
Iztok Fajfar in Jernej Olenšek
OSVOJIMO
C89, C99
in primeri
programiranja
mikrokrmilnikov
Univerza v Ljubljani
Fakulteta za elektrotehniko
Založba FE

OSVOJIMO C

OSVOJIMO C

C89, C99 in primeri programiranja mikrokrmilnikov

Iztok Fajfar in Jernej Olenšek

Univerza *v Ljubljani*
Fakulteta *za elektrotehniko*

Založnik: Založba Fakultete za elektrotehniko, Ljubljana
Izdajatelj: Univerza v Ljubljani, Fakulteta za elektrotehniko, Ljubljana
Urednik: prof. dr. Sašo Tomažič
Recenzija: prof. dr. Tadej Tuma, izr. prof. dr. Marko Jankovec
Jezikovni pregled: Jana Kolarič
Ilustracija in oblikovanje naslovnice: Ciril Horjak, Omar Horjak

1. elektronska izdaja
Način dostopa (url): http://fajfar.eu/Osvojimo-C.pdf

CIP – Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

COBISS.SI-ID=29091587
ISBN 978-961-243-409-0 (pdf)

Fibonacciju,
hvala za ničlo.

PRAZNA STRAN

VSEBINA

ZAHVALA

Od velikega poka je minilo precej časa. Vseh dogodkov, ljudi, živali, rastlin in celo virusov, ki so od takrat privedli do nastanka tega učbenika, je preveč, da bi lahko omenili vsakega posebej. Zato zahvala predvsem vam, dragi bralci, ki boste učbenik vzeli v roke in tako osmislili njegov obstoj.

PRAZNA STRAN

UVOD

Učinkovitost, prenosljivost, prilagodljivost in obsežna knjižnica standardnih funkcij so samo nekatere odlike programskega jezika C. Dobra novica za inženirja elektrotehnike je, da je C nizkonivojski jezik, ki omogoča enostaven dostop do strojnih konceptov (kot na primer strojni registri in pomnilniški naslovi), ki jih skušajo drugi programski jeziki skriti. Poleg tega jezik C vključuje operacije, ki so tesno povezane s strojnimi ukazi računalnika, zato lahko z njim pišemo hitre in učinkovite programe. Jezik C je sestavni del operacijskega sistema UNIX (in seveda tudi njegove popularne različice Linux). Nekatera orodja v UNIXu celo predpostavljajo znanje jezika C. Operacijski sistemi, ki jih poganjajo vgrajeni sistemi, v veliki večini temeljijo na jedru Linux. Te sisteme najdemo praktično povsod, od zabavne elektronike, bele tehnike, omrežne opreme, industrijske avtomatike, do medicinske opreme in celo navigacijskih naprav v letalstvu in vesoljskih plovilih. Operacijski sistem za pametne telefone Android prav tako temelji na Linuxovemu jedru. Leta 2015 je bil svetovni delež teh sistemov med pametnimi telefoni več kot 80-odstoten.

Jezik C je začel nastajati leta 1969 kot stranski produkt operacijskega sistema UNIX, ki sta ga v Bellovih laboratorijih razvijala Ken Thompson in Dennis Ritchie s sodelavci. Thompson je lastnoročno napisal izvorno različico UNIXa za računalnik DEC PDP-7, zgodnji mikroračunalnik z vsega 18,4 KB(!) glavnega pomnilnika. Nič nenavadnega ni, da je jezik, ki se je obdržal tako dolgo, vplival na razvoj vrste drugih jezikov (npr. C++, Java, JavaScript, PHP, C#, Perl, Objective-C). Tudi če se učimo katerega od teh sodobnejših jezikov, ki temeljijo na jeziku C, nam znanje jezika C precej pomaga pri globljem razumevanju njihovih funkcionalnosti. Na primer, brez trdnega razumevanja principa delovanja in uporabe kazalcev se je praktično nemogoče naučiti jezika C++.

Mislim, da smo našteli dovolj razlogov, zakaj se učiti jezik C. Ker smo časovno omejeni, se bomo učili le pomembnejših in pogosteje uporabljenih elementov jezika, v glavnem

tistih, ki jih predpisuje prvi uradno zapisani standard ANSI C (znan tudi kot C89 oz. ANSI X3.159-1989 ali C90 oz. ISO/IEC 9899:1990). Še en razlog, zakaj se bomo držali tega standarda, je ta, da je to trenutno še vedno najbolje podprt standard za jezik C.

V drugem delu učbenika obravnavamo osnovne tehnike programiranja mikrokrmilnikov. Spet zaradi časovnih omejitev smo se odločili, da v ta namen uporabimo razvojno okolje Arduino, ki ni namenjeno profesionalni rabi. Zato pa je to okolje enostavno in široko dostopno. Hkrati omogoča, da z njim na strnjen in slikovit način predstavimo in preizkusimo vsa osnovna načela, ki jih srečujemo pri programiranju mikrokrmilnikov in profesionalnih mikrokrmilniških sistemov.

Učbenik je namenjen predvsem študentom elektrotehnike, ki so poslušali katerega od predmetov *Osnove programiranja* ali *Programiranje 1*, in pa vsem, ki so jim domači vsaj osnovni pojmi računalniškega programiranja. Učbenik lahko uporabite kot samostojno gradivo, verjetno pa se bo za večino najbolje obnesel kot dopolnitev predavanj in vaj pri predmetih *Programiranje mikrokrmilnikov* in *Programiranje 2*.

I. DEL

UVOD V PROGRAMSKI JEZIK C

PRAZNA STRAN

1. POGLAVJE

PRVI KORAKI

V tem poglavju bomo spoznali nekaj osnovnih pojmov, ki jih moramo poznati za uspešno pisanje in oživljanje celo najenostavnejših cejevskih programov.

1.1 Pisanje in zagon programa

Začnimo s preprostim primerom programa, ki izpiše dobrodošlico:

```
#include <stdio.h>
int main(void) {
  printf("Dobrodošli nazaj!");
  return 0;
}
```

Program napišemo s katerimkoli urejevalnikom besedila[1] (angl. text editor) in ga shranimo v datoteko s poljubnim imenom, ki pa se mora običajno končati s podaljškom `.c`. Koda, kakršno vidimo zgoraj, je razumljiva človeku, ne moremo pa je ***zagnati*** (angl. run) na računalniku. Takšni kodi pravimo ***izvorna koda*** (angl. source code), iz katere v postopku ***gradnje*** (angl. build) dobimo tako imenovano ***izvršilno kodo*** (angl. executable). Izvršilno kodo lahko potem zaženemo na računalniku.

[1] Za razliko od ***oblikovalnika besedil*** (angl. word processor), ki je namenjen oblikovanju besedil z uporabo različnih pisav in omogoča vstavljanje raznih objektov, kot so slike ali tabele, se ***urejevalnik besedil*** (angl. text editor) uporablja zgolj za pisanje besedilnih (tekstovnih) datotek.

Ne glede na to, kako obsežna je izvorna koda, je postopek gradnje izvršilne kode vedno sestavljen iz treh korakov:

- ***Predprocesiranje*** (angl. preprocessing) V tem koraku se izvršijo tako imenovane ***predprocesorske direktive*** oziroma preprosto ***direktive*** (angl. directives). To so ukazi, ki se začnejo z znakom lojtra (#, angl. hash). Lojtri včasih rečemo tudi ograjica ali lestev. Delo, ki ga opravi predprocesor, spominja na običajno urejanje besedila: predprocesor lahko na določena mesta v datoteki bodisi doda nove dele kode bodisi spremeni obstoječo kodo. Na primer, direktiva `#include` (slov. vključi) doda v datoteko deklaracije[2] funkcij, ki jih kličemo iz te datoteke. Ker v gornjem programu uporabljamo funkcijo `printf`, moramo v datoteko vključiti standardno knjižnico <stdio.h>, ki vsebuje to funkcijo.
- ***Prevajanje*** (angl. compiling) Program z vsemi spremembami, ki jih je vnesel predprocesor, se v tem koraku prevede iz oblike, razumljive človeku, v niz strojnih ukazov, ki jih razume računalnik. Tako prevedeni kodi pravimo ***objektna koda*** (angl. object code). Vendar program v tem trenutku še ni popolnoma pripravljen za zagon.
- ***Povezovanje*** (angl. linking) V tem zadnjem koraku se objektna koda, ki jo je proizvedel prevajalnik, poveže z vso dodatno kodo, ki je potrebna za pravilno delovanje programa. V primeru gornjega programa ta dodatna koda vključuje definicijo knjižnične funkcije `printf`, ki jo uporabljamo v programu.

Na srečo je proces gradnje izvršilnega programa navadno avtomatiziran. Obstaja vrsta brezplačnih in plačljivih programov za prevajanje in povezovanje kode, ki jih dobimo v najrazličnejših oblikah: od preprostih vrstično usmerjenih (angl. command line) programov do kompletov orodij, združenih v tako imenovanih integriranih razvojnih okoljih (angl. integrated development environment – IDE). Integrirano razvojno okolje je program, ki pod eno streho združuje komponente za urejanje izvorne kode, njeno prevajanje, povezovanje in izvajanje ter celo razhroščevanje.

Za svoje delo bomo v tem učbeniku uporabljali brezplačno odprtokodno okolje za različne platforme z imenom *Code::Blocks* (www.codeblocks.org). Če pa želite, lahko uporabite katerokoli drugo orodje, saj vsa obstoječa orodja podpirajo standard ANSI, ki se ga bomo držali v tem učbeniku.

1.2 Osnovna zgradba programa

Vzemimo zdaj nekoliko daljši, vendar še vedno preprost program:

```
#include <math.h>
#include <stdio.h>
int main(void) {
  float x, y;
  x = 10;
  scanf("%f", &y);
  x = pow(x, y);
  printf("%f", x);
  return 0;
}
```

[2] ***Deklaracija*** funkcije določa zgolj način klicanja funkcije, ne vsebuje pa kode, kot jo vsebuje njena ***definicija***. Razlika med obema pojmoma je pojasnjena v razdelku 5.1.

Verjetno bo marsikdo hitro ugotovil, da program izračuna in prikaže potenco (angl. power) števila deset, pri čemer potenco vnese uporabnik sam. Čeprav je program preprost, pa vsebuje kar nekaj sestavnih delov, ki si jih bomo v nadaljevanju podrobneje ogledali.

Vključevanje knjižnic

Iz razlogov, ki jih bomo spoznali kasneje, potrebuje prevajalnik v vsaki datoteki, v kateri se pojavi klic določene funkcije, vsaj njeno deklaracijo. Deklaracije funkcij so shranjene v tako imenovanih ***zglavnih datotekah*** (angl. header files), ki se navadno končajo s podaljškom .h. V gornjem programu smo klicali standardni knjižnični funkciji `pow` (angl. power) in `printf`, katerih deklaraciji se nahajata v zglavnih datotekah[3] <math.h> (deklaracija funkcije `pow`) in <stdio.h> (deklaracija funkcije `printf`). Ti dve datoteki moramo zato vključiti na začetku datoteke z našim programom, kar naredimo s pomočjo direktive `#include`.

Omenili smo že, da pred začetkom prevajanja predprocesor s pomočjo predprocesorskih direktiv (tj. ukazov, ki se začnejo z lojtro) na ustrezen način predela izvorno kodo. Direktiva `#include`, ki smo jo uporabili v gornjem programu, v času predprocesiranja vsebino celotne podane datoteke preprosto prilepi na mesto, kjer se direktiva nahaja.

Pomembno je tudi to, da na koncu vrstice, ki vključuje zglavno datoteko, ne pišemo podpičja. Podpičja v jeziku C so sicer obvezna, kje pa jih pišemo, bomo natančno izvedeli kasneje.

Podatkovni tipi in deklaracije spremenljivk

C je ***statično tipiziran*** (angl. statically typed) jezik. To pomeni, da moramo vsako spremenljivko, ki jo uporabimo v programu, pred prvo uporabo na ustrezen način napovedati oziroma ***deklarirati*** (angl. declare). Spremenljivko deklariramo tako, da pred njeno ime zapišemo oznako ***podatkovnega tipa*** (angl. data type), ki mu bo ta spremenljivka pripadala. Tip spremenljivke določa, kakšne podatke lahko hranimo v njej. Če želimo deklarirati več spremenljivk istega tipa, lahko njihova imena ločimo z vejicami in pred njih zapišemo oznako želenega tipa. Na primer, v gornjem programu smo deklarirali ***realni*** spremenljivki `x` in `y` na naslednji način:

```
float x, y;
```

Ker so realne vrednosti v Ceju shranjene v zapisu s ***plavajočo vejico*** (angl. floating point), smo uporabili oznako podatkovnega tipa `float`. Tako deklarirani spremenljivki `x` in `y` lahko hranita kakršnokoli realno vrednost[4]. Če želimo vrednost spremenljivke omejiti na celoštevilsko, potem moramo takšno spremenljivko deklarirati kot celoštevilsko (angl. integer), za kar potrebujemo oznako `int`. Na primer, celoštevilsko spremenljivko `a` deklariramo takole:

```
int a;
```

[3] Kadar zglavne datoteke vsebujejo deklaracije funkcij iz tako imenovane ***standardne knjižnice*** (angl. standard library), njihova imena običajno pišemo v par kotnih oklepajev (tj. med simbola < in >). S tema oklepajema označimo, da gre za standardno knjižnico, vendar oklepaja nista del dejanskega imena datoteke.

[4] Kasneje, ko bomo govorili o zgradbi zapisa s plavajočo vejico, bomo spoznali, da ne ravno kakršnokoli. Ta hip pa nas omejitve tega zapisa še ne zanimajo.

Inicializacija spremenljivke

Spremenljivko `x` smo v našem zadnjem programu tudi ***inicializirali*** (angl. initialize), kar pomeni, da smo ji določili začetno vrednost (tj. vrednost 10). Vendar ni nujno, da spremenljivko inicializiramo ločeno od njene deklaracije. Spremenljivki lahko že ob deklaraciji dodelimo začetno vrednost. Takrat pravimo, da smo spremenljivko ***definirali*** (angl. define):

```
float x = 10, y;
```

V cejevskem žargonu se vrednost 10 v gornjem primeru imenuje ***inicializator*** (angl. initializer).

Vsaka spremenljivka, ki jo želimo v deklaraciji inicializirati, potrebuje svoj inicializator. V naslednjem primeru nastavimo na vrednost 13 samo spremenljivko `z`, spremenljivki `x` in `y` pa ostaneta neinicializirani:

```
int x, y, z = 13;
```

Identifikatorji

Ko pišemo program, moramo za vsako spremenljivko, funkcijo, konstanto ali drugo enoto izbrati ime. Takšnim imenom pravimo ***identifikatorji*** (angl. identifier). V Ceju lahko identifikator vsebuje zgolj:

- velike in male črke angleške abecede,
- podčrtaj (tj. znak _, angl. underscore) in
- desetiške cifre.

Velja dodatna omejitev, da se identifikator ne sme začeti z desetiško cifro.

To so primeri petih veljavnih identifikatorjev:

```
stevecImpulzov     levi_rob     naslov2     _izbira      _42
```

Naslednja dva identifikatorja pa nista veljavna:

```
2Pac     levi-rob
```

Prvo ime se namreč začne z desetiško cifro, drugo pa vsebuje pomišljaj.

Jezik C je občutljiv na velike in male črke (angl. case-sensitive), kar pomeni, da loči velike in male črke v imenih. Na primer, naslednji identifikatorji označujejo štiri različne enote:

```
Zaloga     zaloga     ZALOGA     zaLoga
```

Razumljivo je tudi, da za identifikatorje ne moremo izbirati imen, ki so že uporabljena v standardnih knjižnicah (npr. `printf` ali `scanf`). Prav tako ne moremo uporabljati ***ključnih besed*** (angl. keywords), ki imajo v jeziku poseben pomen. Spoznali smo že ključni besedi `int` in `float`, v naslednji tabeli pa so zbrane še nekatere druge ključne besede (vse iz standarda C89 in dve iz standarda C99):

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

Standardi od C99 naprej poznajo še nekaj ključnih besed, ki pa se vse začnejo s podčrtajem in z veliko črko, tako da je majhna verjetnost, da bi kdo želel izbrati takšno ime za identifikator v svojem programu.

Funkcija `main`

Cejevski program je lahko sestavljen iz poljubnega števila funkcij, vendar mora vsak program vsebovati funkcijo `main`. Ta funkcija se kliče avtomatično vsakokrat, kadar zaženemo program. V praksi to pomeni, da se bo naš program začel izvajati s prvo vrstico kode v funkciji `main`. V zadnjem primeru na strani 4 funkcija `main` nima nobenih parametrov, kar moramo v Ceju označiti s ključno besedo `void` (slov. prazen). Ker je C statično tipiziran jezik, moramo na začetku definicije funkcije (pred njenim imenom) označiti tudi ***tip funkcije***. To je v resnici tip vrednosti, ki jo funkcija vrača (angl. return). Standard določa, da mora funkcija `main` vrniti celoštevilsko vrednost (`int`). Ker funkcije `main` v svojem programu nikoli ne kličemo, se zastavi vprašanje: komu ta funkcija sploh vrača vrednost? Program navadno izvajamo na določenem operacijskem sistemu, ki lahko na podlagi vrnjene vrednosti ugotovi, kako se je program končal. Program se lahko konča bodisi normalno (brez posebnosti) bodisi kot posledica kakšne napake. Vrednost nič običajno pomeni, da se je program končal normalno.

Prikaz izhodnih podatkov

Funkcija `printf` je namenjena pisanju podatkov na standardni izhod. Standardni izhod oziroma `stdout` (angl. standard output) je abstrakcija izhodnega toka podatkov, ki ga ponuja operacijski sistem. V praksi to pomeni, da funkcije `printf` v resnici ne zanima, kam podatke izpisuje, čeprav predstavlja `stdout` v veliki večini primerov ***računalniško konzolo*** oziroma ***terminal*** (angl. computer console, terminal).

S funkcijo `printf` lahko prikazujemo tako običajno besedilo kot tudi vrednosti določenih spremenljivk ali izrazov. V programu na strani 3 smo na primer izpisali dobrodošlico z naslednjim klicem:

```
printf("Dobrodošli nazaj!");
```

Funkcija `printf` je ustvarjena tako, da prikaže vsebino, ki jo podamo kot argument v dvojnih navednicah. Kakršnemukoli nizu znakov v dvojnih navednicah pravimo ***znakovni niz*** (angl. string), znakovnemu nizu, ki ga uporabimo kot prvi argument funkcije `printf`, pa pravimo ***formatni niz*** (angl. format string). Formatni niz se imenuje zato, ker lahko poleg navadnega besedila vsebuje tudi kakšne druge simbole, ki tako ali drugače oblikujejo (formatirajo) prikaz. Na primer, v programu na strani 4 smo izpisali vrednost spremenljivke `x` s pomočjo naslednjega klica:

```
printf("%f", x);
```

Kombinacija znakov `%f` predstavlja tako imenovano ***formatno določilo*** (angl. format specifier). S formatnim določilom znotraj formatnega niza funkciji povemo, da naj na tem mestu izpiše vrednost spremenljivke, ki jo podamo kot ločen argument. Črka f v formatnem določilu pove, da je spremenljivka, katere vrednost želimo izpisati, realnega tipa (float). Če bi želeli izpisati vrednost celoštevilske spremenljivke, potem bi v formatnem določilu uporabili črko d.

V formatnem nizu lahko kombiniramo poljubno besedilo s poljubno veliko formatnimi določili. Paziti moramo le, da za vsako formatno določilo v formatnem nizu funkciji podamo dodaten argument, ki je spremenljivka ali izraz z vrednostjo ustreznega tipa. Na primer, če bi hoteli izpisati vsoto realnih spremenljivk `a` in `b`, bi lahko to storili takole:

```
printf("%f+%f=%f", a, b, a + b);
```

V gornjem formatnem nizu opazimo tri formatna določila: na mestih prvih dveh se izpišeta vrednosti spremenljivk `a` in `b`, na mestu tretjega pa se izpiše vrednost izraza `a + b`. Znaka + in = v formatnem nizu nimata posebnega pomena, zato se izpišeta kot običajno besedilo. Na primer, če bi imela spremenljivka `a` vrednost 1,3, spremenljivka `b` pa vrednost 2,9, potem bi gornji klic povzročil naslednji izpis:

```
1.300000+2.900000=4.200000
```

Oblika prikaza realnih števil s šestimi decimalnimi mesti je privzeta oblika prikaza. Če nam privzeta oblika izpisa ne ustreza, jo lahko spremenimo tako, da med znak `%` in črko f (ali d) vstavimo ustrezen ***modifikator*** (angl. modifier). Nekaj primerov je prikazanih v naslednji tabeli:

Formatno določilo	Opis
`%5d`	Izpiše celoštevilsko vrednost, pri čemer uporabi vsaj pet mest. Če je vrednost krajša od petih cifer, doda spredaj potrebno število presledkov (tj. vrednost poravna desno).
`%-5d`	Izpiše celoštevilsko vrednost, pri čemer uporabi vsaj pet mest. Če je vrednost krajša od petih cifer, doda zadaj potrebno število presledkov (tj. vrednost poravna levo).
`%05d`	Izpiše celoštevilsko vrednost, pri čemer uporabi vsaj pet mest. Če je vrednost krajša od petih cifer, doda spredaj potrebno število ničel.
`%.4f`	Izpiše realno vrednost, zaokroženo na štiri decimalna mesta.
`%11.4f`	Izpiše realno vrednost, zaokroženo na štiri decimalna mesta. Za izpis porabi vsaj 11 mest (vključno z decimalno piko). Če je izpis krajši od 11 znakov, doda spredaj potrebno število presledkov.
`%011.4f`	Izpiše realno vrednost, zaokroženo na štiri decimalna mesta. Za izpis porabi vsaj 11 mest (vključno z decimalno piko). Če je izpis krajši od 11 znakov, doda spredaj potrebno število ničel.

