako, na kraju svakog potpunog izraza nalazi se sekvencijska točka. Pri tome je **potpuni izraz** svaki izraz koji nije podizraz nekog većeg izraza. To je značajno u strukturnim naredbama koje testiraju neki izraz. Na primjer, u petlji

```
while(brojac++ < 21)
    printf("%d \n",brojac);
```

test petlje čini potpuni izraz i stoga povećanje varijable `brojac` mora biti izvršeno odmah nakon izračunavanja testa, a ne nakon izvršavanja čitave `while` naredbe, kao što bi netko mogao pomisliti.

U primjeru

```
y=(4+x++)+(6-x++);
```

`4+x++` nije potpun izraz i stoga C ne garantira da će `x` biti povećan odmah nako izračunavanja podizraza `4+x++`. Čitav izraz pridruživanja je potpuni izraz, a sekvencijska točka je točka-zarez te stoga C garantira da će `x` biti dva puta povećan za 1 prije prijelaza na izvršavanje nove naredbe. Jezik ne specificira hoće li `x` biti povećan nakon izračunavanja svakog podizraza ili tek nakon izračunavanja čitavog izraza te stoga ovakve izraze treba izbjegavati.

6.2 Naredbe uvjetnog grananja

Naredbe uvjetnog grananja `if`, `if-else` i `switch` omogućavaju izvršavanje pojedinih grupa naredbi u ovisnosti o tome da li su ispunjeni neki uvjeti. Naredba `goto` pruža mogućnost prekida sekvencijalnog izvršavanja programa i skoka na neku drugu lokaciju unutar programa.

6.2.1 `if` naredba

Naredba `if` ima oblik

```
if(uvjet) naredba;
```

gdje je `uvjet` aritmetički (ili pokazivački) izraz. Naredba prvo izračunava izraz `uvjet` i ako je njegova vrijednost različita od nule (što se interpretira kao istina, odn. ispunjenost uvjeta) izvršava se `naredba`. Ako izraz `uvjet` daje nulu (što se interpretira kao laž, odn. neispunjenost uvjeta), onda se `naredba` ne izvršava i program se nastavlja prvom naredbom iza `if` naredbe. Na primjer, u dijelu programa

```
int x;
....
if(x>0) printf("\n x= %d \n",x);
x++;
```

izraz `x>0` imat će vrijednost 1 ako je vrijednost varijable `x` strogo pozitivna i varijabla `x` će biti ispisana naredbom `printf("\n x= %d \n",x)` te zatim, u sljedećoj naredbi, povećana za 1; ako `x` nije strogo pozitivno, vrijednost izraza `x>0` bit će 0 i program će inkrementirati `x` bez poziva funkcije `printf`.

Umjesto jedne naredbe `if` može sadržavati blok naredbi omeđen vitičastim zagradama (složena naredba).

```
if(uvjet) {
   naredba;
   naredba;
   ......
   naredba;
}
```

U ovom se slučaju čitav blok naredbi izvršava ili ne izvršava ovisno o tome da li `uvjet` ima vrijednost istine ili laži.

Uvjet u `if` naredbi je najčešće izraz sastavljen od relacijskih i logičkih operatora. Na primjer, pretvaranje malog u veliko slovo moglo bi biti izvedeno na sljedeći način:

```
char c;
......
if(c>='a' && c<='z')
      c='A'+c-'a';
```

Ovdje koristimo logičko I kako bismo povezali uvjete `c>='a'` i `c<='z'`. Oba uvjeta zadovoljavaju samo mala slova (engleske) abecede koja u naredbi `c='A'+c-'a';` pretvaramo u odgovarajuća velika slova.

Relacijski operatori ne mogu se primijeniti na znakovne nizove (stringove) pa stoga moramo koristiti `strcmp` funkciju kao u sljedećem primjeru:

```
#include <string.h>
 char *string1, *string2;
 ......
 if(strcmp(string1,string2)==0)
      printf("Stringovi su jednaki.\n");
```

Funkcija `strcmp` će vratiti nulu ako su stringovi `string1` i `string2` jednaki. Istu bismo `if` naredbu mogli napisati i u obliku

```
if(!strcmp(string1,string2))
        printf("Stringovi su jednaki.\n");
```

Uočimo općenito da uvjet `if(x!=0)` možemo pisati kao `if(x)`, te isto tako `if(x==0)` možemo zamijeniti s `if(!x)`.

Kod logičkih izraza koji se pojavljuju u uvjetu `if` naredbe ponekad treba koristiti pravilo da se izraz izračunava sve dok njegova vrijednost ne postane poznata. Na primjer, u dijelu kôda

```
int x;
.....
if(x != 0 && (20/x) < 5)
     ......;
```

izračunavanje izraza `x != 0 && (20/x) < 5` u slučaju `x=0` prestat će nakon `x != 0`, jer je vrijednost cijelog izraza u tom trenutku poznata (=laž). Stoga podizraz `(20/x) < 5` neće biti izračunat i do dijeljenja s nulom neće doći.

6.2.2 `if-else` naredba

`if-else` naredba ima oblik

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

Ako izraz `uvjet` ima vrijednost istine, onda se izvršava `naredba1` a u suprotnom `naredba2`; obje naredbe mogu biti blokovi naredbi. Na primjer,

```
#include <stdlib.h>
 ......
 if(!x){
    printf("Djelitelj jednak nuli!\n");
    exit(-1);
 }else
    y/=x;
```

Funkcija

```
void exit(int status)
```

deklarirana je u datoteci zaglavlja `<stdlib.h>` i ona zaustavlja izvršavanje programa. Vrijednost `status` predaje se operacijskom sustavu. Vrijednost nula znači da je program uspješno izvršen, a vrijednost različita od nule signalizira da je program zaustavljen zbog greške.

Uočimo da je sljedeća naredba s uvjetnim operatorom

```
max = a > b ? a : b;
```

ekvivalenta s

```
if(a>b)
    max=a;
else
    max=b;
```

6.2.3 Primjer: Djelitelji broja

Postavimo si sada sljedeći zadatak: za zadani prirodan broja treba ispisati sve njegove djelitelje.

Program ćemo konstruirati kao petlju u kojoj se broj unosi, analizira i ispisuje rezultat. Na taj način nije potrebno pokretati program za testiranje svakog pojedinog broja.

Koristit ćemo sljedeći model:

```
printf("Unos:")
while(scanf vraca vrijednost 1){
   analizirati broj i ispisati rezultat
   printf("Unos:")
}
```

Funkcijom `scanf` učitavamo brojeve. Kada je broj ispravno učitan `scanf` vraća broj učitanih vrijednosti (1 u našem slučaju). Ako pri sljedećem pozivu funkciji `scanf` upišemo npr. `q` ili neko drugo slovo, vrijednost neće biti učitana i funkcija će vratiti nulu što će zaustaviti petlju.

Najjednostavniji način za pronalaženje svih djelitelja zadanog broja `num` je sljedeći:

```
for(div=2; div < num; ++div)
   if(num % div == 0)
      printf("%d je djeljiv s %d\n", num, div);
```

Ovu petlju možemo učiniti puno efikanijom ako primjetimo da svaki puta dobivamo dva djelitelja. Na primjer, `128%2 = 0` što daje da je 128 djeljivo s 2, ali i sa `128/2 = 64`. Dijeleći redom dobivamo parove: 2, 64; 4, 32; 8,16. Dalje od broja čiji je kvadrat veći ili jednak od 128 nije potrebno ići. Ta nam primjedba značajno reducira količinu računaja jer sada možemo pisati:

```
for(div=2; (div*div) <= num; ++div)
   if(num % div == 0)
      printf("%d je djeljiv s %d i %d\n", num, div, num/div);
```

Uočimo da je `num/div` cjelobrojno dijeljenje koje nam daje drugi djelitelj.

U ovom pristupu imamo još dva problema. Prvi se javlja u slučaju da je broj kvadrat nekog broja, kao npr. `144 = 12 * 12`. Tada bismo ispisivali

```
144 je djeljiv s 12 i 12
```

Da bismo to izbjegli uvodimo još jednu `if` naredbu:[1]

```
for(div=2; (div*div) <= num; ++div)
   if(num % div == 0)
   {
     if(div * div != num)
       printf("%d je djeljiv s %d i %d\n", num, div, num/div);
     else
       printf("%d je djeljiv s %d.\n", num, div);
   }
```

[1]Budući da `if-else` čini jednu naredbu, zagrade iza prve `if`-naredbe nisu nužne.

Drugi problem je kako prepoznati prim broj. Jasno je da ako `num` nije djeljiv niti s jednim brojem onda nikad nećemo ući u tijelo `if` naredbe. Stoga možemo uvesti novu varijablu `flag` koju ćemo inicijalizirati nulom i postaviti na 1 u tijelu `if` naredbe. Ako na kraju programa imamo `flag == 0` znači da se radi o prim broju.

Cijeli program sada izgleda ovako:

```
#include <stdio.h>

int main(void)
{
   long num;           // broj koji provjeravamo
   long div;           // potencijalni djelitelj
   unsigned flag = 0;  // prim broj?

   printf("Unesite cijeli broj ili q za izlaz: ");
   while(scanf("%ld",&num) == 1)
   {
      for(div=2; (div*div) <= num; ++div)
      {
         if(num % div == 0)
         {
            if(div * div != num)
               printf("%d je djeljiv s %d i %d\n", num, div,
                                                    num/div);
            else
               printf("%d je djeljiv s %d.\n", num, div);
            flag=1;
         }
      }
      if(flag == 0) printf("%ld je prom broj.\n",num);
      printf("Unesite cijeli broj ili q za izlaz: ");
   }
   return 0;
}
```

6.2.4 Višestruka `if-else` naredba

Više `if-else` naredbi mogu se nadovezati jedna na drugu. Tako pomoću dvije `if - else` naredbe dobivamo složenu naredbu

```
   if(uvjet1)
```

```
        naredba1;
    else if(uvjet2)
        naredba2;
    else
        naredba3;
    naredba4;
```

u kojoj se prvo ispituje `uvjet1` te ako je istinit izvršava se `naredba1`, a zatim se prelazi na naredbu `nareba4`. Ako `uvjet1` nije istinit izračunava se `uvjet2` te se izvršava ili `naredba2` ili `naredba3` ovisno o istinitosti izraza `uvjet2`. Program se zatim nastavlja naredbom `naredba4`. Na primjer, imamo funkciju

```
#include <math.h>
double f(double x)
{
   double y;

   if(x<2.0)
      y=3.0+(x-2.0);
   else if(x<5.0)
      y=3.0+(x-2.0)*exp(x);
   else
      y=3.0*(1.0+exp(5.0));
   return y;
}
```

Uočimo da je višestruka `if-else` naredba u tijelu funkcije samo drukčiji način pisanja dviju ugniježdenih if-else naredbi:

```
   if(x<2.0)
      y=3.0+(x-2.0);
   else
      if(x<5.0)
         y=3.0+(x-2.0)*exp(x);
      else
         y=3.0*(1.0+exp(5.0));
```

Budući da je `if-else` jedna naredba nisu nam bile potrebne zagrade poslije `else` dijela prve `if-else` naredbe. Za drugu `if-else` naredbu kažemo da je ugniježdena u prvu.

6.2.5 Primjer. Višestruki izbor

Ugnijezditi se može bilo koji broj `if-else` naredbi, a `else` dio zadnje `if-else` naredbe može se ispustiti. U sljedećem primjeru učitavaju se dva broja i jedan znak koji predstavlja izbor računske operacije. U ovisnosti o učitanom znaku izvršava se jedna od četiri računske operacije.

```
#include <stdio.h>

int main(void)
{
  float a,b;
  char operacija;

  printf("Upisati prvi broj: ");
  scanf(" %f",&a);
  printf("Upisati drugi broj: ");
  scanf(" %f",&b);
  printf("Upisati operaciju: zbrajanje(z), oduzimanje(o),\n");
  printf("                             mnozenje(m),dijeljenje(d) :");
  scanf(" %c",&operacija);

  if(operacija=='z')
     printf("%f\n",a+b);
  else if(operacija=='o')
     printf("%f\n",a-b);
  else if(operacija=='m')
     printf("%f\n",a*b);
  else if(operacija=='d')
     printf("%f\n",a/b);
  else
     printf("Nedopustena operacija!\n");

  return 0;
}
```

6.2.6 Sparivanje `if` i `else` dijela

Budući da se `else` dio `if-else` naredbe može ispustiti moguće su ambivalentne situacije ako je ugnježdeno više `if-else` naredbi od kojih neke nemaju `else` dio. Stoga vrijedi pravilo:

Svaka `else` naredba pripada sebi najbližoj `if` naredbi.

Pogledajmo sljedeći primjer:

```
if(n>0)
  if(a>b)
    z=a;
  else
    z=b;
```

Ovdje je `else` naredba pridružena unutarnjoj `if` naredbi (`if(a>b)`) jer joj je najbliža. Ako želimo da `else` naredba bude pridružena vanjskoj `if` naredbi moramo koristiti zagrade:

```
if(n>0){
  if(a>b)
    z=a;
}
else
  z=b;
```

Sljedeći način pisanja sugerira da je `else` naredba pridrižena vanjskoj `if` naredbi (`if(n>0)`)

```
if(n>0)
  if(a>b) z=a;
else
    z=b;
```

dok je ona ustvari pridružena unutarnjoj `if` naredbi (`if(a>b)`). Takav stil pisanja treba izbjegavati.

6.3 switch naredba

Naredba `switch` slična je nizu ugnježdenih `if-else` naredbi. Ona nam omogućava da selektiramo kôd koji će se izvršiti na osnovu nekog uvjeta. Naredba ima sljedeći oblik:

```
switch(izraz) {
  case konstanta_1:
             naredba_1;
             .......
             break;
  case konstanta_2:
```

```
                  naredba_2;
                  .......
                  break;
         .
         .
         .
        case konstanta_n:
                 naredba_n;
                 .....
                 break;
        default:
                 naredba;
                 .....
       }
```

- Izraz u `switch` naredbi mora imati cjelobrojnu vrijednost (`char`, `int` ili `enum`).
- Nakon ključne riječi `case` pojavljuju se cjelobrojne konstante ili konstantni izrazi.
- Pri izvršavanju `switch` naredbe prvo se testira vrijednost izraza `izraz`. Zatim se provjerava da li se dobivena vrijednost podudara s jednom od konstanti: `konstanta_1`, `konstanta_2`, ...,`konstanta_n`. Ukoliko je `izraz` = `konstanta_i` program se nastavlja naredbom `naredba_i` i svim naredbama koje dolaze nakon nje, sve do `break` naredbe. Nakon toga program se nastavlja prvom naredbom iza `switch` naredbe.
- Ako `izraz` nije jednak niti jednoj konstanti, onda se izvršava samo `naredba` koja dolazi nakon ključne riječi `default` i sve naredbe iza nje, sve od vitičaste zagrade koja omeđuje `switch` naredbu.
- Slučaj `default` ne mora nužno biti prisutan u `switch` naredbi. Ako nije i ako nema podudaranja izaza i konstanti, program se nastavlja prvom naredbom iza `switch` naredbe.

Naredba `switch` je često jasnija od niza `if-else` naredbi. Program iz prošle sekcije mogli smo napisati pomoću `switch` naredbe na sljedeći način:

```
#include <stdio.h>

int main(void)
{
```

```
   float a,b;
   char operacija;

   printf("Upisati prvi broj: ");
   scanf(" %f",&a);
   printf("Upisati drugi broj: ");
   scanf(" %f",&b);
   printf("Upisati operaciju: zbrajanje(z), oduzimanje(o),\n");
   printf("                     mnozenje(m),dijeljenje(d) :");
   scanf(" %c",&operacija);

   switch(operacija){
   case 'z':
           printf("%f\n",a+b);
           break;
   case 'o':
           printf("%f\n",a-b);
           break;
   case 'm':
           printf("%f\n",a*b);
           break;
   case 'd':
           printf("%f\n",a/b);
           break;
   default:
           printf("Nedopustena operacija!\n");
   }
   return 0;
}
```

Namjera programera je sada jasnije izražena nego s nizom `if-else` naredbi.

Naredba `break` može se ispustiti na jednom ili više mjesta. Efekt ispuštanja naredbe `break` je "propadanje kôda" u niži `case` blok. Na primjer, ako bismo u gornjem kôdu ispustili sve `break` narebe i ako bi `operacija` bila jednaka `'o'`, onda bi bilo ispisano oduzimanje, množenje, dijeljenje i poruka `"Nedopustena operacija!\n"`. Selektirani `case` je stoga ulazna točka od koje počinje izvršavanje kôda. Izvršavanje se nastavlja do prve `break` naredbe ili sve do kraja `switch` naredbe, ako nema `break` naredbi.

Pogledajmo sljedeći primjer:

```
int i;
.....
```

```
    switch(i) {
      case 1:
      case 2:
      case 3:
      case 4:
      case 5:  printf("i < 6\n");
               break;
      case 6: printf("i = 6\n");
               break;
     default:  printf("i > 6\n");
     }
```

"Propadanje" kroz `case` blokove omogućava da se u slučajevima i=1,2,3,4,5 izvrši naredba `printf("i < 6\n")`. Kako se ne bi izvršila naredba `printf("i = 6\n")` morali smo staviti `break` nakon `printf("i < 6\n")`.

6.4 `while` petlja

While petlja ima oblik

```
while (izraz) naredba;
```

`naredba` će se izvršavati sve dok `izraz` ima vrijednost istine, tj. sve dok je različit od nule. `naredba` može biti blok naredbi unutar vitičastih zagrada koji obično modificira `izraz` kako bi kontrolirao petlju.

Na primjer, dio kôda

```
i=0;
while (i<10){
    printf("%d\n",i);
    i++;
}
```

ispisat će brojeve 0,1,...,9. `while` petlja se rijetko koristi u slučaju kada je broj ponavljanja petlje unaprijed poznat (kao u prethodnom primjeru); tada je pogodnija `for` petlja. Petlju `while` ćemo najčešće koristiti kada se broj ponavljanja ne može unaprijed znati, kao u ovom primjeru:

```
#include <stdui.h>
 /* Srednja vrijednost upisanih brojeva (razlicitih od 0). */
int main(void)
{
```

```
    int i=0;
    double sum=0.0,x;

    printf(" Upisite niz brojeva !=0, ili nulu za kraj.\n");
    printf(" x[0]= "); scanf("%lf",&x);
    while (x!=0.0){
         sum+=x;
         printf(" x[%d]= ",++i);
         scanf("%lf",&x);
    }
    sum/=i;
    printf(" Srednja vrijednost = %f\n",sum);
    return 0;
}
```

6.5 for petlja

Petlja `for` ima oblik

```
    for(izraz_1; izraz_2; izraz_3) naredba;
```

i ekvivalentna je konstrukciji

```
    izraz_1;
    while (izraz_2){
          naredba;
          izraz_3;
    }
```

`izraz_1` inicijalizira parametre koji kontroliraju petlju, `izraz_2` predstavlja uvjete koji moraju biti zadovoljeni da bi se petlja nastavila izvršavati, a `izraz_3` mijenja parametre koje je `izraz_1` inicijalizirao.

`for` petlja započinje inicijalizacijom kontrolnih parametara. `izraz_2` se izračunava na početku svakog prolaza kroz petlju, a `izraz_3` se izračunava na kraju svakog prolaza kroz petlju.

Niti jedan od tri izraza ne treba biti prisutan. Ako nema srednjeg izraza pretpostavlja se da je njegova vrijednost 1. Stoga je `for(;;);` beskonačna petlja koja ne radi ništa.

Na primjer, sljedeći kôd računa skalarni produkt dva vektora:

```
   float x[100000], y[100000], xydot;
   double tmp;
```

```
.........
tmp=0.0;   //  inicijalizacija varijable
           //   u kojoj se sumira
for(i=0;i<100000;i++) tmp += x[i]*y[i];
xydot=tmp;
```

Ovdje smo upotrijebili varijablu sumacije u dvostrukoj preciznosti kako bismo smanjili greške zaokruživanja.

Kao složeniji primjer napišimo implementaciju funkcije `atoi` iz standardne biblioteke (zaglavlje `<stdlib.h>`), koja prevodi niz znakova koji predstavlja cjeli broj u njegovu numeričku vrijednost. Funkcija treba preskočiti sve početne bjeline, učitati predznak i redom pročitati sve znamenke te ih prevesti u broj tipa `int`.

```
#include <ctype.h>

/* atoi : prevodi string s[] u cijeli broj */
int atoi(const char s[])
{
   int i,n,sign;

   for(i=0;isspace(s[i]);i++)   /* preskace sve bjeline */
           ;      /* prazna naredba  */
   sign=(s[i]=='-')?-1:1;
   if(s[i]=='+' || s[i]=='-') i++; /* preskoci predznak */
   for(n=0; isdigit(s[i]);i++)
        n=10*n+(s[i]-'0');
   return sign*n;
}
```

Ovdje koristimo funkcije `isspace` i `isdigit` koje su deklarirane u datoteci zaglavlja `<ctype.h>`. Obje funkcije uzimaju jedan znak i vraćaju cijeli broj (`int`). `isspace` vraća broj različit od nule (istina) ako je znak bjelina, prijelaz u novi red ili tabulator. U suprotnom vraća nulu (laž). Analogno, `isdigit` vraća broj različit od nule (istina) ako je znak znamenka, a nulu (laž) ako nije.

U prvoj `for` petlji preskočene su sve bjeline. Zatim je uvjetnim operatorom određen predznak broja (koji ne mora biti prisutan). Ako je predznak prisutan sljedeća `if` naredba ga preskače. Sada znamo da su sve znamenke pred nama brojevi. Posljednja `for` petlja izvršava se dok ne naiđemo na znak koji nije broj. Znamenke se pretvaraju u brojeve u naredbi

```
 n=10*n+(s[i]-'0');
```

Tu se oslanjamo na činjenicu da se u svim skupovima znakova koji su u upotrebi znamenke nalaze jedna iza druge u rastućem poretku. Tada `s[i]-'0'` daje numeričku vrijednost znamenke `s[i]`.

6.5.1 Operator zarez

U jednoj `for` naredbi moguće je izvršiti više inicijalizacija i promjena brojača. U tome nam pomaže operator zarez. Zarez (,) koji separira dva izraza je ustvari operator. Izrazi separirani zarezom izračunavaju se slijeva na desno i rezultat čitavog izraza je vrijednost desnog izraza. Na primjer, mogli bismo pisati

```
i=(i=3,i+4);
```

i dobili bismo rezultat `i=7`. Operator zarez koristi se uglavnom u `for` naredbi. Pokažimo to na funkciji `invertiraj` koja invertira niz znakova.

```
#include <string.h>
/* invertiraj : invertiraj znakovni niz */
void invertiraj(char s[]) {
   int c,i,j;

   for(i=0,j=strlen(s)-1;i<j;i++,j--) {
     c=s[i]; s[i]=s[j]; s[j]=c;
   }
}
```

Funkcija `strlen` deklarirana je u datoteci zaglavlja `<string.h>` i daje duljinu znakovnog niza (bez nul-znaka). Uočimo da smo za pomoćnu varijablu `c` mogli koristiti varijablu tipa `int` (umjesto `char`).

Napomenimo još da zarezi u pozivu funkcije (ako ona ima više argumenata) nisu operatori i stoga nije garantirano izračunavanje argumenata funkcije slijeva na desno.

6.5.2 Datoteka zaglavlja `<stdlib.h>`

Funkcija `atoi` koju smo realizirali u primjeru u prethodnoj sekciji implementirana je u standardnoj biblioteci i njen prototip se nalazi u datoteci zaglavlja `<stdlib.h>`. U toj datoteci postoje i prototipovi nekih drugih funkcija za konerziju od kojih navodimo:

`double atof(const char *s)`	prevodi `s` u `double`,
`int atoi(const char *s)`	prevodi `s` u `int` ,
`long atol(const char *s)`	prevodi `s` u `long` .

Pored ostalih navedimo još i ove funkcije iz `<stdlib.h>`:

`void exit(int status)`	normalno zaustavljanje programa,
`int abs(int n)`	absolutna vrijednost,
`long labs(long n)`	absolutna vrijednost.

6.6 do - while petlja

`do-while` petlja ima oblik

```
do
  naredba;
while (izraz);
```

`naredba` će se izvršavati sve dok `izraz` ima vrijednost istine, tj. sve dok je različit od nule. `naredba` može biti blok naredbi unutar vitičastih zagrada koji obično modificira `izraz` kako bi zaustavio petlju. Za razliku od `while` petlje vrijednost izraza se kontrolira na kraju prolaza kroz petlju. Petlja se stoga izvršava barem jednom.

Na primjer, dio kôda

```
i=0;
do{
  printf("%d\n",i);
  i++;
} while (i<10);
```

ispisat će brojeve 0,1,...,9.

Kao primjer napišimo funkciju `itoa` koja pretvara cijeli broj u znakovni niz (string).

```
/* itoa : pretvara n  u znakovni niz s */
void itoa(int n, char s[])
{
  int i,sign;

  if((sign=n)<0) n=-n; /* zapamtimo predznak */
  i=0;
  do {
      s[i++]=n%10+'0';
  } while((n/=10)>0);
  if(sign<0) s[i++]='-';
  s[i]='\0';
```

```
  invertiraj(s);
}
```

Algoritam slaže znamenke broja u obrnutom redoslijedu pa stoga koristimo funkciju `invertiraj()` iz prethodne sekcije. Ostatak u dijeljenju s 10 svakog cijelog broja je njegova posljednja dekadska znamenka. Stoga znamenke dobivamo pomoću operatora `%`. Nako što izdvojimo posljednju znamenku broj dijelimo s 10 (cjelobrojno dijeljenje) kako bismo ju eliminirali. Taj se postupak obavlja u `do--while` petlji. Nakon završetka te procedure dobivenom nizu znakova dodajemo predznak i nul-znak te ga prosljeđujemo funkciji `invertiraj`.

6.7 Naredbe `break` i `continue`

Naredba `break` služi za zaustavljanje petlje i izlazak iz `switch` naredbe. Može se koristiti sa `for`, `while` i `do-while` petljom. Pri nailasku na naredbu `break` kontrola programa se prenosi na prvu naredbu iza petlje ili `switch` naredbe unutar koje se `break` nalazi.

Naredba `continue` koristi se unutar `for`, `while` i `do-while` petlji. Nakon nailaska na `continue` preostali dio tijela petlje se preskače i program nastavlja sa sljedećim prolazom kroz petlju. U slučaju `while` i `do-while` petlje to znači da se program nastavlja s izračunavanjem testa petlje. Kod `for` petlje program se nastavlja s naredbom inkrementacije brojača, budući da ona dolazi nakon tijela petlje. Nakon toga se prelazi na izračunavanje testa petlje.

Primjer upotrebe `break` naredbe u `while` petlji:

```
int i;
while(1){
   scanf("%d",&i);
   if (i<0) break;
   .........
}
```

`while(1)` je beskonačna petlja. Iz nje se izlazi ukoliko se učita negativan broj. Kôd koji bi samo preskakao negativne vrijednosti (i ne bi ih obrađivao) mogao bi se izvesti naredbom `continue`:

```
int i;
while{1){
   scanf("%d",&i);
   if (i<0) continue;
```

```
    .........
}
```

Sada nam u dijelu kôda koji obrađuje pozitivne brojeve treba neki drugi način izlaza iz petlje.

Pogledajmo još jedan primjer `continue` naredbe:

```
i=0;
while(i < 10)
{
   ch=getchar();
   if(ch == '\n') continue;
   putchar(ch);
   i++;
}
```

Ovaj će kôd pročitati 10 znakova ne računajući znak za prijelaz u novi red. Naime, nakon `continue`, petlja se nastavlja iza posljednje naredbe u tijelu. U `while` petlji to znači da se preskače naredba `i++;` koja povećava brojač i prva sljedeća naredba je ispitivanje uvjeta `i < 10`.

Naredba `continue` u `for` petlji malo je drugačiji. Pogledajmo primjer

```
for(i=0; i < 10; ++i)
{
   ch=getchar();
   if(ch == '\n') continue;
   putchar(ch);
}
```

Nakon izvršenja naredbe `continue` dolazimo ponovo iz posljednje naredbe u tijelu petlje, tj. `putchar(ch);` se preskače. No u ovom slučaju prva sljedeća naredba je povećanje brojača, `++i;` a zatim testiranje `i < 10`. Znak za prijelaz u novi red je sada uključen u ukupan zbroj učitanih znakova.