Poleg običajnega besedila in formatnih določil lahko v funkciji `printf` uporabimo tudi tako imenovane ***ubežne sekvence*** (angl. escape sequences). Ubežne sekvence uporabljamo za oblikovanje izpisa (kot na primer premik v novo vrstico) ali za izpis znakov,

ki imajo sicer kakšno posebno vlogo (kot na primer ", ki konča znakovni niz). Ubežne sekvence se vedno začnejo z ***vzvratno poševnico*** (angl. backslash). Nekaj ubežnih sekvenc prikazuje naslednja tabela:

Ubežna sekvenca	Opis
\b	Pojdi en znak nazaj (angl. backspace).
\n	Pojdi v novo vrstico (angl. newline).
\r	Pojdi na začetek vrstice (angl. return).
\t	Vstavi tabulator.
\"	Prikaži dvojno navednico.
\\	Prikaži vzvratno poševnico.

Ubežne sekvence se uporabljajo tudi v drugih znakovnih nizih, in ne le v formatnem nizu funkcije `printf`.

Prav poseben primer predstavlja znak za odstotek (`%`), ki v formatnem nizu pomeni začetek formatnega določila. Zanj ne obstaja ubežna sekvenca v obliki `\%`. Če želimo ta znak prikazati s funkcijo `printf`, moramo uporabiti dva znaka za odstotek. Na primer, besedilo *100 %* prikažemo s klicem `printf("100 %%")`.

Naloga 1.1 Za vajo napišite program, ki ustvari naslednji izpis:

```
Za pomik v novo vrstico potrebujemo ubežno sekvenco \n.
Brez težav znamo izpisati tudi znaka " in %.
```

Naloga 1.2 Vzemimo, da imamo v programu deklarirano realno spremenljivko, katere vrednost je 0,99911. Za vajo napišite program, ki ustvari naslednji izpis (druga in tretja vrstica predstavljata izpis vrednosti prej omenjene spremenljivke):

```
0123456789
    0.9991
01.00
```

Opomba: Za drugo in tretjo vrstico izpisa je dovoljeno uporabiti samo po eno formatno določilo z ustreznim modifikatorjem in ubežno sekvenco za pomik v novo vrstico. Prvo vrstico izpišite kot navadno besedilo.

Branje vhodnih podatkov

Funkcija `scanf` je namenjena branju vhodnih podatkov s standardnega vhoda. Standardni vhod oziroma `stdin` (angl. standard input) je abstrakcija vhodnega toka podatkov, ki ga ponuja operacijski sistem. Podobno, kot smo videli v primeru funkcije `printf`, tudi funkcije `scanf` v resnici ne zanima, s kakšne naprave podatki prihajajo. Res pa je, da predstavlja `stdin` v veliki večini primerov vnos podatkov s tipkovnice.

V našem zadnjem programu na strani 4 smo funkcijo `scanf` klicali takole:

```
scanf("%f", &y);
```

Podobno kot pri funkciji `printf` je tudi pri funkciji `scanf` prvi argument formatni niz. V primeru branja podatkov formatni niz pove, kakšno obliko podatkov lahko pričakujemo na vhodu, kar je določeno z ustrezno kombinacijo formatnih določil in včasih tudi

drugih znakov. Vsako formatno določilo pričakuje na vhodu en podatek ustreznega tipa. Na primer, formatno določilo `%f` pričakuje realno, formatno določilo `%d` pa celoštevilsko vrednost. Za vsako formatno določilo v formatnem nizu moramo funkciji `scanf` dodati še en argument. Ta argument je sestavljen iz operatorja `&` (njegov pomen bomo spoznali kasneje) in imena spremenljivke, v katero želimo shraniti prebrano vrednost. Gornji klic tako pričakuje eno realno vrednost, ki se bo shranila v spremenljivko `y`.

Z naslednjim klicem lahko hkrati preberemo eno celoštevilsko in eno realno vrednost ter ju shranimo v spremenljivki `x1` (celoštevilsko) in `x2` (realno):

```
scanf("%d%f", &x1, &x2);
```

Kadar vnašamo več vrednosti, jih med seboj ločimo tako, de med njih postavimo poljubno število presledkov, tabulatorjev ali prehodov v novo vrstico (angl. enter, return). Drugih znakov vmes ne sme biti. Če bi radi uporabniku omogočili, da pri vnosu loči podatke na drugačen način, lahko ta način določimo v formatnem nizu. Na primer, če želimo, da so vneseni podatki ločeni z vejicami, potem moramo vejice vstaviti na ustrezno mesto v formatni niz:

```
scanf("%d ,%f", &x1, &x2);
```

Zdaj moramo pri vnosu podatkov med celoštevilsko in realno vrednost nujno vpisati vejico (dodamo lahko še poljubno število presledkov in praznih vrstic), sicer funkcija podatkov ne bo mogla pravilno prebrati. Pomembno je, da pred vejico v formatnem določilu pišemo presledek. Če tega ne storimo, potem mora tudi pri vnosu podatkov vejica stati takoj za prvim vnesenim podatkom (tj. brez vmesnega presledka).

Težava funkcije `scanf` je ta, da v primeru nepričakovanega vhodnega podatka tega podatka nikoli ne odstrani iz vhodnega toka. V praksi to pomeni, da v primeru napačnega vnosa nikakor ne moremo nadaljevati z branjem novih podatkov. Naslednji primer kaže, kako ta problem rešimo. Primer je preprost: na vhodu pričakuje pet celoštevilskih vrednosti in vsako uspešno prebrano vrednost takoj izpiše na izhod:

```
#include <stdio.h>
int main(void) {
  int i, x;
  for (i = 0; i < 5; i = i + 1) {
    if (scanf("%d", &x) == 1) {
      printf("%d\n", x);
    }
    else {
      printf("Napačen vnos\n");
      while (getchar() != '\n') {}
    }
  }
  return 0;
}
```

Ko se klic funkcije `scanf` konča, funkcija vrne število uspešno prebranih podatkov (vsako formatno določilo ustreza enemu podatku). V gornjem primeru mora funkcija ob uspešno prebranem podatku vrniti vrednost ena. Če se to zgodi (tj. pogoj `scanf("%d", &x) == 1` je izpolnjen), potem se v prvem delu stavka `if` izpiše prebrana vrednost. V nasprotnem primeru se izpiše sporočilo *Napačen vnos*. Poglejmo, kaj se zgodi, če namesto celega števila vnesemo na primer realno vrednost 3,14159. Funkcija `scanf` zaradi formatnega določila `%d` pričakuje celo število in ga tudi prebere, vendar samo do tja, do koder to zna: funkcija prebere vrednost 3, decimalna pika pa ni več del celoštevilske vrednosti,

zato ta pika ostane v vhodnem toku podatkov. Ob naslednjem klicu funkcija spet skuša prebrati decimalno piko, ki je ostala v vhodnem toku in je prva na vrsti za branje. Ker decimalna pika ni celo število, se funkcija konča in vrne vrednost nič, ker ni mogla prebrati zahtevanega podatka. Zato se zdaj izpiše opozorilo *Napačen vnos*. Takoj zatem ponavljanje klica funkcije `getchar`[5] iz vhodnega toka drugega za drugim prebere vse preostale podatke, dokler ne prebere znaka `\n`. To je znak, ki ga pošlje tipka za potrditev (angl. enter oz. return), ki jo vedno pritisnemo na koncu vnosa podatkov. Če napačnih podatkov ne bi odstranili na ta način, potem bi se do konca izvajanja zanke `for` izpisovalo sporočilo *Napačen vnos*, ker bi funkcija `scanf` neprestano skušala (neuspešno) prebrati decimalno piko. Nobene možnosti ne bi imeli, da bi ta napačni vnos popravili.

V zadnjem programu je videti, da moramo vnesti vsako celo število posebej (za vsakim številom moramo pritisniti tipko za potrditev). Če to dejansko storimo, potem program vsako vneseno vrednost takoj tudi izpiše. Lahko pa vnesemo tudi vseh pet celoštevilskih vrednosti naenkrat. To storimo tako, da med njimi pišemo presledke in šele na koncu pritisnemo tipko za potrditev. Videti je, da program zdaj prebere in izpiše vseh pet števil hkrati. V resnici program še vedno prebere in izpiše vsako število posebej. Da lahko to razumemo, moramo vedeti, da je vhodni tok podatkov v operacijskem sistemu izveden kot vrsta oziroma medpomnilnik tipa FIFO. Funkcija `scanf` iz tega medpomnilnika z vsakim formatnim določilom prebere podatek, ki je na vrsti, podatki s tipkovnice pa se zapišejo vanj šele ob pritisku na tipko za potrditev.

Opombe

Kot vsak programski jezik tudi C omogoča vnašanje ***opomb*** (angl. comments). C pozna dve vrsti opomb. Enovrstične opombe se začnejo z dvojno poševnico (`//`) in se končajo na koncu trenutne vrstice. Večvrstične opombe se začnejo s kombinacijo znakov poševnica in zvezdica (`/*`) in končajo s kombinacijo znakov zvezdica in poševnica (`*/`). Prevajalnik preprosto ignorira vse, kar se v programu pojavi kot opomba. Opombe uporabljamo za pojasnjevanje določenih delov kode, da se kasneje v kodi lažje znajdemo. Mehanizem opomb je uporaben tudi za začasno izklapljanje delov kode med preizkušanjem programa.

Konstante

Mnogo programov je takšnih, da v njih nastopajo določene ***konstantne vrednosti*** (angl. constants). Na primer, vrednost 3 v programu[6] lahko pomeni največje dovoljeno število vnosov napačne kode PIN. Če se konstantna vrednost pojavi neposredno v delu kode, kjer preverjamo pravilnost vnesene kode PIN, potem je takšna koda manj razumljiva, kot če uporabimo za konstantno vrednost določeno ime (npr. `maksPoskusovPIN`).

Poleg tega lahko predstavlja ista vrednost v istem programu tudi nekaj čisto drugega. Na primer, 3 je lahko tudi zaporedna številka priključne sponke, na katero je vezan zvočnik. To pomeni, da nastopa vrednost 3 tudi v vseh delih programa, ki upravljajo z zvočnikom. Predstavljajmo si, da se podjetje odloči posodobiti strojno opremo, na kateri iz določenih razlogov zvočnik prestavijo s sponke 3 na sponko 4. Zdaj moramo posodobiti tudi programsko opremo. Če nismo uporabili različnih imen za dve konstanti s sicer enako vred-

[5] Funkcija `getchar` prebere in iz vhodnega toka odstrani znak, ki je prvi na vrsti za branje.

[6] Dobesedno zapisani konstantni vrednosti pravimo včasih tudi ***dobesedna navedba*** (angl. literal). Primeri dobesednih navedb so: `3.1416`, `42`, `'\n'` in `"Konec"`.

nostjo 3, kako zdaj vedeti, katere vrednosti spremeniti s 3 na 4? Preproste funkcije urejevalnika *Najdi in zamenjaj* ne moremo uporabiti.

Konstante lahko poimenujemo na dva načina. Prvi je z uporabo predprocesorskega ukaza `#define`:

```
#define MAKS_POSKUSOV_PIN 3
#define SPONKA_ZVOCNIK 3
```

Taki definiciji pravimo ***makro*** (angl. macro). Makro deluje tako, da pred začetkom prevajanja v datoteki poišče vse identifikatorje, ki so enaki identifikatorju, ki sledi ukazu `#define`. Najdene identifikatorje potem nadomesti s koščkom kode, ki ga najdemo na koncu definicije makra. Na primer, prvi od gornjih makrov nadomesti vse identifikatorje `MAKS_POSKUSOV_PIN`, ki jih najde v datoteki, s konstantno vrednostjo 3. Isto naredi z identifikatorji `SPONKA_ZVOCNIK`. Bodite pozorni, da na koncu definicije makra ne sme biti podpičja. Opazimo tudi, da je identifikator v makru zapisan z velikimi črkami. Nenapisano pravilo pravi, da definicije makrov vedno pišemo z velikimi črkami, da se ločijo od ostalih identifikatorjev, ki se v Ceju navadno pišejo z malimi črkami (razen mogoče velikih začetnic, če uporabljamo kamelje Črke (angl. camelCase)).

Slabost makrov je, da njihovih vrednosti ne moremo videti v razhroščevalniku. Zato za definicije konstant včasih raje uporabimo rezervirano besedo `const`, ki jo postavimo pred običajno definicijo spremenljivke:

```
const int maksPoskusovPIN = 3;
const int sponkaZvocnik = 3;
```

Zaradi uporabe besede `const` prevajalnik nikjer v programu ne bo dopustil spreminjanja vrednosti identifikatorjev `maksPoskusovPIN` in `sponkaZvocnik`. Dodatna prednost takšne definicije konstante je, da je podvržena cejevskim mehanizmom statičnega preverjanja ustreznosti podatkovnih tipov.

Naloga 1.3 Za vajo napišite program, ki prebere dolžino v palcih (angl. inches, okrajšano in) in izpiše isto dolžino v centimetrih. V programu uporabite konstanto `palecCentimetrov`, ki jo nastavite na 2,54. Toliko centimetrov namreč znaša en palec.

Primer delovanja programa (mastni tisk predstavlja vnos s tipkovnice):

```
Vnesi dolžino (in): 12.4
Vnesena dolžina znaša 31.50 cm.
```

1.3 Naloge

Naloga 1.4 Napišite program, ki prebere neko realno vrednost in jo izpiše na načine, ki jih prikazujeta naslednja dva primera (mastni tisk predstavlja vnos s tipkovnice):

```
Vnesi realno vrednost: 1234.56789
    1234.57
001235-----1234.568
"12345.679"

Vnesi realno vrednost: 1.5
       1.50
000002-----1.500
"15.000"
```

Namig: Zaokroževanje na celo število lahko dosežete tako, da zahtevate izpis realnega števila z nič decimalnimi mesti.

Naloga 1.5 Napišite program, ki prebere celoštevilski znesek v evrih in izpiše najmanjše število posameznih bankovcev in kovancev, s katerimi lahko ta znesek plačamo.

Primer delovanja programa (mastni tisk predstavlja vnos s tipkovnice):

```
Vnesi celoštevilski znesek v evrih: 469

Izplačilo:
-------------------
Nominala   Št. enot
   (EUR)
-------------------
     200   2
      50   1
      10   1
       5   1
       2   2
```

Naloga 1.6 V skladu z mednarodnim standardom za zapis datuma in časa ISO 8601 se datum zapisuje v obliki LLLL-MM-DD, kjer L, M in D predstavljajo posamezne števke za leto, mesec in dan. Napišite program, ki prebere datum v obliki zapisa ISO 8601 in ga prikaže v obliki, ki ga določa slovenski pravopis. V slovenskem pravopisu se zapisuje najprej dan, potem mesec in na koncu leto. Za številko dneva in meseca morata biti pika in presledek, pred njima pa ne sme biti vodilne ničle (v primeru, da sta dan ali mesec enomestni števili).

Primer delovanja programa (mastni tisk predstavlja vnos s tipkovnice):

```
Vnesi datum v zapisu ISO 8601: 2007-05-30
Pretvorjen zapis: 30. 5. 2007
```

Naloga 1.7 Napišite program, ki izpiše tabelo sinusov in kosinusov kotov od nič do 180 stopinj v koraku po 30 stopinj. V tabeli naj bodo koti izpisani tako v stopinjah kot tudi v radianih.

Primer delovanja programa (bodite pozorni na poravnavo števil in število decimalnih mest):

```
x (st)  x (rad)  sin(x)  cos(x)
------------------------------
     0    0.000    0.00    1.00
    30    0.524    0.50    0.87
    60    1.047    0.87    0.50
    90    1.571    1.00    0.00
   120    2.094    0.87   -0.50
   150    2.618    0.50   -0.87
   180    3.142    0.00   -1.00
```

Pomoč: Funkciji `sin` in `cos` se nahajata v standardni knjižnici <math.h>. Argumenti kotnih funkcij v Ceju morajo biti podani v radianih.

Naloga 1.8 Napišite program, ki prebere dva ulomka in ju sešteje.

Primer delovanja programa (mastni tisk predstavlja vnos s tipkovnice):

```
Prvi ulomek: 1/8
Drugi ulomek: 5/6
Vsota: 46/48
```

Dopolnite program tako, da bo mogoče vnesti oba ulomka v eni vrstici ter da bo program izpisal vsoto v okrajšani obliki. Na primer:

```
Vnos: 1/8 + 5/6
Vsota: 23/24
Vnos: 17/6 + 5/15
Vsota: 19/6
```

Naloga 1.9 Napišite program, ki prebere pet celoštevilskih vrednosti in prikaže njihovo vsoto. V primeru, da program med petimi vnesenimi števili naleti na napako, naj prosi uporabnika za vnovični vnos petih celih števil. To naj ponavlja toliko časa, dokler mu končno ne uspe brez napake prebrati pet celih števil.

Primer delovanja programa (mastni tisk predstavlja vnos s tipkovnice):

```
Vnesi pet celih števil: 2 54 1.2 8
Napačen vnos.
Vnesi pet celih števil: b 66 3 1 9
Napačen vnos.
Vnesi pet celih števil: 4 7 22 16 9
Vsota: 58
```

2. POGLAVJE

OPERATORJI IN IZRAZI

Osnovni gradnik cejevskega programa je ***izraz*** (angl. expression). Izraz ni nič drugega kot formula, ki izračuna oziroma ***vrne*** (angl. return) določeno vrednost. Tako kot v matematiki tudi v jeziku C izraze gradimo iz konstant, spremenljivk in (klicev) funkcij, ki jih povezujemo z ustreznimi operatorji. C pozna ogromno različnih operatorjev, ki jih bomo spoznavali postopoma. V tem poglavju bomo srečali le nekaj najosnovnejših operatorjev skupaj z osnovnimi pravili računanja vrednosti izrazov.

2.1 Aritmetični operatorji

Kot večina programskih jezikov pozna C naslednje aritmetične operatorje:

Operator	Opis
+ (unarni)	predznak
− (unarni)	predznak
+ (binarni)	seštevanje
− (binarni)	odštevanje
*	množenje
/	deljenje
%	ostanek pri deljenju

Prva dva operatorja delujeta kot predznak in zato sprejmeta en sam operand. Takim operatorjem pravimo ***unarni*** (angl. unary) operatorji:

```
-1
+2
```

Negativen predznak spremeni predznak vrednosti na svoji desni, pozitiven predznak pa ne naredi ničesar. Uporabi se lahko zgolj za to, da poudarimo, da je konstanta pozitivna. Predznak lahko uporabimo tudi pred spremenljivko:

```
-y
```

Pomembno je, da tak izraz samo ***vrne*** vrednost spremenljivke `y`, pomnoženo z -1. Sama vrednost spremenljivke `y` ostane nespremenjena.

Preostalih pet aritmetičnih operatorjev iz gornje tabele spada v skupino tako imenovanih binarnih (angl. binary) operatorjev. To pomeni, da sprejmejo dva operanda. Tudi za teh pet operatorjev velja, da zgolj vrnejo rezultat operacije, na vrednosti operandov pa ne vplivajo. Medtem ko delujejo operatorji seštevanja, odštevanja in množenja tako, kot smo navajeni iz matematike, pa je treba o operatorjih deljenja in ostanka pri deljenju povedati še nekaj besed.

Kadar sta pri deljenju tako števec kot tudi imenovalec celi števili, bo rezultat prav tako celo število. Pri tem je pomembno to, da rezultat ni zaokrožen na najbližjo celoštevilsko vrednost, temveč se del za decimalno piko ***odreže*** (angl. truncate). Tako je na primer vrednost izraza `8 / 9` enaka nič. Po drugi strani operator ostanka pri deljenju (`%`) ***vedno*** zahteva, da sta oba operanda ***celi števili***. V nasprotnem primeru se program ne bo prevedel. Za oba operatorja (deljenje in ostanek pri deljenju) velja, da je njuno ***obnašanje nedoločeno*** (angl. undefined behavior), kadar je njun desni operand enak nič. O nedoločenem obnašanju govorimo takrat, ko lahko dobimo v programu, prevedenem z istim prevajalnikom, v enakih situacijah različne (nepredvidljive) rezultate[1].

V primeru, ko je kateri od operandov operacije deljenja dveh celih števil ali ostanka pri deljenju negativno število, rezultat operacije v standardu C89 ni enoumno določen. Pri deljenju je lahko rezultat zaokrožen navzgor ali navzdol. Kako se bo rezultat obnašal, je ***odvisno od izvedbe prevajalnika*** (angl. implementation-defined behavior). Za razliko od nedoločenega obnašanja dobimo v tem primeru v programu, prevedenem z istim prevajalnikom, vedno iste rezultate. Šele z drugim prevajalnikom se lahko začne program obnašati drugače. Na primer, vrednost izraza `-5 / 4` je lahko bodisi -1 bodisi -2. Prav tako je lahko vrednost izraza `-5 % 4` bodisi -1 bodisi 3, odvisno od uporabljenega prevajalnika. Če želimo, da bo naš program prenosljiv, je najbolje, da se takšnim primerom izognemo oziroma zanje napišemo svojo funkcijo. Standard C99 sicer enoumno določa[2] obnašanje obeh omenjenih primerov, vendar se na to ni priporočljivo zanašati. Še vedno lahko namreč naletimo na prevajalnike, ki tega standarda ne podpirajo.

Prednost in asociativnost operatorjev

Kadar v istem izrazu nastopa več operatorjev, je pomembno, v kakšnem vrstnem redu se ti operatorji izvajajo. Vrstni red izvajanja operatorjev urejata pojma ***prednosti*** (angl.

[1] Glej tudi dodatek C, kjer so zbrani nekateri pogosti primeri, katerih obnašanja standard ne določa povsem natančno ali jih sploh ne določa.

[2] Rezultat celoštevilskega deljenja je dejanska vrednost kvocienta brez decimalnega dela, čemur pravimo tudi ***zaokroževanje proti ničli*** (angl. truncation toward zero). Poleg tega mora biti vrednost izraza `(a / b) * b + a % b` enaka `a`.

precedence) in ***asociativnosti*** (angl. associativity). V tabeli v dodatku A so zbrani vsi operatorji, ki jih bomo obravnavali v tem učbeniku. Tisti, ki so postavljeni višje v tabeli, imajo višjo prednost. To pomeni, da se bodo izvedli pred operatorji, ki so nižje v tabeli in imajo zato nižjo prednost. Na primer, ker ima množenje prednost pred seštevanjem, se bo v izrazu:

```
x + 6 * y
```

najprej izračunala vrednost izraza `6 * y`, temu pa se bo na koncu prištela vrednost spremenljivke `x`. Če imamo v istem izrazu več operatorjev z isto prednostjo, potem se operacije izvajajo glede na njihovo asociativnost: z leve proti desni (leva asociativnost) oziroma z desne proti levi (desna asociativnost). Ker imajo operatorji seštevanja in odštevanja isto prednost in levo asociativnost, računamo vrednost naslednjega izraza od leve proti desni:

```
10 - 2 + 3 - 4 - 5
```

Izraz zato vrne vrednost dve.

Prednosti in asociativnosti osnovnih aritmetičnih operatorjev poznamo že iz nižjih razredov osnovne šole in nam verjetno ne bodo delale težav. Vendar jezik C pozna več kot 40 različnih operatorjev, katerih prednosti in asociativnosti se ni smiselno učiti na pamet. Če smo glede tega v dvomih, je najbolje, da uporabimo oklepaje. Z oklepaji pa lahko prednost in asociativnost celo spreminjamo. Na primer, v naslednjem izrazu s parom oklepajev dosežemo, da se najprej izračuna vrednost izraza `4 - 5`:

```
10 - 2 + 3 - (4 - 5)
```

Zato zdaj izraz vrne vrednost 12 (tj. $10 - 2 + 3 - (-1)$).

2.2 Priredilni operator

Omenili smo že, da aritmetični operatorji ne spreminjajo vrednosti svojih operandov, temveč zgolj vračajo izračunane vrednosti. Pravimo, da takšni operatorji nimajo ***stranskih učinkov*** (angl. side effects). Lahko si predstavljamo, da operacija brez učinka v programu ne igra prav nobene vloge. Zato operatorje, ki nimajo stranskih učinkov, navadno kombiniramo z drugimi operatorji, ki takšne učinke imajo. Na primer, izračunano vrednost izraza lahko shranimo v določeno spremenljivko. To lahko storimo s pomočjo ***priredilnega operatorja*** (angl. assignment operator), katerega stranski učinek je spremenjena vrednost spremenljivke na njegovi levi. Na primer, v naslednjem izrazu bo spremenljivka `x` dobila vrednost osem:

```
x = 10 - 2
```

Pomembno je, da ima priredilni operator skoraj najnižjo prednost (nižjo prednost ima le še vejični operator, ki ga bomo še spoznali). Zaradi tega lahko vedno računamo na to, da se bo najprej izračunal celoten izraz na desni strani priredilnega operatorja, čisto na koncu pa se bo izračunana vrednost prenesla v spremenljivko na levi strani priredilnega operatorja.

V jeziku C je priredilni operator operator v pravem pomenu besede. To pomeni, da mora priredilni izraz vrniti vrednost. Vrednost, ki jo priredilni izraz vrne, je enaka vrednosti, ki se prenese v spremenljivko na levi strani priredilnega operatorja. To lahko preverimo v naslednjem kosu programa:

```
int x;
printf("%d\n", x = 10);
printf("%d\n", x);
```

Prvi klic funkcije `printf` izpiše vrednost izraza `x = 10`, ki je enaka deset. V tem istem izrazu postane `x` enak deset, zato izpiše tudi naslednji `printf` vrednost deset.

Ker priredilni izraz vrne vrednost in ker ima priredilni operator desno asociativnost, lahko priredilne operatorje verižimo na naslednji način:

```
x = y = z = 0
```

S tem na eleganten način nastavimo več spremenljivk hkrati na isto vrednost. V gornjem izrazu se najprej izračuna vrednost izraza `z = 0`, ki vrne vrednost nič. (Poleg tega se kot stranski učinek izraza nastavi vrednost spremenljivke `z` na nič.) Vrnjena vrednost prvega izraza se potem prenese prek drugega priredilnega operatorja v spremenljivko `y` in ta drugi izraz zato spet vrne vrednost nič. Na koncu se ta vrednost zapiše še v spremenljivko `x`.

V skladu s pravili jezika C lahko katerikoli izraz, ki vrne vrednost ustreznega tipa, uporabimo, kjerkoli se takšna vrednost pričakuje. Tako lahko na primer zapišemo izraz, ki hkrati nastavi spremenljivko `x` na vrednost spremenljivke `y`, spremenljivki `z` pa priredi vrednost, ki je desetkrat večja od vrednosti `y`:

```
z = 10 * (x = y)
```

Vendar se je takšnim in podobnim »akrobacijam« najbolje izogibati. Če drugega ne, je takšna koda težko razumljiva in zlahka vodi do napak, ki jih je izjemno težko odkriti.

Leve vrednosti

Za razliko od aritmetičnih (in večine ostalih) operatorjev, ki lahko za operande sprejmejo tako spremenljivke kot tudi konstante, je lahko na levi strani priredilnega operatorja zgolj tako imenovana ***leva vrednost*** (angl. left value, okrajšano lvalue). Leva vrednost pomeni kakršenkoli objekt, shranjen v pomnilniku (od tega, kar smo spoznali doslej, je to lahko le spremenljivka). Leva vrednost ne more biti na primer niti konstanta niti izraz ali klic funkcije, ki vrneta običajno številsko vrednost. Naslednji trije izrazi se ne bodo prevedli, ker na levi strani priredilnih operatorjev niso leve vrednosti:

```
int a, b, c, x, y;
2 = x       /* Napaka: priredilni operator zahteva levo vrednost.  */
a + b = c   /* Napaka: priredilni operator zahteva levo vrednost.  */
sin(x) = y  /* Napaka: priredilni operator zahteva levo vrednost.  */
```

Ker ima priredilni operator nižjo prednost od seštevanja, se na levi strani priredilnega operatorja v drugem od gornjih treh izrazov nahaja izraz `a + b`, kar seveda ni leva vrednost.

Sestavljeni priredilni operatorji

Veliko je primerov, v katerih želimo novo vrednost spremenljivke izračunati glede na njeno staro vrednost. Na primer, če želimo pomnožiti vrednost spremenljivke `x` s šest, naredimo to takole:

```
x = x * 6
```

V takšnih primerih lahko operator množenja združimo s priredilnim operatorjem na naslednji način:

```
x *= 6  /* Isto kot x = x * 6.  */
```

Operator $*=$ v gornjem izrazu je sestavljen iz dveh operatorjev (in tudi izvede dve operaciji: množenje in prirejanje), zato mu pravimo ***sestavljen priredilni operator*** (angl. compound assignment operator).

Na enak način lahko priredilni operator združimo z vsemi binarnimi aritmetičnimi operatorji:

```
x += 2   /* x-u prišteje 2.                        */
x -= 7   /* Od x-a odšteje 7.                      */
x *= 5   /* x pomnoži s 5.                         */
x /= 6   /* x deli s 6.                            */
x %= 12  /* V x shrani ostanek deljenja x / 12.    */
```

Operatorja povečanja in zmanjšanja

Dve najpogostejši operaciji nad spremenljivko sta ***povečanje*** (angl. increment) in ***zmanjšanje*** (angl. decrement) njene vrednosti za ena:

```
x = x + 1
x = x - 1
```

Z uporabo sestavljenih operatorjev $+=$ in $-=$ lahko gornja dva izraza zapišemo krajše:

```
x += 1
x -= 1
```

Obstaja pa še krajši način. Za povečanje vrednosti spremenljivke za ena lahko uporabimo operator povečanja ($++$), za zmanjšanje vrednosti spremenljivke za ena pa lahko uporabimo operator zmanjšanja ($--$):

```
x++  /* Poveča vrednost x-a za 1.   */
x--  /* Zmanjša vrednost x-a za 1.  */
```

Pri operatorjih povečanja in zmanjšanja moramo biti previdni, kadar nastopata v izrazu, v katerem so še drugi operatorji. Pomembno je, ali operator povečanja ali zmanjšanja stoji ***pred spremenljivko*** (angl. prefix) ali ***za spremenljivko*** (angl. postfix). V obeh primerih se vrednost spremenljivke spremeni za ena, vendar vrne izraz vsakokrat drugačno vrednost: če je operator za spremenljivko, potem vrne izraz staro vrednost spremenljivke, sicer vrne njeno novo vrednost. Poglejmo si primer (predpostavimo, da ima spremenljivka `x` v vsakem od naslednjih izrazov na začetku vrednost pet):

```
x++  /* x postane 6, izraz vrne 5.  */
++x  /* x postane 6, izraz vrne 6.  */
x--  /* x postane 4, izraz vrne 5.  */
--x  /* x postane 4, izraz vrne 4.  */
```

Razširimo gornje izraze v naslednje priredilne izraze:

```
y = x++  /* x postane 6, y postane 5.  */
y = ++x  /* x postane 6, y postane 6.  */
y = x--  /* x postane 4, y postane 5.  */
y = --x  /* x postane 4, y postane 4.  */
```

Spremenljivka `y` dobi seveda vsakokrat vrednost, ki jo vrne izraz na desni strani priredilnega operatorja.

Tako kot za priredilne operatorje velja tudi za operatorje povečanja in zmanjšanja, da ni pametno pretiravati z njihovim kombiniranjem z ostalimi operatorji. Hitro lahko pridemo do težko razumljivih izrazov, ki postanejo vir napak, ki jih je težko odkriti.

Vrstni red računanja delnih izrazov

Kadar je izraz sestavljen iz več enakovrednih izrazov, vrstni red njihovega računanja ni predpisan. Na primer, v izrazu:

```
(x + y) * (q + w)
```

ne moremo vedeti, kateri od izrazov `x + y` ali `q + w` se bo prvi izračunal. V gornjem primeru to sicer ni težava, ker vrstni red računanja ne vpliva na končni izid. Naslednji primer pa je glede tega drugačen:

```
2 * a  + (a += 2)
```

Vzemimo, da ima spremenljivka `a` na začetku vrednost pet. Ker ne vemo, kateri od delnih izrazov `2 * a` ali `a += 2` se bo izvedel prvi, ima lahko gornji izraz vrednost bodisi 17 bodisi 21. Če se namreč najprej izračuna izraz `a += 2`, potem ima spremenljivka `a` pri računanju izraza `2 * a` vrednost sedem, in ne pet, kakršno bi imela, če bi se najprej izračunal izraz `2 * a`. Za razliko od deljenja z nič, kjer obnašanje ni določeno, govorimo v tem primeru o ***nepredpisanem obnašanju*** (angl. unspecified behavior). O takšnem obnašanju govorimo, kadar veleva standard dve ali več možnosti, nič pa ne govori o tem, katera naj se uporabi. Seveda se je tudi pisanju takšne kode najbolje izogibati.

Pojma vrstnega reda računanja delnih izrazov ne smemo zamešati s pojmoma prednosti in asociativnosti operatorjev. Na primer, v naslednjem izrazu se od vrednosti, ki jo vrne funkcija `f1`, najprej odšteje vrednost, ki jo vrne funkcija `f2`. Na koncu se od dobljene razlike odšteje še vrednost, ki jo vrne funkcija `f3`. To je vedno tako. Po drugi strani pa vrstni red, v katerem se kličejo funkcije `f1`, `f2` in `f3`, ni predpisan. Te tri funkcije se lahko kličejo v kakršnemkoli vrstnem redu:

```
f1() - f2() - f3()
```

2.3 Logični izrazi

V mnogih primerih (kot na primer v stavkih `if` ali `while`) je treba preveriti, ali je določena trditev ***pravilna*** (angl. true) ali ***napačna*** (angl. false). Jezik C ne pozna posebnega logičnega podatkovnega tipa, prav tako ne vrednosti true in false[3]. Namesto tega vračajo logični operatorji celoštevilski vrednosti ena (pravilno) oziroma nič (napačno).

Primerjalni operatorji

Primerjalni operatorji so namenjeni primerjanju vrednosti dveh številskih izrazov po velikosti. Primerjalne operatorje sestavljajo štirje ***relacijski operatorji*** (angl. relational opera-

[3]Standard C99 sicer pozna poseben podatkovni tip `_Bool`, ki je v resnici celoštevilski tip, katerega vrednosti so omejene na nič in ena. Poleg tega določa standard C99 novo standardno knjižnico <stdbool.h>, ki vsebuje makro, ki `_Bool` preimenuje v `bool`. Knjižnica <stdbool.h> vsebuje tudi dva makra `false` in `true`, ki določata vrednosti nič in ena.

tors), ki ustrezajo matematičnim primerjalnim operatorjem $>$, $<$, $\geq$ in $\leq$, in dva ***operatorja enakosti*** (angl. equality operators), ki primerjata enakost oziroma različnost. Operatorji so zbrani v naslednji tabeli:

Operator	Opis
>	večji
<	manjši
>=	večji ali enak
<=	manjši ali enak
==	enak
!=	različen

Vsi ti operatorji vrnejo vrednost ena v primeru, ko je relacija, ki jo preverjajo, resnična. V nasprotnem primeru vrnejo vrednost nič. Na primer, vrednost izraza `2.6 < 13` je ena, vrednost izraza `2 != 2` pa je nič. Primerjalni operatorji imajo levo asociativnost ter nižjo prednost od aritmetičnih. To pomeni, da je izraz:

```
x + 1 < y - 2
```

enakovreden izrazu:

```
(x + 1) < (y - 2)
```

Poleg tega imajo relacijski operatorji višjo prednost od obeh operatorjev enakosti. Zato je izraz:

```
x > 5 == y < 10
```

enakovreden izrazu:

```
(x > 5) == (y < 10)
```

Naloga 2.1 Napišite program, ki izpiše vrednost izraza `2 < x < 5` za tri različne vrednosti spremenljivke `x`: 0, 3 in 6. Ali dobite vrednosti, ki ste jih pričakovali? Poskusite pojasniti, kako se izračuna vrednost takšnega izraza.

Logični operatorji

Z uporabo logičnih operatorjev IN (angl. and, uporabljen simbol &&), ALI (angl. or, uporabljen simbol ||) in negacija (angl. negation, uporabljen simbol !) lahko gradimo bolj zapletene logične izraze. Operatorja IN in ALI sta binarna, medtem ko je negacija unarni operator. Logični operatorji vrnejo bodisi nič bodisi ena, eno od teh dveh vrednosti pa imajo pogosto tudi njihovi operandi. Vendar slednje ni nujno – operandi logičnih operatorjev imajo lahko kakršnokoli številsko vrednost, pri čemer se bo vrednost nič obravnavala kot napačna (angl. false), katerakoli neničelna vrednost pa kot pravilna (angl. true).

Logični operatorji delujejo takole:

- Izraz `!izr` vrne vrednost ena, če je `izr` enak nič. Sicer vrne nič.

- Izraz `izr1 && izr2` vrne ena, če sta tako `izr1` kot tudi `izr2` oba različna od nič. Sicer vrne nič.
- Izraz `izr1 || izr2` vrne ena, če je vsaj eden od izrazov `izr1` in `izr2` različen od nič. Sicer vrne nič.

Operator `!` ima enako prednost, kakor jo imata pozitiven in negativen predznak. Operator `&&` ima prednost pred operatorjem `||`, oba pa imata nižjo prednost kakor primerjalni operatorji. Tako je na primer izraz:

```
x > 1 && x < 10
```

enakovreden izrazu:

```
(x > 1) && (x < 10)
```

Naloga 2.2 Za vajo napišite program, ki bo izpisal ***resničnostno tabelo*** (angl. truth table) za logične operatorje `&&`, `||` in `!`. Resničnostna tabela je tabela, v kateri so zapisane vrednosti, ki jih vrne določena logična operacija (oz. funkcija) za vse možne kombinacije vhodnih spremenljivk.

Primer delovanja programa:

```
x  y   (x && y)   (x || y)    (!x)
--------------------------------
0  0          0          0       1
0  1          0          1       1
1  0          0          1       0
1  1          1          1       0
```

Kratkostično računanje

Pomembno je, da operatorja `&&` in `||` upoštevata pravilo tako imenovanega ***kratkostičnega računanja*** (angl. short-circuit evaluation). To pomeni, da najprej izračunata vrednost levega operanda. Če je iz izračunane vrednosti levega operanda mogoče sklepati na vrednost celotnega izraza, potem se vrednost desnega operanda sploh ne računa. Vzemimo za primer naslednji izraz:

```
(x != 0) && (y % x == 0)
```

Tu se najprej izračuna vrednost izraza `x != 0`. Če je vrednost tega izraza nič, potem operator `&&` ne more vrniti vrednosti ena ne glede na vrednost izraza `y % x == 0`. Zato operator `&&` vrne vrednost nič, ne da bi računal vrednost izraza na svoji desni strani. Uporabnost pravila kratkostičnega računanja je pri tem očitna: prepreči računanje ostanka pri deljenju z nič, katerega izid je nedoločen. Tako bo gornji logični izraz vrnil ena le v primeru, ko je `x` različen od nič in hkrati brez ostanka deli `y`. Pri tem ne tvegamo nevarnosti, da bi se kdajkoli izvedlo deljenje z nič.

Naloga 2.3 Za vajo razmislite, kakšna bo vrednost spremenljivke `y`, ko se izvede naslednji del cejevskega programa:

```
int x = 0, y = 10;
if (x == 0 || ++y > 0) {
  y = y + 10;
}
```

Pravilnost svoje ugotovitve preverite tako, da program zaženete. Poskusite rezultat tudi pojasniti.

2.4 Stavki iz izrazov

Strogo gledano izraza ne moremo izvesti. Cejevski programi so v celoti sestavljeni iz ***stavkov*** (angl. statements). Na srečo lahko iz prav vsakega ***izraza naredimo stavek*** (angl. expression statement) preprosto tako, da na konec izraza postavimo podpičje:

```
izraz;
```

Na primer, iz izrazov `x = 1` in `x == 3` lahko naredimo naslednja dva stavka:

```
x = 1;   /* x postane enak 1.  */
x == 3;  /* Brez učinka.       */
```

Ko se stavka izvedeta, prvi od obeh stavkov priredi spremenljivki `x` vrednost ena. Kot že vemo, izraz `x = 1` to vrednost tudi vrne. Vendar se ta vrnjena vrednost v gornjem primeru zavrže, ker izraz `x = 1` ni del kakšnega kompleksnejšega izraza.

Stavki torej za razliko od izrazov ne vračajo vrednosti (tudi če vrednost obstaja, se ta zavrže), zato stavki brez stranskega učinka nimajo nikakršnega smisla. Primer takšnega stavka je drugi od gornjih dveh stavkov: izraz v njem sicer vrne vrednost nič (ker ima `x` vrednost ena), vendar se ta vrednost zavrže.