Ako se naredba `break` nalazi u petlji koja je ugnježdena u nekoj drugoj petlji, onda se pomoću `break` izlazi samo iz dublje petlje.

6.8 `goto` naredba

`goto` naredba prekida sekvencijalno izvršavanje programa i nastavlja izvršavanje s naredbom koja je označena labelom koja se pojavljuje u `goto`. Oblik joj je

```
goto label;
```

gdje je `label` identifikator koji služi za označavanje naredbe kojom se nastavlja program. Sintaksa je

```
label: naredba;
```

Na primjer,

```
  scanf("%d",&i);
  while{i<=100){
     .......
     if (i<0) goto error;
     .......
     scanf("%d",&i);
  }
       .......
 /* detekcija greske */
error: {
         printf("Greska : negativna vrijednost!\n");
         exit(-1);
       }
```

Labela na koju se vrši skok mora biti unutar iste funkcije kao i `goto` naredba. Drugim riječima, pomoću `goto` se ne može izaći iz funkcije.

Naredbe `break` i `continue` mogu se izvesti pomoću `goto` naredbe. Na primjer, kôd

```
for(...) {
  .......
  if (...) continue;
  ........
}
```

je ekvivalentan s

```
for(...) {
  .......
  if (...) goto cont;
  .......
  cont: ;
}
```

Slično vrijedi i za `continue` unutar `while` i `do-while` petlje.

Program napisan uz upotrebu `goto` naredbe općenito je teže razumjeti od programa koji ju ne koriste. Nadalje, neke upotrebe `goto` naredbe su

posebno zbunjujuće. Na primjer, skok u `if` ili `else` blok izvan `if` naredbe, ili skok u blok naredbu s preskokom deklaracija lokalnih varijabli na početku bloka. Općenito stoga upotrebu `goto` naredbe treba izbjegavati.

Poglavlje 7

Funkcije

Funkcija je programska cjelina koja uzima neke ulazne podatke, izvršava određen niz naredbi i vraća rezultat svog izvršavanja pozivnom programu.

Pomoću funkcija razbijamo složene programske zadaće na niz jednostavnijih cjelina. Time postižemo veću jasnoću programa i olakšavamo buduće modifikacije. Svaka funkcija treba biti osmišljena tako da obavlja jednu dobro definiranu zadaću te da korisnik funkcije ne mora poznavati detalje njene implementacije da bi ju koristio. Tada je funkciju moguće koristiti u različitim programima, kao što je to slučaj s funkcijama iz standardne biblioteke.

7.1 Definicija funkcije

Definicija funkcija ima oblik

```
tip_podatka ime_funkcije(tip_1 arg_1, ... ,tip_n arg_n)
{
        tijelo funkcije
}
```

gdje je `tip_podatka` tip podatka koji će funkcija vratiti kao rezultat svog izvršavanja. Unutar zagrada nalazi se deklaracija formalnih argumenata funkcije. Prvi argument `arg_1` je varijabla tipa `tip_1` itd. Deklaracije pojedinih argumenata međusobno se odvajaju zarezom. Unutar vitičastih zagrada pojavljuje se tijelo funkcije koje se sastoji od deklaracija varijabli i izvršnih naredbi.

Funkcija vraća rezultat svog izvršavanja pomoću naredbe `return`. Opći oblik te naredbe je

```
return izraz;
```

Vrijednost izraza se vraća dijelu programa koji poziva funkciju. Izraz se može staviti u oble zagrade ali to nije nužno. Vrijedi sljedeće pravilo:

- Funkcija može vratiti aritmetički tip, strukturu, uniju ili pokazivač ali ne može vratiti drugu funkciju ili polje.

Ako je tip izraza u naredbi `return` različit od tipa podatka koji funkcija vraća, izraz će biti konvertiran u tip podatka. Takvu situaciju treba naravno izbjegavati.

Na primjer, sljedeća funkcija pretvara mala slova (engleske abecede) u velika. Formalni argument je samo jedan i tipa je `char`; vraćena vrijednost je također tipa `char`. Ime funkcije je `malo_u_veliko`.

```
char malo_u_veliko(char z)
{
   char c;
   c= (z >= 'a' && z <= 'z') ? ('A' + z - 'a') : z;
   return c;
}
```

Funkcija pretpostavlja da slova abecede imaju rastuće uzastopne kodove te da mala slova prethode velikim ili obratno. Ta je pretpostavka ispunjena za ASCII način kodiranja, koji je najrašireniji, ali nije na primjer za EBCDIC način kodiranja.

Funkcija se poziva navođenjem svog imena i liste stvarnih argumenata. Funkciju iz prethodnog primjera u glavnom programu pozvamo na sljedeći način:

```
int main(void)
{
   char malo, veliko;

   printf("Unesite malo slovo: ");
   scanf("%c", &malo);
   veliko = malo_u_veliko(malo);
   printf("\nUneseno slovo pretvoreno u veliko = %c\n",veliko);
   return 0;
}
```

U gornjem programu `malo` je stvarni argument koji se kopira na mjesto formalnog argumenta `z`. Znak koji funkcija vraća smješten je u varijablu `veliko` prilikom poziva

```
veliko = malo_u_veliko(malo);
```

Važno je primijetiti da funkcija prima kopiju stvarnog argumenta tako da nikakva manipulacija unutar same funkcije ne može promijeniti stvarni argument (`malo` u našem primjeru).

Funkcija koja vraća neku vrijednost najčešće se poziva u izrazu pridruživanja kao u gornjem primjeru, no funkciju je moguće pozvati i unutar svakog drugog izraza koji uzima tip podatka koji ona vraća. Na primjer, u glavnom programi mogli smo pisati

```
printf("\nUneseno slovo pretvoreno u veliko = %c\n",
                                 malo_u_veliko(malo));
```

i tako eliminirati varijablu `veliko`. Analogno, pri pozivu funkcije na mjestu stvarnih argumenata mogu se nalaziti izrazi umjesto varijabli. Pri tome će izrazi prvo biti izračunati, a onda će izračunate vrijednosti biti proslijeđene funkciji. Na primjer, funkciju iz matematičke biblioteke

```
double sqrt(double)
```

koja računa drugi korijen broja možemo pozvati na sljedeći način:

```
double x,y;
 .......
y=sqrt(2*x-3)
```

i varijabla `y` će sadržavati vrijednost $\sqrt{2x-3}$.

Ukoliko se programski tok unutar funkcije grana, onda je moguće imati više `return` naredbi unutar iste funkcije. Na primjer, prethodna funkcija mogla je biti napisana u obliku

```
char malo_u_veliko(char z)
{
   if(z >= 'a' && z <= 'z')
       return('A' + z - 'a');
   else
       return z;
}
```

Kada funkcija ne vraća nikakvu vrijednost onda se za tip “vraćene vrijednosti” koristi ključna riječ `void`. Na primjer, u funkciji

```
void maximum(int x, int y)
{
    int z;
    z=(x>=y) ? x : y;
    printf("\nMaksimalna vrijednost =%d",z);
    return;
}
```

`void` označava da nikakva vrijednost neće biti vraćena u pozivni program. Naredba `return` samo transferira programski tok u pozivni program i stoga uz nju ne stoji `izraz`. U takvoj situaciji naredba `return` može biti izostavljena, no radi preglednosti programa bolje ju je zadržati.

Funkcija koja ne uzima nikakve argumente definira se na sljedeći način:

```
tip_podatka ime_funkcije(void)
{
      tijelo funkcije
}
```

Ključna riječ `void` unutar zagrada označava da funkcija ne uzima argumente. Takva se funkcija poziva s praznim zagradama kao

```
ime_funkcije();
```

ili najčešće u izrazu pridruživanja

```
varijabla=ime_funkcije();
```

Zagrade su obavezne, one ustvari informiraju prevodioc da je simbol koji stoji ispred zagrade, `ime_funkcije`, ime neke funkcije.

Tijelo funkcije sastoji se od deklaracija varijabli i izvršnih naredbi. Deklaracije varijabli moraju prethoditi prvoj izvršnoj naredbi. Budući da na početku svakog bloka možemo deklarirati varijable, koje su onda lokalne za taj blok (vidi sekciju 9.1.1), imamo mogućnost definiranja lokalnih varijabli na početku svakog bloka unutar tijela funkcije; na primjer, u tijelu `if` naredbe i slično.

To je pravilo izmijenjeno u standardu C99. Prema njemu deklaracije varijabli ne moraju prethoditi prvoj izvršnoj naredbi u bloku, tako da se izvršne naredbe i deklaracije mogu ispreplitati.

Napomena. U tradicionalnom C-u definicija funkcije je imala oblik

```
tip_podatka ime_funkcije(arg_1, ... ,arg_n)
tip_1 arg_1;
...
tip_n arg_n;
{
        tijelo funkcije
}
```

Na primjer,

```
char malo_u_veliko(z)
char z;
{
   char c;
   c= (z >= 'a' && z <= 'z') ? ('A' + z - 'a') : z;
   return c;
}
```

Takva definicija je dozvoljena i u ANSI C-u, no standard C99 ju proglašava zastarjelom, što znači da ju u budućnosti prevodioci možda neće podržavati. U svakom slučaju takvu sintaksu treba izbjegavati.

7.2 Deklaracija funkcije

Svaka bi funkcija prije svoga poziva u programu trebala biti *deklarirana*. Deklaracija informira prevodilac o imenu funkcije, broju i tipu argumenata te tipu povratne vrijednosti (tipu funkcije). Uloga deklaracije (prototipa) je omogućiti prevodiocu kontrolu ispravnosti poziva funkcije.

Ako je funkcija definirana u istoj datoteci u kojoj se poziva, prije svog prvog poziva, onda definicija služi kao deklaracija te zasebna deklaracija nije potrebna. Takva se situacija nalazi u jednostavnim programima koji su smješteni u jednu datoteku i u kojima su sve funkcije definirane prije `main()` funkcije. Ukoliko funkcija `f1()` poziva funkciju `f2()` treba paziti da je `f2()` definirana prije `f1()`. U takvoj situaciji nikakve posebne deklaracije nisu potrebne.

Ako želimo da su funkcije koje koristimo definirane nakon `main()` funkcije ili općenitije iza mjesta na kojem se pozivaju, trebamo koristiti njihove deklaracije, odnosno prototipove.

Deklaracija funkcije ima oblik

```
tip_podatka ime_funkcije(tip_1 arg_1, ... ,tip_n arg_n);
```

Vidimo da je posve ista definiciji s tom razlikom da nema tijela funkcije i deklaracija završava točka-zarezom. Deklaracija se treba pojaviti prije poziva funkcije ukoliko funkcija nije prije toga definirana. Definicija i deklaracija moraju biti u skladu.

Ilustrirajmo to na jednom primjeru. Prvo, funkcija može biti definirana prije svog poziva:

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

```
void maximum(int x, int y)
{
    int z;
    z=(x>=y) ? x : y;
    printf("\nMaksimalna vrijednost =%d\n",z);
    return;
}
int main(void)
{
    int x,y;

    printf("Unesite dva cijela broja: ");
    scanf("%d %d", &x,&y);
    maximum(x,y);
    return 0;
}
```

U trenutku svog poziva prevodilac zna da je `maximum` funkcija koja uzima dva argumenta tipa `int` i ne vraća ništa.

Ako želimo definiciju funkcije smjestiti nakon funkcije `main()` uvodimo deklaraciju, odn. prototiop funkcije:

```
#include <stdio.h>

int main(void)
{
    int x,y;
    void maximum(int x, int y);

    printf("Unesite dva cijela broja: ");
    scanf("%d %d", &x,&y);
    maximum(x,y);
    return 0;
}

void maximum(int x, int y)
{
     int z;
     z=(x>=y) ? x : y;
     printf("\nMaksimalna vrijednost =%d\n",z);
     return;
}
```

Ovdje je funkcija `maximum()` deklarirana kao i druge varijable u glavnom programu. Primijetimo da varijable `x` i `y` deklarirane u deklaraciji

`void maximum(int x, int y);`

imaju samo formalni značaj, i mogu se ispustiti. Formalno govoreći te dvije varijable nisu vidljive izvan prototipa funkcije i stoga nisu u koliziji s varijablama istog imena deklariranim u funkciji `main()` (vidi sekciju 9.1.3).

Druga mogućnost, koja se češće koristi, je deklarirati funkciju izvan `main()` funkcije kao u ovom slučaju.

```
#include <stdio.h>
void maximum(int, int);

int main(void)
{
    int x,y;

    printf("Unesite dva cijela broja: ");
    scanf("%d %d", &x,&y);
    maximum(x,y);
    return 0;
}

void maximum(int x, int y)
{
    int z;
    z=(x>=y) ? x : y;
    printf("\nMaksimalna vrijednost =%d\n",z);
    return;
}
```

Uočimo da smo ovdje u prototipu ispustili nazive varijabli. Prevodiocu su potrebni samo broj i tipovi varijabli koje funkcija uzima. Spomenimo još da ovakvo ispuštanje imena varijabli u prototipu ne treba raditi ako imena varijabli podsječaju na njihovu funkciju i time služe dokumentiranju programa.

Prototipovi su uvedeni u C sa standardom C90 (ANSI standardom). Radi kompatibilnosti sa starijim programima dozvoljeno je koristiti i funkcije koje nisu prethodno deklarirane. Pri tome vrijedi sljedeće pravilo:

- Prevodilac pretpostavlja da funkcija vraća podatak tipa `int` i ne pravi nikakve pretpostavke o broju i tipu argumenata.

U tom slučaju kažemo da je funkcija definirana implicitno pravilima prevodioca. Činjenica da prevodilac ne pravi nikakve pretpostavke o broju i tipu argu-

menata znači samo da nije u stanju provjeriti je li poziv funkcije korektan ili nije. Ukoliko nije to će biti otkriveno tek prilikom izvršavanja programa. Za korektan poziv potrebno je da se stvarni i formalni argumenti slažu u broju i tipu (preciznije u sljedećoj sekciji).

U starijim programima moguće je naći deklaraciju funkcije tipa

```
double f();
```

Takva deklaracija informira prevodilac o tipu funkcije (`double`), ali ne i o argumentima. Prevodilac ponovo ne radi nikakve pretpostavke o broju i tipu argumenta. Takve deklaracije treba izbjegavati.

Spomenimo na ovom mjestu još i to da u C++ gornja deklaracija znači

```
double f(void);
```

U C++ sve funkcije moraju imati prototip.

Napomena. Mogućnost pozivanja funkcija koje nisu prethodno deklarirane (prototipom ili definicijom) te funkcija s nepotpunim deklaracijama dozvoljeno je jedino zbog kompatibilnosti sa starijim programima pisanim u tradicionalnom C-u. U svim novim programima imperativ je koristiti prototipove.

7.3 Prijenos argumenata

Argumenti deklarirani u definiciji funkcije nazivaju se formalni argumenti. Izrazi koji se pri pozivu funkcije nalaze na mjestima formalnih argumenata nazivaju se stvarni argumenti.

Prilikom poziva funkcije stvarni argumenti se izračunavaju (ako su izrazi) i kopiraju u formalne argumente. Funkcija prima kopije stvarnih argumenata što znači da ne može izmijeniti stvarne argumente. Sljedeći primjer ilustrira način prijenosa argumenata.

```
#include <stdio.h>

void f(int x) {
    x+=1;
    printf("\nUnutar funkcije x=%d",x);
    return;
}
int main(void)
{
    int x=5;

    printf("\nIzvan funkcije x=%d",x);
    f(x);
    printf("\nNakon poziva funkcije x=%d",x);
```

```
    return 0;
}
```

Rezultat izvršavanja programa će biti:

```
Izvan funkcije x=5
Unutar funkcije x=6
Nakon poziva funkcije x=5
```

Istaknimo najvažnija pravila o prijenosu argumenata.

- Broj stvarnih argumenata pri svakom pozivu funkcije mora biti jednak broju formalnih argumenata.
- Ako je funkcija ispravno deklarirana (ima prototip), tada se stvarni argumenti čije se tip razlikuje od odgovarajućih formalnih argumenta konvertiraju u tip formalnih argumenata, isto kao pri pridruživanju. Takva konverzija pri tome mora biti moguća.
- Ukoliko je funkcija na mjestu svog poziva deklarirana implicitno pravilima prevodioca, tada prevodioc postupa na sljedeći način: Na svaki stvarni argument cjelobrojnog tipa primijenjuje se integralna promocija (konverzija argumenata tipa `short` i `char` u `int`), a svaki stvarni argument tipa `float` konvertira se u tip `double`. Nakon toga broj i tip (konvertiranih) stvarnih argumenta mora se podudarati s brojem i tipom formalnih argumenata da bi poziv funkcije bio korektan.
- Redosljed izračunavanja stvarnih argumenat nije definiran i može ovisiti o implemetaciji.

Pogledajmo na primjer sljedeći program:

```
#include <stdio.h>

int main(void)
{
   float x=2.0;

   printf("%d\n",f(2));  // Neispravno
   printf("%d\n",f(x));  // Ispravno
   return 0;
}
```

```
int f(double x) {
   return (int) x*x;
}
```

U prvom pozivu funkcije `int f(double)` glavni program će funkciji poslati cjelobrojno 2 kao argument, a funkcija će argument interpretirati kao realan broj dvostruke preciznosti. To će dati ili pogrešan rezultat ili prekid izvršavanja programa. U drugom pozivu funkcije argument tipa `float` bit će konvertiran u `double` i funkcija će primiti ispravan argument. Uočimo da ako definiramo funkciju `f()` tako da uzima argument tipa `float`, onda niti jedan poziv ne bi bio korektan.

Uvedemo li funkcijski prototip (što dobar stil programiranja nalaže)

```
int f(double);
int main(void)
    {
   float x=2.0;

   printf("%d\n",f(2));
   printf("%d\n",f(x));
   return 0;
}

int f(double x) {
  return (int) x*x;
}
```

funkcija će u oba slučja biti ispravno pozvana jer će cjelobrojni argument 2 biti konvertiran u tip `double`.

S druge strane kôd

```
int main(void)
{
   float x=2.0;

   printf("%d\n",f(2));    /* greska */
   printf("%d\n",f(x));    /* ispravno */
   return 0;
}

double f(double x) {
   return  x*x;
}
```

ne bi bio uspješno preveden. Naime, funkcija `f` je uvedena u glavnom programu bez eksplicitne deklaracije pa prevodilac pretpostavlja da se radi o funkciji tipa `int` (tj. o funkciji koja vraća vrijednost tipa `int`). Stoga definiciju `double f(double x)` prevodilac shvaća kao redefiniranje simbola `f`, što nije dozvoljeno.

Prijenos argumenata funkciji u C-u vrši se po vrijednosti. To znači da se stvarni argumenti kopiraju u formalne argumente te funkcija stvarne argumente ne može dohvatiti. Ukoliko se želi da funkcija mijenja stvarne argumente, onda ona mora dobiti njihove adrese (vidi sekciju 11.2).

Ako je polje argument funkcije, onda se ono ne prenosi funkciji "po vrijednosti", već funkcija dobiva pokazivač na prvi element polja pomoću kojeg može dohvatiti svaki element polja putem indeksa polja. Razlog za ovakav način prijenosa polja je u tome što se dimenzija polja ne zna unaprijed, a kopiranje svih elemenata velikih polja bilo bi vrlo neefikasno (vidi sekciju 10.5). Ukoliko funkcija ima argument tipa polja koje se u njoj ne mijenja, onda ga treba deklarirati s modifikatorom `const` (sekcija 11.5).

7.4 Inline funkcije

Svaki poziv funkcije predstavlja određen utrošak procesorskog vremena. Procesor treba zaustaviti izvršavanje glavnog programa, spremiti sve podatke nužne za njegov nastavak nakon izlaska iz funkcije, predati funkciji argumente i početi izvršavati kôd funkcije. Kod malih funkcija, kao što je to na primjer

```
double f(double x) {
   return  x*x;
}
```

sam poziv funkcije može uzeti više procesorskog vremena nego izvršavanje kôda funkcije i time značajno usporiti program ako se funkcija često poziva.

Da bi se to izbjeglo C99 dozvoljava da se funkcija deklarira `inline`:

```
inline double f(double x) {
   return  x*x;
}
```

Ključna riječ `inline` je sugestija prevodiocu da ekspandira tijelo funkcije na mjestu na kojem se ona poziva, izbjegavajući tako poziv funkcije. Prevodilac nije dužan ispuniti taj zahtijev na svakom mjestu.

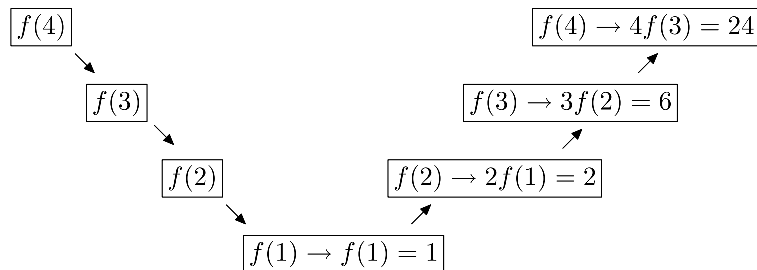
Osnovno ograničenje kod upotrebe `inline` funkcije je što njena definicija (a ne samo deklaracija) mora biti vidljiva na mjestu poziva funkcije. To ne predstavlja problem kod statičih funkcija, no ako je funkcija definirana u nekoj drugoj datoteci, taj će uvjet biti narušen. Tada se postupa tako da se definicija funkcije stavi u datoteku zaglavlja koja se potom uključi u svaku `.c` datoteku koja funkciju koristi.

7.5 Rekurzivne funkcije

C dozvoljava da se funkcije koriste rekurzivno, odnosno da pozivaju same sebe. Na primjer, za računanje $n! = 1 \cdot 2 \cdot 3 \cdots n$ možemo napisati rekurzivnu funkciju

```
long faktorijeli(long n) {
    if(n<=1) return 1;
    else return n*faktorijeli(n-1);
}
```

Funkcija (rekurzivno) poziva samu sebe `n-1` puta kako bi izračunala `n!`. Uočimo da nakon poziva funkciji `faktorijeli` s argumentom `n`, strogo većim od 1, dolazi do poziva funkcije `faktorijeli` s argumentom `n-1`. U toj funkciji dolazi do poziva funkcije `faktorijeli` s argumentom `n-2` i tako dalje, sve dok ne dođe do poziva funkcije `faktorijeli` s argumentom 1. Tada najdublje ugnježđena funkcija `faktorijeli` vrati vrijednost 1. Funkcija koja ju je pozvala vrati vrijednost `2*1`, sljedeća vrati `3*2*1` itd. dok konačno prvo pozvana funkcija ne vrati vrijednost `n!`.



Uočimo da svaka rekurzivna funkcija mora imati određen kriterij izlaska iz rekurzije. U ovom slučaju je to uvjet `n<=1`.

Rekurzivno korištenje funkcija često rezultira vrlo elegantnim algoritmima no ono ima i svojih nedostataka. Osnovni je taj što je npr. u ovom slučaju bilo potrebno n poziva funkcije da bi se izračunao broj $n!$. Nasuprot tome, faktorijele možemo računati i nerekurzivnim postupkom kao npr. u funkciji

```
long faktorijeli(long n) {
     long f=1;
     for(;n>1;n--) f*=n;
     return f;
}
```

što je efikasnije jer trebamo samo jedan poziv funkcije.

Pogledajmo još jedan primjer rekurzivne funkcije:

```
void unos(void){
     char znak;
     if((znak=getchar())!='\n') unos();
     putchar(znak);
}
```

Funkcija učitava znakove sa standardnog ulaza sve dok ne naiđe do prijelaza u novu liniju i ispisuje učitane znakove. Ako izvršimo program

```
#include <stdio.h>
void unos(void);

int main(void)
{
   printf("\n Unesite niz znakova: ");
   unos();
   return 0;
}
```

dobit ćemo ovaj rezultat:

```
 Unesite niz znakova: Zdravo

ovardZ
```

Prvo je ispisan znak za prijelaz u novi red, a zatim niz `Zdravo` u obrnutom redoslijedu.

Svaki poziv funkcije `unos` učita jedan znak sa standardnog ulaza i spremi ga u lokalnu varijablu `znak`. Ukoliko učitani znak nije znak prijelaza u novi red ponovo se poziva funkcija `unos`.

Svaka varijabla definirana unutar funkcije vidljiva je samo u toj funkciji (vidi sekciju 9.1.1). Ona se kreira kod poziva funkcije i nestaje nakon izlaza iz funkcije. Stoga pri svakom novom pozivu funkcije `unos` kreira se nova varijabla s imenom `znak`. Budući da varijabla definirana u funkciji (unutarnja varijabla) nije vidljiva izvan te funkcije, prisutnost više različitih varijabli s imenom `znak` je dozvoljeno.

Dakle, pri svakom novom pozivu funkcije `unos` bit će pročitan i pohranjen jedan znak sve dok ne naiđemo do znaka za prijelaz u novi red. Tada pozvana funkcija ispiše znak za prijelaz u novi red i izađe. Funkcija koja ju je pozvala ispisat će znak koji je pročitala (znak `'o'`) i predaje kontrolu funkciji

koja ju je pozvala. Na taj se način svi upisani znakovi ispisuju u obrnutom redoslijedu.

U ovom jednostavnom primjeru vidi se drugi nedostatak rekurzivnog programa. Pri svakom novom rekurzivnom pozivu funkcije kreiraju se ponovo sve lokalne varijable. U gornjem primjeru koristili smo onoliko varijabli koliko je bilo upisanih znakova, mada bi jedna bila sasvim dovoljna.

7.6 Funkcije s varijabilnim brojem argumenata

Funkcije `scanf` i `printf` primjeri su funkcija koje primaju varijabilan broj argumenata. Datoteka zaglavlja `<stdarg.h>` sadrži neke definicije i makro naredbe (vidi sekciju 8) koje omogućavaju programeru pisanje valstitih funkcija s varijabilnim brojem argumenata.

U `<stdarg.h>` je definiran tip podatka `va_list` koji predstavlja pokazivač na listu nedeklariranih argumenata. Stoga na početku funkcije treba definirati varijablu tipa `va_list`. Na primjer:

```
va_list pa;
```

Varijabla `pa` se inicijalizira pomoću funkcije `va_start`. Ona uzima kao prvi argument varijablu `pa`, a kao drugi argument posljednji deklarirani argument funkcije. To znači da funkcija mora imati barem jedan deklarirani argument. Kao rezultat varijabla `pa` će pokazivati na prvi nedeklarirani argument funkcije.

Kada se želi pročitati jedan nedeklarirani argument koristi se `va_arg`. Pri tome moramo znati tip argumenta. `va_arg` uzima kao prvi argument pokazivač `pa`, a kao drugi tip argumenta te vraća vrijednost argumenta i povećava pokazivač `pa` tako da pokazuje na sljedeći nedeklarirani argument. Na primjer, ako je argument na koji `pa` pokazuje tipa `int` i `vali` varijabla tipa int, onda bismo imali poziv oblika

```
vali=va_arg(pa,int);
```

Nakon njega `pa` će pokazivati na sljedeći nedeklarirani argument funkcije.

Nedeklarirani argumenti funkcije pri pozivu funkcije podvrgnuti su standardnoj promociji koja vrijedi za funkcije bez prototipa: manji integralni tipovi konvertiraju se u `int`, a `float` u `double`.

Konačno, deklaracija funkcije s varijabilnim brojem argumenata ima oblik:

```
tip ime_funkcije(tip_1 arg1, tip_2 arg_2,...);
```

Ovdje je dat primjer s dva deklarirana argumenata, mada broj deklariranih argumenata može biti bili koji broj veći ili jednak 1. Tri točke iza posljednjeg deklariranog argumenta označava da funkcija uzima varijabilan broj argumenata. Pokažimo jedan jednostavan primjer. Napisat ćemo funkciju koja sumira `n` brojeva tipa `double` koji su joj dani kao argumenti. Pri tome je `n` zadan kao prvi argument.