2.5 Naloge

Naloga 2.4 Kaj bo izpisal naslednji kos cejevskega programa?

```
int x = 2, y = 3, z = 4;
printf("%d %d %d\n", x++, --y, ++z);
printf("%d %d %d\n", x, y, z);
```

Naloga 2.5 Kakšne bodo vrednosti spremenljivk `x` in `y` po tem, ko se izvede vsak od naslednjih stavkov? Predpostavite, da ima spremenljivka `x` na začetku vsakokrat vrednost 5, spremenljivka `y` pa 2. Nekaterih od spodnjih stavkov prevajalnik ne bo prevedel. Zakaj? Poleg tega nekateri stavki nimajo učinka. Kateri in zakaj?

```
(a) x %= y;
(b) x = 1 + (y = 6);
(c) x = 1 + (y == 6);
(č) x + 2 - 3 * (y - 1);
(d) x += y;
(e) x =+ y;
(f) x =/ y;
(g) x + --y;
(h) x + 1 = y;
(i) x > 1 || y == 13;
(j) y = y / x;
(k) y = 10 - 6 / 3 / 2;
(l) x = y < 0 || y > 10;
(m) y = y && x -= 5;
(n) y = y && (x -= 5);
```

```
(o) x = 10 < x < 100;
(p) x = x / 6 * 6;
(r) x = (x != 5) < (y == 2);
(s) x = 20 || y++;
```

Naloga 2.6 Samo eden od izrazov `x++` in `++x` je enakovreden izrazu `x += 1`. Kateri? Utemeljite odgovor.

Naloga 2.7 Ali imata izraza `-(7 / 6)` in `-7 / 6` vedno enako vrednost? Utemeljite odgovor.

Naloga 2.8 Napišite program, ki bo izpisal tabelo vrednosti izrazov `(x && y || z)` in `(x && (y || z))` za vse možne kombinacije logičnih vrednosti spremenljivk `x`, `y` in `z`.

Primer delovanja programa:

```
x  y  z   (x && y || z)   (x && (y || z))
-----------------------------------------
0  0  0                0                 0
0  0  1                1                 0
0  1  0                0                 0
0  1  1                1                 0
1  0  0                0                 0
1  0  1                1                 1
1  1  0                1                 1
1  1  1                1                 1
```

Naloga 2.9 Napišite program, ki s tipkovnice prebere deset številskih vrednosti in izpiše, koliko jih je večjih od nič. Od krmilnih stavkov lahko v programu uporabite zgolj en ponavljalni stavek. Odločitvenega stavka (tj. stavka `if`) ne smete uporabiti.

Pomoč: Za rešitev problema uporabite načelo kratkostičnega računanja.

Naloga 2.10 Napišite program, ki s tipkovnice prebere dva para celoštevilskih vrednosti x_{start} in x_{stop} ter y_{start} in y_{stop} (predpostavimo, da velja $x_{\text{start}} \leq x_{\text{stop}}$ in $y_{\text{start}} \leq y_{\text{stop}}$). Program naj izpiše vsa cela števila med vključno x_{start} in x_{stop}, pri čemer naj z zvezdico označi vsa števila, ki ležijo med vključno y_{start} in y_{stop}. V programu lahko uporabite en ponavljalni stavek in en stavek `if` brez dela `else`.

Dva primera delovanja programa:

```
Območje za izpis: 3 12
Označi vrednosti: 6 8
3  4  5  *6  *7  *8  9  10  11  12

Območje za izpis: 2 6
Označi vrednosti: 0 4
*2  *3  *4  5  6
```

Naloga 2.11 Ugotovite, kaj naredi naslednji kos cejevskega programa:

```
float a, b;
scanf("%f%f", &a, &b);
printf("%f\n", (a >= b) * a + (a < b) * b);
```

Pravilnost ugotovitve preizkusite tako, da kodo zaženete. Ugotovitev tudi pojasnite.

Naloga 2.12 Knjižnica <stdlib.h> vsebuje funkcijo `rand`, ki ob klicu vrne naključno celoštevilsko vrednost med nič in `RAND_MAX` (makro, določen v knjižnici <stdlib.h>). Napišite program, ki ob zagonu izpiše deset naključnih celih števil med vključno ena in šest.

Opomba: Ko boste program zagnali, boste opazili vsakokrat isto zaporedje števil. To se zgodi zato, ker vrednosti, ki jih vrača funkcija `rand`, niso povsem naključne. Vsaka naslednja vrednost se izračuna iz prejšnje po določenem algoritmu. Takšna števila so zgolj videti naključna, zato jih imenujemo ***psevdonaključna*** števila (angl. pseudorandom number). Težavo lahko rešimo tako, da poskrbimo, da je vsaj prvo število resnično naključno. Ker je algoritem izračunavanja psevdonaključnih števil takšen, da se zaporedje števil začne ponavljati šele po zelo dolgem času, je takšna rešitev večinoma sprejemljiva.

Prvo naključno vrednost za generator psevdonaključnih števil izberemo s klicem funkcije `srand`, ki ji kot argument podamo sistemsko uro. Ker ne vemo, kdaj bo uporabnik program zagnal, lahko ta podatek vzamemo kot naključen:

```
#include <stdlib.h>  /* Funkciji rand in srand.          */
#include <time.h>    /* Funkcija time.                   */
//...
srand(time(NULL));   /* Kličemo samo enkrat, in to pred
                        prvim klicem funkcije rand.      */
```

PRAZNA STRAN

3. POGLAVJE

STAVKI

Najmanjša enota, ki jo v ceju lahko izvedemo, je ***stavek*** (angl. statement). Spoznali smo že stavek `return` in stavke, ki so izrazi, zaključeni s podpičjem. Jezik C pozna še nekaj stavkov, ki jih bomo spoznali v tem poglavju in jih lahko razdelimo v naslednje štiri kategorije:

- ***Odločitveni oziroma izbirni stavki*** (angl. selection statement): Sem spadata stavka `if...else` in `switch`, ki glede na določen pogoj izbereta, kateri stavek (oz. katere stavke) bomo izvedli.
- ***Ponavljalni stavki ali zanke*** (angl. iteration statement, loop): S stavki `for`, `while` in `do...while` lahko dosežemo iterativno izvajanje (oz. ponavljanje) določenega zaporedja stavkov.

  Odločitvenim in ponavljalnim stavkom pravimo včasih tudi ***krmilni stavki*** (angl. control statement).
- ***Skoki*** (angl. jump statement): C pozna tudi stavke, s katerimi lahko brezpogojno preusmerimo izvajanje programa na kakšno drugo točko. To lahko dosežemo s stavki `return`, `break`, `continue` in `goto`. Seveda je uporaba teh stavkov smiselna le v kombinaciji z odločitvenim ali izbirnim stavkom.
- ***Prazen in sestavljen stavek*** (angl. null and compound statement): To sta dva posebna stavka, od katerih prvi ne stori ničesar, drugi pa združi več zaporednih stavkov v enega samega.

3.1 Stavek `if...else`

Stavek `if...else` zapišemo takole:

```
if ( izraz ) stavek1 else stavek2
```

Seveda lahko drugi del stavka (tj. `else` *stavek*2) tudi izpustimo.

Naj spomnimo, da C ne pozna logičnih (Boolovih) vrednosti true in false. Stavek `if` se zato odloča glede na ***številsko*** vrednost izraza *izraz*: če je njegova vrednost različna od nič, se izvede *stavek*1, sicer se izvede *stavek*2. Na primer, naslednji del programa bo dvakrat izpisal sporočilo *Pogoj je izpolnjen*:

```
int x = 2;
if (x != 0) printf("Pogoj je izpolnjen");
else printf("Pogoj ni izpolnjen");

if (x) printf("Pogoj je izpolnjen");
else printf("Pogoj ni izpolnjen");
```

Namreč, tako izraz `x != 0` kot tudi izraz `x` imata oba vrednost, ki je različna od nič.

Sestavljen stavek

Opazimo, da v gornjem zapisu stavka `if...else` besedi *stavek*1 in *stavek*2 nastopata v ednini. To pomeni, da lahko na vsako od obeh mest vstavimo zgolj po en stavek. Če želimo vstaviti več stavkov, lahko uporabimo ***sestavljen stavek*** (angl. compound statement):

```
{ stavki }
```

Takoj ko zapišemo več zaporednih stavkov v par zavitih oklepajev, bo prevajalnik vse te stavke obravnaval kot en sam stavek. Kadar v stavkih `if` in `if...else` v vsakem delu nastopa le po en stavek, zavitih oklepajev ne potrebujemo, vendar jih bomo vseeno pisali. Na tak način se lahko izognemo neprijetnim napakam, ki jih je včasih težko izslediti. Primer v naslednjem razdelku kaže eno takšnih napak.

Naloga 3.1 Preden pa nadaljujemo, poskusite odgovoriti na naslednje vprašanje: Ali se bo naslednji cejevski program prevedel? Svoj odgovor utemeljite.

```
#include <stdio.h>
int main(void) {
  int a = 1, b = 2, c = 3;
  if (a > b)
    a++;
    c++;
  else
    b++;
  printf("%d %d %d", a, b, c);
  return 0;
}
```

Kaj pa se zgodi, če odstranimo drugi del gornjega stavka `if...else` (tj. `else b++;`)? Ali se program zdaj prevede? Če se, kakšne vrednosti spremenljivk `a`, `b` in `c` se bodo izpisale, ko ga zaženemo?

Problem visečega stavka `else`

Poglejmo si naslednji primer dveh gnezdenih odločitvenih stavkov:

```
if (q != 0)
  if (z > 0)
    z = z / q;
else
  printf("Napaka: q mora biti različen od nič.");
```

Iz kode je videti, kakor da del `else` spada k prvemu stavku `if`, vendar ni tako. Ko prevajalnik naleti na besedo `else`, jo poveže z najbližjim stavkom `if`, ki še nima dela `else`. Težavi, ki nastopi zaradi takšne dvoumne situacije, pravimo problem ***visečega stavka `else`*** (angl. dangling `else`).

Če želimo del `else` povezati s prvim od obeh stavkov `if`, potem to storimo tako, da na ustrezna mesta postavimo zavite oklepaje:

```
if (q != 0) {
  if (z > 0) {
    z = z / q;
  }
}
else {
  printf("Napaka: q mora biti različen od nič.");
}
```

Zdaj ni nobenega dvoma več glede logike gornje kode.

Pogojni operator

Poleg odločitvenega stavka `if...else`, ki omogoča izbiro med dvema opraviloma, pozna C tudi ***pogojni operator*** (angl. conditional operator). Pogojni operator ***vrne*** eno od dveh vrednosti v odvisnosti od vrednosti pogoja. Sestavljen je iz vprašaja in dvopičja (tj. iz simbolov ? in :), ki ju uporabimo na naslednji način:

$izr1$ `?` $izr2$ `:` $izr3$

Pri tem so izrazi $izr1$, $izr2$ in $izr3$ izrazi poljubnega tipa, izraz, ki ga sestavljajo skupaj s simboloma ? in :, pa se imenuje ***pogojni izraz*** (angl. conditional expression). Pogojni operator je edini cejevski operator, ki zahteva tri operande, zato mu včasih rečemo tudi ***ternarni operator*** (angl. ternary operator).

Vrednost izraza $izr1$ `?` $izr2$ `:` $izr3$ se računa na naslednji način: Najprej se izračuna vrednost izraza $izr1$. Če je njegova vrednost različna od nič, potem se izračuna vrednost izraza $izr2$, ki postane tudi vrednost celotnega pogojnega izraza. V nasprotnem primeru – če je vrednost izraza $izr1$ enaka nič – je vrednost celotnega pogojnega izraza enaka vrednosti izraza $izr3$. Podobno kot v stavku `if...else` se tudi v pogojnem izrazu izvede samo eden od izrazov $izr2$ ali $izr3$. To dejstvo je pomembno, kadar ima kateri od obeh izrazov stranski učinek.

Naslednji del kode ilustrira, kako pogojni operator deluje:

```
int x, a = -2;
x = a >= 0 ? a : -a;  /* x postane enak 2.  */
```

Ker je v gornji kodi `a` manjši od nič, pogojni operator vrne vrednost izraza `-a`, sicer bi vrnil vrednost izraza `a`. To pomeni, da operator v gornjem primeru vrne absolutno vrednost

spremenljivke `a`. Ker ima pogojni operator višjo prednost, kot jo ima priredilni operator, je druga vrstica gornje kode enakovredna stavku:

```
x = (a >= 0 ? a : -a);
```

S pretirano uporabo pogojnega operatorja lahko naredimo kodo nerazumljivo, v določenih primerih pa je vseeno prikladen. Na primer, če želimo iz funkcije vrniti absolutno vrednost spremenljivke, lahko namesto takšnega stavka `if...else`:

```
if (x >= 0) return x;
else return -x;
```

uporabimo kompaktnejši zapis:

```
return x >= 0 ? x : -x;
```

3.2 Stavek `switch`

Kadar moramo primerjati vrednost izraza z več različnimi vrednostmi, lahko uporabimo zaporedje stavkov `if...else`. Na primer, če želimo izpisati doseženo oceno na izpitu z besedo, lahko to storimo na naslednji način:

```
if (ocena == 10) {
  printf("odlično");
}
else if (ocena == 9 || ocena == 8) {
  printf("prav dobro");
}
else if (ocena == 7) {
  printf("dobro");
}
else if (ocena == 6) {
  printf("zadostno");
}
else {
  printf("nezadostno");
}
```

V takšnih primerih lahko uporabimo tudi stavek `switch`, ki ima naslednjo obliko:

```
switch ( izr ) {
  case konstanten-izraz : stavki
  case konstanten-izraz : stavki
  //...
  case konstanten-izraz : stavki
  default : privzeti-stavki
}
```

Stavek `switch` se odloča glede na vrednost izraza *izr*, ki se izračuna čisto na začetku. Velja omejitev, da mora imeti *izr* ***celoštevilsko vrednost***. Izračunana vrednost se po vrsti primerja z vrednostmi ***oznak `case`*** (angl. `case` labels), ki se izračunajo kot vrednosti ***konstantnih izrazov***, ki stojijo za posameznimi besedami `case`. Ti izrazi lahko vsebujejo zgolj konstantne vrednosti, njihove izračunane vrednosti pa morajo biti ***unikatne***. Tako kot vrednost izraza *izr* morajo biti tudi vrednosti konstantnih izrazov v oznakah `case` celoštevilskega tipa.

Kakor hitro je vrednost izraza *izr* enaka kateri od vrednosti oznak `case`, se začne izvajati prvi stavek, ki stoji za to oznako `case`. Od tod naprej se izvedejo vsi stavki do konca stavka `switch`. V primeru, da vrednost izraza *izr* ni enaka vrednosti nobene od naštetih oznak `case`, se izvedejo stavki, ki stojijo za oznako `default` (tj. *privzeti-stavki*). Oznaka `default` (slov. privzeto) ima podobno vlogo kot jo ima del `else` v stavku `if...else`. Pokrije namreč vse primere, kjer vrednost izraza *izr* ni enaka vrednosti nobene od naštetih oznak `case`. In prav tako kot del `else` pri stavku `if...else` tudi oznaka `default` ni obvezen del stavka `switch`.

Omeniti velja še, da lahko za vsako od oznak `case` (za dvopičjem) postavimo poljubno mnogo stavkov, pri čemer ne potrebujemo zavitih oklepajev, da bi jih združili v sestavljen stavek.

Poglejmo si zdaj na konkretnem zgledu, kako stavek `switch` deluje. Naslednji primer posnema vožnjo z dvigalom do pritličja. Najprej z vhoda prebere številko nadstropja, v katerem vstopimo v dvigalo, potem pa izpiše, mimo katerih nadstropij se peljemo:

```
int nadstropje;
printf("V katerem nadstropju vstopiš? ");
scanf("%d", &nadstropje);
printf("Peljemo se preko ... ");
switch (nadstropje) {
  case 4 : printf("četrtega, ");
  case 3 : printf("tretjega, ");
  case 2 : printf("drugega in ");
  case 1 : printf("prvega nadstropja ");
  default : printf("v pritličje.");
}
```

Naslednja dva primera kažeta, kako program deluje:

```
V katerem nadstropju vstopiš? 2
Peljemo se preko ... drugega in prvega nadstropja v pritličje.

V katerem nadstropju vstopiš? 0
Peljemo se preko ... v pritličje.
```

Zdaj nastopi vprašanje, kako lahko s stavkom `switch` nadomestimo zaporedje stavkov `if...else` v programu za izpisovanje ocene z besedo na strani 30. Za to potrebujemo stavek `break`.

Stavek `break`

Videli smo, da stavek `switch` sam po sebi ne ponuja izbire med več možnostmi. Stavek zgolj preskoči vse stavke pred oznako `case`, katere vrednost je enaka vrednosti izraza, ki ga preizkušamo. Da bi lahko stavek `switch` uporabili kot izbirni stavek, moramo v njem uporabiti stavek `break` (slov. prekini). Ta stavek lahko uporabimo tako v stavku `switch` kot tudi v kateremkoli od ponavljalnih stavkov. Stavek `break` deluje tako, da brezpogojno preusmeri izvajanje programa na stavek, ki sledi neposredno stavku, v katerem ga uporabimo.

Program za izpis ocene izpita z besedo s strani 30 lahko zdaj zapišemo takole:

```
switch (ocena) {
  case 10 : printf("odlično"); break;
  case  9 : printf("prav dobro"); break;
  case  8 : printf("prav dobro"); break;
```

```
  case  7 : printf("dobro"); break;
  case  6 : printf("zadostno"); break;
  default : printf("nezadostno"); break;
}
```

Ker se z zadnjim klicem funkcije `printf` gornji stavek `switch` konča, čisto zadnji stavek `break` ni potreben. Navada je, da ga vseeno pišemo, saj bi lahko kasneje z dodajanjem novih oznak `case` nanj pozabili.

Če upoštevamo dejstvo, da se brez stavka `break` izvedejo tudi stavki ob naslednjih oznakah `case`, lahko zadnji primer nekoliko poenostavimo:

```
switch (ocena) {
  case 10 : printf("odlično"); break;
  case  9 :
  case  8 : printf("prav ");
  case  7 : printf("dobro"); break;
  case  6 : printf("zadostno"); break;
  default : printf("nezadostno"); break;
}
```

V primeru, da ima `ocena` vrednost devet ali osem, se najprej izpiše beseda *prav*. Ker tu še ni stavka `break`, se zatem izpiše še beseda *dobro*.

3.3 Stavek `while`

Stavek `while` je prvi od treh ponavljalnih stavkov oziroma ***zank*** (angl. loop), ki jih pozna C. Naloga ponavljalnega stavka je, da ponavlja izvajanje nekega drugega stavka, ki mu pravimo tudi ***telo zanke*** (angl. loop body). Vsakokratno izvajanje telesa zanke se imenuje ***iteracija*** (angl. iteration). Ponavljanje zanke nadzoruje ***krmilni izraz*** (angl. controlling expression), čigar vrednost mora biti različna od nič, da se izvede naslednja iteracija.

Stavek `while` zapišemo v Ceju takole:

```
while ( izraz ) stavek
```

Tu predstavljata *izraz* krmilni izraz, *stavek* pa telo zanke. Krmilni izraz mora biti zapisan v par okroglih oklepajev, med besedo `while` in oklepajem pa ne sme biti ničesar (razen presledkov, tabulatorjev ali praznih vrstic). Prav tako ne sme biti ničesar med zaklepajem in telesom zanke. Kadar želimo, da je telo zanke sestavljeno iz več stavkov, te stavke zapišemo v par zavitih oklepajev, s čimer ustvarimo sestavljen stavek.

3.4 Stavek `do...while`

Stavek `do...while` deluje podobno kot stavek `while` s to razliko, da se vrednost krmilnega izraza preveri ***po tem***, ko se izvede telo stavka. To pomeni, da se – ne glede na vrednost krmilnega izraza – telo zanke v vsakem primeru izvede vsaj enkrat. Stavek zapišemo takole:

```
do stavek while ( izraz ) ;
```

Stavek `do...while` je edini od krmilnih (tj. odločitvenih in ponavljalnih) stavkov, ki mora imeti na koncu podpičje. Tudi pri tem stavku velja, da mora biti njegovo telo (tj.

stavek v gornjem zapisu) en sam stavek. Če želimo izvajati več stavkov, jih združimo z zavitima oklepajema.