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

double suma(int, ...);

int main(void)
{
   double s,t;

   s=suma(3,1.0,2.0,3.0);
   t=suma(5,1.0,2.0,3.0,4.0,5.0);
   printf("%g %g\n",s,t);
   return 0;
}

double suma(int n,...)
{
   va_list ap;
   double total=0.0;
   int i;

   va_start(ap,n);
   for(i=0; i<n; ++i)
      total += va_arg(ap,double);
   va_end(ap);
   return total;
}
```

Uočimo da kroz deklarirane argumente moramo na neki način funkciji dati informaciju sa koliko je argumenata pozvana i kojeg su tipa. Funkcije `printf` i `scanf` to rade putem kontrolnog stringa.

Nakon što su svi parametri pročitani potrebno je pozvati funkciju `va_end` koja završava proces čitanja argumenta.

Poglavlje 8

Preprocesorske naredbe

Prije prevođenja izvornog koda u objektni ili izvršni izvršavaju se preprocesorske naredbe. Svaka linija izvornog koda koja započinje znakom # predstavlja[1] jednu preporocesorsku naredbu. Njih izvršava zaseban dio prevodioca koji se naziva preprocesor, i koji prije samog procesa prevođenja na osnovu preprocesorskih naredbi mijenja izvorni kôd.

Opći oblik preprocesorskih naredbi je

```
#naredba parametri
```

i one nisu sastavni dio jezika C te ne podliježu sintaksi jezika. Svaka preprocesorska naredba završava krajem linije (a ne znakom točka-zarez). Neke od preprocesorskih naredbi su

```
#include #define #undef #if #ifdef #ifndef #elif #else
```

8.1 Naredba #include

Naredba `#include` može se pojaviti u dva oblika:

```
#include "ime_datoteke"
```

ili

```
#include <ime_datoteke>
```

U oba slučaja preprocesor će obrisati liniju s `#include` naredbom i uključiti sadržaj datoteke `ime_datoteke` u izvorni kôd, na mjestu `#include` naredbe.

[1]Znaku # mogu prethoditi bjeline, no neki stariji prevodioci (makroprocesori) zahtijevaju da # bude prvi znak u liniji.

Ako je ime_datoteke navedeno unutar navodnika, onda preprocesor datoteku traži u direktoriju u kojem se izvorni program nalazi. Ime datoteke navedeno između oštrih zagrada signalizira da se radi o sistemskoj datoteci (kao npr. stdio.h), pa će preprocesor datoteku tražiti na mjestu određenom operacijskim sustavom. Pod UNIX-om to je najčešće direktorij /usr/include a postoji i mogućnost da se prevodiocu da instrukcija (-I) da te datoteke traži u unaprijed zadanom direktoriju.

Datoteke zaglavlja se najčešće koriste za uključivanje sljedećih veličina:

- Simboličkih konstanti — U stdio.h tako imamo EOF, NULL itd.
- Makro funkcija — Na primjer getchar() koji je obično definiran kao getc(stdin) gdje je getc() makro funkcija.
- Deklaracije funkcija — U string.h je deklariran niz funkcija za rad sa stringovima.
- Deklaracije struktura — U stdio.h se definira struktura FILE.
- Definicije tipova — Na primjer size_t, time_t, itd.

Datoteka uključena u izvorni kôd pomoću #include naredbe može i sama sadržavati #include naredbe.

8.2 Naredba #define

Njena je forma sljedeća:

```
#define ime tekst_zamjene
```

Preprocesor će od mjesta na kome se #define naredba nalazi do kraja datoteke svako pojavljivanje imena ime zamijeniti s tekstom tekst_zamjene. Do zamjene neće doći unutar znakovnih nizova, tj. unutar dvostrukih navodnika. Tako će na primjer dio izvornog kôda

```
#define PI 3.14
   ......
  x=2*r*PI;
```

prije prevođenja biti zamijenjen s

```
   ......
  x=2*r*3.14;
```

no u naredbi `printf("PI");` do zamjene ne bi došlo. Svako ime definirano u nekoj `#define` naredbi nazivamo makro.

Naredba `#define` može se koristiti i bez teksta zamjene kao

```
#define ime
```

Nakon te naredbe `ime` je definirano, što se može ispitivati pomoću `#if` naredbe. Ako je neko ime definirano pomoću `#define`, definicija se može poništiti pomoću naredbe `#undef` (vidi sekciju 8.4).

8.3 Parametrizirana `#define` naredba

Naredba `#define` osim za definiranje simboličkih konstanti služi i za definiranje parametriziranih makroa.

U parametriziranoj `#define` naredbi simboličko ime i tekst koji zamijenjuje simboličko ime sadrže argumente koji se prilikom poziva makroa zamijenjuju stvarnim argumentima. Sintaksa je sljedeća:

```
#define  ime(argumenti) text_zamjene
```

Jedan primjer parametriziranog makroa je

```
#define  max(A,B) ((A)>(B) ? (A) : (B))
```

gdje su `A` i `B` argumenti. Ako se u kodu pojavi naredba

```
  x=max(a1,a2);
```

preprocesor će ju zamijeniti s

```
  x=((a1)>(a2) ? (a1) : (a2));
```

Formalni argumenti (parametri) `A` i `B` zamijenjeni su sa stvarnim argumentima `a1` i `a2`. Ako pak na drugom mjestu imamo naredbu

```
  x=max(a1+a2,a1-a2);
```

ona će biti zamijenjena s

```
  x=((a1+a2)>(a1-a2) ? (a1+a2) : (a1-a2));
```

Vidimo da je parametrizirani makro vrlo sličan funkciji no u njemu nema funkcijskog poziva i prenošenja argumenata, pa je stoga efikasniji.

Sličnost makroa i funkcije može zavarati. Ako bismo makro `max` pozvali na sljedeći način

```
  max(i++,j++);
```

varijable i, j ne bi bile inkrementirane samo jednom (kao pri funkcijskom pozivu) već bi veća varijabla bila inkrementirana dva puta.

Argumente makroa treba stavljati u zagrade kako bi se izbjegla situacija ilustrirana u sljedećem primjeru: ako definiramo

```
#define  kvadrat(x)  x * x
```

onda bi `kvadrat(x+1)` dao `x+1 * x+1`, što očito nije ono što smo htjeli.
Zadatak. Pretpostavimo da smo definirali

```
#define  max(A,B) (A)>(B) ? (A) : (B)
```

Što bi bio rezultat poziva

```
#define  x=y+max(0.1,0.2);
```

ovisno o y?

Unutar dvostrukih navodnika parametar makroa neće biti zamijenjen stvarnim argumentom. Na primjer, ako imamo definiciju

```
#define  PSQR(X)  printf("Kvadrat od X je %d.\n",((X)*(X)));
```

onda bi poziv

```
 PSQR(5);
```

ispisao

```
Kvadrat od X je 25.
```

To očito nije ono što smo htjeli, jer je X u znakovnom nizu tretiran kao običan znak, a ne kao parametar makroa. To se može ispraviti pomoću operatora #. To je operator koji makro parametar pretvara u string. Korektana definicija bi bila:

```
#include <stdio.h>
#define  PSQR(X)  printf("Kvadrat od " #X " je %d.\n",((X)*(X)));

int main(void)
{
    int y=5;

    PSQR(y);
    PSQR(2+4);
    return 0;
}
```

Program će ispisati

```
Kvadrat od y je 25.
Kvadrat od 2+4 je 36.
```

Napomena. Neke su "funkcije" deklarirane u `<stdio.h>` ustvari makroi, na primjer `getchar` i `putchar`. Isto tako, funkcije u `<ctype.h>` uglavnom su izvedene kao makroi.

Definiciju nekog imena može se poništiti pomoću `#undef`. Na primjer,

```
#undef getchar
```

U `#define` naredbi tekst zamjene se prostire od imena koje definiramo do kraja linije. Ako želimo da ime bude zamijenjeno s više linija teksta moramo koristiti kosu crtu (\) na kraju svakog reda osim posljednjeg. Na primjer, makro za inicijalizaciju polja možemo definirati na sljedeći način:

```
#define INIT(polje, dim)  for(int i=0; i < (dim); ++i) \
                                 (polje)[i]=0.0;
```

Osnovna prednost parametriziranih makroa pred običnim funkcijama je u brzini izvršavanja – makro nam štedi jedan funkcijski poziv. To može biti značajno ukoliko se makro poziva veliki broj puta. Prednost može biti i to što mako ne kontrolira tip svojih argumenata. Tako npr. makro `kvadrat(x)` možemo koristiti s `x` bilo kojeg skalarnog tipa. S druge strane osnovni je nedostatak što prevodilac ne može provjeriti korektnost poziva makroa za vrijeme prevođenja. U tom smislu je `inline` funkcija bolje rješenje.

Generalna je konvencija da se parametrizirani makroi pišu velikim slovima.

8.4 Uvjetno uključivanje

Pomoću preprocesorskih naredbi `#if`, `#else`, `#elif` možemo uvjetno uključivati ili isključivati pojedine dijelove programa. Naredba `#if` ima sljedeći oblik:

```
#if uvjet
    blok naredbi
#endif
```

Ukoliko je `uvjet` ispunjen blok naredbi između `#if uvjet` i `#endif` bit će uključen u izvorni kôd; ako `uvjet` nije ispunjen blok neće biti uključen.

Uvjet koji se pojavljuje u `#if` naredbi je konstantan cjelobrojni izraz. Nula se interpretira kao laž, a svaka vrijednost različita od nule kao istina. Simbolička imena se prije izračunavanja izraza zamijenjuju svojim vrijednostima. Ukoliko se u uvjetu pojavi simboličko ime koje nije prije toga definirano nekom `#define` naredbom, ono se zamijenjuje nulom.

Najčešće se uključivanje odnosno isključivanje dijela kôda pomoću naredbe `#if` čini u ovisnosti o tome da li je neka varijabla definirana ili nije. Tu nam pomaže izraz `defined(ime)` koji daje 1 ako je `ime` definirano, a 0 ako nije. Na primjer,

```
#if !defined(__datoteka.h__)
#define __datoteka.h__

  /* ovdje dolazi datoteka.h  */

#endif
```

Ako varijabla `__datoteka.h__` nije definirana ona će u sljedećoj `#define` naredbi biti definirana i `datoteka.h` će biti uključena. U suprotnom će cijela `datoteka.h` biti jednostavno preskočena. To je standardna tehnika kojom se izbjegava višestruko uključivanje `.h` datoteka u program (provjerite npr. datoteku `stdio.h`).

Budući da se konstrukcije `#if defined` i `#if !defined` često pojavljuju postoje kraći izrazi s istim značenjem: `#ifdef` i `#ifndef`. Tako smo prethodnu konstrukciju mogli napisati u obliku

```
#ifndef __datoteka.h__
#define __datoteka.h__

  /* ovdje dolazi datoteka.h  */

#endif
```

Zagrade oko varijabli nisu obavezne.

Napomena. `#ifdef ime` i `#if ime` su ekvivalentne naredbe ako je `ime` simboličko ime koje, kad je definirano, daje cjelobrojni konstantan izraz, različit od nule. ☐

Složene `if` naredbe grade se pomoću `#else` i `#elif`, koji ima značenje `else if`. Na primjer,

```
#if SYSTEM == SYSV
   #define DATOTEKA "sysv.h"
#elif SYSTEM == BSD
```

```
   #define DATOTEKA "bsd.h"
#elif SYSTEM == MSDOS
   #define DATOTEKA "msdos.h"
#else
   #define DATOTEKA "default.h"
#endif
```

Ovdje se testira ime SYSTEM kako bi se uključila prava datoteka zaglavlja.

U razvoju programa korisno je ispisivati što veći broj međurezultata kako bismo mogli kontrolirati korektnost izvršavanja programa. Nakon što je program završen i testiran sav suvišan ispis treba eliminirati. U tome nam pomaže uvjetno uključivanje kôda kao što se vidi na sljedećem primjeru:

```
int x;
......
scanf("%d",&x);
#ifdef DEBUG
 printf("Debug:: x=%d\n",x);
#endif
```

Ukoliko je varijabla DEBUG definirana, učitana vrijednost će biti ispisana. Prevodioci pod UNIX-om obično imaju -D opciju, koja se koristi u obliku -Dsimbol, i koja dozvoljava da se simbol definira na komandnoj liniji. Na primjer, pretpostavimo da je program koji sadrži prikazani dio kôda smješten u datoteku prog.c. Tada će kompilacija naredbom

```
cc -o prog prog.c
```

proizvesti program u koji ispis varijable x nije uključen. Kompilacija s naredbom

```
cc -DDEBUG -o prog prog.c
```

dat će izvršni kôd koji uključuje printf naredbu, jer je sada varijabla DEBUG definirana.

Tehnika koja se koristi u razvoju programa je sljedeća: svi ispisi međurezultata ubacuju se između para #ifdef DEBUG i #endif naredbi i program se u razvojnoj fazi kompilira s -DDEBUG opcijom. Kada je program konačno završen i testiran kompilira se bez -DDEBUG opcije. Na taj način se iz izvršnog kôda izbacuju sve suvišne printf (i ostale) naredbe.

Napomena. Pomoću #if naredbe možemo isključiti dio koda iz programa na sljedeći način:

```
#if 0
  dio programa
  koji iskljucujemo
#endif
```

Time se izbjegava problem ugniježdenih komentara. □

8.5 Predefinirani makroi

C standard specificira nekoliko makroa koje moraju biti definirane. Neki od njih su

Makro	Značenje
`__DATE__`	Datum preprocesiranja
`__TIME__`	Vrijeme preprocesiranja
`__FILE__`	Ime datoteke s izvornim kôdom
`__LINE__`	Trenutna linija kôda
`__func__`	Ime funkcije.

Zadnji makro je uveden standardom C99. Na primjer,

```
#include <stdio.h>

int main(void)
{
    printf("Ime datoteke: %s.\n",__FILE__);
    printf("Datum: %s.\n",__DATE__);
    printf("Vrijeme:  %s.\n",__TIME__);
    printf("Linija koda:   %d.\n",__LINE__);
    printf("Ime funkcije:   %s.\n",__func__);
    return 0;
}
```

Ovaj kôd na jednom sustavu ispisuje

```
Ime datoteke: makro_2.c.
Datum: Feb 12 2003.
Vrijeme:  19:28:06.
Linija koda:   9.
Ime funkcije:   main.
```

Ovi se makroi najčešće koriste za ispis poruka o greškama kao u sljedećem primjeru:

```
 if(n != m)
    printf("Greska: linija %d, datoteka %s\n",__LINE__, __FILE__);
```

8.6 assert

Mnoge funkcije očekuju da će dobiti argumente koji zadovoljavaju određene uvjete. Na primjer, funkcija koja prima jedan argument tipa `double` može očekivati samo pozitivne vrijednosti. Predaja negativne vrijednosti vrlo često signalizira da se desila greška u izvršavanju programa. U takvim situacijama željeli bismo provjeriti na početku izvršavanja funkcije jesu li uvjeti koje argumenti moraju zadovoljavati ispunjeni. Budući da takvih provjera može biti jako puno u većim programima namjera nam je isključiti sve te provjere u konačnoj verziji kôda. Tu nam ponovo može pomoći tehnika s `#ifdef DEBUG`, no ANSI-C nam nudi bolju opciju, a to je makro `assert`.

Da bismo iskoristili makro `assert` trebamo uključiti standardnu datoteku zaglavlja `<assert.h>`. Zatim se `assert` koristi kao da se radi o funkciji oblika

```
void assert(int izraz)
```

Ako je `izraz` jednak nuli u trenutku kada se izvršava naredba

```
assert(izraz);
```

`assert` će ispisati poruku

```
Assertion failed: izraz,  file ime_datoteke, line br_linije
```

gdje je `ime_datoteke` ime datoteke u kojoj se naredba nalazi i `br_linije` broj linije u kojoj se naredba nalazi. Nakon toga `assert` zaustavlja izvršavanje programa. Na primjer,

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

int f(int x)
{
   assert(2*x-1 >= 0);
   return sqrt(2*x-1);
}
int main(void)
{
   int x=-1;
   printf("x=%d\n",f(x));
   return 0;
}
```

U ovom slučaju program na jednom sustavu ispisuje sljedeće:

```
Assertion failed: 2*x-1 > 0,  file C:\prjC\test\testA\a1.cpp, line 6
```

Želimo li isključiti `assert` naredbe iz programa dovoljno je prije uključivanja datoteke zaglavlja `<assert.h>` definirati `NDEBUG` kao u ovom primjeru:

```
#include <stdio.h>
#define NDEBUG
#include <assert.h>
  ........
```

Svaki `assert` makro će sada biti reduciran na nul-naredbu.

Poglavlje 9

Struktura programa

U programskom jeziku C sve varijable moramo deklarirati prije njihove upotrebe. Deklaracija definira tri svojstva varijable: tip, doseg i vijek trajanja. Pojedini elementi deklaracije zadaju se eksplicitno, pomoću ključnih riječi jezika, ili implicitno, položajem deklaracije u programu. Isto vrijedi i za deklaracije funkcija.

Tip varijable uvijek se uvodi eksplicitno ključnim riječima `int, float, double` i drugim. Doseg i trajanje varijable određeni su položajem deklaracije u programu, a mogu se modificirati ključnim riječima `static` i `extern`.

9.1 Doseg varijable

Doseg varijable je dio programa u kojem je njena deklaracija vidljiva i u kojem se stoga varijabli može pristupiti putem njenog imena. Dva su osnovna tipa dosega: to može biti blok ili datoteka. Varijable s dosegom bloka nazivamo lokalnim varijablama, dok varijable s dosegom datoteke nazivamo globalnim. Ako je izvorni kod razbijen u više datoteka, onda su globalne varijable, deklarirane `extern`, vidljive i izvan datoteke u kojoj su definirane.

9.1.1 Lokalne varijable

Blok naredbi čini svaki niz naredbi omeđen vitičastim zagradama. Na primjer, tijelo funkcije je jedan blok naredbi.

Programski jezik C dozvoljava da se u svakom bloku deklariraju varijable. Takve varijable nazivamo lokalnim. Deklaracija lokalne varijable unutar

nekog bloka mora prethoditi prvoj izvršnoj naredbi u tom bloku (prema standardu C90). U primjeru

```
if(n>0) {
   int i;   /* deklaracija varijable */
   for(i=0; i<n; ++i)
      ....
}
```

nova varijabla `i` definirana je u bloku `if` naredbe koji se izvršava u slučaju istinitosti uvjeta `n>0`.

Varijabla definirana unutar nekog bloka vidljiva je samo unutar tog bloka. Izvan bloka ne može joj se pristupiti, tj. njeno ime izvan bloka nije definirano. Štoviše, izvan bloka može biti deklarirana varijabla istog imena. Ta je varijabla tada unutar bloka nedostupna, jer ju skriva varijabla deklarirana u bloku. Na primjer,

```
int x,y;
......
void f(double x) {
   double y;
      ....
}
```

Formalni argument funkcije vidljiv je unutar funkcije i nije dohvatljiv izvan nje. Doseg formalnog argumenta je dakle isti kao i doseg varijable definirane na početku funkcije. Stoga u prethodnom primjeru formalni argument `x` i `double` varijabla `y`, deklarirana u funkciji, skrivaju `int` varijable `x` i `y`, deklarirane izvan funkcije. Njih unutar funkcije nije moguće dosegnuti. Nakon izlaza iz funkcije, cjelobrojne varijable `x` i `y` su ponovo vidljive i imaju nepromijenjene vrijednosti, dok `double` varijable `x` i `y` više ne postoje.

Ukratko: Doseg varijable definirane unutar nekog bloka je taj blok. Doseg formalnog argumenta je tijelo funkcije.

9.1.2 Globalne varijable

Varijabla deklarirana u vanjskom bloku vidljiva je u svakom unutarnjem bloku, ako u njemu nije definirana varijabla istog imena. Ako je varijabla definirana izvan svih blokova, onda je njen doseg čitava datoteka u kojoj je definirana. Takvu varijablu nazivamo globalnom. Na primjer,

```
#include <stdio.h>
int x=3;
```

```
void ispisi(void) {
   int y=4;
   printf("x=%d, y=%d\n",x,y);
}
int main(void){
  ispisi();
  return 0;
}
```

Varijabla `x` vidljiva je unutar funkcije `ispisi()`. S druge strane kôd

```
#include <stdio.h>  /* pogresno */
void ispisi(void) {
   int y=4;
   printf("x=%d, y=%d\n",x,y);
}
int main(){
   int x=3;
   ispisi();
   return 0;
}
```

nije ispravan jer varijabla `x` nije definirana u bloku koji sadrži definiciju funkcije `ispisi()`.

Globalne varijable se definiraju izvan svih funkcija i njihova je svrha prijenos podataka između funkcija. Svaka funkcija može doseći globalnu varijablu i promijeniti njenu vrijednost. Na taj načni više funkcija može komunicirati bez upotrebe formalnih argumenta. U sljedećem primjeru tri funkcije rade na istom polju znakova:

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

char string[64];

void ucitaj(void);
void malo_u_veliko(void);
void ispisi(void);

int main(void) {
   ucitaj();
   malo_u_veliko();
```

```
   ispisi();
   return 0;
}
void ucitaj() {
   fgets(string,sizeof(string),stdin);
}
void malo_u_veliko() {
  int i;
  for(i=0;string[i] !='\0';i++)
     string[i]=toupper(string[i]);
}
void ispisi() {
  printf("%s\n",string);
}
```

Uočimo da sve funkcije rade s istom vanjskom varijablom `string`. (Funkcija `fgets()` učitava string iz datoteke, ovdje standardnog ulaza; vidi sekciju 13.3.2).

Naglasimo još da je globalna varijabla vidljiva od mjesta svoje deklaracije do kraja datoteke u kojoj se nalazi. Stoga globalne varijable deklariramo na početku datoteke prije svih funkcija. U primjeru,

```
int a;
void f(int);
int main(void) {
    ....
}
int b;
void f(int i) {
   .....
}
```

varijabla `a` je vidljiva i u funkciji `main()` i u funkciji `f()`, dok je varijabla `b` vidljiva u funkciji `f()`, ali ne i u funkciji `main()`

Funkcije su po svojoj prirodi globalni objekti. Funkcija definirana na bilo kojem mjestu može se dohvatiti iz bilo kojeg dijela programa ako ima prototip. Definicija funkcije unutar druge funkcije nije dozvoljena.

Zadatak. Kakav će biti izlaz iz sljedećeg programa?

```
#include <stdio.h>

int main(void) {
```

```
    int x=30;

    printf("x u vanjskom bloku = %d\n",x);
    while(x++ < 33) {
      int x =100;
      ++x;
      printf("x u unutarnjem bloku = %d\n",x);
    }
    return 0;
}
```

Napomena. Funkcija koja zavisi od neke globalne varijable i vrijednosti koju će u nju postaviti neka druga funkcija nije samostalna cjelina i ne može se bez modifikacija koristiti u različitim programima. K tomu, ovisnost o globalnoj varijabli može biti slabo uočljiva, što otežava održavanje programa. Dobar stil programiranja stoga nalaže što manju upotrebu globalnih varijabli.

9.1.3 Argumenti funkcijskog prototipa

Argumenti navedeni u funkcijskom prototipu imaju doseg koji ne seže dalje od prototipa. To znači da imena navedena u prototipu nisu važna i mogu se podudarati s imenima drugih varijabli u programu. Štoviše, prevodilac nam dopušta da ispustimo imena varijabli. Jedina iznimka je deklaracija polja varijabilne dimenzije (sekcija 10.7).

9.1.4 Lokalne varijable i standard C99

Standard C99 je uveo izmjene u varijable blokovnog dosega koje C čine kompatibilnijim s C++om. Izmjene su sljedeće:

- Varijabla može biti deklarirana bilo gdje unutar bloka, a ne samo na njegovom početku. Deklaracije i izvršne naredbe mogu se sada slobodno ispreplitati.
- Proširuje se pojam bloka kako bi obuhvatio petlje `for`, `while` i `do while`, te naredbu `if`, i u slučaju kada se ne koriste vitičaste zagrade.

Ideja prvog pravila je omogućiti deklaraciju varijable što bliže mjestu na kojem se koristi. Time se postiže veća čitljivost kôda i olakšava prevodiocu zadatak optimizacije.

Drugo pravilo omogućava definiciju varijable unutar `for` naredbe kao u slučaju

```
for(int i=0; i<10; ++i)
   printf("Prema standardu C99 i= %d\n",i);
printf("i=%d\n",i);  // Greska, i nije definirano
```

Varijabla `i` definirana je unutar `for` petlje i njen doseg je `for` petlja. To znači da već u sljedećoj naredbi `i` nije definirano.

9.1.5 Funkcijski doseg

Napomenimo još da postoji i tzv. *funkcijski doseg*, ali se on primijenjuje samo na labele u `goto` naredbi. Funkcijski doseg znači da je `goto` labela vidljiva u cijeloj funkciji, bez obzira na blok u kojem se pojavljuje. Kao što je rešeno u sekciji 6.8, pomoći `goto` naredbe ne može se izaći iz funkcije.

9.2 Vijek trajanja varijable

Svakoj varijabli prevodilac pridružuje određeni memorijski prostor. Vijek trajanja varijable je vrijeme za koje joj je pridružena njena memorijska lokacija, dakle ukupno vrijeme njene egzistencije. Prema vijeku trajanja varijable dijelimo na automatske i statičke.

9.2.1 Automatske varijable

Svaka varijabla kreirana unutar nekog bloka (dakle unutar neke funkcije), koja nije deklarirana s ključnom riječi `static`, je **automatska** varijable. Automatske varijable se kreiraju na ulasku u blok u kome su deklarirane i uništavaju na izlasku iz bloka. Memorija koju je automatska varijabla zauzimala oslobađa se za druge varijable. Na primjer,

```
......
void f(double x) {
    double y=2.71;
    static double z;
        ....
}
```

varijable `x` i `y` su automatske dok `z` nije, jer je deklarirana s ključnom riječi `static`.

Automatske varijable mogu se inicijalizirati, kao što je to slučaj s varijablom `y`. Inicijalizacija se vrši pri svakom novom ulazu u blok u kome je varijabla definirana. Tako će varijabla `y` biti ponovo kreirana i inicijalizirana pri svakom novom pozivu funkcije `f()`.

Automatska varijabla koja nije inicijalizirana na neki način, na ulasku u blok u kome je definirana dobiva nepredvidivu vrijednost. Najčešće je to vrijednost koja se zatekla na pridruženoj memorijskoj lokaciji.

Osim konstantnim izrazom, inicijalizaciju automatsje varijable moguće je izvršiti i izrazom koji nije konstantan kao u ovom slučaju:

```
void f(double x, int n) {
    double y=n*x;
      ....
}
```

Inicijalizacija varijable y ovdje je samo pokrata za eksplicitno pridruživanje y=n*x;.

9.2.2 Identifikatori memorijske klase

Identifikatori memorijske klase su auto, extern, static i register. Oni služe preciziranju vijeka trajanja varijable. Postavljaju se u dekelaraciji varijable prije identifikatora tipa varijable, tako da je opći oblik deklaracije varijable:

```
identifikator_mem_klase tip_varijable ime_varijable;
```

Na primjer,

```
extern double l;
static char polje[10];
auto int *pi;
register int z;
```

9.2.3 auto

Identifikator memorijske klase auto deklarira automatsku varijablu. Vrlo se rijetko susreće u programima jer za njegovu upotrebu nema drugog razloga osim iskazivanja namjere programera. Naime, sve varijable definirane unutar nekog bloka, a bez ključne riječi static, su automatske varijable. Sve varijable definirane izvan svih blokova su statičke varijable,

9.2.4 register

Identifikator memorijske klase register može se primijeniti samo na automatske varijable i formalne argumente funkcije. Na primjer

```
f(register int m, register long n)
{
    register int i;
```

```
    ........
}
```

Ključna riječ `register` sugerira prevodiocu da će varijabla biti često korištena i da bi trebala biti alocirana tako da se smanji vrijeme pristupa. To najčešće znači smjestiti varijablu u registar mikroprocesora. Prevodilac nije dužan poštovati deklaraciju `register`, tako da je ona samo sugestija prevodiocu. Na varijablu tipa `register` ne može se primijeniti adresni operator.

9.2.5 Statičke varijable

Statičke varijable alociraju se i inicijaliziraju na početku izvršavanja programa, a uništavaju se tek na završetku programa. Vijek trajanja statičke varijable je cijelo vrijeme izvršavanja programa. Prostor za statičke varijable alocira se u dijelu memorije različitom od dijela u kojem se alociraju automatske varijable (što je standardno programski stog).

Svaka varijabla definirana izvan svih funkcija je statička. Varijabla deklarirana u nekom bloku (npr. funkciji) s identifikatorom memorijske klase `static` je također statička varijabla.

Ukoliko statička varijabla nije inicijalizirana eksplicitno prevodilac će je inicijalizirati nulom. Statičke je varijable moguće inicijalizirati samo konstantnim izrazima tako da sljedeći kôd nije ispravan:

```
int f(int j)
{
    static int i=j;  // neispravno
    ........
}
```

9.2.6 Statičke lokalne varijable

Statička lokalna varijabla je lokalna varijabla deklarirana s identifikatorom memorijske klase `static`. Ona postoji za cijelo vrijeme izvršavanja programa ali se može dohvatiti samo iz bloka u kojem je definirana. K tome vrijedi i sljedeće pravilo: Statička varijabla definirana unutar nekog bloka inicijalizira se samo jednom i to pri prvom ulazu u blok. Pogledajmo kako se to svojstvo koristi u jednom primjeru.

Želimo napisati program koji ispisuje prvih 20 Fibonaccijevih brojeva. To su brojevi definirani rekurzijom

$$F_i = F_{i-1} + F_{i-2}, \quad (i = 3, 4, \ldots) \quad F_1 = F_2 = 1.$$

Glavni program imat će sljedeći oblik:

```
#include <stdio.h>
long fibonacci(int);
int main(void) {
   int i;
   for(i=1;i<=20;i++) printf("\n i= %d, F= %ld",i,fibonacci(i));
   return 0;
}
```

Dakle, treba nam funkcija koja će za svaki i izračunati F_i uz uvjet da se brojevi računaju redom od F_1 do F_{20}. Tu će nam pomoći statičke varijable:

```
long fibonacci(int i)
{
    static long f1=1, f2=1;
    long f;

    f=(i<3) ? 1 : f1+f2;
    f2=f1;
    f1=f;
    return f;
}
```

Statičke varijable `f1` i `f2` bit će inicijalizirane jedinicama samo pri prvom pozivu funkcije `fibonacci`. Između svaka dva poziva funkciji `fibonacci` one zadržavaju svoju vrijednost i stoga pri i-tom pozivu funkcije imamo `f1`$= F_{i-1}$ i `f2`$= F_{i-2}$.

Upozorenje. Ključna riječ `static` ispred varijable definirane izvan svih blokova ne označava statičku varijablu već reducira njen doseg (vidi sljedeću sekciju).

9.3 Vanjski simboli

C program može biti smješten u više datoteka. Na primjer, svaka funkcija definirana u programu može biti smještena u zasebnu `.c` datoteku. Takvo razbijanje izvornog kôda većih programa olakšava njihovo održavanje i nadogradnju. Pri tome je nužno da se funkcije i globalne varijable definirane u jednoj datoteci mogu koristiti i u svim ostalima.

Proces prevođenja izvornog kôda smještenog u više datoteka ima dvije faze. U prvoj prevodilac prevodi izvorni kôd svake pojedine datoteke u objektni kôd (datoteku s ekstenzijom `.o`); u drugoj linker povezuje više datoteka

s objektnim kodom u izvršni program. Zadatak je linkera da pronađe one simbole (imena funkcija i globalnih varijabli) koji se u pojedinoj objektnoj datoteci koriste, ali nisu u njoj definirani. Svaki takav simbol mora imati jednu i samo jednu definiciju u nekoj od datoteka, s kojom linker onda povezuje simbol.

Na primjer, funkcija može biti definirana u jednoj datoteci, a pozivati se u više drugih. U svakoj takvoj datoteci navodi se samo prototip funkcije. Linker će povezati simbol naveden u prototipu s definicijom funkcije. Pri tome svi prototipovi (deklaracije) moraju odgovarati definiciji, i definicija treba biti samo jedna. Analogno je s vanjskim varijablama.

Osnovno je pravilo da su imena svih vanjskih varijabli i funkcija dostupna linkeru; sva su ta imena vanjski simboli. C nam dozvoljava da neka imena ne eksportiramo linkeru, kako bi se smanjio broj vanjskih simbola (vidi sekciju 9.3.3). To se čini pomoću ključne riječi `static`.

9.3.1 Funkcije

Vidjeli smo već da funkcije uvijek imaju doseg datoteke, tj. funkcija definirana u datoteci može se pozvati bilo gdje u toj datoteci ako je na mjestu poziva vidljiva njena deklaracija. Štoviše, ime funkcije je automatski vidljivo linkeru, što znači da se funkcija može pozivati i u drugim datotekama.

Na primjer, pretpostavimo da u prvoj datoteci imamo kôd:

```
#include <stdio.h>         /****** Datoteka 1 ******/
int g(int);

void f(int i) {
   printf("i=%d\n",g(i));
}
int g(int i) {
  return 2*i-1;
}
```

i da se funkcija `main()`, koja poziva funkciju `f()`, nalazi u drugoj datoteci. Tada je u toj datoteci funkciju `f()` potrebno deklarirati, odnosno navesti njen prototip:

```
extern void f(int);       /****** Datoteka 2 *******/

int main(void) {
   f(3);
}
```

Ključna riječ `extern` je identifikator memorijske klase koji označava da deklarirano ime (ovdje `f`) vanjski simbol, tj. da je poznato linkeru. Budući da su sva imena funkcija automatski poznata linkeru, `extern` možemo ispustiti. To znači da je deklaracija

```
extern void f(int);
```

ekvivalentan s

```
void f(int);
```

Ključna riječ `extern` kod funkcija ima stoga samo ulogu dokumentiranja programa. Stavit ćemo ju uvijek u prototipu funkcije koja je definirana u nekoj drugoj datoteci, kako bi označili da se radi o *vanjskoj* funkciji.

Funkcija može biti deklarirana i s identifikatorm memorijske klase `static`. Efekt takve deklaracije je da ime funkcije neće biti eksportirano linkeru. Takvu funkciju nije moguće pozivati iz druge datoteke. Na primjer, ako bismo željeli onemogučiti korištenje funkcije `g` izvan prve datoteke, modificirali bismo prvu datoteku na sljedeći način:

```
#include <stdio.h>      /****** Datoteka 1 *******/
static int g(int);
void f(int i)
{
    printf("i=%d\n",g(i));
}
static int g(int i) {
   return 2*i-1;
}
```

Sada funkciju `g` više ne možemo dohvatiti iz druge datoteke, pa je sljedeći program neispravan:

```
extern void f(int);      /****** Datoteka 2 *******/
extern int g(int);   /* pogresno */

int main(void) {
    f(3);                        /* Ispravno   */
    printf("g(2)=%d\n",g(2)); /* Neispravno */
}
```

jer linker ne bi pronašao funkciju `g`.

Funkcije koje ne koristimo izvan datoteke u kojoj su definirane treba u principu deklarirati kao *statičke* kako njihova imena ne bi bila poznata u čitavom programu.

Uobičajena je praksa deklaracije svih funkcija koje su vanjski simboli staviti u datoteku zaglavlja koja se onda uključuje u svaku datoteku koja te funkcije želi koristiti, čak i u onu u kojoj su funkcije definirane. Na taj se način izbjegava moguće nepodudaranje prototipova i definicija. Uključivanje zaglavlja u datoteku u kojoj su funkcije definirane omogućava prevodiocu da provjeri jesu li prototipovi u skladu s definicijama.

To je postupak koji se koristi s funkcijama iz standardne biblioteke. Ako na primjer, želimo koristiti funkciju `printf()`, koja je definirana u standardnoj biblioteci, onda moramo uključiti datoteku `<stdio.h>` koja sadrži njen prototip.

9.3.2 Globalne varijable

Svaka globalna varijabla, jednako kao i svaka funkcija, je automatski vanjski simbol; njeno ime je vidljivo linkeru. Takvo ponašanje možemo promijeniti pomoću identifikatora memorijske klase `static`, kao i kod funkcije. U primjeru,

```
static int d;
int main(void)
{
  ........
}
```

Globalna varijabla `d` vidljiva je samo u svojoj datoteci (nije *vanjska varijabla*).

Uočimo da je ključna riječ `static` ima drugačije značenje kad se primjeni na lokalnu varijablu, odnosno na globalnu varijablu. Kod lokalne varijable `static` mijenja vijek trajanja varijable, a kod globalne reducira doseg.

Kada se jedna globalna varijabla koristi u više datoteka, onda ona mora biti definirana u jednoj od njih, a deklarirana u svim ostalima. Posve isto pravilo vrijedi i za funkcije.

Kod funkcija je to pravilo lako primijeniti jer se definicija i deklaracija funkcije jasno razlikuju. Definicija funkcije ima tijelo funkcije, dok deklaracija (prototip) nema. Kod varijabli razlika nije tako očita.

Prevodilac mora jasno razlikovati definiciju od deklaracije varijable. Kod definicije se za varijablu rezervira memorijski prostor, dok se kod deklaracije samo uvodi ime varijable, i smatra se da definicija dana negdje drugdje. Stoga je jasno da mora postojati točno jedna definicija. Svaka deklaracija varijable u kojoj se varijabla i inicijalizira nužno je njena definicija. No, ako inicijalizacija nije prisutna, onda nije jasno koja je od više deklaracija iste varijable u više datoteka njena definicija.

Na žalost, različiti prevodioci koriste različite načine za razlikovanje definicije i deklaracije. Da bi se izbjegli problemi treba usvojiti sljedeći model.

- Definicija varijable ima eksplicitnu inicijalizaciju i identifikator memorijske klase `extern` nije prisutan. Na primjer,

```
double a   =0.0;
int    z[3]={0,0,0};
```

- Sve deklaracije sadrže identifikator memorijske klase `extern` i ne sadrže inicijalizaciju:

```
extern double a;
extern int z[];
```

Ključna riječ `extern` ovdje indicira da se radi o vanjskoj varijabli, definiranoj u nekoj drugoj datoteci. Njena uloga ovdje nije isključivo dokumentiranje koda.

9.3.3 Vanjska imena

Standard C99 propisuje da prevodilac treba prepoznati prva 63 znaka svakog lokalnog identifikatora (imena funkcije, varijable,...) i prvih 31 znakova vanjskih identifikatora. Standard C90 propisuje 31 znak lokalnog identifikatora i samo 6 znakova vanjskog identifikatora. Imena vanjskih varijabli treba stoga držati dovoljno kratkim.

Poglavlje 10

Polja

Polje je niz varijabli istog tipa koje su numerirane i mogu se dohvatiti pomoću cjelobrojnog indeksa. Na primjer,

```
double x[3];
```

je deklaracija polja od tri varijable tipa `double` koje čine polje `x`. Prva varijabla je `x[0]`, druga je `x[1]` i treća je `x[2]`; u C-u su polja uvijek indeksirana počevši od nule.

Radi efikasnosti pristupa elementi polja smještaju se na uzastopne memorijske lokacije. Stoga je polje element jezika C pomoću kojeg se realiziraju vektori i matrice.

10.1 Definicija i inicijalizacija polja

Polje se definira jednako kao i skalarna varijabla s tom razlikom da dimenzija polja (broj elemenata) mora biti zadana. Dimenzija se zadaje kao pozitivni cjelobrojni izraz u uglatim zagradama.

Jednodimenzionalno polje definira se na sljedeći način:

```
mem_klasa tip ime[izraz];
```

gdje je `mem_klasa` memorijska klasa, `tip` je tip podatka, `ime` je ime polja, a `izraz` mora biti konstantan cjelobrojni pozitivni izraz.

Deklaracija memorijske klase nije obavezna. Unutar funkcije polje deklarirano bez memorijske klase je automatska varijabla, a izvan svih funkcija je statička varijabla. Unutar funkcije polje se može učiniti statičkim pomoću identifikatora memorijske klase `static`.

`izraz` u definiciji polja je najčešće pozitivna konstanta ili simbolička konstanta.

Polje definirano naredbom

```
float v[3];
```

je polje od tri elementa v[0], v[1], v[2]. Uočimo da prvi element uvijek ima indeks 0, drugi indeks 1 itd.

Polja se mogu inicijalizirati navođenjem vrijednosti elemenata unutar vitičastih zagrada. Sintaksa je sljedeća:

```
mem_klasa tip ime[izraz]={v_1,v_2,...,v_n};
```

gdje je v_1 vrijednost koja će biti pridružena prvom elemetu polja (ime[0]), v_2 vrijednost pridružena drugom (ime[1]) itd. Na primjer,

```
float v[3]={1.17,2.43,6.11};
```

daje pridruživanje v[0]=1.17, v[1]=2.43, v[2]=6.11. Prilikom inicijalizacije polja dimenzija ne mora biti specificirana već će biti automatski izračunata. Stoga možemo pisati

```
float v[]={1.17,2.43,6.11};
```

i prevodilac će kreirati polje dimenzije 3.

Ako je broj inicijalizacijskih vrijednosti veći od dimenzije polja javlja se greška. Ako je manji, onda će preostale vrijednosti biti inicijalizirane nulom.

Sljedeći program za dano polje x[dim] računa

$$a = \frac{1}{\mathrm{dim}} \sum_{i=0}^{\mathrm{dim}-1} x_i, \quad b = \frac{1}{\mathrm{dim}} \sqrt{\sum_{i=0}^{\mathrm{dim}-1} (x_i - a)^2}.$$

Polje x je definirano kao globalna varijabla.

```
#include <stdio.h>
#include <math.h>
float x[]={1.3, 2.4, 5.6, 6.7, 7.8};
int main(void) {
   int i,dim=sizeof(x)/sizeof(float);
   double a=0.0,b=0.0;

   for(i=0;i<dim;++i) a+=x[i];
   a/=dim;

   for(i=0;i<dim;++i) b+=(x[i]-a)*(x[i]-a);
   b=sqrt(b)/dim;

   printf("Srednja vrijednost = %f, odstupanje = %f\n", a,b);
  return 0;
}
```

Dimenziju polja dobivamo naredbom

```
dim=sizeof(x)/sizeof(float);
```

Operator `sizeof` primijenjen na varijablu daje broj okteta potreban da za memoriranje varijable. Ako je varijabla polje, onda `sizeof` daje broj okteta potreban da se zapamti cijelo polje, pa smo stoga broj elemenata polja dobili dijeljenjem s brojem okteta koji zauzima jedna varijabla tipa `float`.

Nažalost, ovakav način računanja broja elemenata polja nije moguće promijeniti na argument funkcije tipa polja, budući da se on automatski konvertira u pokazivač.

Polje `x` je tipa `float` dok su varijable `a` i `b` u kojima se vrši sumacija tipa `double`. Stoga u naredbi `a+=x[i]` i `b+=(x[i]-a)*(x[i]-a)` imamo konverziju iz užeg tipa u širi tip podatka. Sumiranje u varijabli šireg tipa korisno je ako se sumira veliki broj članova jer se na taj način smanjuju greške zaokruživanja.

Obrnuta konverzija, iz šireg u uži tip se dešava pri inicijalizaciji polja `x` jer su sve konstante tipa `double`, na što će nas prevoditelj eventualno upozoriti. Da bi smo izbjegli upozorenje mogli smo koristiti konstante tipa `float`:

```
float x[]={1.3f, 2.4f, 5.6f, 6.7f, 7.8f};
```

U računu srednjeg odstupanja koristimo funkciju `sqrt` (drugi korijen) iz standardne biblioteke. Kao i sve matematičke funkcije koje rade s realnim brojevima i `sqrt` uzima argument tipa `double` i vraća vrijednost tipa `double`. Verzija funkcije `sqrt` koja uzima `float` i vraća `float` naziva se `sqrtf`. Nazivi drugih `float` funkcija dobivaju se na isti način.

Standard C99 omogućava parcijanu inicijalizaciju polja. Sintaksa je kao u sljedećem primjeru

```
float x[5]={[3]=1.3f};
```

Ovdje se treći element inicijalizira s `1.3`. Neinicijalizirani elementi polja dobit će vrijednost nula, kao i kod obične inicijalizacije.

Napomena. U definiciji polja dimenzije polja moraju biti konstantni izrazi. Zbog toga C ne dozvoljava deklaraciju polja unutar funkcije na sljedeći način:

```
void f(int n)
{
   double A[n][n];  // pogresno
   ........
}
```

U funkciji `f()` smo pokušali deklarirati matricu čiji je red zadan argumentom `n`. Zbog gornjeg pravila, to je nemoguće. To ograničenje ANSI C-a (C90) uklonio je standard C99 uvođenjem polja varijabilne duljine (vidi sekciju 10.7). □

10.2 Polja znakova

Polja znakova mogu se inicijalizirati stringovima. Na primjer, deklaracijom

```
char c[]="tri";
```

definirano je polje od 4 znaka: `c[0]='t', c[1]='r', c[2]='i', c[3]='\0'`. Takav način pridruživanja moguć je samo pri definiciji varijable. Nije dozvoljeno pisati

```
c="tri"; /* pogresno */
```

već trebamo koristiti funkciju `strcpy` deklariranu u `<string.h>`.

Sljedeći program učitava ime i prezime i ispisuje ih u jednom retku:

```
#include <stdio.h>
#include <string.h>
char ime          [128];
char prezime      [128];
char ime_i_prezime[128];
int main(void)
{
   printf("Unesite ime:");
   gets(ime);
   printf("Unesite prezime:");
   gets(prezime);
   strcpy(ime_i_prezime,ime);
   strcat(ime_i_prezime," ");
   strcat(ime_i_prezime,prezime);
   printf("Ime i prezime: %s\n", ime_i_prezime);
   return 0;
}
```

Definirali smo tri globalna polja znakova, svako od po 128 znakova. Nizove znakova unosimo pomoću `gets` naredbe. Ona automatski zamijenju znak za prijelaz u novi red s nul znakom. U naredbi

```
strcpy(ime_i_prezime,ime);
```

funkcija `strcpy` kopira string `ime` u string `ime_i_prezime`. Pri tome se kopira i nul znak kojim string završava. Funkcija vraća pokazivač na prvi string, no u programu tu vrijednost zanemarujemo.

U naredbi

```
strcat(ime_i_prezime," ");
```

koristimo funkciju `strcat` koja povezuje dva stringa. U ovom slučaju rezultat je novi string koji sadrži vrijednost varijable `ime` i jedan razmak iza zadnjeg slova. Dobiveni niz znakova opet završava nul znakom. I ova funkcija vraća pokazivač na prvi string no u programu ga zanemarujemo. Ponovnom upotrebom funkcije `strcat` dobivamo konačan string koji zatim ispisujemo.

10.3 Funkcije za rad sa stringovima

Datoteka zaglavlja `<string.h>` deklarira niz funkcija za rad sa stringovima. Najčešće upotrebljavane su

```
strlen(), strcat(), strncat(), strcmp(), strncmp(),
strcpy(), strncpy(). strchr(), strstr().
```

Funkciju `strlen()` smo već sreli. Ona vraća duljinu stringa, tj. broj znakova u stringu, bez zadnjeg null znaka. Prototip funkcije je

```
size_t strlen(const char *s);
```

Cjelobrojni tip `size_t` je definiran u `<stdio.h>` npr. kao

```
typedef unsigned long   size_t;
```

Ta je definicija ovisna o računalu pa je stoga stavljena u standardnu datoteku zaglavlja. Funkcija `strlen` vraća vrijednost za jedan manju od operatora `sizeof`.

Funkcija `strcat` povezuje dva stringa u jedan. Njen prototip je

```
char *strcat(char *s1, const char *s2);
```

String `s2` se nadovezuje (eng. *concatenation*) na string `s1` koji se pri tome povećava dok se `s2` ne mijenja. Prvi znak u polju znakova na koje pokazuje `s2` bit će prepisan preko null-znaka kojim završava `s1`. Ova funkcija ne vodi računa o tome da li u polju na koje pokazuje `s1` ima dovoljno mjesta za oba stringa. Ukoliko nema doći će do pisanja preko susjednih memorijskih lokacija.

Funkcija `strncat` uzima treći argument koji ima značenje maksimalnog broja znakova koji će biti dodani stringu `s1`. Ona nam omogućava pisanje pouzdanijih programa. Prototip funkcije je

```
char *strncat(char *s1, const char *s2, size_t n);
```

Ako u prvih `n` znakova na koje pokazuje `s2` nema nul-znaka, onda `strncat` dodaje `n` znakva iz `s2` na kraj niza `s1` te dodaje nul-znak; dakle, ukupno se dodaje `n+1` znakova.

Sljedeći program ilustrira upotrebu funkcije `strncat`. Ako cijeli ime i prezime ne stane u polje `ime_i_prezime` program će naprosto staviti ono što stane (kvalitetnije rješenje bi bilo pomoću dinamički alocirane memorije).

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

#define SIZE 30
char ime_i_prezime[SIZE];
char prezime[SIZE];

int main(void){
   int n;

   puts("Vase ime: ");
   gets(ime_i_prezime);
   if(strlen(ime_i_prezime) < SIZE-2)
      strcat(ime_i_prezime," ");
   puts("Vase prezime: ");
   gets(prezime);
   n=SIZE-strlen(ime_i_prezime)-1;
   strncat(ime_i_prezime,prezime,n);
   puts(ime_i_prezime);
   return 0;
}
```

Pri spajanju nizova `ime_i_prezime` i `prezime` moramo imati

```
strlen(ime_i_prezime) + strlen(prezime) + 1 <= SIZE.
```

Za uspoređivanje stringova imamo dvije funkcije

```
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

Uspoređivanje se vrši znak po znak. Ukoliko su dva stringa jednaka funkcija vraća nulu. Ako prvi string leksikografski prethodi drugom funkcija će vratiti negativnu vrijednost, a u suprotnom pozitivnu. Sljedeći program ilustrira leksikografski poredak stringova.

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

int main(void){
    printf("%d\n", strcmp("A","A"));
    printf("%d\n", strcmp("A","B"));
    printf("%d\n", strcmp("B","A"));
    printf("%d\n", strcmp("C","A"));
    printf("%d\n", strcmp("a","A"));
    printf("%d\n", strcmp("aaac","aaab"));
    return 0;
}
```

Program će redom ispisati vrijednosti 0, -1, 1, 2, 32 i 1.

Funkcija `strncmp` ima dodatni argument `n` koji predstavlja broj znakova koji će biti uspoređeni. Funkcija će redom uspoređivati znakove u dva stringe sve dok ne usporedi `n` parova ili ne naiđe do kraja jednog od stringova. Funkcija `strcmp` uvijek uspoređuje parove znakova sve do kraja jednog od dva stringa. Nova funkcija omogućava veću efikasnost uspoređivanja jer ako, na primjer, provjeravamo da li riječ počine s `"astro"`, dovoljno je limitirati usporedbu na prvih pet znakova.

Za kopiranje jednog znakovnog niza u drugi koristimo funkcije

```
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
```

Obje funkcije kopiraju niz znakova na koji pokazuje `s2` na lokaciju na koju pokazuje `s1`. Druga verzija funkcije kopira najviše `n` znakova. Funkcija `strcpy` kopira uvijek i nul-znak, dok `strncpy` neće kopirati nul-znak ako je `n <= strlen(s2)`. U tom slučaju treba naknadno dodati nul-znak. Obje funkcije vraćaju `s1`.

Funkcija

```
char *strchr(const char *s, int c);
```

vraća pointer na prvo pojavljivanje znaka `c` u stringu na koji pokazuje `s` (nul-znak se također može tražiti). Ako znak `c` nije prisutan u `s` bit će vraćen nul-pointer.

Funkcija

```
char *strstr(const char *s1, const char *s2);
```

vraća pointer na prvo pojavljivanje stringa `s2` u stringu na koji pokazuje `s1`. Ako `s2` nije prisutan u `s1` bit će vraćen nul-pointer.

Datoteka zaglavlja `<string.h>` deklarira i mnoge druge funkcije za rad sa stringovima. Za potpuniju informaciju vidi man-stranicu na računalu ili [4], sekcija B3.

10.4 sscanf(), sprintf()

Ove dvije funkcije deklarirane su u zaglavlju `<stdio.h>`.

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

Za razliku od funkcija `scanf` i `printf` one kao prvi argument uzimaju pokazivač na znakovni niz. Jedina razlika prema tim funkcijama je u tome što se čitanje i pisanje vrši iz i u znakovni niz `s`, koji je dan kao prvi argument. Tako funkcija `sscanf` čita iz stringa `s` (umjesto s tastature) prema zadanom formatu; funkcija `sprintf` piše u string `s` (umjesto na ekran) prema zadanom formatu. Ove funkcije prvenstveno služe za formatiranje stringova.

Ako smo, na primjer, pomoću kôda

```
char line[256];
  .........
  .........
  gets(line);
```

učitali jedan string koji sadrži dva broja tipa `double` i jedan tipa `int`, ona možemo pomoću `sscanf` izvršiti daljnju konverziju

```
sscanf(line,"%lf %lf %d",&x,&y,&n);
```

gdje su `x` i `y` varijable tipa `double`, a `n` varijabla tipa `int`. Slično, ako je realan broj `x` potrebno pretvoriti u string najlakše je to učiniti pomoću funkcije `sprintf`:

```
char x_str[64];
float x;
.......
sprintf(x_str,"%12.7f",x);
```

10.5 Polje kao argument funkcije

Polje može biti formalni argument funkcije. U tom slučaju argument se ne prenosi po vrijednosti već funkcija dobiva pokazivač na prvi element polja. Funkcija tada može dohvatiti i promijeniti svaki elemet polja. U deklaraciji (jednodimenzionalnog) polja kao formalnog argumenta dimenziju polja nije potrebno navoditi. Na primjer, funkcija koja uzima polje realnih brojeva i izračunava srednju vrijednost može biti napisana na sljedeći način:

```
double srednja_vrijednost(int n, double v[]) {
    int i;
    double rez=0.0;

    for(i=0;i<n;i++) rez+=v[i];
    return rez/n;
}
```

Polje `v` je argument funkcije deklariran s `double v[]`, bez navođenja dimenzije polja. Dimenzija je mogla biti navedena ali to nije nužno. Broj elemenata polja čiju srednju vrijednost treba izračunati također je argument funkcije. Unutar funkcije sa `v[i]` dohvaćamo i-ti element polja `v`, a ne kopije polja, jer je funkcija dobila pokazivač na prvi element polja. Pri pozivu funkcije koja ima polje kao formalni argument, stvarni argument je ime polja. Tako na primjer,

```
int main(void) {
   int n;
   double v[]={1.0,2.0,3.0},sv;

   n=3;
   sv=srednja_vrijednost(n,v);
   return 0;
}
```

Prevodilac ime polja `v` pri pozivu funkcije pretvara u pokazivač na prvi element polja. Time se dobiva na efikasnosti jer se izbjegava kopiranje (velikih) polja.

Ako funkcija uzima polje kao formalni argument i ne mijenja njegove elemente, onda polje treba deklarirati kao `const` polje. Na taj se način jasno pokazuje da funkcija ne mijenja polje. Prevodilac neće dozvoliti promjenu polja unutar tijela funkcije, štiteći nas tako od vlastitih pogrešaka.

Naredba `return` ne može biti iskorištena za vraćanje polja u pozivni program jer funkcija može vratiti samo skalarnu vrijednost ili strukturu.

10.6 Višedimenzionalna polja

Deklaracija višedimenzionalnog polja ima oblik

```
mem_klasa tip ime[izraz_1][izraz_2]...[izraz_n];
```

gdje je `mem_klasa` memorijska klasa, `tip` je tip podatka, `ime` je ime polja, a `izraz_1`, ..., `izraz_n` su konstantni cjelobrojni pozitivni izrazi koji određuju broj elementa polja vezanih uz pojedine indekse. Tako se prvi indeks kreće od 0 do `izraz_1 - 1`, drugi od 0 do `izraz_2 - 1` itd.

Na primjer, polje `m` deklarirano sa

```
static float m[2][3];
```

predstavlja matricu s dva retka i tri stupca. Njene elemente možemo prostorno zamisliti na sljedeći način:

```
m[0][0]  m[0][1]  m[0][2]
m[1][0]  m[1][1]  m[1][2]
```

U prvom retku su elementi `m[0][i]` za `i=0,1,2`, a u drugom `m[1][i]` za `i` $= 0, 1, 2$. Element na mjestu `(i,j)` matrice `m` je `m[i][j]`.