Stavek `do...while` se uporablja mnogo redkeje kot stavek `while`. Kljub temu je stavek prikladen, kadar je treba telo izvesti vsaj enkrat, preden se lahko odločimo, ali bomo s ponavljanjem nadaljevali ali ne. Na primer, naslednji program pričakuje od uporabnika, da vnaša števila preštetih zabojev, dokler ne vnese vrednosti nič. Takrat se program konča in izpiše vsoto vseh vnesenih vrednosti:

```
#include <stdio.h>

int main(void) {
  int vnos, vsota = 0;
  do {
    printf("Vnesi število zabojev (0 konča): ");
    scanf("%d", &vnos);
    vsota += vnos;
  } while (vnos != 0);
  printf("Skupaj imamo %d zabojev.\n", vsota);
  return 0;
}
```

Narava naloge je tu takšna, da moramo uporabnika vprašati za vnos števila, ***preden*** se lahko odločimo, ali bomo nadaljevali ali ne. (Uporabnik se lahko na primer že takoj ob zagonu programa odloči, da bo končal.) Seveda bi lahko nalogo rešili tudi s stavkom `while`, le da bi morali na začetku spremenljivko `vnos` inicializirali na neko (določeno) vrednost, različno od nič (npr. ena). Vendar bi bil tak program za odtenek manj razumljiv, saj ne bi bilo takoj očitno, zakaj mora imeti ta spremenljivka na začetku vrednost ravno ena.

3.5 Neskončna zanka

Če poskrbimo, da je vrednost krmilnega izraza v zanki vedno različna od nič, se zanka nikoli ne bo ustavila. Dobimo ***neskončno zanko*** (angl. infinite loop):

```
while (1) {  /* Neskončna zanka.  */
  //...
}
```

Takšno zanko lahko končamo tako, da uporabimo katerega od skočnih ukazov `break` ali `return` (izjemoma `goto`) v kombinaciji s stavkom `if`. Na ta način lahko preverjamo vrednost krmilnega izraza tudi na sredini telesa (in ne le na njegovem začetku oziroma koncu, kar počneta stavka `while` oziroma `do...while`).

Na primer, naslednji kos programa omogoča uporabniku, da vanj vnaša izmerjene dolžine feritnih jeder z namenom preverjanja njihove kakovosti na koncu proizvodne linije. Program šteje, koliko jeder odstopa za več kot 10 % od zahtevane dolžine (največje dopustno odstopanje) in na koncu izračuna odstotek izmeta:

```
#include <stdio.h>
#define TOL 0.1

int main(void) {
  float zahtevanaDolzina;
  float izmerjenaDolzina;
  int vsi = 0, izmet = 0;
```

```
  printf("Vnesi zahtevano dolžino: ");
  scanf("%f", &zahtevanaDolzina);
  while (1) {
    printf("Vnesi izmerjeno dolžino (-1 konča): ");
    scanf("%f", &izmerjenaDolzina);
    if (izmerjenaDolzina < 0) break;
    if (izmerjenaDolzina < (1 - TOL) * zahtevanaDolzina ||
      izmerjenaDolzina > (1 + TOL) * zahtevanaDolzina) {
      izmet++;
    }
    vsi++;
  }
  printf("Izmet je %d odstoten.\n", vsi > 0 ? 100 * izmet / vsi : 0);
  return 0;
}
```

V zadnjem stavku `printf` smo uporabili pogojni izraz, da se izognemo morebitnemu deljenju z nič. Če uporabnik ne vnese nobenega podatka in takoj izbere izhod iz programa, program enostavno izpiše, da je izmet ničodstoten.

3.6 Stavek `for`

Zadnji od treh ponavljalnih stavkov v jeziku C je stavek `for`. Uporabimo ga vedno, kadar imamo opravka z določenim štetjem, saj njegov zapis omogoča, da v en par okroglih oklepajev pregledno združimo vse tri operacije nad števcem (tj. nastavljanje začetne in preverjanje končne vrednosti ter štetje). Stavek zapišemo takole:

`for (` $izr1$ `;` $izr2$ `;` $izr3$ `)` $stavek$

Z izrazi $izr1$, $izr2$ in $izr3$ običajno nadzorujemo štetje, $stavek$ pa predstavlja telo zanke. Če želimo ponavljati več stavkov, spet uporabimo sestavljen stavek. Stavek `for` deluje takole: Najprej se izračuna izraz $izr1$. Ta izraz ni vključen v ponavljanje, zato tu navadno nastavimo začetne vrednosti števcev in drugih spremenljivk, ki jih bomo potrebovali v zanki. Izraz $izr2$ je krmilni izraz, katerega vrednost mora biti različna od nič, da se izvajanje zanke nadaljuje. Za izrazom $izr2$ se vedno izvede $stavek$, za njim se izračuna izraz $izr3$ (tu navadno večamo ali manjšamo vrednost števca), na koncu pa se znova preveri vrednost izraza $izr2$.

Ker se v stavku `for` preveri vrednost krmilnega izraza, ***preden*** se izvede telo zanke, je stavek `for` v tem smislu podoben stavku `while` in različen od stavka `do...while`.

Naloga 3.2 Za vajo razmislite, kaj bo izpisal naslednji kos cejevskega programa:

```
int n, fakt = 1;
for (n = 1; n <= 5; n++)
  fakt *= n;
  printf("%d! = %d\n", n, fakt);
```

Utemeljite odgovor. Popravite kodo tako, da bo program izpisal:

```
1! = 1
2! = 2
3! = 6
```

```
4! = 24
5! = 120
```

Standard C99 dopušča, da prvi izraz v stavku `for` nadomestimo z deklaracijami spremenljivk:

```
for (int i = 0; i < 10; i++) {
  printf("%d ", i);  /* V redu: i je deklariran znotraj zanke.  */
}
i = 10;              /* Napaka: i ni deklariran.                */
```

Tako deklarirane spremenljivke so »vidne« zgolj znotraj zanke, v kateri so deklarirane. Zato zadnja vrstica v gornjem delu kode povzroči napako pri prevajanju. Mnogi programerji zagovarjajo prakso, da na tak način deklariramo vse števce in druge pomožne spremenljivke, ki jih uporabljamo lokalno v zanki `for`.

Opuščanje izrazov v stavku `for`

Izrazi $izr1$, $izr2$ in $izr3$ v stavku for niso obvezni in kateregakoli od njih lahko izpustimo (čeprav moramo podpičja še vedno pisati). Če na primer izpustimo izraza $izr1$ in $izr3$, lahko dobimo takšen stavek:

```
for (;x < 10;) {
  x--;
}
```

Hitro ugotovimo, da je ta stavek popolnoma enakovreden naslednjemu stavku `while`:

```
while (x < 10) {
  x--;
}
```

Kadar v stavku `for` izpustimo izraz $izr2$, se zanj uporabi privzeta vrednost, ki je različna od nič. Naslednji primer stavka je tako neskončna zanka:

```
for (;;) {  /* Neskončna zanka.  */
//...
}
```

Vejični operator

Včasih se zgodi, da želimo v prvem izrazu stavka `for` inicializirati več kot eno spremenljivko. To lahko storimo z uporabo ***vejičnega operatorja*** (angl. comma operator). Vejični operator je operator, ki preprosto združi dva izraza v enega. Tako dobimo ***vejični izraz*** (angl. comma expression):

```
izr1 , izr2
```

Pri tem sta $izr1$ in $izr2$ poljubna izraza. V vejičnem izrazu se najprej izračuna vrednost izraza $izr1$, ki se zavrže, potem pa se izračuna še vrednost izraza $izr2$, ki je hkrati vrednost celotnega vejičnega izraza. Glede na to, da se vrednost levega izraza zavrže, ima le-ta smisel samo v primeru, kadar povzroči stranski učinek.

Vejični operator ima najnižjo prednost od vseh operatorjev in levo asociativnost. Tako bo vrednost naslednjega izraza enaka šest, kar je vrednost zadnjega od treh izrazov, združenih z vejičnima operatorjema:

```
y = 3, y += 2, y + 1
```

Hkrati se bo vrednost spremenljivke `y` nastavila na pet. Če želimo vrednost vejičnega izraza prirediti kakšni spremenljivki, moramo uporabiti oklepaje, ker ima sicer priredilni operator prednost pred vejičnim:

```
x = (y = 3, y += 2, y + 1);  /* x postane enak 6.  */
```

Naloga 3.3 Za vajo razmislite, kakšna bo vrednost spremenljivke `x`, če iz gornjega primera odstranimo par oklepajev:

```
x = y = 3, y += 2, y + 1;
```

Preverite pravilnost svoje ugotovitve z računalnikom in pojasnite, zakaj je tako.

Uporabnost vejičnega operatorja je v praksi omejena na primere, kjer se sicer zahteva en sam izraz, mi pa bi jih želeli več[1]. Eden redkih primerov, kjer nam vejični operator pride prav, je pri inicializaciji števcev in ostalih spremenljivk v stavku `for`. Vzemimo primer, ki preveri, ali je `n` praštevilo:

```
int n = 1153, i, delitelji;
for (i = 2, delitelji = 0; i < n; i++) {
  if (n % i == 0) {
    delitelji++;
  }
}
if (delitelji) {
  printf("%d ni praštevilo.", n);
}
else {
  printf("%d je praštevilo.", n);
}
```

Tu smo z uporabo vejičnega operatorja v prvem izrazu stavka `for` hkrati inicializirali obe spremenljivki (tako `i` kot tudi `delitelji`), ki ju potrebujemo za preverjanje, ali je `n` praštevilo.

Vejični operator je lahko nevaren, če po pomoti namesto decimalne pike uporabimo decimalno vejico, ki se v slovenščini uporablja za zapisovanje decimalnih števil. Na primer, v naslednjem kosu programa bo `pi` postal enak tri:

```
float pi;
pi = 3,14159;  /* Napaka: vejični operator namesto decimalne pike.  */
```

Prazen stavek

Samo podpičje ali prazen par zavitih oklepajev predstavljata veljaven stavek, ki pa ne naredi ničesar. Takšnemu stavku pravimo ***prazen stavek*** (angl. null statement). Prazen stavek lahko uporabimo na primer pri ponavljalnem stavku, v katerem ne potrebujemo telesa (vendar ga pravilo zahteva). Na primer, gornji program, ki preverja, ali je `n` praštevilo, lahko zapišemo na krajši način:

[1] V tem pogledu spominja vejični izraz na sestavljeni stavek, ki ga uporabimo tam, kjer je dovoljen en sam stavek, mi pa jih potrebujemo več.

```
int n = 1153, i;
for (i = 2; i < n && n % i != 0; i++);
if (i < n) {
  printf("%d ni praštevilo.", n);
}
else {
  printf("%d je praštevilo.", n);
}
```

Podpičje na koncu stavka `for` predstavlja prazen stavek, ki je hkrati obvezno telo zanke. Brez tega stavka bi telo zanke postal stavek `if...else`, ki sledi.

Sicer gornji program deluje takole: V krmilnem izrazu stavka `for` se vedno najprej preveri, ali je `i` manjši od `n`. Če to drži, potem se preveri še, ali deljenje `n` z `i` pusti ostanek, različen od nič. Če drži tudi to, se izvajanje zanke nadaljuje: `i` se poveča za ena, potem pa se spet začne računati krmilni izraz. Zanka se lahko konča na dva načina: bodisi ko `i` postane enak `n` bodisi ko se deljenje `n` z `i` izide brez ostanka. Če se zgodi prvi primer, potem je jasno, da `n` nima nobenega delitelja razen ena in samega sebe. To pomeni, da je `n` praštevilo. Če se zgodi drugi primer, mora biti `i` nujno manjši od `n`. V tem primeru `n` ni praštevilo.

Nevarnost predstavlja nenamerno vstavljen prazen stavek na koncu ponavljalnega stavka, ki ga je težko opaziti:

```
x = 10;
while (x > 0);  /* Napaka: neskončna zanka zaradi praznega stavka.  */
  printf("%d ", x--);  /* Ta vrstica se nikoli ne bo izvedla.       */
```

Gornji stavek `while` se nikdar ne bo ustavil. Zaradi nenamerno vstavljenega praznega stavka (podpičje na koncu druge vrstice kode) klic funkcije `printf` namreč ni več del ponavljalnega stavka. Posledično `x` nikoli ne bo postal enak ali manjši od nič, zaradi česar dobimo neskončno zanko brez izhoda.

3.7 Dodatno krmiljenje izvajanja ponavljalnih stavkov

Kot že vemo, se ponavljalni stavek konča, kakor hitro je vrednost njegovega krmilnega izraza enaka nič. Vrednost krmilnega izraza lahko preverjamo bodisi pred telesom ponavljalnega stavka (v stavkih `while` in `for`) bodisi za njim (v stavku `do...while`). Ponavljalne stavke pa lahko krmilimo tudi drugače.

Stavek `break`

V razdelku 3.5 smo videli, kako lahko uporabimo stavek `break` v kombinaciji s stavkom `if` za preverjanje vrednosti krmilnega izraza v sredini telesa zanke. Če želimo, lahko na ta način izvedemo preverjanje vrednosti (različnih) krmilnih izrazov tudi na več mestih v telesu zanke. Pri tem je pomembno vedeti, da stavek `break` v primeru gnezdenih zank konča zgolj izvajanje zanke, v kateri se nahaja neposredno.

Poglejmo si primer programa, ki izpiše poštevanko števil do n v obliki kvadratne tabele, pri čemer izpiše zgolj vrednosti pod diagonalo:

```
#include <stdio.h>

int main(void) {
  int n;
```

```
  printf("Vpiši celo število: ");
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      printf("%2d ", i * j);
      if (i == j) break;
    }
    printf("\n");
  }
  return 0;
}
```

Stavek `break` v gornjem programu konča izvajanje notranje zanke `for` vsakokrat, ko pridemo do diagonalnega elementa. Vendar se zunanja zanka `for` zaradi tega ne konča. Zato se v naslednjem koraku `i` poveča za ena in notranja zanka se začne znova izvajati (če je seveda `i` še vedno manjši ali enak `n`). Program deluje takole:

```
Vpiši celo število: 8

1
2  4
3  6  9
4  8 12 16
5 10 15 20 25
6 12 18 24 30 36
7 14 21 28 35 42 49
8 16 24 32 40 48 56 64
```

Naloga 3.4 Če v zadnjem programu odstranimo vrstico `if (i == j) break`, lahko z minimalno spremembo preostale kode še vedno dosežemo enako delovanje programa. Poskusite to doseči sami.

Spomnimo se, da lahko uporabimo stavek `break` tudi za izhod iz stavka `switch`. Tudi tu velja isto pravilo: če je stavek `switch` gnezden v kakšnem ponavljalnem stavku, potem stavek `break` v tem stavku `switch` ne bo končal izvajanja ponavljalnega stavka, temveč zgolj izvajanje stavka `switch`.

Stavek `continue`

Stavek `continue` (slov. nadaljuj) je na nek način podoben stavku `break`. Oba stavka preusmerita izvajanje programa na kasnejšo točko. Razlika je le v tem, da stavek `break` preusmeri izvajanje na konec zanke, medtem ko stavek `continue` preusmeri izvajanje na konec ***telesa*** zanke. V primeru stavka `continue` se potemtakem izvajanje zanke nadaljuje (od tod ime) s preverjanjem vrednosti krmilnega izraza oziroma z računanjem tretjega izraza ($izr3$) v stavku `for`.

Naslednji kos programa izpiše vse prafaktorje pozitivnega celega števila `n`:

```
i = 2;

while (n > 1) {
  if (n % i++) continue;
  i--;
  n /= i;
  printf("%d ", i);
}
```

Naj omenimo, da se stavek `continue` v praksi uporablja precej redkeje kakor stavek `break`.

Stavek `goto`

Videli smo, da stavek `break` ne more končati več gnezdenih ponavljalnih stavkov hkrati. Takšno kombinacijo stavkov pa lahko vedno končamo s stavkom `return`, vendar ta stavek hkrati konča tudi izvajanje funkcije. Kadar tega ne želimo, lahko uporabimo stavek `goto`. Stavek deluje tako, da preusmeri izvajanje programa na stavek, ki sledi ustrezni ***oznaki*** (angl. label). Podobno kot pri stavku `switch` tudi oznako stavka `goto` pišemo pred stavkom, na katerega želimo prenesti izvajanje programa, med oznako in stavek pa postavimo dvopičje:

oznaka `:` *stavek*

Pred en stavek lahko postavimo več različnih oznak, podobno kot smo to videli tudi v stavku `switch`. Izvajanje iz kakšnega drugega dela programa preusmerimo na stavek *stavek* na naslednji način:

`goto` *oznaka* `;`

Velja omejitev, da morata biti stavek `goto` in pripadajoča oznaka v isti funkciji. V praksi pa se stavek `goto` uporablja izključno namesto stavka `break`, kadar želimo izstopiti iz več gnezdenih zank hkrati. V nasprotnem primeru lahko z uporabo tega stavka hitro prekršimo načela strukturiranega programiranja, s čimer postane koda težko obvladljiva[2]. Oznako pišemo takoj za zanko, iz katere želimo izstopiti s stavkom `goto`. Na primer:

```
for (/* ... */) {
  for (/* ... */) {
    //...
    goto adijo;
    //...
  }
}
adijo:
//...
```

Podobno lahko ukrepamo, kadar želimo končati zanko iz gnezdenega stavka `switch`:

```
for (/* ... */) {
  switch (/* ... */) {
    //...
    goto adijo;
    //...
  }
}
adijo:
//...
```

[2]Kodi, v kateri se prekomerno uporablja stavek `goto` v smislu, da ruši načela strukturiranega programiranja, pravimo tudi ***špagetna koda*** (angl. spaghetti code). Takšna koda je zmedena, težka za vzdrževanje in podvržena napakam, ki jih je težko izslediti.

Zaviti oklepaji

Kadar z odločitvenim stavkom `if...else` krmilimo le posamezne stavke ali kadar vsebuje telo zanke en sam stavek, ne potrebujemo zavitih oklepajev. Vendar je priporočljivo, da se zavite oklepaje kljub temu uporablja. Za to je kar nekaj razlogov:

- ***Čitljivost kode*** Z uporabo zavitih oklepajev je koda precej lažje berljiva in njena struktura precej očitnejša. Na primer, del kode na strani 36, ki računa, ali je `n` praštevilo, bi bil brez uporabe zavitih oklepajev videti takole:

```
for (i = 2, delitelji = 0; i < n; i++) if (n % i == 0) delitelji++;
if (delitelji) printf("%d ni praštevilo.", n);
else printf("%d je praštevilo.", n);
```

- ***Viseči stavek `else`*** Primerna uporaba zavitih oklepajev lahko prepreči neželeni pojav visečega stavka `else`, ki smo ga srečali na strani 29.
- ***Nenamerno vstavljen prazen stavek*** Na strani 37 smo videli, kako lahko nehote s praznim stavkom predčasno zaključimo ponavljalni stavek. Če se navadimo, da telo zanke vedno začnemo z zavitim oklepajem, bo takšna napaka očitnejša, saj (nenavadna) kombinacija podpičja in zavitega oklepaja hitreje pade v oko:

```
x = 10;
while (x > 0); {  /* Napačno postavljeno podpičje je
                     ob zavitem oklepaju očitnejše.   */
  printf("%d ", x--);
}
```

- ***Namerno uporabljen prazen stavek*** Včasih želimo za telo zanke namerno uporabiti prazen stavek (kot smo to storili v primeru preverjanja praštevila na strani 36). Če namesto podpičja za prazen stavek uporabimo prazen sestavljen stavek, je očitneje, da smo to storili namerno:

```
for (i = 2; i < n && n % i != 0; i++) {}
```

  Zaklepaj lahko postavimo tudi v novo vrstico:

```
for (i = 2; i < n && n % i != 0; i++) {
}
```

- ***Izpadel stavek*** Če med dela stavka `if` in `else` ali med dela stavka `do` in `while` nehote postavimo dva ali več stavkov (glej npr. nalogo 3.1 na strani 28), nas na to opozori prevajalnik. To je namreč tako imenovana ***sintaktična napaka*** (angl. syntax error), ki krši osnovna pravila jezika. V ostalih primerih, kjer se v krmilnih stavkih prav tako zahteva le en stavek, pa prevajalnik ne more ugotoviti, da smo uporabili več stavkov nehote: za ustrezen zaključek krmilnega stavka se preprosto uporabi le prvi od stavkov, ki sledijo. Preostali stavki iz tega krmilnega stavka ***izpadejo***. Takšen primer kaže naslednja koda, kjer je izpadel zadnji stavek. Zato se zanka `while` nikoli ne bo ustavila:

```
vsota = 0;
x = 10;
while (x > 0)
  vsota += x;  /* Ta zanka se nikoli ne konča.       */
  x--;         /* Ta stavek ni več del zanke while.  */
```

Podoben primer smo videli že v nalogi 3.2 na strani 34. Če se navadimo, da za telo zanke vedno uporabimo zavite oklepaje, potem je takšna napaka veliko manj verjetna.