Elementi višedimenzionalnog polja pamte se u memoriji računala kao jedno jednodimenzionalno polje. Pri tome su elementi poredani po recima što znači da se pri smještanju elemenata u memoriju najdesniji indeks najbrže varira. Kod dvodimenzionalnog polja `m` poredak elemenata u memoriji bio bi

```
m[0][0]  m[0][1]  m[0][2]  m[1][0]  m[1][1]  m[1][2]
```

Preciznije element `m[i][j]` biti će na `l`-tom mjestu u memoriji, gdje je

```
l=i*MAXY+j,
```

a `MAXY=3` je broj stupaca matrice. Poznavanje te činjenice važno je pri konstrukciji numeričkih algoritama koji rade s matricama te za razumijevanje inicijalizacije polja. Na primjer, polje `m` može biti inicijalizirano na sljedeći način:

```
static float m[2][3]={1.0,2.0,3.0,4.0,5.0,6.0};
```

Inicijalne vrijednosti će biti pridružene elementima matrice po recima:

```
m[0][0]=1.0,  m[0][1]=2.0,  m[0][2]=3.0,
m[1][0]=4.0,  m[1][1]=5.0,  m[1][2]=6.0.
```

Kako je takav način inicijalizacije jako nepregledan, inicijalne se vrijednosti mogu pomoću vitičastih zagrada formirati u grupe koje se pridružuju pojedinim recima. Tako možemo pisati

```
static float m[2][3]={{1.0,2.0,3.0},
                      {4.0,5.0,6.0}
                     };
```

što daje istu inicijalizaciju polja `m`, ali je namjera programera puno jasnija. Ukoliko neka od grupa ima manje elementa od dimenzije retka, ostali elemeti će biti inicijalizirani nulama.

Višedimenzionalna polja se definitaju rekurzivno:

- Višedimenzionalno polje je jednodimenzionalno polje čiji su elementi polja dimenzije manje za jedan.

Tako je npr. dvodimenzionalno polje deklarirano naredbom

```
float m[2][3];
```

jedno jednodimenzionalno polje dimenzije 2 čiji su elementi `m[0]` i `m[1]` tipa `float[3]` (jednodimenzionalna polja dimenzija 3); to su reci matrice `m`. Trodimenzionalno polje

```
float m[2][3][4];
```

je jednodimenzionalno polje dimenzije 2 čiji su elementi `m[0]` i `m[1]` dvodimenzionalna polja tipa `float[3][4]`. Elementi polja `m` bit će stoga smješteni u memoriju u redoslijedu `m[0][][]`, `m[1][][]`, odnosno prvo prva matrica dimenzije 3×4, a zatim druga. Preciznije element `m[i][j][k]` bit će smješten na mjesto

```
i*MAXY*MAXZ+j*MAXZ+k,
```

gdje su `MAXX=2`, `MAXY=3` i `MAXZ=4` pojedine dimenzije polja.

Kada je višedimenzionalno polje argument funkcije ono se može deklarirati sa svim svojim dimenzijama ili sa svim dimenzijama osim prve. Na primjer, funkcija koja čita matricu s `MAXX` redaka i `MAXY` stupaca može biti deklarirana kao

```
void readinput(int m[MAXX][MAXY], int n, int m)
```

gdje su `n` i `m` stvarni brojevi redaka i stupaca koje treba učitati. Ta ista funkcija može biti deklarirana na sljedeći način:

```
void readinput(int m[][MAXY], int n, int m)
```

Naime, broj redaka nije bitan za adresiranje elemenata matrice. Sve što funkcija mora znati je da se element `m[i][j]` nalazi na `l`-tom mjestu, gdje je `l=i*MAXY+j`. Stoga je samo `MAXY` nužan pri pozivu funkcije.

Konačno, budući da je u deklaraciji funkcije polje isto što i pokazivač na prvi element polja (stoga što se pri pozivu funkcije vrši implicitna konverzija polja u pokazivač) možemo istu funkciju deklarirati na treći način:

```
void readinput(int (*m)[MAXY], int n, int m)
```

Zagrade su nužne da nebismo dobili "polje pokazivača".

Posve analogno, kod višedimenzionalnih polja funkcija mora znati sve dimenzije polja osim prve. Pri tome treba uočiti da dimenzije moraju biti konstantni izrazi. To je bitno ograničenje ANSI-C- jezika.

Sljedeći program ilustrira inicijalizaciju trodimenzionalnog polja.

```
#include <stdio.h>
char A[][2][2]={ {{'a','b'},{'c','d'}},{{'e','f'},{'g','h'}}};

void f(char a[2][2]);

int main(void)
{
   printf("Matrica A[0]:\n");
   f(A[0]);
   printf("Matrica A[1]:\n");
   f(A[1]);
   return 0;
}

void f(char a[2][2])
{
   printf("%c %c\n",a[0][0],a[0][1]);
   printf("%c %c\n",a[1][0],a[1][1]);
}
```

Uočimo da prvu dimenziju u deklaraciji

```
char A[][2][2]={ {{'a','b'},{'c','d'}},{{'e','f'},{'g','h'}}};
```

prevodioc može izračunati, ali sve ostale dimenzije su nužne.

Budući da je trodimenzionalno polje zapravo (jednodimenzionalno) polje dvodimenzionalnih polja, to funkciji koja uzima dvodimenzionalno polje (`char a[2][2]`) kao argument možemo predati `A[0]` i `A[1]`. Rezultat izvršavanja programa bit će:

```
Matrica A[0]:
a b
c d
Matrica A[1]:
e f
g h
```

Sljedeći program množi dvije matrice. Matrica A je dimenzije 2×2, a matrica B ima dimenziju 2×3. Produkt je matrica $C = AB$ dimenzije 2×3.

```
#include <stdio.h>
double A[2][2]={{1.0,2.0},{3.0,4.0}};
double B[2][3]={{0.0,1.0,0.0},{1.0,0.0,1.0}};
double C[2][3];

int main(void) {
    int i,j,k;

    for(i=0;i<2;++i)
       for(j=0;j<3;++j)
          for(k=0;k<2;++k)
             C[i][j]+=A[i][k]*B[k][j];

    printf("Matrica C:\n");
    for(i=0;i<2;++i){
       for(j=0;j<3;++j) printf("%f ",C[i][j]);
       printf("\n");
    }
    return 0;
}
```

Množenje matrica vrši se u trostrukoj petlji. Budući da poredak petlji može biti proizvoljan imamo $3! = 6$ verzija tog algoritma. Uočimo da smo u programu iskoristili činjenicu da se statička varijabla (polje `C`) automatski inicijalizirana nulama. Rezultat izvršavanja programa je

```
Matrica C:
2.000000 1.000000 2.000000
4.000000 3.000000 4.000000
```

U unutarnjoj petlji redak matrica `A` množi se sa stupcem matrice `B`. Većina računala danas ima hijerarhijski organiziranu memoriju u kojoj se bliske memorijske lokacije mogu dohvatiti brže od udaljenih.[1] Pri množenju jednog retka matrice `A` i jednog stupca matrice `B` treba dohvatiti elemente stupca matrice `B`. Ali, uzastopni elementi jednog stupca nalaze se na memorijskim lokacijama koje su međusobno udaljene za duljinu retka. Kod velikih matrica ta je udaljenost velika, što rezultira sporijim kôdom. Stoga je efikasnija verzija algoritma u kojoj je petlja po `j` unutarnja:

```
......
for(i=0;i<2;++i)
   for(k=0;k<2;++k)
      for(j=0;j<3;++j)
         C[i][j]+=A[i][k]*B[k][j];

......
```

Sada se u unutarnjoj petlji radi s recima matrica `C` i `B`.

10.7 Polja varijabilne duljine

Polja varijabilne duljine (engl. variable-length arrays) uvedena su u jezik u standardu C99 i mnogi prevodioci ih još ne implementiraju u potpunosti.

Programski jezik C (C90) pokazuje ozbiljan nedostatak u radu s matricama. Da bismo to ilustrirali pokušajmo napisati funkciju koja računa Frobeniusovu normu matrice. Ako je $A = (a_{i,j})_{i,j=1}^n$ matrica reda n, onda je njena Frobeniusovu norma

$$\|A\| = \sqrt{\sum_{i,j=1}^{n} a_{i,j}^2}.$$

Funkcija bi mogla izgledati ovako:

```
#include <math.h>
#include <assert.h>
#define M 9

double Fnorm(int n, double A[][M]) {
     double norm=0.0;
     int i,j;
```

[1]Kada treba dohvatiti varijablu na nekoj memorijskoj lokaciji dohvati se čitav blok memorije i smjesti se u bržu memoriju, tzv. *cache*. Tražena varijabla se sada čita iz cachea. Ako je sljedeća varijabla koju treba dohvatiti blizu prethodne, onda je ona već u cacheu i njen dohvat je vrlo brz. Ako pak nije, onda se mora dohvatiti novi blok memorije i spremiti u cache, što predstavlja dodatni utrošak vremena.

```
     assert(n < M);
     for(i=0;i<n;++i)
        for(j=0;j<n;++j)
            norm += A[i][j]*A[i][j];

     return sqrt(norm);
}
```

Prevodilac zahtijeva da su dimenzije matrice u deklaraciji argumenta funkcije konstantni izrazi. To nas prisiljava da esencijalnu dimenziju, broj stupaca, predamo funkciji kao simboličku konstantu. Drugim riječima prevodilac nam neće dozvoliti da funkciji `Fnorm` umjesto konstante `M` damo varijablu koja sadrži red matrice. Posljedica toga je da naša funkcija može raditi samo s matricama reda 9, a da za matricu drukčije dimenzije moramo promijeniti `M` i rekompilirati kôd.

Zbog tih razloga C99 standard uvodi novi element jezika, tzv. polje varijabilne duljine. Polje varijabilne duljine može biti jedino automatsko, te stoga mora biti definirano u nekoj funkciji ili kao argument funkcije. Osnovna razlika prema standardnim poljima u C-u je što dimenzije polja mogu biti zadane varijablama. Tako možemo definirati

```
  int n=3;
  int m=3;
  double A[n][m];   // PVD
```

Uočimo da `n` i `m` moraju biti definirani prije `A`. Kao argument funkcije, polje varijabilne duljine bilo bi deklarirano na sljedeći način:

```
  double Fnorm(int n, int m, double A[n][m]); // a je PVD
```

Deklaracija

```
  double Fnorm(double A[n][m], int n, int m); // neispravno
```

je neispravna zbog pogrešnog poretka argumenata. Standard dozvoljava da se u prototipu ne pišu imena argumenata. U polju varijabilne duljine ih tada treba zamijeniti zvjezdicama:

```
  double Fnorm(int, int, double A[*][*]); // a je PVD
```

Pogledajmo kako bi izgledao program koji računa Frobeniusovu normu matrice.

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

double Fnorm(int n, int m, double A[n][m]);
void vandermond(int n, int m, double A[n][m]);
void print(int n, int m, double A[n][m]);

int main(void)
{
    int n=3,m=3;
```

```
    double A[n][m];
    vandermond(n,m,A);
    print(n,m,A);
    printf("norma matrice = %f\n", Fnorm(n,m,A));
    return 0;
}

void vandermond(int n, int m, double A[n][m])
{
     int i,j;

     for(i=0;i<n;++i)
        for(j=0;j<m;++j)
             A[i][j] = 1.0/(2.0+i+j);
}

double Fnorm(int n, int m, double A[n][m])
{
     double norm=0.0;
     int i,j;

     for(i=0;i<n;++i)
        for(j=0;j<m;++j)
            norm += A[i][j]*A[i][j];

     return sqrt(norm);
}

void print(int n, int m, double A[n][m])
{
     int i,j;

     for(i=0;i<n;++i){
        for(j=0;j<m;++j)
             printf("%f ",A[i][j]);
        printf("\n");
    }
}
```

Izlaz iz programa bi bio

```
0.500000 0.333333 0.250000
0.333333 0.250000 0.200000
0.250000 0.200000 0.166667
norma matrice = 0.876071
```

Polje varijabilne duljine može biti deklarirano samo unutar neke funkcije. Ono se alocira na programskom stogu i zato ne može biti deklarirano `static`. Ono k tome podliježe jednom broju ograničenja u odnosu na obična polja, vidi [3],

Konačno, spomenimo da sličnu funkcionalnost osigurava u brojnim implementacijama jezika nestandardna funkcija `alloca()`.

Poglavlje 11

Pokazivači

Svakoj varijabli u programu pridružena je memorijska lokacija čija veličina ovisi o tipu varijable. Za varijablu tipa `int` tipično se rezervira 16 ili 32 bita, za varijablu tipa `double` 64 bita itd. Program dohvaća memorijsku lokaciju na kojoj je varijabla pohranjena pomoću jedinstvene adrese koja je toj lokaciji pridružena. Pri manipulacijama s varijablom tu adresu ne moramo eksplicitno poznavati nego u tu svrhu služi ime varijable.

Programski jezik C nudi mogućnost rada neposredno s adresama varijabli putem pokazivača (pointera). Pokazivač na neki tip je varijabla koja sadrži adresu varijable danog tipa. Na primjer, pokazivač na `int` je varijabla koja sadrži adresu varijable tipa `int`.

11.1 Deklaracija pokazivača

Da bi se dohvatila adresa neke varijable koristi se *adresni operator* &. Ako je `v` varijabla danog tipa, a `pv` pokazivač na taj tip, onda je naredbom

```
pv=&v;
```

pokazivaču `pv` pridružena adresa varijable `v`. Pored adresnog operatora koristimo još i *operator dereferenciranja* `*` koji vraća vrijednost spremljenu na adresu na koju pokazivač pokazuje. Tako, ako je `pv=&v`, onda je `*pv` isto što i `v`.

Pokazivač na neki `tip` deklarira se na sljedeći način:

```
tip  *ime;
```

gdje je `ime` pokazivača, a `tip` je tip podatka na koji pokazuje. Zvijezdica označava da se radi o pokazivaču, a ne o vrijabli tipa `tip`. Na primjer varijable `pi` deklarirana naredbom

```
int  *pi;
```

je pokazivač na `int`. Prilikom definicije varijable ona može biti inicijalizirana kao u ovom slučaju

```
int  i=5;
int  *pi=&i;
```

Naravno, varijabla `i` čija se adresa uzima mora biti definirana prije nego što se na nju primjeni adresni operator.

Adresni operator može se primijeniti na operande kojima je pridružena jedinstvena adresa. Stoga ga ne možemo primijeniti na npr. aritmetičke izraze i slično. Operator dereferenciranja djeluje samo na pokazivačke varijable.

Adresni operator i operator dereferenciranja su unarni operatori i imaju isti prioritet kao ostali unarni operatori. Njihov prioritet je veći od prioriteta aritmetičkih operatora tako da u aritmetičkim izrazima `*pi` nije potrebno stavljati u zagradu. Npr. uz gornje deklaracije izraz

```
i=2*(*pi+6);
```

dat će `i=22` jer se `*pi` izračunava i daje 5 prije aritmetičkih operacija.

Operator dereferenciranja može se pojaviti na lijevoj strani jednakosti tj. možemo imati

```
*pi=6;
```

što je ekvivalentno s `i=6`. Adresni operator, s druge strane, ne može se pojaviti na lijevoj strani jednakosti.

Deklaracija `int *pi` indicira da je `*pi` objekt tipa `int`. Sintaksa deklaracije varijable imitira sintaksu izraza u kojem se ona pojavljuje. Isto se odnosi i na deklaraciju funkcija. U primjeru

```
int  *f(char *);
```

f je funkcija koja uzima pokazivač na char i vraća pokazivač na int. Deklaracija sugerira da *f(s) mora biti tipa int (s je pokazivač na char ili jednostavno niz znakova).

Pokazivač se u printf naredbi ispisuje s kontrolnim znakom %p:

```
#include <stdio.h>

int main(void) {
    int i=5;
    int *pi=&i;

    printf("i= %d, adresa od i= %p\n",i,pi);
    return 0;
}
```

11.2 Pokazivači i funkcije

Pokazivači mogu biti argumenti funkcije. U tom slučaju funkcija može promijeniti vrijednost varijable na koju pokazivač pokazuje.

Uzmimo da želimo napisati funkciju zamjena koja uzima dva cjelobrojna argumenta x i y i zamijenjuje njihove vrijednosti: x preslikava u y, a y u x. Funkciju bismo mogli ovako napisati:

```
 void zamjena(int x, int y) {  /* POGRESNO  */
     int temp=x;
     x=y;
     y=temp;
}
```

Ova funkcija ne daje traženi rezultat zbog prijenosa argumenta po vrijednosti. Pri pozivu funkcije zamjena(a,b) ona dobiva kopije stvarnih argumenata a i b koje međusobno zamijenjuje, ali to nema nikakvog utjecaja na stvarne argumente. Stoga, funkcija treba uzeti pokazivače na varijable čije vrijednosti treba zamijeniti. Poziv funkcije treba biti

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

a funkcija treba imati oblik

```
void zamjena(int *px, int *py) {
    int temp=*px;
    *px=*py;
    *py=temp;
}
```

Kada je polje argument funkcije, onda funkcija ne dobiva kopiju čitavog polja već samo pokazivač na prvi element polja. Pri pozivu fukciji se daje samo ime polja (bez uglatih zagrada) jer ono predstavlja pokazivač na prvi element.

Činjenica da funkcija koja uzima polje kao argument očekuje pokazivač na prvi element polja može se iskoristiti da se funkciji dade samo dio polja. Na primjer, funkcija `f` u programu

```
char z[100];
void f(char *);
.....
f(&z[50]);
```

dobit će samo zadnjih 50 elemenata polja `z`.

Uočimo da funkciju koja uzima kao argument jednodimenzionalno polje možemo deklarirati i kao funkciju koja uzima pokazivač na tip polja, kao što to pokazuje sljedeći primjer:

```
#include <stdio.h>

int a[3]={1,2,3};
void f(int *);
void g(int []);

int main(void) {
    f(a); g(a);
    return 0;
}
void f(int *x) {
   int j;
   for(j=0;j<3;++j) printf("%d ",x[j]);
   printf("\n");
}
void g(int x[]) {
   int j;
   for(j=0;j<3;++j) printf("%d ",x[j]);
   printf("\n");
}
```

U oba slučaja bit će ispisano polje `a`.

Nadalje, funkcija može vratitit pokazivač kao u sljedećem primjeru.

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

char *tocka(char *niz)
{
    char *p;
    for(p=niz; *p !='\0'; ++p)
        if(*p == '.') return p;
    return NULL;
}

int main(void) {
    char *p="bigjob.com";

    printf("Pokazivac na prvi znak=%p,\n"
           "pokazivac na tocku=    %p\n",p,tocka(p));
    return 0;
}
```

Funkcija `tocka` vraća pokazivač na `char`. Tu je važno primijetiti da funkcija ne smije vratiti pokazivač na lokalnu varijablu. Takav pokazivač ne pokazuje na korektnu memorijsku lokaciju jer se lokalna varijabla uništava nakon izlaska iz funkcija. Iznimka tog pravila je statička lokalna varijabla. Naime, statička varijabla postoji za cijelo vrijeme izvršavanja programa i ako funkcija vrati pokazivač na nju, ona se može koristiti i izvan funkcije; vidi primjer u sekciji 11.6.

11.3 Operacije nad pokazivačima

11.3.1 Povečavanje i smanjivanje

Aritmetičke operacije dozvoljene nad pokazivačima konzistentne su sa svrhom pokazivača da pokazuju na varijablu određenog tipa. Ako je `pi` pokazivač tipa `int`, onda će `pi+1` biti pokazivač na sljedeću varijablu tipa `int` u memoriji. To znači da dodavanje jedinice pokazivaču ne povećava adresu koju on sadrži za jedan, već za onoliko koliko je potrebno da nova vrijednost pokazuje na sljedeću varijablu istog tipa u memoriji.

```
#include <stdio.h>

int main(void)
```

```
{
    float  x[]={1.0,2.0,3.0},*px;
    px=&x[0];
    printf("Vrijednosti: x[0]=%g, x[1]=%g, x[2]=%g\n",
                                        x[0],x[1],x[2]);
    printf("Adrese      : x[0]=%x, x[1]=%x, x[2]=%x\n",
                                          px,px+1,px+2);
    return 0;
}
```

U ovom primjeru vidimo da će pokazivač biti inkrementiran dovoljno da pokaže na sljedeću `float` vrijednost. Uočimo da smo pokazivače ispisali u formatu `%x` kao heksadecimalni cijeli broj (usporedite s ispisom u formatu `%p`).

Svakom pokazivaču moguće je dodati i oduzeti cijeli broj. Stoga ako je `px` pokazivač i `n` varijabla tipa `int`, onda su dozvoljene operacije

```
++px   --px   px+n   px-n
```

Pokazivač `px+n` pokazuje na `n`-ti objekt nakon onog na kog pokazuje `px`.

Unarni operatori `&` i `*` imaju viši prioritet od aritmetičkih operatora i operatora pridruživanja. Stoga u izrazu

```
*px += 1;
```

dolazi do povećanja za jedan vrijednosti na koju `px` pokazuje, a ne samog pokazivača. Isti izraz bismo mogli napisati kao

```
++*px;
```

stoga što je asocijativnost unarnih operatora zdesna na lijevo pa se prvo primijenjuje dereferenciranje, a zatim inkrementiranje. Iz tog razloga, želimo li koristiti postfiks notaciju operatora inkrementiranja, moramo koristiti zagrade:

```
(*px)++;
```

Izraz `*px++` inkrementirao bi pokazivač nakon što bi vratio vrijednost na koju `px` pokazuje.

11.3.2 Pokazivači i cijeli brojevi

Pokazivaču nije moguće pridružiti vrijednost cjelobrojnog tipa. Iznimku jedino predstavlja nula. Naime, C garantira da nula nije legalna adresa i omogućava da se nula pridruži bilo kojoj pokazivačkoj varijabli s ciljem da se signalizira kako varijabla ne sadrži legalnu adresu. Legalno je pisati

```
double *p=0;
```

To je naročito korisno kod automatskih varijabli koje pri pozivu funkcije imaju nedefiniranu vrijednost. Često se koristi u ovu svrhu simbolička konstanta `NULL`

```
#define NULL 0
   ........
 double *p=NULL;
```

(simbolička konstanta `NULL` definirana je u `<stdio.h>`). Pokazivače je osim s drugim istovrsnim pokazivačem moguće uspoređivati i s nulom, tako da je moguće pisati

```
if(px != 0) ...
```

Uspoređivanje s drugim cijelim brojevima nije dozvoljeno:

```
if(px == 0xBFFFF986) ... // POGRESNO
```

11.3.3 Uspoređivanje pokazivača

Pokazivače istog tipa možemo međusobno uspoređivati pomoću relacijskih operatora. Takva operacija ima smisla ako pokazivači pokazuju na isto polje. Ako su `px` i `py` dva pokazivača istog tipa, onda je moguće koristiti izraze

```
px < py    px > py    px == py   px != py
```

Rezultat tih operacija je 1 ili 0 ovisno o tome da li je reacija zadovoljena ili ne.

11.3.4 Oduzimanje pokazivača

Jedan pokazivač može se oduzeti od drugoga ukoliko oni pokazuju na isto polje. Ako su `px` i `py` dva pokazivača na isto polje te ako je `py > px`, tada je `py-px+1` broj elemenata između `px` i `py`, uključujući krajeve. Uočimo da je `py-px` vrijednost cjelobrojnog tipa (a ne pokazivačkog). Na primjer, funkcija koja daje broj znakova u stringu može biti napisana na sljedeći način:

```
int strlen(char *s)
{
    char *p=s;

    while(*p != '\0') p++;
    return p-s;
}
```

s						p
1	2	3	4	5	6	\0

p-s=6

Moguće je da na nekim sustavima razlika dva pokazivača ne stane nužno u varijablu tipa `int`. Stoga se u datoteci zaglavlja `<stddef.h>` definira cjelobrojni tip podatka `ptrdiff_t` čija je širina dovoljna da primi razliku bilo koja dva pokazivača.

Gore navedene operacije su jedine koje su dozvoljene s pokazivačima. Navedimo ih sve ponovo.

- Pokazivaču može biti pridružena adresa (npr. `px=&x`);
- Pokazivaču može biti pridružen pokazivač istog tipa (npr. `px=py`);
- Pokazivaču može biti pridružena nula (npr. `px=0` ili `px=NULL`);
- Pokazivaču može biti dodana ili oduzeta cjelobrojna varijabla (npr. `px+3, ++px, --px` itd.);
- Dva pokazivača mogu biti oduzeta ukoliko pokazuju na isto polje;
- Dva pokazivača mogu biti povezana relacijskim operatorom ako pokazuju na isto polje.

11.3.5 Primjer

Kao primjer napišimo verziju funkcije `strcpy` iz standardne biblioteke. Prva verzija ima oblik

```
void strcpy(char *s, char *t)
{
    int i=0;
    while((s[i] = t[i]) != '\0') i++;
}
```

Funkcija kopira polje znakova na koje pokazuje `t` u polje na koje pokazuje `s`. Kopiranje se zaustavlja kada se kopira nul znak `'\0'`.

Istu funkciju možemo izvesti i bez uglatih zagrada, koristeći aritmetiku pokazivača.

```
void strcpy(char *s, char *t)
{
    while((*s = *t) != '\0') {
        s++; t++;
    }
}
```

Ovdje koristimo viši prioritet operatora dereferenciranja od operatora pridruživanja i inkrementiramo pokazivače umjesto indeksa. Budući da unarni operatori imaju asocijativnost zdesna na lijevo gornji kôd možemo skratiti i pisati

```
void strcpy(char *s, char *t)
{
    while((*s++ = *t++) != '\0') ;
}
```

Pokazivači `s` i `t` bit će povećani nakon što pridruživanje bude izvršeno. Konačno, kôd možemo još malo skratiti ako uočimo da je `...!='\0'` uspoređivanje izraza s nulom, pa možemo pisati

```
void strcpy(char *s, char *t)
{
    while(*s++ = *t++) ;
}
```

Naravno, ovakav kôd slabo izražava namjeru programera te ga treba izbjegavati.

11.3.6 Generički pokazivač

Pokazivači na različite tipove podatatka općenito se ne mogu pridruživati. Na primjer,

```
char *pc;
int *pi;
  .....
pi=pc;  /* NEISPRAVNO */
```

Razlog je u tome što konverzija pokazivača na jedan tip u pokazivač na neki drugi tip može dovesti do promjene interne reprezentacije pokazivača. Ipak, svako takvo pridruživanje je dozvoljeno uz eksplicitnu promjenu tipa, tj. upotrebu *cast* operatora:

```
char *pc;
int *pi;
.....
pi=(int *) pc;  /* ISPRAVNO */
```

Ako ponovna izvršimo konverziju u polazni tip pokazivača, nije garantirano da se vrijednost pokazivača neće promijeniti.

Pokazivač može biti deklariran kao pokazivač na `void` i tada govorimo o generičkom pokazivaču.

```
void *p;
```

Pokazivač na bilo koji tip može se konvertirati u pokazivač na `void` i obratno, bez promjene pokazivača.

```
double *pd0,*pd1;
void *p;
.....
p=pd0;   /* ISPRAVNO */
pd1=p;   /* ISPRAVNO */
```

Osnovna uloga generičkog pokazivača je da omogući funkciji da uzme pokazivač na bilo koji tip podatka.

```
double *pd0;
void f(void *);
.....
f(pd0);   /* O.K. */
```

S druge strane, ako se želi da funkcija primi upravo pokazivač na `double`, onda formalni argument treba deklarirati kao pokazivač na `double`.

Generički pokazivač se ne može dereferencirati, povećavati i smanjivati.

U sljedećem primjeru funkcija `print` uzima generički pokazivač `ptr` i jedan argument tipa `char` koji govori na kakav objekt `ptr` pokazuje. Argument se zatim ispisuje na odgovarajući način.

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

```
void print(const char c, void *ptr)
{
    int    *pi;
    double *px;
    char   *pc;

    if(c == 'i'){
        pi=ptr; printf("i=%d\n",*pi);
    }
    else if(c == 'f'){
        px=ptr; printf("x=%f\n",*px);
    }
    else{
        pc=ptr; printf("c=%s\n",pc);
    }
}

int main(void) {
    double a=8.0;
    int    j=8;
    char  *s="string";
    void  *p;

    p=&a; print('f',p);
    p=&j; print('i',p);
    p=s;  print('s',p);

    return 0;
}
```

11.4 Pokazivači i jednodimenzionalna polja

Pokazivači i polja su usko vezani. Ime jednodimenzionalnog polja je konstantan pokazivač na prvi elemet polja. Ako imamo polje `x` i pokazivač istog tipa `px`, onda `px` nakon naredbe

```
px=&x[0];
```

pokazuje na prvi element polja `x`. Isti efekt postižemo ako napišemo

```
px=x;
```

Štoviše, imamo da je `*(px+i)==x[i]` i dvije forme možemo koristiti ravnopravno. U skladu s time, ako imamo pokazivač `px`, onda možemo koristiti notaciju `px[i]` umjesto `*(px+i)`.

Pokazivači i polja su stoga gotovo ekvivalentni. Svaka funkcija `f()` koja kao argument uzima jednodimenzionalno polje nekog tipa može se deklarirati kao

```
tip_rez f(tip x[])
```

ili kao

```
tip_rez f(tip *x)
```

Ipak postoje i bitne razlike. Polje `x` nije l-vrijednost, tj. naredbe tipa

```
x= ...;
```

nisu dozvoljene. Isto tako, ime polja je konstantan pokazivač pa nije dozvoljeno pisati `x++, x--` i slično. Ali, u izrazu `x+1` bit će iskorištena pokazivačka aritmetika. To možemo iskoristiti u sljedećoj situaciji: uzmimo da imamo funkciju `f(float *)` koja očekuje polje tipa `float` i da kao stvarni argument funkciji želimo predati dio polja `x` od šestog do zadnjeg elemeta. Tada funkciju `f()` možemo pozvati s

```
f(&x[5]);
```

ili

```
f(x+5);
```

Da bismo naglasili razliku između polja i pokazivača pogledajmo sljedeće dvije deklaracije:

```
char poruka[]="Dolazim odmah.";
char *pporuka="Dolazim odmah.";
```

U prvoj se deklarira i inicijalizira polje od 15 znakova koje je moguće dohvatiti i mijenjati putem indeksa u uglatim zagradama. U drugoj deklaraciji deklarira se pokazivač na `char` i inicijalizira adresom znakovnog niza `"Dolazim odmah."`. Sve članove niza možemo dohvatiti putem pokazivača, ali modificiranje niza putem pokazivača je nedefinirana operacija. Naime, prevodilac ima slobodu konstantan niz znakova smjestiti bilo gdje u memoriji pa stoga ne može garantirati da će se vrijednosti niza moći mijenjati. S druge strane pokazivač `pporuka` može pokazivati i na svaku drugu `char` varijablu.

11.5 Pokazivači i const

Vidjeli smo da modifikator `const` možemo koristiti za definiciju konstanti. Na primjer

```
const double pi=3.14159;
```

Jednako tako možemo ga primijeniti na pokazivače. Uzmimo sljedeći primjer

```
double polje[5]={0.1,0.2,0.3,0.4,0.5};
const double *pp=polje;
```

Pokazivač `pp` deklariran je kao pokazivač na konstantan `double`. To znači da on pokazuje na konstantnu varijablu tipa `double`. Pokazivač `pp` smo inicijalizirali s nekonstantnim poljem `polje`, što je dozvoljeno. Prevodilac nam jedino neće dozvoliti mijenjanje elemenata polja putem pokazivača `pp`.

```
*pp=3.2;     //  nije dozvoljeno
pp[3]=1.0;  // nije dozvoljeno
polje[3]=1.0;  // O.K.
```

Sam pokazivač možemo slobodno mijenjati

```
pp++;   // O.K.
```

Ako definiramo konstantnu varijablu, onda samo pokazivač na konstantan tip može pokazivati na tu varijablu. Na primjer,

```
const double polje[5]={0.1,0.2,0.3,0.4,0.5};
const double *pp=polje;  /*  ISPRAVNO  */
double *pt=polje;        /*  NIJE DOZVOLJENO */
```

Pokazivač na konstantan tip se deklarira kao argument funkcije da bi se pokazalo da funkcija neće koritit pokazivač za mijenjanje varijable na koju pokazuje. Funkcija koja samo ispisuje elemente polja mogla bi biti deklarirana na ovaj način:

```
void print_array(const double *array, int n);
```

Takva će funkcija prihvatiti konstantno i nekonstantno polje kao argument.

Moguće je definirati konstantan pokazivač na nekonstantan tip. Treba samo pomaknuti `const` u definiciji. Na primjer,

```
double polje[5]={0.1,0.2,0.3,0.4,0.5};
double * const  pp=polje;
pp = &polje[1];  /*  NIJE DOZVOLJENO  */
*pp=56.9;        /* DOZVOLJENO  */
```

Ovakav pokazivač pokazuje uvijek na istu lokaciju. Konačno, možemo definirati i konstantan pokazivač na konstantan tip:

```
double polje[5]={0.1,0.2,0.3,0.4,0.5};
const double * const  pp=polje;
pp = &polje[1];  /*  NIJE DOZVOLJENO  */
*pp=56.9;        /*  NIJE DOZVOLJENO  */
```

11.6 Polja pokazivača

Polje pokazivača ima deklaraciju

```
tip_pod *ime[izraz];
```

Na primjer

```
int  *ppi[10];
```

`ppi` je polje od 10 pokazivača na `int`. Analogno se definiraju i višedimenzionalna polja pokazivača. Uočimo da uglate zagrade imaju viši prioritet od operatora dereferenciranja. Stoga gornja deklaracija predstavlja polje od 10 elemenata koji su pokazivači na tip `int`, a ne pokazivač na polje od 10 varijabli tipa `int`. U ovom drugom slučaju morali bismo pisati

```
int  (*ppi)[10];
```

Polja pokazivača koriste se umjesto dvodimenzionalnih polja kada treba zapamtiti niz polja različite dužine. Na primjer,

```
char *mjeseci[]={ "sijecanj", "veljaca", "ozujak",
                  "travanj",  "svibanj", "lipanj",
                  "srpanj",   "kolovoz", "rujan",
                  "listopad", "studeni", "prosinac"};
```

Ovdje smo iskoristili činjenicu da pokazivač na `char` možemo inicijalizirati stringovima. Pojedine znakove možemo dohvatiti indeksiranjem. Na primjer, `mjeseci[2][2]` je jednako `'u'`.

Polje `mjeseci` možemo iskoristiti u funkciji koja za zadani broj vraća pripadni mjesec.

```
char *ime_mjeseca(int n)
{
 static char *mjeseci[]={ "Nekorektan broj mjeseca",
                          "sijecanj", "veljaca", "ozujak",
```

```
                        "travanj","svibanj", "lipanj",
                        "srpanj","kolovoz", "rujan",
                        "listopad", "studeni", "prosinac"};
            return (n<1 || n>12) ? mjeseci[0] : mjeseci[n];
}
```

Uočimo da smo polje `mjeseci` deklarirali `static` kako ne bi bilo uništeno na izlazu iz funkcije.

11.7 Pokazivači i višedimenzionalna polja

U C-u dvodimenzionalno polje je polje čiji je svaki element jedno jednodimenzionalno polje. Stoga, ako je

```
static int x[MAXX][MAXY];
```

jedno dvodimenzionalno polje, onda je `x[i][j]` element na mjestu (i,j), dok je `x[i]` polje od `MAXY` elemenata. Slično je i s višedimenzionalnim poljima. Ako imamo

```
static int x[2][3][4];
```

onda je `x` polje dimenzije $2 \times 3 \times 4$ dok je `x[i]` polje od tri elementa, svaki od kojih je polje od 4 elemeta; `x[i][j]` je polje od 4 elemeta.

Operacija indeksiranja `E1[E2]` identična je s `*(E1+E2)` i stoga je to komutativna operacija (što nije naročito korisno). Na primjer, ako je `x` polje onda drugi element polja možemo zapisati kao `x[1]` ili kao `1[x]`.

Kod dvodimenzionalnog polja izraza oblika `x[2][3]` interpretira se na sljedeći način:

```
x[2][3]   ->   *(x[2]+3)   ->   *(*(x+2)+3).
```

Ovdje je

`x` pokazivač na jednodimenzionalno polje čiji su elemeti polja;

`x+2` je pokazivač na treće polje;

`*(x+2)` treće polje, dakle pokazivač na prvi element trećeg polja;

`*(x+2)+3` je tada pokazivač na četvrti element tog polja;

`*(*(x+2)+3)` je sam taj element.

Na analogan način tretiraju se tri i višedimenzionalna polja.

Prilikom deklaracije višedimenzionalnog polja (ali ne i definicije) mogu se koristiti dvije ekvivalentne forme: potpuna

```
tip_pod ime[izraz_1][izraz_2]....[izraz_n];
```

ili bez prve dimenzije

```
tip_pod ime[][izraz_2]....[izraz_n];
```

ili pomoću pokazivača

```
tip_pod (*ime)[izraz_2]....[izraz_n];
```

U zadnjem primjeru su zagrade nužne jer bi u suprotnom imali polje pokazivača na `tip`.

11.7.1 Matrica kao pokazivač na pokazivač

Osnovni nedostatak programskog jezika C u numeričkim primjenama je način na koji se matrice predaju funkcijama. Standard C99 je to riješio uvodeći polja varijabilne duljine. Ovdje ćemo pokazati jedno rješenje problema koje se ne oslanja na PVD.

Neka je definirano na primjer polje

```
double A[1024][1024];
```

Tada definiramo polje pokazivača

```
double *aa[1024];
```

Ove pokazivače inicijaliziramo pokazivačima na retke matrice `A`:

```
for(i=0;i<1024;++i) aa[i]=A[i];
```

Zatim funkciju koja uzima matricu `A` deklariramo kao

```
void f(double **aa, int n, int m)
```

gdje su `n` i `m` dimenzije matrice. Sama matrica je deklarirana kao pokazivač na pokazivač na `double`. Funkciju `f()` ne pozivamo s poljem `A` već s poljem pokazivača `aa`. Na primjer,

```
f(aa,n,m);
```

Unutar funkcije `f` element `A[i][j]` možemo dohvatiti kao `aa[i][j]`. Slijedi jedan kompletan primjer:

```
#include <stdio.h>
double A[1024][1024];
void f(double **aa, int n, int m);

int main(void)
{
    int i;
    double *aa[1024];

    A[56][1000]=123.445;
    for(i=0;i<1024;++i) aa[i]=A[i];
    f(aa,56,1000);
    return 0;
}

// ispisi element na mjestu (i,j)
void f(double **aa, int i, int j)
{
    printf("%f\n",aa[i][j]);
}
```

Program će naravno ispisati `123.44500`. Ovom tehnikom postižemo to da funkcija `f` ne mora znati stvarne dimenzije matrice `A` da bi s njom mogla raditi. Cijena koju smo morali platiti je alokacija jednog polje pokazivača duljine `n`, gdje je `n` broj redaka matrice.

Uočimo da bi poziv

```
 f(A,n,m);  /* POGRESNO */
```

bio nekorektan i prevodilac bi javio grešku. Naime, pri pozivu funkcije dolazi do konverzije polja u pokazivač na prvi element, ali ta se konverzija vrši samo jednom. Dvodimenzionalno polje tada postaje pokazivač na jednodimenzionalno polje, što nije isto što i pokazivač na pokazivač.

11.8 Dinamička alokacija memorije

Varijable i polja možemo alocirati za vrijeme izvršavanja programa prema potrebama za memorijskim prostorom. U tu svrhu služimo se funkcijama `malloc` i `calloc` deklariranim u `<stdlib.h>`. Funkcije `malloc` i `calloc` deklarirane su na sljedeći način

```
void *malloc(size_t n);
void *calloc(size_t n, size_t size);
void *realloc(void *ptr, size_t n);
void free(void *ptr);
```

`size_t` je cjelobrojni tip bez predznaka definiran u `<stddef.h>`, dovoljno širok da primi vrijednost koju vraća `sizeof` operator.

Funkcija `malloc` uzima jedan argument `n` koji predstavlja broj bajtova koji treba alocirati i rezervira memorijski blok od `n` bajtova.[1] Funkcija vraća pokazivač na rezervirani blok memorije ili `NULL` ako zahtijev za memorijom nije mogao biti ispunjen. Vraćeni pokazivač je generički, tj. tipa `void*` pa ga stoga prije upotrebe treba konvertirati u potrebni tip pokazivača. Tipičan primjer upoterbe funkcije `malloc` je sljedeći:

```
double *p;
......
p=(double *) malloc(128*sizeof(double));
if(p==NULL) {
   printf("Greska: alokacija memorije nije uspjela!\n");
   exit(-1);
}
```

Uočimo da operatorom `sizeof` postižemo neovisnost o stvarnoj duljini varijable `double`. Ispitivanje je li pokazivač koji `malloc` vraća `NULL` pointer je nužno za ispravno funkcioniranje programa.

Funkcija `calloc` uzima dva argumenta. Prvi je broj je broj varijabli za koje želimo rezervirati memoriju, a drugi je broj bajtova koji svaka varijabla zauzima. Funkcija vraća pokazivač na blok memorije dovoljno velik da primi polje od `n` objekata veličine `size`, ili `NULL` ako zahtijev za memorijom nije mogao biti ispunjen. Memorija se inicijalizira nulama tako da je svaki bit postavi na nulu. Primjer upotrebe je

```
int *p;
......
p=(int *) calloc(128,sizeof(int));
if(p==NULL)
{
   printf("Greska: alokacija memorije nije uspjela!\n");
   exit(-1);
}
```

[1]Jedan bajt u C-u je duljina varijable `char` u bitovima. Najčešće iznosi uobičajenih osam bitova.

Primijetimo da svi bitovi postavljeni na nulu ne reprezentiraju nužno, na svakoj hardwareskoj platformi, nulu u sustavu s pokretnim zarezom i nul-pokazivač.

Funkcija `realloc` uzima kao prvi argument pokazivač na memoriju alociranu s `malloc` ili `calloc` funkcijom i mijenja njenu veličinu, čuvajući sadržaj memorije. Ako je potrebno, sadržaj će biti kopiran na novu lokaciju. Nova veličina memorijskog bloka (u bajtovima) dana je u drugom argumentu. Funkcija vraća pokazivač na alocirani blok, ili `NULL` ako realokacija nije uspjela. U slučaju neuspjeha realokacije stari memorijski blok ostaje nepromijenjen. Ukoliko `realloc` vrati pokazivač različit od prvog argumenta, onda je stari memorijski blok dealociran, a stari sadržaj je premješten na novu lokaciju.

Ako je nova dimenzija memorijskog bloka veća od stare, onda se dodaje nova memorija koja, kao i kod `malloca`, nije inicijalizirana. Ako je nova dimenzija bloka manja od stare, onda se samo dealocira dio memorije.

Memoriju alociranu pomoću funkcija `malloc` i `calloc` potrebno je delocirati pomoću funkcije `free`. Ona uzima pokazivač na alocirani blok memorije i oslobađa memoriju.

```
free(p);
```

Memorijski prostor treba delocirati čim više nije potreban.

11.9 Pokazivač na funkciju

Pokazivač na funkciju deklarira se kao

```
tip_pod (*ime)(tip_1 arg_1,tip_2 arg_2, ...,tip_n arg_n);
```

`ime` je tada pokazivač na funkciju koja uzima n argumenata tipa `tip_1` do `tip_n` i vraća vrijednost tipa `tip_pod`. Zagrade su obavezne. U primjeru

```
int (*pf)(char c, double a);
```

`pf` je pokazivač na funkciju koja uzima dva argumenta, prvi tipa `char`, a drugi tipa `double` te vraća `int`. Kod deklaracija (koje nisu definicije) imena varijabli se mogu ispustiti. Tako ako funkcija `g` uzima kao argument pokazivač na funkciju gornjeg tipa i ne vraća ništa, onda bismo deklarirali

```
void g(int (*)(char, double));
```

Ako je `pf` pokazivač na funkciju onda je `(*pf)` može koristiti kao ime funkcije.

Uzmimo kao primjer funkciju koja računa približno integral zadane funkcije po trapeznoj formuli.

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

double integracija(double, double, double (*)(double));
int main(void)
{
    printf("Sinus: %f\n",integracija(0,1,sin));
    printf("Kosinus: %f\n",integracija(0,1,cos));
    return 0;
}

double integracija(double a, double b, double (*f)(double))
{
  return 0.5*(b-a)*((*f)(a)+(*f)(b));
}
```

Funkcija `integracija` uzima donju i gornju granicu integracije i pokazivač na funkciju koju treba integrirati. Kao i kod polja, ime funkcije se kao stvarni argument neke funkcije konvertira u pokazivač na funkciju. Zato smo funkciju `integracija` moglo pozvati kao

```
 integracija(0,1,sin)
```

Na funkciju možemo primijeniti adresni operator tako da smo mogli pisati

```
 integracija(0,1,&sin)
```

i dobili bismo isti rezultat.

Unutar same funkcije integracija pokazivač na funkciju mora se nalaziti unutar zagrada je bi `*f(a)`, zbog višeg prioriteta zagrada, bio interpretiran kao dereferenciranje objekta `f(a)`.

11.10 Argumenti komandne linije

U sistemskim okruženjima kao što su DOS i UNIX, programu se mogu predati određeni parametri, tzv. argumenti komandne linije, tako da se navedu na komandnoj liniji iza imena programa, separirani razmacima. Na primjer, naredba (program) `cp` u operacijskom sustavu UNIX koja vrši kopiranje datoteka poziva se sa dva parametra:

```
    cp ime1 ime2
```

gdje je `ime1` ime datoteke koju želimo kopirati, a `ime2` ime datoteke na koju `ime1` želimo kopirati. Programi pisani u C-u također mogu primiti argumente komandne linije. Tu funkcionalnost osigurava funkcija `main()`.

Funkcija `main()` prihvaća dva argumenta: `argc` tipa `int` i polje pokazivača na `char`, obično nazvan `argv`. Nova forma funkcije `main()` glasi:

```
int main(int argc, char *argv[])
{ .... }
```

Mehanizam je sljedeći: operacijski sustav u varijablu `argc` smješta broj argumenta komandne linije koji su utipkani pri startanju programa, uvećan za jedan. Ako nema argumenata komandne linije, onda je `argc=1`. U `argv` se nalaze pokazivači na argumente komandne linije, spremljeni u `argv[0]` do `argv[argc-1]`. Pri tome je `argv[0]` uvijek pokazivač na string koji sadrži ime programa koji se izvršava. Ostali parametri su smješteni redom kojim su upisani na komandnoj liniji.[2] Nadalje `argv[argc]` mora biti nul-pokazivač.

Na primjer, program koji samo ispisuje argument komandne linije koji su mu dani izgledao bi ovako:

```
/* program args   */
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    for(i=0;i<argc; i++)
        printf("%s%s", argv[i],(i<argc-1) ? "," : ".");
    printf("\n");
    return 0;
}
```

Pozovemo li program `args` narebom

```
 args ovo su neki parametri
```

on će ispisati

```
 args,ovo,su,neki,parametri.
```

Operacijski sustav UNIX, odnosno *shell* koji se koristi ekspandirat će specijalne znakove `?`, `*` i `[]` prije nego što ih preda pozivnom programu. Više detalja može se naći priručniku za dani *shell* (`sh`, `ksh`, `csh` ili `bash`, vidi još odgovarajuće `man`-stranice).

[2] Ako shvatimo ime programa kao argument komandne linije, onda `argc` daje točno broj argumenata komandne linije.

11.11 Složene deklaracije

Pri čitanju složenih deklaracija osnovno pravilo je da je deklaracija objekta u skladu s načinom korištenja objekta. Ako deklariramo

```
int *f(void);
```

onda `*f()` mora biti tipa `int`. To znači da je `f` funkcija koja ne uzima argumente i vraća pokazivač na `int`.

Prilikom interpretacije deklaracije uzimaju se u obzir prioriteti pojedinih operatora. Ti prioriteti mogu se promijeniti upotrebom zagrada, što komplicira čitanje deklaracija. Tako je

```
int (*f)(void);
```

deklaracija pokazivača na funkciju koja ne uzima argumente i vraća `int`.

Evo nekoliko primjera:

```
int *p;         /* p je pokazivac na int */

int *p[10];    /* p je polje od 10 pokazivaca na int    */

int (*p)[10];  /* p je pokazivac na polje od 10 elemenata
                    tipa int */

int *f(void);  /* p je funkcija koja ne uzima argumente i
                    vraca pokaziva\v c na  int */

int p(char *a); /* p je funkcija koja uzima pokazivac na
                     char i vraca int  */

int *p(char *a); /* p je funkcija koja uzima pokazivac na
                      char i vraca pokazivac na int  */

int (*p)(char *a); /* p je pokazivac na funkciju koja uzima
                        pokazivac na char i vraca  int  */

int (*p(char *a))[10]; /* p je funkcija koja uzima pokazivac
                            na char i vraca pokazivac na polje
                            od 10 elemenata tipa int int  */

int p(char (*a)[]); /* p je funkcija koja uzima pokazivac
                         na polje znakova i vraca int */
```

```
int (*p)(char (*a)[]); /* p je pokazivac na funkciju koja
                          uzima pokazivac na polje znakova
                          i vraca int */

int *(*p)(char (*a)[]); /* p je pokazivac na funkciju koja
                          uzima pokazivac na polje znakova
                          i vraca pokazivac na int */

int *(*p[10])(char *a); /* p je polje od 10 pokazivaca na
                           funkciju koja uzima pokazivac na
                           char i vraca pokazivac na int */
```

Poglavlje 12

Strukture

Polja grupiraju veći broj podataka istog tipa. Strukture služe grupiranju više podataka različitih tipova.

12.1 Deklaracija strukture

Struktura se deklarira na sljedeći način:

```
struct ime {
      tip_1 ime_1;
      tip_2 ime_2;
      ....   ....
      tip_n ime_n;
};
```

`struct` je rezervirano ime, `ime` je naziv koji dajemo strukturi, a unutar vitičastih zagrada su pojedini članovi strukture. Na primjer, možemo definirati strukturu `tocka` koja definira točku u ravnini

```
struct tocka {
      int x;
      int y;
};
```

Ova struktura sadrži dva člana istog tipa, no članovi strukture mogu biti bilo kojeg tipa, osnovni tipovi, pokazivači, polja i druge strukture.

12.1.1 Varijable tipa strukture

Deklaracija strukture samo definira tip podatka, ali ne rezervira memorijski prostor. Nakon što je struktura deklarirana možemo definirati varijable

tog tipa:

```
mem_klasa struct ime  varijabla1, varijabla2,...;
```

gdje je `mem_klasa` memorijska klasa, `ime` je naziv strukture, a `varijabla1, varijabla2,..` varijable koje definiramo. Na primjer,

```
struct tocka  p1,p2;
```

Ovdje su varijable `p1` i `p2` tipa `tocka` i za njih je ovdje rezerviran potreban memorijski prostor.

Varijable tipa strukture moguće je definirati unutar deklaracije strukture na sljedeći način:

```
mem_klasa struct ime {
      tip_1 ime_1;
      tip_2 ime_2;
      ....   ....
      tip_n ime_n;
} varijabla1, varijabla2,...;
```

tako smo mogli pisati

```
struct tocka {
      int x;
      int y;
} p1,p2;
```

Takav način deklaracije strukture i definicije varijabli pogodan je kada varijable tipa struture uvodimo samo na jednom mjestu. U tom slučaju ne moramo davati ime strukturi, tj. možemo pisati

```
struct {
      int x;
      int y;
} p1,p2;
```

Naravno, druge varijable tog tipa na drugom mjestu nije moguće definirati.

12.1.2 Inicijalizacija strukture

Varijabla tipa strukture može se inicijalizirati prilikom definicije na ovaj način:

```
mem_klasa struct ime  varijabla={v_1, v_2,..., v_n};
```

pri čemu se v_1 pridružuje prvom članu strukture, v_2 drugom itd.

Uzmemo li strukturu

```
struct racun {
      int broj_racuna;
      char ime[80];
      float stanje;
};
```

onda možemo inicijalizirati varijablu `kupac` na ovaj način:

```
struct racun kupac={1234,"Goran S.", -234.00};
```

Slično se može inicijalizirati i polje struktura

```
struct racun kupci[]={34,"Ivo R.",    456.00,
                      35,"Josip S.",  234.00,
                      36,"Dalibor M.",00.00};
```

12.1.3 Struktura unutar strukture

Strukture mogu sadržavati druge strukture kao članove. Pravokutnik može biti određen donjim lijevim (`pt1`) i gornjim desnim vrhom (`pt2`):

```
struct pravokutnik {
       struct tocka pt1;
       struct tocka pt2;
};
```

Pri tome deklaracija strukture `tocka` mora prethoditi deklaraciji strukture `pravokutnik`.

Ista imena članova mogu se koristiti u različitim strukturama.

12.2 Rad sa strukturama

12.2.1 Operator točka

Članovima strukture može se pristupiti individualno putem operatora točka (.). Ako je `var` varijabla tipa strukture koja sadrži član `memb`, onda je

```
var.memb
```

član `memb` u strukturi `var`. Ako je `pt` varijabla tipa `struct tocka`,

```
struct tocka pt;
```

onda su `pt.x` i `pt.y` `int` varijable koje daju x i y koordinatu točke `pt`. Ako je `rect` varijabla tipa `struct pravokutnik`,

```
struct pravokutnik rect;
```

onda su `rect.pt1.x` i `rect.pt1.y`, x i y koordinate donjeg lijevog vrha pravokutnika itd. Udaljenost točke `pt` do točke (0,0) izračunali bismo ovako

```
double dist, sqrt(double);
dist=sqrt((double) pt.x * pt.x + (double) pt.y * pt.y);
```

Operator točka (.) separira ime varijable i ime člana strukture. Spada u najvišu prioritetnu grupu i ima asocijativnost slijeva na desno. Zbog najvišeg prioriteta točka operatora izraz

```
++varijabla.clan;
```

je ekvivalentan s `++(varijabla.clan)`; isto tako `&varijabla.clan` je ekvivalentan s `&(varijabla.clan)`. Adresni operator može se primijeniti na čitavu strukturu i na pojedine njene članove. Broj memorijskih lokacija koje struktura zauzima može se dobiti pomoću `sizeof` operatora kao i za jednostavne tipove podataka. Taj broj može biti veći od sume broja memorijskih lokacija za pojedine članove strukture.

Kada struktura sadrži polje kao član strukture, onda članovi polja dosežu izrazom

```
varijabla.clan[izraz]
```

Na primjer, ako je `kupac` varijabla tipa `struct racun`, onda osmi znak u imenu kupca možemo dohvatiti izrazom `kupac.ime[7]`. Ako pak imamo polje struktura, onda pojedini član elementa polja dosežemo izrazom

```
polje[izraz].clan
```

Na primjer, ako je varijabla `kupci` polje tipa `struct racun`, onda broj računa osmog kupca dosežemo izrazom `kupci[7].broj_racuna`.

U zadnja dva primjera dolazi do izražaja socijativnost, jer su uglate zagrade i operator točka istog prioriteta. Njihova asocijativnost je slijeva na desno, tako da se operandi grupiraju prvo oko lijevog operatora.

S članovima strukture postupa se jednako kao i s običnim varijablama istog tipa. Isto vrijedi i za složene članove strukture, polja, strukture itd.

12.2.2 Struktura i funkcije

Jedine operacije koje se mogu provoditi na strukturi kao cjelini su pridruživanje i uzimanje adrese pomoću adresnog operatora. Struktura može biti argument funkcije koja tada dobiva kopiju strukture koja je stvarni argument funkcije. Isto tako funkcija može vratiti strukturu u pozivni program. Na primjer, sljedeća funkcija uzima dvije strukture tipa `tocka` kao argumente i vraća strukturu tipa `tocka` koja je suma dvaju argumenata:

```
struct tocka suma(struct tocka p1, struct tocka p2)
{
     p1.x += p2.x;
     p1.y += p2.y;
     return p1;
}
```

12.3 Strukture i pokazivači

Pokazivač na strukturu definira se kao i pokazivač na osnovne tipove varijabli. U primjeru

```
struct tocka {
      int x;
      int y;
} p1, *pp1;
```

`p1` je varijabla tipa `struct tocka`, a `pp1` je pokazivač na strukturu tipa `tocka`. Pokazivač možemo inicijalizirati adresom varijable:

```
pp1= &p1;
```

Pojedini član strukture može se putem pokazivača doseći izrazom `(*pp1).x` i `(*pp1).y`. Uočimo da su zagrade nužne jer operator točka ima viši prioritet od operatora dereferenciranja. Izraz `*pp1.x` je ekvivalenta s `*(pp1.x)`, što nije korektno formiran izraz.

12.3.1 Operator strelica (->)

C nudi semantički jednostavniji način dohvaćanja člana strukture putem pokazivača: ako je `ptvar` pokazivač na strukturu, a `clan` jedan član strukture, onda je izraz

```
ptvar->clan
```

ekvivalentan s `(*ptvar).clan`. Tako je `p1.x` isto što i `pp1->x` itd.

Ako je član strukture i sam stuktura, onda možemo kombinirati operatore `->` i `.` da bismo došli podčlana uložene strukture:

```
ptvar->clan.podclan
```

Ako je neki član strukture polje, onda elemente polja dostižemo izrazom

```
ptvar->clan[izraz]
```

Struktura može sadržavati članove koji su pokazivači. Budući da `.` i `->` operatori imaju viši prioritet od `*` operatora, vrijednost na koju član pokazuje možemo dohvatiti pomoću

```
*var.clan
```

ili

```
*ptvar->clan
```

Slično, zbog najvišeg prioriteta koji imaju operatori `.` i `->` izrazi poput

```
++ptvar->clan   i  ++ptvar->clan.podclan
```

ekvivalentni su izrazima

```
++(ptvar->clan)   i  ++(ptvar->clan.podclan).
```

Izraz

```
(++ptvar)->clan
```

povećat će pokazivač na strukturu prije nego što se dohvati `clan` strukture. Pri tome se adresa koju sadrži `ptvar` poveća za onoliko okteta koliko iznosi veličina strukture.

Evo još nekoliko promjera izraza s `->` i `.`; neka je

```
struct pravokutnik r, *pr=&r;
```

Tada su sljedeći izrazi ekvivalentni:

```
 r.pt1.x
 pr->pt1.x
(r.pt1).x
(pr->pt1).x
```

Tu se koristi asocijativnost operatora `.` i `->`, koja je slijeva na desno.

12.3.2 Složeni izrazi

Kada imamo pokazivač na strukturu s članom koji je pokazivač mogu se javiti složeniji izrazi. Na primjer, neka je

```
struct {
       int n;
       char *ch;
} *pt;
```

Tada možemo imati izraze

```
(++pt)->ch    (pt++)->ch
```

pri čemu prvi povećava pokazivač `pt`, a zatim vraća `ch` član strukture, dok drugi prvo vraća `ch` član strukture, i nakon toga povećava pokazivač `pt`. U ovom drugom primjeru zagrada nije potrebna, tj. možemo pisati

```
pt++->ch
```

(asocijativnost od `++` je zdesna na lijevo, a `->` slijeva na desno).

Izraz

```
*pt->ch++
```

povećat će clan `ch` nakon što dohvati vrijednost na koju `pt->ch` pokazuje. Naime, operator `->` ima najviši prioritet i prvi se primijenjuje. Unarni operatori imaju isti prioritet i asocijativnost `D->L`. To znači da se prvo primijenjuje operator inkrementiranja na pokazivač `pt->ch`, a tek onda operator dereferenciranja. Kako je operator inkrementiranja u postfiks formi, to će do inkrementiranja doći nakon što se dohvati vrijednost na koju `pt->ch` pokazuje. Da bismo inkrementirali sam objekt na koji `pt->ch` pokazuje treba pisati `(*pt->ch)++`. Konačno,

```
*pt++->ch
```

će inkrementirati `pt` nakon što dohvati objekt na koji `ch` pokazuje.

Ne treba niti spominjati da ovakve izraze treba izbjegavati, razbijati ih u više izraza i maksimalno se koristiti zagradama kako bi značenje izraza bilo jasnije.

12.4 Samoreferentne strukture

Za implementaciju tipova podataka kao što su vezane liste i stabla koristimo samoreferentne strukture. To su strukture u kojima je jedan ili više članova pokazivač na strukturu istog tipa. Jedan primjer je

```
struct element {
    char ime[64];
    struct element *next;
};
```

Struktura `element` ima član `next` koji je pokazivač na strukturu tipa `element`. Taj nam pokazivač služi za povezivanje istovrsnih podataka u listu tako što će `next` pokazivati na sljedeći element u listi; posljednji element ima u polju `next` upisanu vrijednost `NULL`. Na taj način dobivamo jednostruko povezanu listu.

12.4.1 Jednostruko povezana lista

Ilustrirajmo upotrebu samoreferentne strukture na jednom jednostavnom primjeru jednostruko povezane liste čiji su elementi znakovni nizovi. Početak programa ima sljedeći oblik:

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

/* oznaka kraja liste */
#define KRAJ (struct ime *) 0

/* deklaracija samoreferentne strukture */
struct ime {
    char *p_ime;
    struct ime *next;
};

/* pokazivac na prvi element liste (globalna varijabla) */
struct ime *start=KRAJ;
```

Struktura `ime` sadrži pokazivač na niz znakova `p_ime` i pokazivač na sljedeći element u listi `next`.