- ***Napačna interpretacija stavka `do...while`*** Če je telo zanke `do...while` brez zavitih oklepajev, obstaja možnost, da zaključek zanke `do...while` pomotoma tolmačimo kot samostojno zanko `while`, zaključeno s praznim stavkom. Na primer:

```
do
  printf("%d", x);
while (x++ < y--);
y++;
```

Če uporabimo oklepaje, je veliko očitneje, da predstavlja koda `while (x++ < y--)` zaključek stavka `do...while`:

```
do {
  printf("%d", x);
} while (x++ < y--);
y++;
```

3.8 Naloge

Naloga 3.5 Kakšna bo vrednost spremenljivk `x` in `y` po tem, ko se izvede vsak od naslednjih stavkov? Predpostavite, da ima spremenljivka `y` na začetku vsakokrat vrednost deset.

```
(a) x = (y++, y + 2);
(b) x = y++, y + 2;
(c) x = y > 0 ? (y--, y + 10) : (y++, y - 10);
(č) x = y < 9 || y > 11 ? y : 1 - y;
```

Naloga 3.6 Podan je naslednji stavek `switch`:

```
switch (x) {
  case 10:
  case 20: y *= 2; break;
  default: y /= 2;
}
```

Kako bi isti učinek dosegli z uporabo stavka `if...else`?

Naloga 3.7 Podan je naslednji del kode:

```
if (x > 0) {
  y = 1;
}
else {
  y = 0;
}
```

Kako bi isti učinek dosegli z uporabo stavka `switch`?

Opomba: Rešitev, ki jo boste dobili, se v praksi nikoli ne uporablja. Naloga je namenjena zgolj poglabljanju razumevanja delovanja stavka `switch` in logičnih izrazov.

Naloga 3.8 Napišite program, ki prebere celoštevilsko vrednost med vključno 30 in 99 ter prebrano vrednost izpiše z besedo.

Primer delovanja programa:

```
Vpiši celo število med 30 in 99: 30
Vpisal/a si število trideset.

Vpiši celo število med 30 in 99: 45
Vpisal/a si število petinštirideset.
```

Naloga 3.9 Dopolnite program iz naloge 3.8 tako, da bo deloval tudi za vrednosti med vključno ena in 29.

Primer delovanja programa:

```
Vpiši celo število med 1 in 99: 3
Vpisal/a si število tri.

Vpiši celo število med 1 in 99: 28
Vpisal/a si število osemindvajset.
```

Naloga 3.10 Napišite program, ki z vhoda prebere, koliko dni ima mesec in s katerim dnem v tednu se začne (ena za ponedeljek, dve za torek ... sedem za nedeljo). Nato naj program izpiše koledar za podani mesec.

Primer delovanja programa:

```
Število dni v mesecu: 28
Prvi dan v mesecu (1: pon, 2: tor ... 7: ned): 5

 P  T  S  Č  P  S  N
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28
```

Naloga 3.11 Dopolnite program iz naloge 3.10 tako, da bo kot vhodne podatke sprejel leto in mesec gregorijanskega koledarja ter podatek o tem, kateri dan v tednu je prvi januar vnesenega leta. Program naj potem izpiše koledar za vneseni mesec. Program naj nalogo ponavlja toliko časa, dokler za leto ne vpišemo vrednosti -1.

Primer delovanja programa:

```
Vpiši leto (-1 za izhod): 2018
Vpiši mesec: 3
1. januar 2018 (1: pon, 2: tor ... 7: ned): 1

 P  T  S  Č  P  S  N
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
```

```
26 27 28 29 30 31

Vpiši leto (-1 za izhod): -1
Hvala, ker si uporabil/a naš program.
```

Naloga 3.12 Izdelajte kalkulator za razreševanje trikotnika. Program naj prek številskih ukazov omogoča uporabniku naslednje operacije:

- vnos dolžin stranic a, b in c,
- izračun ploščine,
- izračun obsega in
- izračun kotov A, B in C.

Primer delovanja programa:

```
Dobrodošel/la v programu za razreševanje trikotnikov.
Uporabiš lahko naslednje ukaze:
(0) Vnos stranic, (1) Ploščina, (2) Obseg, (3) Koti, (4) Izhod
Vnesi ukaz: 1
Najprej vnesi dolžine stranic.
Vnesi ukaz: 0
Vnesi stranice (a b c): 5.3 7.9 11.4
Vnesi ukaz: 1
Ploščina trikotnika je 18.46
Vnesi ukaz: 2
Obseg trikotnika je 24.60
Vnesi ukaz: 0
Vnesi stranice (a b c): 1.5 8 2.5
Vnesi ukaz: 1
Tak trikotnik ni mogoč.
Vnesi ukaz: 5
Neveljaven ukaz. Uporabiš lahko naslednje ukaze:
(0) Vnos stranic, (1) Ploščina, (2) Obseg, (3) Koti, (4) Izhod
Vnesi ukaz: 4
Nasvidenje!
```

Naloga 3.13 Izdelajte program, ki proti človeškemu igralcu igra igro jemanja žetonov s kupa. Igra poteka tako, da igralca s kupa izmenično jemljeta enega, dva ali tri žetone. Izgubi tisti, ki s kupa vzame zadnji žeton.

Primer delovanja programa:

```
Vnesi začetno število žetonov: 15
Kdo bo prvi na potezi (0: človek, 1: računalnik): 0
Na kupu je 15 žetonov. Tvoja poteza: 1
Računalnik vzame 1 žetonov.
Na kupu je 13 žetonov. Tvoja poteza: 2
Računalnik vzame 2 žetonov.
Na kupu je 9 žetonov. Tvoja poteza: 8
Ne goljufaj! Vzameš lahko en, dva ali tri žetone!
Na kupu je 9 žetonov. Tvoja poteza: 3
Računalnik vzame 1 žetonov.
Na kupu je 5 žetonov. Tvoja poteza: 1
Računalnik vzame 3 žetonov.
Na kupu je 1 žetonov. Izgubil/a si!
```

Pomoč: Če je n število žetonov na kupu, potem je idealna poteza, ki vodi do zmage, enaka $(n-1)\ \%\ 4$ žetonov. Kadar je izraz $n-1$ deljiv s štiri brez ostanka, bi bilo najbolje, da igralec, ki je na potezi, ne vzame nobenega žetona, vendar to ni dovoljeno. V tem primeru je vseeno, koliko žetonov vzame igralec, saj je v vsakem primeru v izgubljenem položaju. Ko boste pisali program, lahko za potezo računalnika, ki je v izgubljenem položaju, vedno izberete en žeton. Da pa igra zaradi tega ne bi postala dolgočasna, lahko namesto enega izberete naključno število žetonov (tj. enega, dva ali tri). Za izbiranje naključnih vrednosti lahko uporabite rešitev naloge 2.12 na strani 25.

Razširitev: Računalnik se ne bo nikoli zmotil, zato že ena človeška napaka vodi v poraz človeškega igralca. Igrico lahko dopolnite tako, da se bo računalnik sem in tja zmotil, verjetnost računalnikove napake pa naj bo odvisna od izkušenosti igralca. Svoj nivo izkušenosti naj igralec vnese na začetku igre.

4. POGLAVJE

SKALARNI PODATKOVNI TIPI

V tem poglavju bomo spoznali ***skalarne podatkovne tipe*** (angl. scalar data types) jezika C. Značilno za te podatkovne tipe je, da predstavljajo eno samo številsko vrednost. Delimo jih na ***aritmetične tipe*** (angl. arithmetic types) in ***kazalčne tipe*** (angl. pointer types). Aritmetične tipe delimo naprej na ***celoštevilske tipe*** (angl. integer types) in ***realne tipe*** (angl. real types). Pod celoštevilske tipe spadajo cela števila in ***znakovni tipi*** (angl. character types), pod realne tipe pa cela in realna števila.

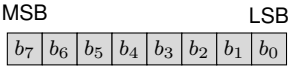
4.1 Celoštevilski tipi

V tem razdelku bomo obravnavali le predznačena in nepredznačena cela števila, znakovnim tipom pa se bomo posvetili posebej. ***Celoštevilski tipi*** (angl. integer types), ki jih pozna jezik C, se razlikujejo glede na ***območje vrednosti*** (angl. value range), ki jih lahko predstavljajo, in glede na predznak: cela števila so lahko bodisi ***predznačena*** (angl. signed) bodisi ***nepredznačena*** (angl. unsigned). Da bomo te pojme bolje razumeli, si najprej poglejmo, kako so cela števila zapisana v pomnilniku.

Bit in bajt

Najmanjša enota informacije v računalništvu je ***bit*** (angleška prekrivanka iz besed ***bi***nary digi***t***, slov. dvojiška številka). Kadar nastopajo biti v skupinah, jih navadno številčimo od desne proti levi, pri čemer začnemo šteti z nič. Skrajno desnemu bitu pravimo tudi ***najmanj***

pomemben bit (angl. least significant bit, LSB), skrajno levemu bitu pa ***najpomembnejši bit*** (angl. most significant bit, MSB). To poimenovanje in številčenje od desne proti levi sovpada s težo (tj. z velikostjo prispevka k skupni vrednosti), ki jo ima posamezen bit v nepredznačenem celem številu, kot bomo videli v naslednjem razdelku. Naslednja slika kaže primer podatka, zapisanega z osmimi biti:



Skupini osmih[1] bitov pravimo tudi ***bajt*** (angl. byte) oziroma ***zlog***. Prvotno je pomenil bajt število bitov, potrebnih za zapis enega znaka. Zaradi tega je v mnogih računalniških arhitekturah bajt tudi ***najmanjša enota pomnilnika, ki jo lahko naslavljamo*** (angl. smallest addressable unit of memory). Z drugimi besedami, z večino operatorjev ne moremo izvajati operacij nad podatki, ki bi bili krajši od enega bajta.

Dvojiški zapis celih števil

V pomnilniku je vsako celo število zapisano v dvojiškem zapisu, kar pomeni, da je sestavljeno iz n zaporednih bitov. Desetiško vrednost tako zapisanega števila izračunamo tako, da seštejemo potence števila dve, pomnožene z vrednostmi ustreznih bitov. Vzemimo, da imamo dvojiško vrednost $b_{n-1}b_{n-2}\cdots b_0$ in želimo izračunati njeno desetiško vrednost N. To storimo po naslednji enačbi:

$$N = \sum_{i=0}^{n-1} b_i 2^i. \tag{4.1}$$

Na primer:

$$1011001_2 = 2^6 + 2^4 + 2^3 + 2^0 = 89_{10},$$

pri čemer indeksa 2 in 10 označujeta osnovo številskega sistema, v katerem je število zapisano.

Matematično gledano lahko za zapis dvojiške vrednosti uporabimo poljubno število bitov, v računalniku pa smo večinoma omejeni na bajte, kar pomeni, da so takšni zapisi navadno mnogokratniki števila osem. Gornjo dvojiško vrednost bi tako v pomnilniku shranili kot na primer 01011001 ali morda kot 00000000 01011001.

Dvojiški komplement

Dvojiški komplement (angl. two's complement) je preprosta operacija, ki jo mnogi računalniški sistemi uporabljajo za zapisovanje negativnih števil. Operacija je sestavljena iz negacije bitov[2] in prištevanja konstantne vrednosti ena. Dvojiški komplement vrednosti 01011001 je tako 10100111.

[1] V preteklosti je bila dolžina bajta pogojena s hardverom in je lahko zajemala dolžine od enega do 48 bitov. Sodobni de facto standard določa, da pomeni bajt osem bitov.

[2] Operacija negacije bitov enostavno zamenja vsak bit, ki ima vrednost nič, z bitom, ki ima vrednost ena, in obratno – vsak bit, ki ima vrednost ena, zamenja z bitom, ki ima vrednost nič.

Če originalni (osembitni) vrednosti prištejemo njen dvojiški komplement, dobimo vrednost $1\,00000000$. Deveti, skrajno levi bit, je nastal kot posledica prenosa pri seštevanju najvišjih dveh bitov obeh seštevancev. Če se omejimo na osem bitov, potem ta bit odpade in ostane nam vrednost nič. Če je originalna dvojiška vrednost predstavljala desetiško vrednost 89, potemtakem mora njen dvojiški komplement predstavljati desetiško vrednost -89, da je vsota obeh enaka nič. Iz tega lahko sklepamo, da je dvojiški komplement operacija, ki ustreza ***množenju z minus ena***.

Predznačena in nepredznačena cela števila

Pravkar smo ugotovili, da lahko z dvojiškim komplementom zapišemo negativna cela števila. V zadnjem primeru smo tako zapisali število -89. Prav lahko pa bi si zamislili tudi naslednji primer:

$$10100111_2 = 2^7 + 2^5 + 2^2 + 2^1 + 2^0 = 167_{10}.$$

Dvojiški komplement tega števila je 01011001, kar bi moralo biti zdaj enako -167.

Znašli smo se v neprijetnem položaju, ko ena in ista dvojiška vrednost predstavlja dve različni desetiški vrednosti. Na primer, 01011001 je lahko bodisi 89 bodisi -167. Prav tako je lahko 10100111 bodisi 167 bodisi -89. Da se takšnemu dvoumnemu položaju izognemo, se moramo natančno dogovoriti o tem, kaj so ***nepredznačena*** in kaj so ***predznačena*** števila:

- ***Nepredznačena cela števila*** (angl. unsigned integers) nimajo predznaka in so vedno nenegativna. Za njihov zapis torej ne potrebujemo dvojiškega komplementa, njihovo desetiško vrednost pa računamo po enačbi (4.1).
- ***Predznačena cela števila*** (angl. signed integers) imajo predznak in so lahko bodisi pozitivna bodisi negativna. Predznak predznačenega števila določa prvi (skrajno levi) bit dvojiškega zapisa: nič predstavlja pozitiven, ena pa negativen predznak. Kadar je število negativno, je zapisano v dvojiškem komplementu[3].

Zdaj ni dvoma: če zapis 10100111 tolmačimo kot nepredznačeno število, potem dobimo desetiško vrednost 167. Če isti zapis tolmačimo kot predznačeno število, potem dobimo desetiško vrednost -89. Po drugi strani predstavlja zapis 01011001 v obeh primerih desetiško vrednost 89. Ker je prvi bit v zapisu enak nič, je ta vrednost namreč pozitivna tudi, če jo tolmačimo kot predznačeno število.

Naloga 4.1 Za vajo pretvorite osembitne vrednosti v naslednji tabeli v desetiške vrednosti:

Dvojiška vrednost	Desetiška vrednost	
	(Predznačena)	(Nepredznačena)
00011101		
10010101		
01111111		
11101010		

[3]Nobena od obstoječih različic standarda jezika C ne zagotavlja, da so predznačena števila zapisana v dvojiškem komplementu. Vendar bomo v resnici redko naleteli na okolje, ki ne uporablja tega zapisa. O tem, kako pogosto se v praksi uporablja dvojiški komplement, priča dejstvo, da obstajajo resne pobude, da bi priporočilo o uporabi dvojiškega komplementa vključili v standard.

10000000
11111111

Območje vrednosti

Ker smo v računalniškem pomnilniku omejeni s številom bitov, je pomembno, da vemo, kakšno območje desetiških vrednosti lahko zapišemo s posameznim podatkovnim tipom. Za ***nepredznačena*** števila hitro ugotovimo, da je največja desetiška vrednost, ki jo lahko zapišemo z n biti, enaka $2^n - 1$. Velja namreč:

$$11\cdots1_2 = 2^{n-1} + 2^{n-2} + \cdots + 2^0 = 2^n - 1.$$

Najmanjša vrednost je seveda nič.

Največje pozitivne vrednosti, ki jo lahko zapišemo kot n-bitno ***predznačeno*** število, tudi ni težko izračunati:

$$011\cdots1_2 = 2^{n-2} + 2^{n-3} + \cdots + 2^0 = 2^{n-1} - 1.$$

Za zapis pozitivnih števil imamo pri predznačenem zapisu na voljo en bit manj kot pri nepredznačenem zapisu. Zato je gornji rezultat pričakovan. Skupaj z vrednostjo nič imamo tako 2^{n-1} različnih nenegativnih vrednosti. Ker pa lahko z n biti zapišemo 2^n različnih vrednosti, nam ostane še 2^{n-1} negativnih vrednosti, ki so vrednosti od -2^{n-1} do -1. Najmanjša predznačena desetiška vrednost, ki jo tako lahko zapišemo z n biti, je potemtakem -2^{n-1}.

Naloga 4.2 Za vajo zapišite naslednje vrednosti v osembitnem (predznačenem in/ali nepredznačenem) dvojiškem zapisu:

Desetiška vrednost	**Dvojiška vrednost**	
	(Predznačena)	**(Nepredznačena)**
214		
127		
-21		
45		
-128		
128		

Opomba: Vseh podanih vrednosti ni mogoče zapisati v obeh zapisih.

Celoštevilski tipi jezika C

ANSI C pozna tri različne celoštevilske tipe: `short`, `int` in `long`. Tip `short` ima ***bitno dolžino*** (angl. storage size) vsaj 16, tip `long` pa vsaj 32 bitov. Dolžina tipa `int` je določena zelo ohlapno, delno je pogojena z okoljem, v katerem se izvaja program. Na primer, na osem- ali 16-bitnih sistemih imajo podatki tipa `int` običajno dolžino 16 bitov, na 32-bitnih sistemih pa 32 bitov. Poleg minimalnih bitnih dolžin standard določa tudi, da tip `long` ne sme biti krajši od `int`, ki ne sme biti krajši od `short`.

Privzeti zapis vseh treh celoštevilskih tipov je predznačen. Če želimo uporabiti nepredznačen zapis, moramo pri deklaraciji uporabiti besedo `unsigned`. Naslednja tabela prikazuje tipične bitne dolžine in območja vrednosti za celoštevilske podatkovne tipe:

Tip	Bitna dolžina	Najmanjša vrednost	Največja vrednost
`short`	16	$-32\,768$ (-2^{15})	$32\,767$ $(2^{15}-1)$
`unsigned short`	16	0	$65\,535$ $(2^{16}-1)$
`int`	32	$-2\,147\,483\,648$ (-2^{31})	$2\,147\,483\,647$ $(2^{31}-1)$
`unsigned int`	32	0	$4\,294\,967\,295$ $(2^{32}-1)$
`long`	32	$-2\,147\,483\,648$ (-2^{31})	$2\,147\,483\,647$ $(2^{31}-1)$
`unsigned long`	32	0	$4\,294\,967\,295$ $(2^{32}-1)$

Kadar želimo v programu izvedeti, koliko pomnilnika zasede določen podatek, lahko uporabimo operator `sizeof`. Ta operator vrne nepredznačeno celoštevilsko vrednost, ki predstavlja število bajtov, ki jih zasede določen podatek ali podatkovni tip. Operator `sizeof` je unarni operator, ki ga pišemo pred njegovim operandom:

```
sizeof izraz
sizeof ( ime-tipa )
```

Bodite pozorni, da potrebujemo oklepaje, kadar je operand ime podatkovnega tipa. Naslednji primer kaže, kako lahko operator `sizeof` uporabimo:

```
long x;
printf("%u", sizeof x);                /* Izpiše: 4  */
printf("%u", sizeof(unsigned short));  /* Izpiše: 2  */
```

Ker operator `sizeof` vrne nepredznačeno celo število, smo za izpis uporabili formatno določilo `%u`. Podrobneje bomo o različnih formatnih določilih za cela števila govorili nekoliko kasneje.

Prekoračitev

Pri aritmetičnih operacijah nad celimi števili se lahko pripeti, da pade rezultat zunaj območja vrednosti podatkovnega tipa. Takrat govorimo, da je prišlo do ***prekoračitve***[4] (angl. overflow). Če pride do prekoračitve pri računanju s ***predznačenimi*** števili, obnašanje programa ni določeno. Najverjetneje bo rezultat enostavno napačen, v skrajnem primeru se lahko program tudi sesuje. V primeru ***nepredznačenih*** števil pa je rezultat operacije določen. Vedno dobimo pravilen rezultat po modulu 2^n (tj. ostanek pri (Evklidovem) deljenju pravilnega rezultata z 2^n)[5], pri čemer je n bitna dolžina uporabljenega podatkovnega tipa. Na primer:

[4] V tem učbeniku uporabljamo izraz prekoračitev tako za predznačena kot tudi za nepredznačena števila. Na nivoju bitov pa se običajno za prekoračitev pri nepredznačenih številih uporablja pojem ***prenos*** (angl. carry), za prekoračitev pri predznačenih številih pa ***preliv*** (angl. overflow).

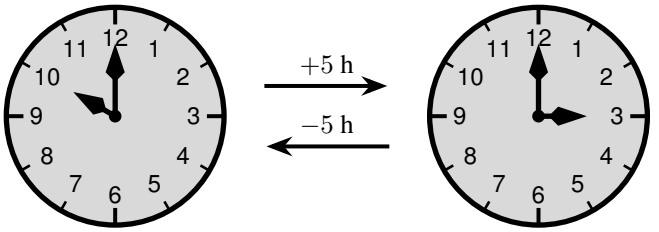
[5] Tak ostanek dobimo, če pravilnemu rezultatu prištevamo (ali odštevamo) vrednost 2^n toliko časa, dokler ne dobimo vrednosti, ki leži med vključno nič in $2^n - 1$.

```
unsigned short x = 40000;
x *= 2;  /* x postane enak 14464 (= 80000 mod 65536).  */
```

Prekoračitev se lahko zgodi tudi v drugo (negativno) smer:

```
unsigned short x = 1;
x -= 4;  /* x postane enak 65533 (= -3 mod 65536).     */
```

V gornjih opombah pomeni `mod` ostanek pri Evklidovem deljenju. Takšnemu sistemu računanja, kjer se števila po tem, ko dosežejo določeno mejno vrednost, vrnejo na izhodiščno vrednost (bodisi največjo bodisi najmanjšo, odvisno od smeri prekoračitve), pravimo ***modularna aritmetika*** (angl. modular arithmetic). Najbolj znan primer modularne aritmetike je računanje z uro. Naslednja slika prikazuje, kaj dobimo, če deseti uri prištejemo pet ur oziroma pet ur odštejemo od tretje ure (tj. $10 + 5 = 3$ in $3 - 5 = 10$):



Naloga 4.3 Za vajo napišite program, ki izpiše toliko členov Fibonaccijevega zaporedja, kolikor je to mogoče. Razmislite, kako bi program sam zaznal, kdaj v postopku računanja je prišlo do prekoračitve.

Naloga 4.4 Brez uporabe operatorja `sizeof` napišite program, ki izpiše število bitov, ki jih zaseda vsak od treh nepredznačenih celoštevilskih tipov, ki smo jih spoznali.

Namig: Nalogo lahko rešite z večkratnim zaporednim množenjem z dve.

Celoštevilske konstante

Celoštevilske konstante v jeziku C običajno zapisujemo v desetiškem zapisu, obstajata pa še ***osmiški*** (angl. octal) in ***šestnajstiški*** (angl. hexadecimal) zapis:

- ***Desetiški zapis*** Konstante v desetiškem zapisu zapisujemo s števkami med nič in devet, vendar se konstanta ne sme začeti z nič:

  ```
  42  1999 13
  ```

- ***Osmiški zapis*** V osmiškem zapisu sestavimo konstanto iz števk med nič in sedem, pri čemer mora biti prva števka vedno nič:

  ```
  052  03717 015
  ```

  Velja opozoriti, da se lahko zgodi, da na prvo mesto vrednosti, za katero želimo, da bi bila desetiška, nehote zapišemo ničlo. Prevajalnik bo seveda takšno vrednost tolmačil kot osmiško. To je napaka, ki jo je izredno težko odkriti.

- ***Šestnajstiški zapis*** V šestnajstiškem zapisu uporabljamo števke med nič in devet ter črke med A in F, ki predstavljajo vrednosti med deset in 15. Uporabljamo lahko tako velike kot tudi male črke. Zapis šestnajstiške konstante se vedno začne z `0x`:

```
0x2A  0x7CF 0xD
```

Osmiški in šestnajstiški zapis sta zgolj dva drugačna načina zapisovanja števil – nobenega vpliva nimata na to, kako se v pomnilniku vrednosti dejansko zapišejo. V gornjih treh primerih imamo tako v pomnilniku vsakokrat zapisane enake tri dvojiške vrednosti (ki predstavljajo desetiške vrednosti 42, 1999 in 13). Celoštevilske konstante so v osnovi tipa `int`. Če pa je vrednost prevelika za zapis v tem tipu, se prevajalnik po določenih pravilih trudi najti ustreznejši celoštevilski tip.

Naj pripomnimo še, da se osmiške vrednosti redko uporabljajo. Po drugi strani so šestnajstiške vrednosti zelo uporabne, saj predstavljajo pregleden kompakten zapis dvojiških vrednosti v pomnilniku – vsak šestnajstiški simbol preprosto nadomesti določeno kombinacijo štirih bitov, kakor kaže naslednja tabela:

Šestn.	Dvoj.	Šestn.	Dvoj.	Šestn.	Dvoj.	Šestn.	Dvoj.
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Naloga 4.5 Pretvorite dvojiško vrednost 11000001 01011010 10101000 01111111 v šestnajstiški zapis. Poleg tega pretvorite šestnajstiško vrednost AFC07511 v dvojiški zapis.

Branje in pisanje celih števil

Spoznali smo že, da pri branju ali pisanju celoštevilskih vrednosti s pomočjo funkcij `scanf` in `printf` uporabljamo formatno določilo `%d`. To formatno določilo uporabljamo le za branje in pisanje predznačenih števil tipa `int`. Za ostale celoštevilske tipe uporabljamo namesto črke d druge črke ali kombinacije črk:

- Za branje oziroma pisanje podatkov tipa `unsigned int` uporabljamo črke u, o ali x, odvisno od tega, ali želimo podatek brati oziroma pisati v desetiški, osmiški ali šestnajstiški obliki. Na primer:

```
unsigned int x;
scanf("%x", &x);  /* Vnesemo šestnajstiško vrednost. Na primer 2A.  */
printf("%u", x);  /* Izpiše isto vrednost v desetiški obliki: 42.   */
```

  V funkciji `printf` lahko v formatnem določilu uporabimo bodisi veliki bodisi mali x. Funkcija bo glede na velikost x-a za izpis šestnajstiške vrednosti ustrezno uporabila velike oziroma male črke.

- Pri branju in pisanju podatka tipa `short` postavimo pred črke d, u, o ali x črko h:

```
short x;
scanf("%hd", &x);
printf("%hd", x);
```

- Pri branju in pisanju podatka tipa `long` postavimo pred črke d, u, o ali x črko l:

```
unsigned long x;
long y;
scanf("%lu%ld", &x, &y);
printf("%lu %ld", x, y);
```

4.2 Realna števila

V cejevskih programih so realna števila shranjena v tako imenovanem zapisu s ***plavajočo vejico*** (angl. floating point). Zapis se imenuje tako zaradi decimalne vejice, ki »plava« levo oziroma desno, odvisno od vrednosti eksponenta, ki nastopa v zapisu. Dva najpogosteje uporabljena realna tipa sta `float` in `double`, ki se razlikujeta po območju vrednosti, predvsem pa po ***natančnosti*** (angl. precision). Sam cejevski standard ne določa natančnosti omenjenih podatkovnih tipov, večina izvedb prevajalnikov pa se ravna po standardu IEEE 754. Ta standard pozna ***enojno natančnost*** (angl. single precision), ki mu v Ceju ustreza podatkovni tip `float`, in ***dvojno natančnost*** (angl. double precision), ki mu v Ceju ustreza podatkovni tip `double`. Naslednja tabela prikazuje natančnost in območje vrednosti obeh tipov po standardu IEEE 754:

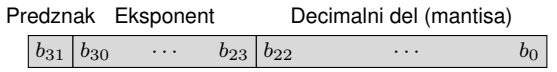
Tip	Bitna dolžina	Najmanjša vrednost	Največja vrednost	Natančnost
`float`	32	$1{,}17549 \times 10^{-38}$	$3{,}40282 \times 10^{38}$	6 des. mest
`double`	64	$2{,}22507 \times 10^{-308}$	$1{,}79769 \times 10^{308}$	15 des. mest

Najmanjše in največje vrednosti so v tabeli podane kot absolutne vrednosti, ki jim lahko dodamo tudi negativen predznak. V zapisu s plavajočo vejico je seveda mogoče zapisati tudi vrednost nič[6].

Iz tabele vidimo, da obsegajo števila, ki jih lahko zapišemo z obema realnima tipoma, izjemno širok razpon vrednosti. V praksi nas zato območje vrednosti realnih tipov ne zanima tako pogosto, kakor nas zanima njihova natančnost. Ta zajema pri tipu `float` le šest desetiških mest, kar je za kakšno resnejše računanje z realnimi števili mnogokrat premalo. Tip `double` nam ponuja precej boljšo natančnost, ki obsega 15 desetiških mest.

Zapis s plavajočo vejico je sestavljen iz ***predznaka*** (angl. sign), ***eksponenta*** (angl. exponent) in ***decimalnega dela*** (angl. fraction). Decimalni del se imenuje tudi ***mantisa*** (angl. mantissa). Število bitov, ki se uporabljajo za eksponent, določa, kako velika (oziroma majhna) števila lahko zapišemo, število bitov decimalnega dela pa določa, kako natančno lahko število zapišemo. V zapisu enojne natančnosti (32 bitov) se uporablja osem bitov za eksponent in 23 bitov za decimalni del. Naslednja slika kaže primer zapisa z 32-bitno plavajočo vejico:

[6]Poleg tega je mogoče zapisati tudi določene posebne vrednosti, kot sta vrednost neskončno ali NaN (not a number, slov. ni številka). Slednja predstavlja neveljavno vrednost, kakršno dobimo kot rezultat neveljavne operacije, kot je na primer kvadratni koren negativnega števila.



Pomen bitov v gornji sliki je naslednji:

- Bit b_{31} predstavlja predznak: če je ta bit enak nič, je predznak pozitiven, sicer je negativen.
- Biti b_{30} do b_{23} (osem bitov) predstavljajo osembitno nepredznačeno vrednost $eksp$, iz katere se izračuna dejanski eksponent (glej enačbo (4.2)).
- Biti b_{22} do b_0 (23 bitov) predstavljajo decimalni del (del za decimalno vejico). Pred te bite se pri izračunu vrednosti vedno doda še en bit (pred decimalno vejico), katerega privzeta vrednost je ena.

Končno realno vrednost izračunamo po naslednji enačbi:

$$vrednost = (-1)^{b_{31}} \times 1{,}b_{22} \cdots b_0 \times 2^{eksp-127}. \tag{4.2}$$

Na primer, iz dvojiškega zapisa 1 10000001 00101000000000000000000 izračunamo desetiško vrednost takole:

$$-1{,}00101 \times 2^{129-127} = -100{,}101_2 = -(2^2 + 2^{-1} + 2^{-3}) = -4{,}625_{10}.$$

Ko mantiso pomnožimo z 2^2 (tj. z $2^{129-127}$), se decimalna vejica (v dvojiškem zapisu) premakne za dve mesti v desno. Pri računanju desetiške vrednosti moramo upoštevati, da biti, postavljeni za decimalno vejico, k skupni vrednosti prispevajo negativne (celoštevilske) potence števila dve. Zato lahko v zapisu s plavajočo vejico brez napake zapišemo le števila, katerih decimalni del lahko zapišemo kot vsoto negativnih celoštevilskih potenc števila dve. Ostala števila se zaokrožijo na vrednosti, ki jih je mogoče zapisati, način zaokroževanja pa je odvisen od izvedbe prevajalnika.

Na primer, s plavajočo vejico ni mogoče brez napake zapisati vrednosti $0{,}4$. O tem se lahko prepričamo, če preizkusimo naslednjo kodo:

```
float x = 0.4;
double y = 0.4;
printf("%d", x == y);  /* Izpiše: 0  */
```

Vrednost izraza `x == y` je enaka nič, kajti primerjamo dva ***različno dobra približka*** vrednosti $0{,}4$.

Če gornjo kodo prevedemo, nas nekateri prevajalniki celo opozorijo, da smo primerjali dve realni števili z operatorjem ==. Zaradi omejene natančnosti pri zapisovanju – in še bolj pri računanju – navadno ni smiselno, da z realnimi števili uporabljamo operatorja enakosti (tj. operatorja == in !=).

Naloga 4.6 Za vajo razmislite in preverite z računalnikom, katere od naslednjih vrednosti je mogoče zapisati v zapisu s plavajočo vejico brez napake: $6{,}1$, $0{,}75$, $0{,}5078125$, $79{,}3125$, $0{,}725$.

Pomoč: Celi del lahko vedno pretvorimo brez napake, zato se osredotočimo zgolj na decimalne dele gornjih vrednosti. Če decimalni del v dvojiškem zapisu množimo z dve, se biti v njem pomaknejo za eno mesto v levo. Pri tem se bit, ki je bil pred množenjem tik za decimalno vejico, premakne tik pred decimalno vejico. Zaradi tega

lahko pretvorimo ***decimalni del*** vrednosti iz desetiškega zapisa v dvojiškega preprosto tako, da ga večkrat zaporedoma množimo z dve. Pri tem vsakokrat na konec dvojiškega zapisa dodamo bit, ki se je po množenju pojavil tik pred decimalno vejico. Pretvorbo lahko izvedemo z uporabo naslednjega algoritma:

```
Ponavljaj, dokler je decimalni del števila različen od nič:
{
  Pomnoži decimalni del z dve;
  Če je zmnožek manjši od ena:
  {
    Na konec dvojiškega zapisa dodaj ničlo;
  }
  sicer:
  {
    Na konec dvojiškega zapisa dodaj enko;
  }
}
```

Opomba: Če se gornji algoritem nikdar ne ustavi, potem vrednosti ni mogoče pretvoriti brez napake. Algoritem se ne bo ustavil, brž ko se začne zaporedje bitov, ki jih dodajamo na konec vrednosti, ki jo pretvarjamo, ponavljati. Na primer, če po gornjem algoritmu pretvorimo vrednost $0{,}45$, dobimo dvojiško vrednost $0{,}01110\overline{0011}$. To vrednost moramo na koncu seveda še ustrezno zamakniti, da dobimo pravilno mantiso.

Realne konstante

Realne konstante se ločijo od celoštevilskih po tem, da vsebujejo decimalno piko. Če je pred ali za decimalno piko samo ničla, potem te ničle ni treba pisati:

- `.125` je isto kakor `0.125`,
- `66.` je isto kakor `66.0`.

Poleg pike lahko uporabimo tudi eksponent, ki predstavlja potenco števila deset. Eksponent označujemo bodisi z veliko bodisi z malo črko e. Na primer:

- `1.2e3` pomeni $1{,}2 \times 10^3$,
- `3.E-4` pomeni $3{,}0 \times 10^{-4}$.

Realne konstante so v Ceju shranjene v zapisu z dvojno natančnostjo. Občasno želimo prevajalnik prisiliti, da shrani konstanto v obliki enojne natančnosti. V tem primeru za konstanto postavimo malo ali veliko črko f. Naslednji primer kaže razliko v številu bajtov, ki ga zasede ena ali druga konstanta:

```
printf("%u", sizeof 5.9);   /* Izpiše: 8  */
printf("%u", sizeof 5.9f);  /* Izpiše: 4  */
```

V drugem primeru zasede konstanta $5{,}9$ zgolj 4 bajte, saj je zapisana z enojno natančnostjo.

Pisanje in branje realnih števil

Za pisanje realnih vrednosti s funkcijo `printf` lahko uporabimo formatni določili `%f` in `%e`. Prvo izpiše realno število v običajni, drugo pa v eksponentni obliki:

```
float x = 0.0071;
printf("%f", x);  /* Izpiše: 0.007100       */
printf("%e", x);  /* Izpiše: 7.100000e-003  */
```

Isti formatni določili uporabimo tako za podatke tipa `float` kot tudi `double`. Pri branju podatkov s funkcijo `scanf` pa moramo v primeru spremenljivke tipa `double` pred črki e in f postaviti še črko l:

```
double x;
scanf("%lf", &x);
scanf("%le", &x);
```

Omeniti velja, da v funkciji `scanf` med črkama e in f ni razlike. Obe prepoznata vnos vrednosti tako v običajni kot tudi v eksponentni obliki.

O natančnosti zapisa realnih števil

V tabeli na strani 52 smo videli, da so števila, zapisana z enojno oziroma dvojno natančnostjo, zagotovo natančna le do šest oziroma 15 desetiških mest. Desetiškim mestom, ki določajo natančnost zapisa, pravimo ***pomembna*** oziroma ***signifikantna mesta*** (angl. significant digits). Tega pojma ne smemo mešati s pojmom ***decimalnih mest***, ki predstavljajo mesta za decimalno vejico. Pomembna mesta so tista številska mesta, katerih prispevek ima glede na dano natančnost pomen. Začnemo jih šteti s prvim mestom, ki je različno od nič. Poglejmo si primer:

```
float h = 6.626070040e-34;  /* Planckova konstanta (J s).           */
float c = 299792458.0;      /* Hitrost svetlobe v vakuumu (m / s).  */
printf("%.43f\n", h);
printf("%.1f\n", c);
```

Če gornjo kodo zaženemo, dobimo takšen izpis:

```
0.0000000000000000000000000000000006626070179
299792448.0
```

Ne glede na število decimalnih mest je v obeh primerih zagotovo pravilnih le šest mest od prvega neničelnega mesta (sedmo mesto je slučajno pravilno). Prispevek naslednjih mest nima več nikakršnega pomena. Seveda, če tip spremenljivk `h` in `c` spremenimo iz `float` v `double`, dobimo pravilni vrednosti:

```
0.0000000000000000000000000000000006626070040
299792458.0
```

Pojem natančnosti je pomemben tudi pri računanju. Poglejmo si primer, kjer želimo sešteti maso Sonca in Lune ter pri tem uporabimo enojno natančnost:

```
float sonce = 1.99e30;  /* kg  */
float luna = 7.35e22;   /* kg  */
float vsota = sonce + luna;
printf("Masa Sonca:  %.10e kg\n", sonce);
printf("Masa Lune:   %.10e kg\n", luna);
printf("Skupna masa: %.10e kg\n", vsota);
```

Vrednosti mase Sonca in Lune v 11-mestnem izpisu (celi del in 10 decimalnih mest) seveda vsebujeta napako[7]:

```
Masa Sonca:  1.9899999468e+030 kg
Masa Lune:   7.3499997919e+022 kg
Skupna masa: 1.9899999468e+030 kg
```

Še pomembnejše opažanje pa je, da se masa Lune v skupni vsoti sploh ne pozna. Masa Lune je za več kot sedem velikostnih razredov manjša od mase Sonca. Prispevek njene mase k masi Sonca je zato na mestih, ki v zapisu mase Sonca z enojno natančnostjo sploh več ne obstajajo. Stanje lahko popravimo, če tipe vseh spremenljivk v gornjem primeru zamenjamo iz `float` v `double`:

```
Masa Sonca:  1.9900000000e+030 kg
Masa Lune:   7.3500000000e+022 kg
Skupna masa: 1.9900000735e+030 kg
```

Naloga 4.7 Za vajo pojasnite, zakaj se naslednja zanka `while` nikoli ne bo ustavila:

```
float x = 1e8;
while (x < 2e8) {
  x = x + 1;
}
```

Če tip spremenljivke `x` zamenjamo iz `float` v `double`, se zanka na koncu ustavi. Zakaj?

Primer programa: logistična preslikava

Logistična preslikava je diferenčna enačba, ki se mnogokrat uporablja kot primer, ki kaže, kako se lahko enostavni nelinearni sistemi začnejo obnašati kaotično. Namen logistične preslikave je, da izračuna, kako se s časom spreminja velikost določene populacije v danem okolju. Preslikavo določa naslednja enačba, ki iz stanja populacije v trenutku n (x_n) izračuna njeno stanje v naslednjem trenutku (x_{n+1}):

$$x_{n+1} = r x_n (1 - x_n). \tag{4.3}$$

Pri tem je x_n vrednost med nič in ena ter predstavlja razmerje velikosti populacije v trenutku n in njene maksimalne velikosti, ki jo razpoložljivi viri še dopuščajo. Enačba se trudi popisati vzajemni učinek rasti populacije, ki je premo sorazmeren z njeno velikostjo, ter učinek upadanja populacije, ki je obratno sorazmeren z viri, ki so na voljo za preživetje.

Najzanimivejši v enačbi (4.3) je parameter r, katerega vrednost navadno spreminjamo med nič in štiri, kar vodi do različnih obnašanj enačbe:

- Ko je r med 0 in 1, bo populacija na koncu izumrla.
- Ko je r med 1 in 2, se bo velikost populacije hitro približala vrednosti $(r-1)/r$.
- Ko je r med 2 in 3, se bo velikost populacije tudi približala vrednosti $(r-1)/r$, vendar bo pred tem nekaj časa nihala okoli te vrednosti.

[7]Napaka se pokaže sicer že na tretjem pomembnem mestu, vendar še vedno velja, da je zapis natančen do šestega pomembnega mesta. Če bi namreč izpis zaokrožili na šest pomembnih mest, bi dobili pravilne vrednosti.

- Ko je r med 3 in $1+\sqrt{6}$ (približno $3{,}44949$), bo velikost populacije iz skoraj vseh začetnih vrednosti začela nihati med dvema vrednostma, ki sta odvisni od r.
- Ko je r med $1+\sqrt{6}$ in približno $3{,}54409$, bo velikost populacije iz skoraj vseh začetnih vrednosti začela nihati med štirimi vrednostmi.
- Ko je r med približno $3{,}54409$ in $3{,}56995$, bo velikost populacije iz skoraj vseh začetnih vrednosti začela nihati med osmimi, 16, 32 itd. vrednostmi.
- Ko postane r približno $3{,}56995$ ali več, postane sistem iz skoraj vseh začetnih pogojev ***kaotičen*** (angl. chaotic) in ne moremo več zaznati nihanj s končno periodo. Poleg tega dobimo ob minimalni spremembi začetnih pogojev čez nekaj časa popolnoma drugačne rezultate. To je tudi matematična definicija kaotičnega sistema.