Pokazivač na prvi element liste `start` definiran je kao globalna varijabla i inicijaliziran nul-pokazivačem koji označava kraj liste. U ovom trenutku lista je posve definirana i ne sadrži niti jedan element.

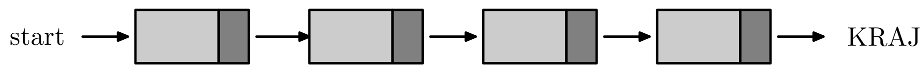
Element se u vezanu listu može dodati i izbrisati na bilo kojem mjestu. Napišimo funkciju `unos` koja dodaje novi element na kraj liste.

```
/* dodavanje novog elementa na kraj liste */
void unos(void) {
  struct ime  *zadnji, *novi;
  char line[128];

  printf("Dodavanje novog elementa na kraj liste.\n");
  novi = (struct ime *) malloc(sizeof(struct ime));
  if(novi == NULL) {
    printf("Nema dovoljno memorije ... ");
    exit(-1);
  }
  novi->next=KRAJ;
  printf("Unesite ime > ");
  scanf(" %[^\n]",line);
  novi->p_ime=(char *) malloc(strlen(line)+1);
  if(novi->p_ime == NULL) {
     printf("Nema dovoljno memorije ...");
     exit(-1);
  }
  strcpy(novi->p_ime,line);

  if(start==KRAJ) /* prazna lista */
  start=novi;
  else {    /* pronadji kraj liste */
    for(zadnji=start; zadnji->next != KRAJ; zadnji=zadnji->next)
    ;  // ne radi nista
  /* neka zadnji pokazuje na novi element */
    zadnji->next=novi;
  }
}
```

Kôd prvo alocira memoriju za novi element. Budući da će on biti zadnji u listi, njegov `next` član se postavlja na `KRAJ`. Zatim se alocira memorija za novo ime koje se učitava u spremnik `line` pomoću `scanf` funkcije i kopira na rezerviranu memoriju sa `strcpy`. Kôd ne uzima u obzir eventualni pretek spremnika `line`.



Novi element je sada inicijaliziran i lokalna varijabla `novi` pokazuje na njega. Ukoliko je lista prazna početak liste inicijaliziramo s `novi`. Ako nije, nalazimo zadnji element i njegov `next` član postavljamo na `novi` koji time postaje zadnji element liste.

Sljedeća funkcija ispisuje listu.

```
void ispis(void) {        /* Ispis liste  */
  struct ime *element;
  int i=1;

  if(start == KRAJ ) {
     printf("Lista je prazna.\n");
     return;
  }
  printf("Ispis liste \n");
  for(element=start; element != KRAJ; element=element->next) {
     printf("%d. %s\n",i,element->p_ime); i++;
  }
    return;
}
```

Ukoliko lista nije prazna u jednoj `for` petlji se prolazi svim njenim članovima.

Za pretraživanje liste koristimo istu `for` petlju kojom obilazimo čitavu listu.

```
void trazi(void) {   /* Pretrazivanje liste */
  struct ime *test;
  char line[128];
  int i;

  printf("Nalazenje imena u listi.\n");
  printf("Unesite ime ");
  scanf(" %[^\n]",line);

  i=1;
  for(test=start; test != KRAJ; test=test->next) {
     if( !strcmp(test->p_ime,line) ) {
       printf("Podatak je %d. u listi\n",i);
       return;
```

```
      }
      i++;
  }
  printf("Podatak nije u listi.\n");
}
```

Funkcija vrši test uspoređivanja upisanog imena sa svakim podatkom u listi linearno, počevši od prvog do zadnjeg elementa liste. Ukoliko ime nije u listi bit će ispisano: `Podatak nije u listi.`

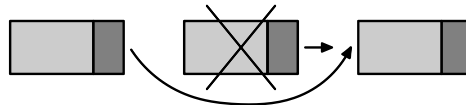
Brisanje elementa iz liste je nešto složenije.

```
void brisi(void)  /* Brisanje elementa iz liste */
{
  struct ime *test, *tmp;
  char line[128];
  int i;

  printf("Brisanje imena iz liste.\n");
  if(start == KRAJ){
    printf("Lista je prazna.\n");
    return;
  }
  printf("Unesite ime ");
  scanf(" %[^\n]",line);

  /* ako je prvi element onaj koji trazimo */
  if( !strcmp(start->p_ime,line) ) {
     printf("Podatak je 1. u listi\n");
     tmp=start;
     start=start->next;
     free(tmp->p_ime);
     free(tmp);
     return;
   }
   i=1;
   for(test=start; test != KRAJ; test=test->next) {
      i++;
      if(!(tmp=test->next)) break; // elm. nije nadjen
      if( !strcmp(tmp->p_ime,line) )
      {
         printf("Brisemo podatak br. %d u listi\n",i);
         test->next=test->next->next;
```

```
          free(tmp->p_ime);
          free(tmp);
          return;
      }
     }
   printf("Podatak nije u listi.\n");
   return;
}
```



Kod brisanja elementa iz liste prvo provjeravamo da li je lista prazna. Ako nije, unosimo ime koje želimo izbrisati i provjeravamo da li je ono na prvom mjestu u listi. To radimo stoga što je brisanje elementa na prvom mjestu nešto jednostavnije od brisanja proizvoljnog elementa. Dovoljno je samo `start` postaviti da pokazuje na drugi element: `start=start->next`. Budući da je memorija za svaki element alocirana dinamički, prilikom brisanja elementa treba ju osloboditi. Stoga se stara vrijednost pokazivača `start` pamti u lokalnoj varijabli `tmp` i memorija se oslobađa pozivom funkcije `free(tmp)`.

Kod brisanja elementa na proizvoljnom mjestu porebno je imati pokazivač na element koji prethodi elementu za brisanje jer njegov `next` član treba preusmjeriti na element iza onog koji brišemo. Stoga u petlji

```
    for(test=start; test != KRAJ; test=test->next)
```

provjeravamo pokazuje li `tmp=test->next` na element za brisanje. Ako da, onda se `test->next` postavlja na `test->next->next`, prvi element iza onog kojeg brišemo i time je izvršeno izbacivanje iz liste. Ostaje još samo delocirati memoriju izbrisanog elementa.

U ovom kôdu postoji problem zadnjeg elemeta. Ako je `test` zadnji element, onda je `tmp=test->next=KRAJ` pa bi `tmp->p_ime` u `strcmp` funkciji dalo grešku. Stoga tesiramo taj pokazivač i izlazimo iz petlje pomoću `break` naredbe ako smo na zadnjem elementu.

Konačno da bismo kompletirali program uvodimo funkciju `menu` koja ispisuje menu s mogućim operacijama

```
int menu(void) {
  int izbor;

  printf("\n\n\n");
```

```
  printf("\n    OPERACIJE : ");
  printf("\n    =========");
  printf("\n 1. Unos novog elemeta ");
  printf("\n 2. Ispis liste ");
  printf("\n 3. Pretrazivanje liste ");
  printf("\n 4. Brisanje iz liste ");
  printf("\n 5. Izlaz ");
  printf("\n          izbor = ");

  scanf("%d",&izbor);
  return izbor;
}
```

i dajemo glavni program:

```
int main(void) {
  int izbor;
  do {
     switch(izbor=menu()) {
      case 1:
           unos();
           break;
      case 2:
           ispis();
           break;
      case 3:
           trazi();
           break;
      case 4:
           brisi();
           break;
      case 5:
           break;
      default:
           printf("\n Pogresan izbor.");
      }
  } while(izbor != 5);
  return 0;
}
```

12.5 typedef

Pomoću ključne riječi `typedef` možemo postojećim tipovima podataka dati nova imena. Na primjer, ako neki podaci tipa `double` imaju značenje mase, onda možemo pisati

```
typedef double Masa;
```

i `Masa` će postati sinonim za `double`. Nakon toga možemo deklarirati varijable tipa `double` kao

```
Masa m1,m2,*pm1;
Masa elementi[10];
```

i tako dalje. Općenito `typedef` ima oblik

```
typedef tip_podatka  novo_ime_za_tip_podatka;
```

Vidimo da je `typedef` vrlo sličan preprocesorskoj naredbi `#define`, no kako `typedef` interpretira prevodilac moguće su složenije supstitucije. Na primjer, pokazivač na `double` možemo nazvati `Pdouble`:

```
typedef  double * Pdouble;
```

i možemo pisati

```
Pdouble px;
void f(Pdouble,Pdouble);
px= (Pdouble) malloc(100*sizeof(Pdouble));
```

Značenje simbola koji se uvodi `typedef` deklaracijom je sljedeće: Ako se ispusti ključna riječ `typedef` iz deklaracije, dobiva se obična deklaracija varijable u kojoj deklarirani simbol postaje varijabla. Deklarirani simbol predstavlja tip koji bi ta varijabla imala.

`typedef` se često koristi sa strukturama kako bi se eliminirala potreba da se pri deklaraciji varijabli navodi `struct`. Na primjer, ako imamo

```
struct tocka {
      int x;
      int y;
};
```

onda varijable tipa `tocka` deklariramo kao

```
struct tocka p1, *pp1;
```

Umjesto toga možemo iskoristiti `typdef`:

```
typedef struct  {
        int x;
        int y;
} tocka;
```

i sada možemo pisati

```
tocka p1, *pp1;
```

ili

```
struct pravokutnik {
      tocka pt1;
      tocka pt2;
};
```

Složenija je situacija ako želimo napraviti `typedef` samoreferentne strukture. Tada moramo uvesti jednu prethodnu deklaraciju tipa. Na primjer,

```
typedef struct element *Pelement;

typedef struct element {
   char ime[64];
   Pelement next;
} Element;
```

Sada možemo definirati

```
Element root;
```

i slično.

Sljedeća česta upotreba `typedef`-ova je za skračivanje kompleksnih deklaracija kao što su to pokazivači na funkcije. Na primjer, nakon

```
typedef int (*PF)(char *, char *);
```

`PF` postaje ime za pokazivač na funkciju koja uzima dva pokazivača na `char` i vraća `int`. Sada umjesto deklaracije

```
void f(double x, int (*g)(char *, char *))
{   ......
```

možemo pisati

```
void f(double x, PF g)
{ ......
```

`typedef` se koristi za parametrizaciju kôda koja omogućava veću prenosivost kôda s jednog na drugo računalo: na primjer `size_t` je izveden kao `typedef`. Drugo, uvođenje smislenih imena za standardne tipove podataka pridonosi boljoj dokumentaciji programa.

12.6 Unija

Unija kao i struktura sadrži članove različitog tipa, ali dok kod strukture svaki član ima zasebnu memorijsku lokaciju, kod unije svi članovi dijele istu memorijsku lokaciju. Memorijska će lokacija biti dovoljno široka da u nju stane najširi član unije.

Sintaksa unije analogna je sintaksi strukture. Općenito imamo

```
union  ime {
      tip_1 ime_1;
      tip_2 ime_2;
      ....   ....
      tip_n ime_n;
};
```

Varijabla tipa `ime` može se deklarirati pri deklaraciji unije ili odvojeno, kao kod struktura.

```
union  ime  x,y;
```

Na primjer,

```
union  pod {
     int  i;
     float x;
} u, *pu;
```

Komponentama varijable `u` možemo pristupiti preko operatora `.` ili operatora `->`. Tako su `u.i` i `pu->i` varijable tipa `int`, a `u.x` i `pu->x` varijable tipa `float`.

Osnovna svrha unije je ušteda memorijskog prostora. Na istoj memorijskoj lokaciji možemo čuvati varijable različitih tipova. Pri tome moramo paziti da uniji pristupamo konsistentno. Na primjer, u dijelu programa u kojem smo uniju `u` inicijalizirali kao varijablu tipa `int` moramo joj pristupiti kao varijabli tipa `int` i obratno. Ako u uniju upišemo vrijednost jednog tipa,

a onda vrijednost unije pročitamo kao varijablu drugog tipa rezultat će ovisiti o računalu i najvjerojatnije će biti besmislen.

Uniju `pod` možemo iskoristiti kako bismo ispisali binarni zapis broja s pokretnim zarezom na sljedeći način:

```
u.x=0.234375;
printf("0.234375 binarno = %x\n",u.i);
```

Ovaj se kôd oslanja na činjenicu da se (na danom računalu) za pamćenje varijabli tipa `float` i `int` koristi isti broj okteta.

Poglavlje 13

Datoteke

U ovom poglavlju opisujemo funkcije iz standardne ulazno-izlazne biblioteke koje služe za rad s datotekama. Program koji koristi ove funkcije mora uključiti datoteku zaglavlja `<stdio.h>`.

13.1 Vrste datoteka

Datoteka je imenovano područje u sekundarnoj memoriji, najčešće tvrdom disku (disketi, CD-u i slično), koje služi za smještaj podataka. Sa stanovišta operacijskog sustava datoteka je složen objekt koji se obično sastoji od više povezanih fragmenata smještenih na različitim lokacijama u sekundarnoj memoriji. Kako operacijski sustav brine o svim detaljima rada s datotekama, C promatra datoteku posve funkcionalno kao cjelinu u koju je moguće upisati niz znakova ili ga iz nje pročitati.

Preciznije, u C-u je datoteka kontinuirani niz okteta[1] (eng. *stream*) koje je moguće dohvatiti individualno. Prilikom čitanja iz datoteke niz okteta je usmjeren od datoteke prema programu; program može čitati ulazni niz znak-po-znak. Kod pisanja u datoteku niz znakova ide od programa prema datoteci. Najmanja jedinica koja može biti pročitan ili upisana je jedan znak (oktet).

Ovakva apstraktna definicija datoteke omogućava da se tastatura i ekran računala tretiraju kao datoteke. S tastature je moguće samo čitati, dok je na ekran moguće samo pisati.

[1]Niz znakova, odn. niz varijabli tipa `char`.

13.1.1 Tekstualne i binarne datoteke

ANSI standard poznaje dvije vrste datoteka: tekstualne i binarne. Razlika među njima je u načinu na koji se interpretira sadržaj datoteke.

Binarna datoteka je niz podataka tipa `char`. Budući da se svaki tip podatka u C-u (`int, double, struct ...`) može preslikati u niz vrijednosti tipa `char`, u binarnu je datoteku moguće upisati podatke onako kako su reprezentirani u računalu.

Teksualna datoteka je niz znakova podijeljenih u linije. Svaka linija sadrži nula ili više znakova iza kojih slijedi znak za prijelaz u novi red `'\n'`. Tekstualne datoteke su namijenjene pohranjivanju tekstualnih podataka.

Vidimo da pojam binarne datoteke odgovara našoj prethodnoj definiciji datoteke kao niza znakova. Tekstualna datoteka je isto tako niz znakova ali se on interpretira na malo drugačiji način. Potreba za razlikovanjem između tekstualnih i binarnih datoteka pojavila se stoga što različiti operacijski sustavi koriste različite načine označavanja kraja linije: programski jezik C označava kraj linije znakom `'\n'`.

Pod operacijskim sustavom UNIX, znak za kraj linije je `'\n'` (kao i u C-u) te stoga pod UNIXom nema razlike između binarne i tekstualne datoteke. Dvije vrste datoteka implementirane su na isti način.

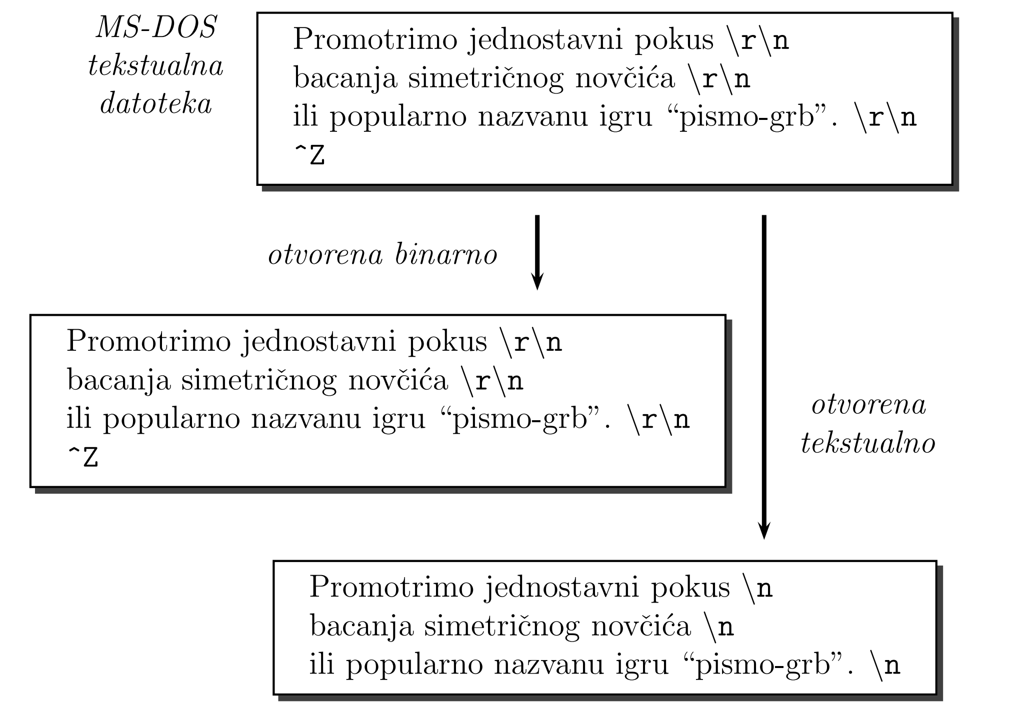
Operacijski sustav MS-DOS kraj linije označava s dva znaka `'\r''\n'`, dok npr. Macintosh koristi znak `'\r'`. Kada C otvori tekstualnu datoteku kreiranu pod MS-DOSom on će svaki par znakova `'\r' '\n'` pretvoriti u znak `'\n'`. Analogna se transformacija provodi s Macintoshovim tekstualnim datotekama. Pored toga neki MS-DOS editori označavaju kraj datoteke znakom `Ctr-Z` dok C nema poseban znak za kraj datoteke.

Treba uočiti da C omogućava da danu datoteku promatramo kao binarnu ili tekstualnu prema našim potrebama, neovisno o tome da li datoteka sadrži tekstualne podatke ili ne. Tekstualnu datoteku formiranu npr. pod MS-DOSom možemo otvoriti i kao tekstualnu i kao binarnu. Razlika je prikazana na slici 13.1.1.

13.1.2 Razine ulaza/izlaza

Postoje dvije razine ulazno-izlaznih operacija. 1) Moguće je neposredno koristiti funkcije za ulaz/izlaz koje nudi operacijski sustava; to je niska razina ulaza/izlaza. 2) Moguće je koristiti standardne ulazno/izlazne funkcije čiji su prototipovi dani u `<stdio.h>` datoteci. To je standardna visoka razina ulaza/izlaz. Funkcije iz standardne biblioteke jezika C skrivaju od korisnika jedan niz detalja vezanih uz operacijski sustav pa su stoga jednostavnije za korištenje od funkcija koje nudi operacijski sustav. Pored toga kôd koji

slika 13.1.1 Tekstualna i binarna datoteka



koristi standardnu biblioteku lakše je prenosiv na druge sustave. Nasuprot tome funkcije operacijskog sustava pružaju programeru više funkcionalnosti, no takav kôd nije prenosiv na druge operacijske sustave. Mi ćemo u ovom poglavlju opisati samo neke funkcije iz standardne biblioteke.

13.1.3 Standardne datoteke

C automatski otvara tri datoteke za korisnika. To su *standardni ulaz*, *standardni izlaz* i *standardni izlaz za greške*. Standardni ulaz je povezan s tastaturom, a standarni izlaz i izlaz za greške s ekranom. Istaknimo da C tastaturu i ekran tretira kao datoteke budući da datoteka predstavlja konceptualno naprosto ulazni ili izlazni niz oktetat. Na nekim operacijskim sustavima (Unix, DOS,..) moguće je standardni ulaz, izlaz i izlaz za greške preusmjeriti.

13.2 Otvaranje i zatvaranje datoteke

```
FILE *fopen(const char *ime, const char *tip)
int fclose(FILE *fp)
```

Komunikacija s datotekama vrši se preko spremnika (buffer) u koji se privremeno pohranjuju informacije koje se šalju u datoteku. Svrha je spremnika smanjiti komunikaciju s vanjskom memorijom (diskom) i tako povećati efikasnost ulazno-izlaznih funkcija. U datoteci `<stdio.h>` deklarirana je struktura `FILE` koja sadrži sve podatke potrebne za komunikaciju s datotekama (uključujući veličinu i položaj spremnika). Program koji želi otvariti datoteku mora prvo deklarirati pokazivač na `FILE` kao u ovom primjeru:

```
FILE *ptvar;
```

Datoteka mora biti otvorena pomoću funkcije `fopen` prije prve operacije pisanja ili čitanja. Tipično se `fopen` koristi na sljedeći način:

```
ptvar=fopen(ime,tip);
if(ptvar==NULL)
{
   printf("Poruka o gresci"):
   ......
}
```

gdje je `ime` ime datoteke koja se otvara, a tip jedan od sljedećih stringova:

tip	Značenje
"r"	Otvaranje postojeće datoteke samo za čitanje
"w"	Kreiranje nove datoteke samo za pisanje.
"a"	Otvaranje postojeće datoteke za dodavanje teksta.
"r+"	Otvaranje postojeće datoteke za čitanje i pisanje.
"w+"	Kreiranje nove datoteke za čitanje i pisanje.
"a+"	Otvaranje postojeće datoteke za čitanje i dodavanje teksta.

Pri tome treba znati da:

- Ako se postojeća datoteka otvori s "w" ili "w+" njen sadržaj će biti izbrisan i pisanje će početi od početka.
- Ako datoteka koju otvaramo s tipom "a" ili "a+" ne postoji bit će kreirana.

- Ako se datoteka otvara s tipom "a" ili "a+" novi tekst će biti dodavan na kraju datoteke ("a" dolazi od eng. *append*).

Pomoću gornjih oznaka tipova datoteka će biti otvorena u tekstualnom modu. Da bi se otvorila u binarnom modu treba treba svakom tipu dodati `b`. Time dolazimo do tipova

- `"rb", "wb", "ab"` = binarno čitanje, pisanje, dodavanje;
- `"rb+"` ili `"r+b"` = binarno čitanje/pisanje (otvaranje);
- `"wb+"` ili `"w+b"` = binarno čitanje/pisanje (kreiranje);
- `"ab+"` ili `"a+b"` = binarno dodavanje.

Funkcija `fopen` vraća pokazivač na strukturu `FILE` povezanu s datotekom ili `NULL` ako datoteka nije mogla biti otvorena.

Na kraju programa datoteka treba biti zatvorena funkcijom `fclose` koja uzima kao argument pokazivač na spremnik:

```
fclose(ptvar);
```

Na primjer, otvaranje i zatvaranje datoteke `primjer.dat` izgledalo bi ovako:

```
#include <stdio.h>
  .......
 FILE *fpt;
 if((fpt=fopen("primjer.dat","w"))==NULL)
       printf("\nGRESKA: Nije moguce otvoriti datoteku.\n");
  .....
 fclose(fpt);
```

Funkcija `fclose` vraća nulu ako je datoteka uspješno zatvorena te `EOF` u slučaju greške.

13.2.1 Standardne datoteke

stdin, stdout, stderr

U datoteci `<stdio.h>` definirani su konstantni pokazivači na `FILE` strukturu povezanu sa standardnim ulazom, izlazom i izlazom za greške. Ti pokazivači imaju imena `stdin` (standardni ulaz), `stdout` (standardni izlaz) i `stderr` (standardni izlaz za greške).

13.3 Funkcije za čitanje i pisanje

Funkcije `getchar`, `putchar`, `gets`, `puts`, `printf` i `scanf` imaju analogne verzije `getc`, `putc`, `fgets`, `fputs`, `fprintf` i `fscanf` koje rade s datotekama. Sve te funkcije kao prvi argument uzimaju pokazivač na spremnik (pokazivač na `FILE`) povezan s datotekom.

13.3.1 Čitanje i pisanje znak po znak

Funkcije

```
int getc(FILE *fp)
int fgetc(FILE *fp)
```

vraćaju sljedeći znak iz datoteke na koju pokazuje `fp`. Razlika između `getc` i `fgetc` je u tome da `getc` može biti implementirana kao makro naredba dok `fgetc` ne smije. U slučaju greške ili kraja datoteke vraća se `EOF`. To je razlog što je tip vraćene vrijednosti `int` umjesto `char`. Funkcija `getchar()` koju smo uveli u Poglavlju 5 implementira se kao `getc(stdin)`.

Program u sljedećem primjeru broji znakove u datoteci koja je navedena kao argument komandne linije.

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

int main(int argc, char *argv[])
{
    int ch,count=0;
    FILE *fpt;

    if(argc==1){  /* ime datoteke nedostaje */
       printf("\nUporaba: %s ime_datoteke\n",argv[0]);
       exit(-1);
    }
    else if((fpt=fopen(argv[1],"r"))==NULL){
        printf("Ne mogu otvoriti datoteku %s\n",argv[1]);
        exit(-1);
    }
    while((ch=fgetc(fpt))!=EOF) count++;
    fclose(fpt);
    printf("\nBroj znakova  = %d\n",count);
```

```
    return 0;
}
```

Znak pročitan sa `getc` ili `fgetc` može biti vraćen u datoteku pomoću funkcije

```
int ungetc(int c, FILE *fp);
```

Prvi argument je znak koji vraćamo. Za više detalja vidi [7].

Znak može biti upisan u datoteku pomoću funkcija

```
int putc(int c, FILE *fp)
int fputc(int c, FILE *fp)
```

Ponovo, razlika između `putc` i `fputc` je u tome što `putc` može biti implementirana kao makro naredba dok `fputc` ne smije. Funkcije vraćaju ispisani znak ili `EOF` ako je došlo do greške prilikom ispisa. Funkcija `putchar(c)` iz Poglavlja 5 definirana je kao `putc(c,stdout)`.

Funkcija koja kopira jednu datoteku u drugu može biti napisana na sljedeći način:

```
void cpy(FILE *fpulaz, FILE *fpizlaz) {
    int c;
    while((c=getc(fpulaz))!=EOF)
              putc(c,fpizlaz);
}
```

Pomoću te funkcije možemo napisati program `cat` koji uzima imena datoteka i ispisuje datoteke na standardnom izlazu u poretku u kojem su njihova imena navedena. Ukoliko se ne navede niti jedno ime, onda se standardni ulaz kopira na standardni izlaz.

```
#include <stdio.h>

void cpy(FILE *, FILE *);

int main(int argc, char *argv[]) {
    FILE *fpt;
    if(argc==1)  /* ime datoteke nedostaje */
          cpy(stdin,stdout);
    else
        while(--argc>0)
```

```
        if((fpt=fopen(*++argv,"r"))==NULL){
           printf("cat: ne mogu otvoriti datoteku %s\n",*argv);
           exit(1);
         }
         else {
           cpy(fpt,stdout);
           fclose(fpt);
        }
    return 0;
}
```

Uočimo da smo u programu posve izbjegli upotrebu brojača time što inkrementiramo i dekrementiramo `argc` i `argv`.

13.3.2 Čitanje i pisanje liniju po liniju

Funkcija koja učitava podatke iz datoteke liniju-po-liniju je

```
char *fgets(char *buf, int n, FILE *fp);
```

Prvi argument je pokazivač na dio memorije (eng. *buffer*) u koji će ulazna linija biti spremljena, a drugi je veličina memorije na koju prvi argument pokazuje. Treći argument je pokazivač na datoteku iz koje se učitava.

Funkcija će pročitati liniju uključujući i znak za prijelaz u novi red i na kraj će dodati nul znak `'\0'`. Pri tome će biti učitano najviše `n-1` znakova, uključivši `'\n'`. Ukoliko je ulazna linija dulja od toga, ostatak će biti pročitan pri sljedećem pozivu funkcije `fgets`.

Funkcija vraća `buf` ako je sve u redu ili `NULL` ako se došlo do kraja datoteke ili se pojavila greška.

Funkcija

```
char *gets(char *buf);
```

uvedena u Poglavlju 5 čita uvjek sa standardnog ulaza. Ona ne uzima veličinu *buffera* kao argument i stoga se može desiti da je ulazna linija veća od memorije koja je za nju rezervirana. Stoga je bolje umjesto `gets(buf)` koristiti `fgetf(buf,n,stdin)`. Pri tome treba uzeti u obzir razliku da `fgets` učitava i znak `'\n'`, dok `gets` znak za prijelaz u novi red učitava, ali na njegovo mjesto stavlja `'\0'`.

Funkcija za ispis podataka u datoteku, liniju-po-liniju je

```
int fputs(const char *str, FILE *fp);
```

Funkcija vraća nenegativnu vrijednost ako je ispis uspio ili `EOF` u slučaju greške.

`fputs` ispisuje znakovni niz na koji pokazuje `str` u datoteku na koju pokazuje `fp`. Zadnji nul znak neće biti ispisan i znak za prijelaz u novi red neće biti dodan.

Funkcija

```
int puts(const char *str);
```

ispisuje znakovni niz na koji pokazuje `str` na standardni izlaz. Na kraj niza ona dodaje znak za prijelaz u novi red što ju razlikuje od `fputs(str,stdout)`.

13.3.3 Prepoznavanje greške

Budući da ulazne funkcije vraćaju `EOF` i u slučaju kada je došlo do greške i u slučaju kada se naiđe na kraj datoteke postoje funkcije

```
int ferror(FILE *fp);
int feof(FILE *fp);
```

koje služe za razlikovanje između pojedinih situacija. `ferror` vraća broj različit od nule (istina) ako je došlo do greške, a nulu (laž) ako nije. Funkcija `feof` vraća broj različit od nule (istina) ako smo došli do kraja datoteke, a nulu (laž) u suprotnom.

Sljedeći program kopira standardni ulaz na standardni izlaz i detektira ulaznu grešku pomoću funkcije `ferror`.

```
#include <stdio.h>
#define MAXLINE 128

int main(void) {
    char buf[MAXLINE];

    while(fgets(buf,MAXLINE,stdin) != NULL)
     if(fputs(buf,stdout)==EOF) {
         fprintf(stderr,"\nIzlazna greska.\n");
         exit(1);
     }
    if(ferror(stdin)) {
```

```
        fprintf(stderr,"\nUlazna greska.\n");
        exit(2);
    }
  return 0;
}
```

13.3.4 Formatirani ulaz/izlaz

Za formatirani ispis u datoteku i upis iz datoteke možemo koristiti funkcije

```
int fprintf(FILE *fp, const char *format,...);
int fscanf(FILE *fp, const char *format,...);
```

One su identične funkcijama `printf` i `scanf` s jedinom razlikom da im je prvi argument pokazivač na datoteku u koju se vrši upis odn. iz koje se čita.

`fscanf` vraća broj učitanih objekata ili `EOF` ako je došlo do greške ili kraja datoteke. `fprintf` vraća broj upisanih znakova ili negativan broj u slučaju greške. Konačno, `printf(...)` je ekvivalentno s `fprintf(stdout,...)`, a `scanf(...)` je ekvivalentno s `fscanf(stdin,...)`.

13.3.5 Binarni ulaz/izlaz

Za zapisivanje struktura mogu se pokazati pogodnije sljedeće funkcije:

```
size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
```

Ove funkcije vrše binarno čitanje i pisanje. To znači da ne dolazi do konverzije podataka iz binarnog zapisa, u kojem se ono nalaze u računalu, u znakovni zapis koji se može čitati.

Funkcije `fread` čita iz datoteke na koju pokazuje `fp` `nobj` objekata dimenzije `size` i smješta ih u polje na koje pokazuje `ptr`. Vrijednost koju funkcija vraća je broj učitanih objekata. Taj broj može biti manji od `nobj` ako je došlo do greške ili kraja datoteke. Da bi se ispitao uzrok treba koristiti funkcije `ferror` i `feof`.

Funkcije `fwrite` upisuje u datoteke na koju pokazuje `fp` `nobj` objekata dimenzije `size` iz polja na koje pokazuje `ptr`. Vrijednost koju funkcija vraća je broj upisanih objekata. Taj broj može biti manji od `nobj` ako je došlo do greške.

Primjer upotrebe je zapis strukture u datoteku dan je u sljedećem programu.

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

#define SIZE 2

struct account{
       short count;
       long  total;
       char  name[64];
 };

int main(void)
{
    struct account lista[SIZE]={{3,1000L,"Ivo"},
                                {7,2000L,"Marko"}};
    FILE *fp;
    int i;

    if((fp=fopen("1.dat","wb"))==NULL){
        fprintf(stderr,"Error: nije moguce otvoriti"
                       " datoteku 1.dat.\n");
        exit(-1);
    }

    for(i=0;i<SIZE;++i){
       if(fwrite(&lista[i],sizeof(lista[i]), 1, fp) != 1){
           fprintf(stderr,"Greska pri upisu u 1.dat\n");
           exit(1);
       }
    }
    fclose(fp);
    return EXIT_SUCCESS;
}
```

Program u jednoj `for` petlji upisuje listu struktura `account` u datoteku. Sadržaj datoteke može biti izlistan sljedećom funkcijom.

```
void Print(const char *file, int n)
{
```

```
    FILE *fp;
    struct account  elm;
    int i;

    if((fp=fopen(file,"rb"))==NULL){
        fprintf(stderr,"Read, error: nije moguce otvoriti"
                " datoteku %s.\n",file);
        exit(-1);
    }

    for(i=0;i<n;++i){
       if(fread(&elm,sizeof(elm), 1, fp) != 1){
           fprintf(stderr,"Greska pri citanju iz %s\n",file);
           exit(1);
       }
       printf("%d : count = %d, total = %ld, name = %s\n",
               i, elm.count, elm.total, elm.name);
    }
    fclose(fp);
    return;
}
```

Poziv funkcije bi bio npr. oblika `Print("1.dat",2);`

Osnovni nedostatak funkcija za binarni ulaz/izlaz je zavisnost o arhitekturi računala i prevodiocu. Naime, na različitim računalima binarni zapis iste strukture podataka može biti različit što onemogućava da podaci zapisani na jednom stroju budu ispravno pročitani na drugom. Prednosti su brzina i kompaktnost zapisa.

13.3.6 Direktan pristup podacima

Pomoću do sada uvedenih funkcija podacima možemo pristupiti samo sekvencijalno. Sljedeće dvije funkcije nam omogućavaju direktan pristup.

```
int fseek(FILE *fp, long offset, int pos);
long ftell(FILE *fp);
```

Funkcija `ftell` uzima kao argument pokazivač na otvorenu datoteku i vraća trenutni položaj unutar datoteke. Kod binarnih datoteka to je broj znakova koji prethode trenutnom položaju unutar datoteke, dok je kod tekstualne datoteke ta veličina ovisna o implementaciji. U svakom slučaju izlaz iz

funkcije `ftell` mora dati dobar drugi argument za funkciju `fseek`. U slučaju greške, `ftell` će vratiti vrijednost `-1L`.

Funkcija `fseek` uzima pokazivač na otvorenu datoteku, te dva parametra koji specificiraju položaj u datoteci. Trceći parametar `pos` indicira od kog mjesta se mjeri `offset` (odmak). U datoteci `stdio.h` su definirane tri simboličke konstante: `SEEK_SET`, `SEEK_CUR` i `SEEK_END`, koje imaju sljedeće značenje:

pos	Nova pozicija u datoteci
SEEK_SET	`offset` znakova od početka datoteke
SEEK_CUR	`offset` znakova od trenutne pozicije u datoteci
SEEK_END	`offset` znakova od kraja datoteke

Na primjer,

Poziv	Značenje
`fseek(fp, 0L, SEEK_SET)`	Idi na početak datoteke
`fseek(fp, 0L, SEEK_END)`	Idi na kraj datoteke
`fseek(fp, 0L, SEEK_CUR)`	Ostani na tekućoj poziciji
`fseek(fp, ftell_pos, SEEK_SET)`	Idi na poziciju je dao prethodni poziv `ftell()` funkcije

Standard postavlja neka ograničenja na funkciju `fseek`. Tako za binarne datoteke `SEEK_END` nije dobro definiran, dok za tekstualne datoreke jedino su pozivi iz prethodne tabele dobro definirani.

Napišimo sada funkciju koja će u datoteci potražiti određeni `account` i povećati njegovo polje `total` za 100, ako je ono manje od 5000.

```
void Povecaj(const char *file, int n)
{
    FILE *fp;
    struct account tmp;
    long           pos;
    int            size=sizeof(struct account);

    if((fp=fopen(file,"r+b"))==NULL){
        fprintf(stderr,"Povecaj, error: nije moguce otvoriti"
                " datoteku %s.\n",file);
        exit(-1);
    }

    pos = n*size;
    fseek(fp, pos, SEEK_SET);
```

```
    if(fread(&tmp, size, 1, fp) != 1){
           fprintf(stderr,"Greska pri citanju.\n");
           exit(EXIT_FAILURE);
    }
    if(tmp.total < 5000L) {
        tmp.total += 100;
        fseek(fp, -size, SEEK_CUR);
        if(fwrite(&tmp, size, 1, fp) != 1){
           fprintf(stderr,"Greska pri upisu.\n");
           exit(1);
        }
    }

    fclose(fp);
    return;
}
```

Funkcija mijenja sadržaj `n+1`-og zapisa (brojanje zapisa počinje od nule). Prvo se otvara datoteka za čitanje i upisivanje (`r+b`), a zatim se pokazivač pozicionira na početak `n+1`-og zapisa (`fseek(fp, pos, SEEK_SET);`). Zapis se pročita i ako je `total` manji od 5000 povečava se za 100. Nakon što je zapis promijenjen, treba ga upisati nazad u datoteke. U tu svrhu se prvo vraćamo na početak `n+1`-og zapisa (`fseek(fp, -size, SEEK_CUR);`) i onda vršimo upis.

Poglavlje 14

Operacije nad bitovima

14.1 Operatori

Programski jezik C ima jedan broj operatora čije je djelovanje definirano na bitovima. Takvi se operatori mogu primijeniti na cjelobrojne tipove podataka `char`, `short`, `int` i `long`. To su sljedeći operatori

Operator	Značenje
`&`	logičko I bit-po-bit
`|`	logičko ILI bit-po-bit
`^`	ekskluzivno logičko ILI bit-po-bit
`<<`	lijevi pomak
`>>`	desni pomak
$\sim$	1-komplement

Prve tri operacije `&`, `|` i `^` uzimaju dva operanda i vrše operacije na bitovima koji se nalaze na odgovarajućim mjestima. Uspoređuju se bitovi na najmanje značajnom mjestu u oba operanda, zatim na sljedećem najmanje značajnom mjestu itd. Definicije operacija dane su u sljedećoj tabeli:

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

Operacije ilustriramo s nekoliko promjera. Logičko I:

```
    a   =  0100 0111 0101 0111
    b   =  1101 0100 1010 1001
 -------------------------------
 a & b  =  0100 0100 0000 0001
```

Logičko ILI:

```
    a   =  0100 0111 0101 0111
    b   =  1101 0100 1010 1001
  a | b =  1101 0111 1111 1111
```

Ekskluzivno logičko ILI:

```
    a   =  0100 0111 0101 0111
    b   =  1101 0100 1010 1001
  a ^ b =  1001 0011 1111 1110
```

Logički operatori najčešće služe *maskiranju* pojedinih bitova u operandu. Maskiranje je transformacija binarnog zapisa u varijabli na određen način. Na primjer, pretpostavimo da želimo šest najmanje značajnih bitova iz varijable `a` kopirati u varijablu `b`, a sve ostale bitove varijable `b` staviti na nulu. To možemo postići pomoću logičkog I operatora. Pretpostavimo da su varijable tipa `int` i da je `int` kodiran u 16 bitova. Tada prvo definiramo *masku*, konstantu koja ima sljedeći raspored bitova:

```
mask = 0000 0000 0011 1111
```

Odmah se vidi da je `mask=0x3f`. Operacija koju trebamo napraviti je

```
         a  =  0100 0111 0101 0111
      mask  =  0000 0000 0011 1111
  a & mask  =  0000 0000 0001 0111
```

i stoga imamo `b=a & 0x3f`. Uočimo da je maska neovisna o broju bitova koji se koristi za tip `int`. Naravno, da smo htjeli izdvojiti dio najznačjnijih bitova (lijevo pozicioniranih) morali bismo precizno znati duljinu tipa `int`. Da bismo izdvojili najznačajnijih šest bitova, a ostale postavili na nulu morali bismo izvršiti operaciju `b=a & 0xfc00` (provjerite), ali taj način je zavisan o širini tipa `int`.

Logičko I također služi postavljanjem na nulu određenih bitova. Ako u nekoj varijabli `a` želimo postaviti na nulu neki bit, dovoljno je napraviti logičko I s konstantom koja na traženom mjestu ima nulu, a na svim ostalim jedinice. Na primjer,

```
         a  =  0100 0111 0101 0111
      mask  =  1111 1101 1111 1111
  a & mask  =  0100 0101 0101 0111
```

Ovdje smo deseti bit postavili na nulu.

Logičko ILI na sličan način može poslužiti maskiranju bitova. Njega ćemo iskoristiti kada želimo iz jedne varijable u drugu kopirati dio bitova, a sve ostale postaviti na jedan. Na primjer, ako želimo izdvojiti šest najmanje značajnih bitova trebamo napraviti

```
         a  =  0100 0111 0101 0111
      mask  =  1111 1111 1100 0000
  a | mask  =  1111 1111 1101 0111
```

i stoga imamo `b=a | 0xffa`. Ova operacija ovisi o duljini tipa `int` (odn. onog tipa koji se koristi) no to se može izbjeći pomoću 1-komplementa.

Unarni operator 1-komplement (∼) djeluje tako da jedinice pretvara u nule u nule u jedinice:

```
~0101 1110 0001 0101 = 1010 0001 1110 1010
```

odnosno, ∼`0x5e15=0xa1ea`.

U prethodnom primjeru bismo stoga pisali `b=a | ~0x3f`.

Logičko ILI evidentno možemo koristiti za stavljanje pojedinih bitova na jedan.

Ekskluzivno ILI možemo koristiti za postavljanje određenih bitova na 1 ako su bili 0 i obratno. Na primjer,

`a`	=	0100 0111 0101 0111
`mask`	=	0000 0000 0011 0000
`a ^ mask`	=	0100 0111 0110 0111

Vidimo da je rezultat da su bitovi 5 i 6 promijenili svoju vrijednost. Ako ponovimo operaciju

`a`	=	0100 0111 0110 0111
`mask`	=	0000 0000 0011 0000
`a ^ mask`	=	0100 0111 0101 0111

dolazimo do polazne vrijednosti.

Operatori pomaka pomiću binarni zapis broja nalijevo ili nadesno. Operator pomaka nalijevo `<<` djeluje na sljedeći način: prvi operand mora biti cjelobrojni tip nad kojim se operacija vrši, a drugi operand je broj bitova za koji treba izvršiti pomak (`unsigned int`). Drugi operand ne smije premašiti broj bitova u lijevom operandu. Na primjer, `b=a<<6` ima sljedeći efekt:

```
     a = 0110 0000 1010 1100
a << 6 = 0010 1011 0000 0000
```

Sve se bitovi pomiću za 6 mjesta ulijevo. Pri tome se 6 najznačajnijih bitova gubi, a sa desne strane se mjesta popunjavaju nulama.

Pomak nadesno `>>` također djeluje nad prvim operandom dok je drugi operand broj bitova za koji treba izvršiti pomak. Bitovi na desnoj strani (najmanje značajni) pri tome se gube, dok se na desnoj strani uvode novi bitovi. Pri tome, ako je varijabla na kojoj se pomak vrši tipa `unsigned`, onda se na lijevoj strani uvode nule. Na primjer

```
     a = 0110 0000 1010 1100
a >> 6 = 0000 0001 1000 0010
```

Ako je varjabla `a` cjelobrojni tip s predznakom onda rezultat može ovisiti o implementaciji. većina prevodioca će na lijevoj strani uvesti bit predznaka broja. To znači da će za negativan broj ispražnjeni bitovi na lijevoj strani biti popunjavani jedinicama, a za pozitivan broj nulama. S druge strane, neki prevodioci će uvijek popunjavati ispražnjena mjesta nulama.

Logički operatori uvedeni u ovom poglavlju formiraju operatore pridruživanaja

```
&=   ^=   |=   <<=   >>=
```

Neka je `a=0x6db7`. Tada je

izraz	ekvivalentan izraz	vrijednost
`a &= 0x7f`	`a = a & 0x7f`	`0x37`
`a ^= 0x7f`	`a = a ^ 0x7f`	`0x6dc8`
`a |= 0x7f`	`a = a | 0x7f`	`0x6dff`
`a <<= 5`	`a = a << 5`	`0xb6e0`
`a >>= 5`	`a = a >> 5`	`0x36d`

Na kraju pokažimo program koji ispisuje binarni zapis cijelog broja tipa `int`.

```
#include <stdio.h>

int main(void) {
    int a,b,i,nbits;
    unsigned mask;

    nbits=8*sizeof(int);  /* duljina tipa int */
    mask=0x1 << (nbits-1); /* 1 na najznacajnijem mjestu*/

    printf("\nUnesite cijeli broj: "); scanf("%d",&a);
    for(i=1;i<=nbits;++i) {
       b=(a & mask) ? 1 : 0;
       printf("%d",b);
       if(i % 4 == 0) printf(" ");
       mask >>= 1;
    }
    printf("\n");
    return 0;
}
```

Program prvo odredi duljinu tipa `int` pomoću `sizeof` operatora i zatim inicijalizira `unsigned` varijablu `mask` tako da bit 1 stavi na najznačajnije mjesto, a sve ostale bitove stavi na nulu.

Operacija `(a & mask) ? 1 : 0` dat će 1 ako je na maskiranom mjestu bit jednak 1, odnosno nulu ako je maskirani bit 0. U naredbi `mask >>= 1;` maskirni bit pomićemo s najznačajnijeg mjesta prema najmanje značajnom i tako ispitujemo sve bitove u varijabli.

14.2 Polja bitova

Polja bitova nam omogućuju rad s pojedinim bitovima unutar jedne kompljutorske riječi. Deklaracija polja bitova posve je slična deklaraciji strukture:

```
struct ime {
           clan 1;
           clan 2;
           .....
           clan n;
};
```

s tom razlikom da članovi strukture imaju drugačije značenje nego u običnoj strukturi. Svaki član polja bitova predstavlja jedno polje bitova unutar kompjutorske riječi. Sintaksa je takva da iz imena varijable dolazi dvotočka i broj bitova koji član zauzima. Na primjer,

```
struct primjer {
           unsigned a : 1;
           unsigned b : 3;
           unsigned c : 2;
           unsigned d : 1;
};
struct primjer v;
```

Prva deklaracija definira strukturu razbijenu u četiri polja bitova, `a`, `b`, `c` i `d`. Ta polja imaju duljinu 1, 3, 2 i 1 bit. Prema tome zauzimaju 7 bitova. Poredak tih bitova unutar jedne kompjutorske riječi ovisi o implementaciji.

Pojedine članove polja bitova možemo dohvatiti istom sintaksom kao i kod struktura, dakle `v.a`, `v.b` itd.

Ako broj bitova deklariran u polju bitova nadmašuje jednu kompjutorsku riječ, za pamćenje polja bit će upotrebljeno više kompjutorskih riječi.

Jednostavan program koji upotrebljava polje bitova je sljedeći:

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

```
int main(void)
{
    static struct{
        unsigned a : 5;
        unsigned b : 5;
        unsigned c : 5;
        unsigned d : 5;
        } v={1,2,3,4};

   printf("v.a=%d, v.b=%d, v.c=%d, v.d=%d\n",v.a,v.b,v.c,v.d);
   printf("v treba %d okteta\n", sizeof(v));
   return 0;
   }
```

Poredak polja unutar riječi može se kotrolirati pomoću neimenovanih članova unutar polja kao u primjeru

```
 struct  {
           unsigned a : 5;
           unsigned b : 5;
           unsigned   : 5;
           unsigned c : 5;
 };
 struct primjer v;
```

Neimenovani član čija je širina deklarirana kao 0 bitova tjera prevodilac da sljedeće polje smjesti u sljedeću računalnu riječ. Na primjer,

```
#include <stdio.h>

int main(void)
{
    static struct{
        unsigned a : 5;
        unsigned b : 5;
        unsigned   : 0;
        unsigned c : 5;
        } v={1,2,3};

   printf("v.a=%d, v.b=%d, v.c=%d\n",v.a,v.b,v.c);
   printf("v treba %d okteta\n", sizeof(v));
   return 0;
}
```

Polja bitova mogu se koristiti kao i druge strukture s određenim ograničenjima. Članovi mogu biti tipa `unsigned` i tipa `int`. Adresni operator se ne može primijeniti na člana polja bitova niti se član može doseći pomoću pokazivača.

Dodatak A

A.1 ANSI zaglavlja

<assert.h>	<inttypes.h>	<signal.h>	<stdlib.h>
<complex.h>	<iso646.h>	<stdarg.h>	<string.h>
<ctype.h>	<limits.h>	<stdbool.h>	<tgmath.h>
<errno.h>	<locale.h>	<stddef.h>	<time.h>
<fenv.h>	<math.h>	<stdint.h>	<wchar.h>
<float.h>	<setjmp.h>	<stdio.h>	<wctype.h>

A.2 ASCII znakovi

```
Char  Dec  | Char  Dec  | Char   Hex | Char Dec
-----------------------------------------------
(nul)   0  | (sp)   32  | @      64  | `      96
(soh)   1  | !      33  | A      65  | a      97
(stx)   2  | "      34  | B      66  | b      98
(etx)   3  | #      35  | C      67  | c      99
(eot)   4  | $      36  | D      68  | d     100
(enq)   5  | %      37  | E      69  | e     101
(ack)   6  | &      38  | F      70  | f     102
(bel)   7  | '      39  | G      71  | g     103
(bs)    8  | (      40  | H      72  | h     104
(ht)    9  | )      41  | I      73  | i     105
(nl)   10  | *      42  | J      74  | j     106
(vt)   11  | +      43  | K      75  | k     107
(np)   12  | ,      44  | L      76  | l     108
(cr)   13  | -      45  | M      77  | m     109
(so)   14  | .      46  | N      78  | n     110
(si)   15  | /      47  | O      79  | o     111
(dle)  16  | 0      48  | P      80  | p     112
(dc1)  17  | 1      49  | Q      81  | q     113
(dc2)  18  | 2      50  | R      82  | r     114
(dc3)  19  | 3      51  | S      83  | s     115
(dc4)  20  | 4      52  | T      84  | t     116
(nak)  21  | 5      53  | U      85  | u     117
(syn)  22  | 6      54  | V      86  | v     118
(etb)  23  | 7      55  | W      87  | w     119
(can)  24  | 8      56  | X      88  | x     120
(em)   25  | 9      57  | Y      89  | y     121
(sub)  26  | :      58  | Z      90  | z     122
(esc)  27  | ;      59  | [      91  | {     123
(fs)   28  | <      60  | \      92  | |     124
(gs)   29  | =      61  | ]      93  | }     125
(rs)   30  | >      62  | ^      94  | ~     126
(us)   31  | ?      63  | _      95  | (del) 127
```

A.3 Matematičke funkcije

```
double cos(double x) -- cos(x)
double sin(double x) -- sin(x)
double tan(double x) -- tg(x)

double acos(double x) -- arccos(x)
double asin(double x) -- arcsin(x)
double atan(double x) -- arctg(x)
double atan2(double y, double x) -- arctg(y/x)

double cosh(double x) -- ch(x)
double sinh(double x) -- sh(x)
double tanh(double x) -- th(x)

double exp(double x) --  e^x
double pow (double x, double y) -- x^y.

double log(double x) -- ln(x).
double log10 (double x ) --  log(x)

double fabs (double x ) -- |x|
labs(long n) --  |n|

double sqrt(double x) -- korijen

double ceil(double x)  -- Najmanji cijeli broj veci od x
double floor(double x) -- Najveci cijeli broj manji od x
```

Bibliografija

[1] C-faq, http://www.eskimo.com/ scs/C-faq/top.html.

[2] B. S. Gottfried, *Theory and Problems of Programming with C*, Schaum's outline series, McGraw-Hill, 1996.

[3] S. P. Harbison III, G. L. Steele Jr., *C, A Reference Manual, Fifth Edition*, Prentice Hall, 2002.

[4] B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, 2nd Edition, 1988.

[5] W. H. Murray, III, C. H. Pappas, *8036/8026 Assembly Language Programming*, Osborne McGraw-Hill, Berkeley, 1986.

[6] S. Prata, *C Primer Plus*, 4th edition, SAMS, 2002.

[7] W. R. Stevens, *Advanced Programming in the UNIX Enviroment*, Addison Wesley, 1992.