Napišimo program, ki bo preizkusil obnašanje logistične preslikave:

```
#include <stdio.h>
int main(void) {
  float x = 0.5, r;
  printf("Vnesi r: ");
  scanf("%f", &r);
  for (int i = 1; i <= 100; i++) {
    x = r * x * (1 - x);
    printf("%f\n", x);
  }
  return 0;
}
```

Desna dva stolpca v naslednji tabeli kažeta, kako se spreminja velikost populacije (`x`), če za `r` vnesemo vrednost $3{,}9$:

Iteracija	**Velikost populacije**	
	(`float`)	**(`double`)**
0	0,500000	0,500000
10	0,104013	0,104010
20	0,325655	0,327834
30	0,832821	0,972843
40	0,737961	0,566158
50	0,957548	0,241355
60	0,824760	0,412782
70	0,960597	0,356455
80	0,545069	0,933496
90	0,973831	0,271254
100	0,802888	0,762038

V srednjem stolpcu so vrednosti, ki smo jih dobili z uporabo tipa `float`. Desni stolpec smo dobili, ko smo za računanje uporabili tip `double` (ko uporabite ta tip, ne pozabite v funkciji `scanf` zamenjati formatnega določila `%f` z `%lf`). Ker je sistem kaotičen, smo pri računanju z dvojno natančnostjo dobili popolnoma drugačne rezultate kot pri računanju z enojno natančnostjo, čeprav smo obakrat računali iz iste začetne vrednosti $(0{,}5)$. Vzrok

za to je različna natančnost računanja, ki v vsakem koraku vodi k za spoznanje različnim vrednostim. Ta mala razlika pa je za kaotičen sistem že dovolj, da dobimo na koncu popolnoma drugačne rezultate.

Stavek `x = r * x * (1 - x)` v gornjem programu lahko nadomestimo z naslednjim matematično ekvivalentnim zaporedjem stavkov:

```
x = x * (1 - x);
x *= r;
```

Zdaj ne bomo več presenečeni, če dobimo po določenem številu iteracij spet popolnoma drugačne številke. Vendar s tem še nismo izčrpali vseh možnosti: če program prevedemo z drugim prevajalnikom, spet obstaja verjetnost, da bodo rezultati drugačni. Različni prevajalniki lahko namreč stavke v izvorni kodi prevedejo v sicer matematično ekvivalentna, vendar kljub temu različna zaporedja strojnih ukazov.

Naloga 4.8 Kadar je dinamičen sistem kaotičen, zaradi napak pri kodiranju in računanju nikoli ne moremo dobiti pravih vrednosti njegovega odziva. Če pa je sistem stabilen ali če niha s stabilno amplitudo in frekvenco, majhne napake v natančnosti večinoma niso usodne. Predelajte program za računanje logistične preslikave s strani 57 tako, da bo zaznal stabilno stanje ali stabilna nihanja s periodami dve, štiri, osem, 16, 32, 64 in tako dalje. Iz zapisa na strani 56 je razvidno, da se bo to dogajalo pri vrednostih konstante r do približno $3{,}56995$. Program naj na izhod izpiše dolžino periode, ki jo zazna, če pa je vnesena vrednost konstante r večja ali enaka $3{,}56995$, naj izpiše, da je obnašanje sistema kaotično.

Namig: Preden začne vaš program preverjati dolžino periode, naj izračuna npr. $5\,000$ iteracij odziva, da postane njegova perioda zares stabilna.

4.3 Znaki

Podatke ***znakovnega tipa*** (angl. character type) zapisujemo v jeziku C v spremenljivke podatkovnega tipa `char`. Podatek znakovnega tipa je v resnici celo število, katerega dolžina je vedno natanko en bajt. Standard ANSI C ne določa, ali je znakovni tip predznačen ali nepredznačen. Kot smo že spoznali, so ostali celoštevilski tipi predznačeni, če z njimi ne uporabimo besede `unsigned` (slov. nepredznačen), predznačenost znakovnega tipa pa je odvisna od izvedbe prevajalnika. Če želimo zagotoviti, da bo znakovni tip predznačen, potem moramo z njim uporabiti besedo `signed` (slov. predznačen). Za nepredznačen tip uporabimo besedo `unsigned`.

Podatek znakovnega tipa hrani en znak. Natančneje – podatek znakovnega tipa hrani ***kodo***, ki predstavlja en znak. Standard ANSI C sicer ne določa načina kodiranja znakov, vendar dandanes verjetno ne bomo našli sistema, ki za kodiranje znakov ne bi uporabljal kode ASCII. ***Koda ASCII*** (American standard code for information interchange, slov. ameriški standardni zapis za izmenjavo informacij) predpisuje sedembitno kodo za vsakega od 128 osnovnih znakov. Tako imajo na primer desetiške številke od nič do devet kode od 0110000 do 0111001, velike črke (angleške abecede) od A do Z pa imajo kode od 1000001 do 1011010. Tabelo prvih 128 znakov ASCII z njihovimi desetiškimi kodami najdete v dodatku B.

Konstantne znake v programski kodi zapisujemo v enojnih (in ne dvojnih) navednicah:

```
char znak1 = 'A';   /* Velika črka A                */
char znak2 = ' ';   /* Presledek                    */
char znak3 = '\t';  /* Tabulator (ubežna sekvenca)  */
```

Med enojne navednice moramo postaviti točno en znak, razen v primeru ubežnih sekvenc. Ubežne sekvence so lahko sestavljene iz več znakov, vendar vedno predstavljajo en sam znak.

Ker podatek znakovnega tipa vedno zasede en bajt, bosta klica operatorja `sizeof` v naslednjem delu kode vedno vrnila vrednost ena:

```
char z = '6';
printf("%u", sizeof(char));  /* Izpiše: 1  */
printf("%u", sizeof z);      /* Izpiše: 1  */
```

Po drugi strani je ***dobesedna navedba znaka*** (angl. character literal) vedno tipa `int`, zato na 32-bitnem sistemu operacija `sizeof` nad dobesedno navedbo znaka vrne vrednost štiri:

```
printf("%u", sizeof '0');  /* Izpiše: 4  */
printf("%u", sizeof 'A');  /* Izpiše: 4  */
printf("%u", sizeof ' ');  /* Izpiše: 4 (presledek)  */
```

Desetiška vrednost znaka je enaka desetiški vrednosti njegove kode ASCII:

```
char znak = '0';
printf("%d", znak);  /* Izpiše: 48              */
printf("%d", 'A');   /* Izpiše: 65              */
printf("%d", ' ');   /* Izpiše: 32 (presledek)  */
```

Operacije z znaki

Pravkar smo spoznali, da so znaki v pomnilniku zapisani kot celoštevilske vrednosti. Zato za nas ne bo nobeno presenečenje, da lahko nad podatki znakovnega tipa izvajamo vse operacije, ki jih lahko izvajamo tudi nad celimi števili. Na primer:

```
char znak = 'a';  /* Mala črka a.                                */
char znak = 97;   /* Isto kot prejšnja vrstica: mala črka a.     */
znak += 2;        /* Znak postane mala črka c (koda ASCII 99).   */
znak--;           /* Znak postane mala črka b (koda ASCII 98).   */
```

Ob tem naj povemo še, da so vse desetiške številke ter male in velike črke angleške abecede urejene po vrsti: vsak naslednji znak ima za eno večjo kodo od svojega predhodnika (glej tabelo znakov ASCII v dodatku B).

Znake lahko seveda tudi primerjamo po velikosti. Tako lahko na primer z naslednjim stavkom `if` pretvorimo znak, ki je mala črka, v veliko črko:

```
if ('a' <= zn && zn <= 'z') {
  zn = zn - 'a' + 'A';
}
```

To deluje, ker so velike in male črke v tabeli ASCII urejene po vrsti in se vse male črke razlikujejo od svojih velikih črk za isto vrednost. Vendar to ni vedno najboljši postopek za pretvorbo malih črk v velike (spomnimo se, da standard ANSI C ne zagotavlja kodiranja znakov s kodo ASCII). Naša koda bo bolj prenosljiva, če uporabimo funkcijo `toupper`, ki jo najdemo v knjižnici <ctype.h>:

```
zn = toupper(zn);
```

Funkcija `toupper` preveri, ali je njen argument mala črka, in vrne kodo ustrezne velike črke. V nasprotnem primeru vrne nespremenjeno kodo.

Branje in pisanje znakov

Za branje in pisanje znakov s funkcijama `scanf` in `printf` uporabljamo formatno določilo `%c`:

```
char znak;
scanf("%c", &znak);
printf("%c", znak);
```

Ker je presledek tudi znak, bo gornja funkcija `scanf` v primeru, da smo vtipkali najprej presledek in šele potem kak drug znak, v spremenljivko `znak` shranila presledek. Če želimo, da funkcija preskoči vodilne presledke, potem moramo pred formatno določilo `%c` dodati vsaj en presledek:

```
scanf(" %c", &znak);
```

Naslednji program bere z vhoda besedilo, dokler ne pritisnemo tipke za potrditev. Na koncu na izhod izpiše vtipkano besedilo, v katerem so vse male črke zamenjane z velikimi:

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

int main(void) {
  char znak;
  printf("Vnesi nekaj besedila: ");
  while (1) {
    scanf("%c", &znak);
    if (znak == '\n') break;
    printf("%c", toupper(znak));
  }
  return 0;
}
```

Program deluje takole:

```
Vnesi nekaj besedila: abc+123@(efg)
ABC+123@(EFG)
```

Naloga 4.9 Za vajo napišite program, ki s tipkovnice prebere poljubno besedilo (zaključeno s tipko za potrditev) in izpiše vneseno besedilo brez presledkov in ločil. Pri pisanju programa si pomagajte s funkcijama `isspace` in `ispunct`. Obe funkciji kot argument sprejmeta en znak. Prva funkcija vrne neničelno vrednost, če je znak presledek (angl. space) ali katerikoli drug znak, ki se prikaže kot prazen prostor. Druga funkcija vrne neničelno vrednost, če je znak ločilo (angl. punctuation). Vsaka od obeh funkcij vrne vrednost nič v primeru, da znak ne ustreza kriteriju, ki ga funkcija preverja. Funkciji se nahajata v knjižnici <ctype.h>.

Primer delovanja programa:

```
Vnesi nekaj besedila: Da vidimo, kako to deluje.
Davidimokakotodeluje
```

Težava pri kombiniranem branju znakov in števil

Bodite pozorni, kadar v istem programu za branje podatkov s funkcijo `scanf` uporabljate formatno določilo `%c` v kombinaciji s katerim od drugih formatnih določil:

```
printf("Vnesi celo število: ");
scanf("%d", &x);
printf("Vnesi znak: ");
scanf("%c", &znak);
```

V gornjem programu prvi klic funkcije `scanf` v medpomnilniku pusti za seboj vse, česar ne more prebrati, vključno z znakom `'\n'`. Drugi klic funkcije `scanf` bo tako prebral naslednji znak, ki je ostal v medpomnilniku. V najboljšem primeru je to znak `'\n'`, nikakor pa ne znak, ki naj bi ga vnesli po naslednjem pozivu. Težavo lahko rešimo takole:

```
printf("Vnesi celo število: ");
scanf("%d", &x);
do {
  scanf("%c", &znak);  /* Počisti, kar je ostalo v medpomnilniku.  */
} while (znak != '\n');
printf("Vnesi znak: ");
scanf("%c", &znak);
```

Namesto zanke `do...while`, ki smo jo dodali za prvim klicem `scanf`, lahko za isti učinek uporabimo tudi naslednji stavek, ki smo ga srečali že v programu na strani 10:

```
while (getchar() != '\n') {}  /* Izplakne vhodni tok.  */
```

Naloga 4.10 Za vajo napišite program, ki uporabniku omogoča, da vnaša bodisi kode ASCII bodisi znake. Program naj za vsako vneseno kodo prikaže ustrezen znak oziroma za vsak vnesen znak izpiše njegovo kodo ASCII.

Primer delovanja programa:

```
Vnesi ukaz (0: izhod, 1: ASCII-->znak, 2: znak-->ASCII): 1
Vnesi kodo ASCII: 125
Koda 125 predstavlja znak }
Vnesi ukaz (0: izhod, 1: ASCII-->znak, 2: znak-->ASCII): 2
Vnesi znak: 8
Znak 8 ima kodo 56.
Vnesi ukaz (0: izhod, 1: ASCII-->znak, 2: znak-->ASCII): 0
Nasvidenje!
```

Opomba: Za branje celoštevilske vrednosti s funkcijo `scanf` uporabite spremenljivko tipa `int`. Isto spremenljivko lahko potem uporabite za izpis ustreznega znaka s funkcijo `printf` in formatnim določilom `%c`. Formatno določilo `%c` v funkciji `printf` sprejme namreč argument, katerega tip je lahko bodisi `char` bodisi `int`. Konec koncev tudi konstantni znaki pripadajo tipu `int`.

4.4 Pretvorbe podatkovnih tipov

Kadar operandi v aritmetičnih izrazih nimajo istega tipa, je računalnik pred izvajanjem operacije primoran v določene pretvorbe. Te pretvorbe se večinoma izvajajo avtomatično in zanje pogosto ni treba vedeti, ker ne vplivajo pomembno na izid računskih operacij. Pretvorbe, ki spadajo v to kategorijo, vedno pretvorijo podatkovni tip z manjšim območjem vrednosti in/ali manjšo natančnostjo v podatkovni tip z večjim območjem vrednosti in/ali večjo natančnostjo. Takšni pretvorbi pravimo tudi ***napredovanje*** (angl. promotion). Na

primer, če seštevamo podatek tipa `double` s podatkom tipa `float`, potem se mora pred seštevanjem podatek tipa `float` pretvoriti v `double`. Prav tako se v računskih operacijah podatek tipa `char` pretvori v podatek tipa `int`. (Kot že vemo, so konstantni znaki že v osnovi tega tipa.)

Obstajajo pa primeri, ko takšna pretvorba ni možna, zaradi česar lahko pride do delne izgube informacije ali celo nesmiselnih rezultatov. Poglejmo si nekaj tipičnih primerov:

- Kadar v istem izrazu nastopata ***predznačena in nepredznačena*** vrednost, se predznačena vrednost pretvori v nepredznačeno. Pri tem se uporablja načelo modularne aritmetike, na katero smo naleteli že pri prekoračitvi območja pri računanju z nepredznačenimi števili na strani 49. Tako se na primer predznačena vrednost -3 pretvori v vrednost $4\,294\,967\,293$ (tj. $-3 \bmod 2^{32}$)[8], kadar se kombinira z 32-bitno nepredznačeno vrednostjo:

  ```
  unsigned int x = -3;  /* Napaka: x postane 4294967293.       */
  unsigned int y = 10;
  y += x;  /* y postane (10 + 4294967293) mod 4294967296 = 7,
              kar je pravilen rezultat.                        */
  ```

  V gornjem primeru je prišlo v resnici dvakrat do prekoračitve območja: v prvi vrstici v negativno smer, v tretji vrstici pa nazaj v pozitivno smer. Zato je končni rezultat pravilen.

  V naslednjem primeru pa je ne odnesemo tako poceni:

  ```
  unsigned int y = 10;
  printf("%d", y < -3);  /* Primerjata se 10 in 4294967293,
                            zato printf izpiše 1.            */
  ```

  Zaradi takšnih pasti je najbolje, da nepredznačene celoštevilske tipe uporabljamo samo takrat, ko je to res treba. V vsakem primeru pa se izogibamo kombiniranju predznačenih in nepredznačenih vrednosti v istem izrazu.

- Pravilo iz prejšnje točke se uporablja tudi za vrednosti, ki za ***večkrat presegajo območje nepredznačenega zapisa***, in sicer tako v pozitivno kot tudi v negativno smer. Na primer:

  ```
  unsigned char x = 10000;  /* x postane 16 (10000 mod 256).  */
  unsigned char x = -917;   /* x postane 107 (-917 mod 256).  */
  ```

- V ostalih primerih standard ne določa, kaj se zgodi, če leži izvorna vrednost ***zunaj območja vrednosti***, ki jih lahko zapišemo. Pri pretvorbi celoštevilske vrednosti v predznačeno celo število je obnašanje odvisno od izvedbe prevajalnika. Pri ostalih pretvorbah pa obnašanje programa ni določeno:

  ```
  short x = 1000000;       /* Izid odvisen od prevajalnika.  */
  int i = 1e30;            /* Nedoločeno obnašanje.          */
  unsigned short = 1e10;   /* Nedoločeno obnašanje.          */
  float f = 1e200;         /* Nedoločeno obnašanje.          */
  ```

[8] Ker so negativna števila praktično vedno zapisana v dvojiškem komplementu, ta pretvorba niti ni tako nenavadna – če namreč izračunamo ostanek pri deljenju negativne desetiške vrednosti z 2^n, dobimo enako vrednost, kot če bi zadnjih n bitov originalnega dvojiškega zapisa te negativne vrednosti enostavno tolmačili kot n-bitno nepredznačeno število.

- Kadar ***celoštevilski spremenljivki prirejamo realno vrednost***, katere decimalni del je različen od nič (celi del pa dovolj majhen, da ga lahko zapišemo), se nujno izgubi nekaj informacije: odreže se del vrednosti za decimalno piko:

```
int x;
x = 2.156;   /* x postane enak 2.   */
x = -2.156;  /* x postane enak -2.  */
```

- Najmanj problematičen je primer, ko spremenljivki tipa ***`float`*** prirejamo vrednost tipa ***`double`***, ki je dovolj majhna, da jo lahko zapišemo v zapisu z enojno natančnostjo. V takem primeru se utegne izgubiti nekaj natančnosti, saj se izvede pretvorba iz dvojne v enojno natančnost. Kadar natančnost ni zelo pomembna, to niti ni težava. Način zaokroževanja pri pretvorbi je odvisen od izvedbe prevajalnika.

Na srečo nas prevajalnik navadno opozori, kadar je prisiljen izvesti katero od zgoraj omenjenih pretvorb, kjer lahko pride do izgube natančnosti ali celo do popolnoma napačnih rezultatov.

Zahtevana pretvorba tipa

Čeprav jezik C veliko pretvorb med podatkovnimi tipi izvaja avtomatično in prikladno, pa včasih vseeno želimo vrsto pretvorbe določiti sami. Pretvorbo lahko zahtevamo z ***operatorjem za pretvorbo tipa*** (angl. cast operator). Operator je sestavljen iz para oklepajev, v katerega zapišemo želeni tip, vse skupaj pa postavimo pred izraz, katerega tip želimo pretvoriti:

```
( ime-tipa ) izraz
```

Takšna pretvorba je seveda samo začasna.

Na primer, če se želimo izogniti opozorilu prevajalnika ob prirejanju konstante tipa `double` spremenljivki tipa `float`, lahko pretvorbo tipa zahtevamo sami:

```
float pi = (float) 3.1416;
```

Če smo se sami odločili za pretvorbo, nas prevajalnik o njej ne bo opozarjal. Kot že vemo, pa lahko v tem primeru isti učinek dosežemo tudi tako, da na koncu konstante dodamo malo ali veliko črko f:

```
float pi = 3.1416f;
```

Z operatorjem za pretvorbo tipa lahko dosežemo tudi realno deljenje celoštevilskih spremenljivk:

```
int x = 1, y = 2;
printf("%f", (double) x / y);  /* Izpiše: 0.500000  */
```

Operator za pretvorbo tipa ima prednost pred operatorjem deljenja, zato se v realen tip pretvori zgolj spremenljivka `x`. Vendar to ni težava, saj operator deljenja vrne realen rezultat, kakor hitro je vsaj eden od njegovih operandov realno število.

Za konec tega razdelka je tu še en primer uporabe operatorja za pretvorbo tipa. Naslednji del kode izpiše decimalni del realnega števila, ki je shranjen v spremenljivki `x`:

```
float x = 13.671;
printf("%f", x - (int) x);  /* Izpiše: 0.671000  */
```

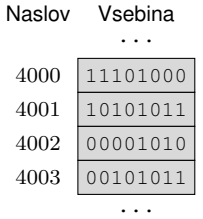
Kot smo v tem poglavju že spoznali, pretvorba iz realne v celoštevilsko vrednost deluje tako, da enostavno odreže del za decimalno piko. Tako se v gornjem delu programa od celotne vrednosti spremenljivke x odšteje njen celi del.

4.5 Kazalci

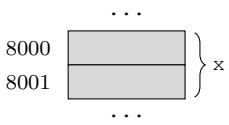
Kazalci (angl. pointer) so eden najpomembnejših (in hkrati najpogosteje napačno razumljenih) elementov jezika C. Zato se jih bomo lotili postopoma in v več poglavjih.

Pomnilnik in naslovni operator

Prvi korak k razumevanju kazalcev je, da razumemo, kaj kazalci predstavljajo na strojnem nivoju. Omenili smo že, da je v večini sodobnih računalnikov pomnilnik razdeljen na enote po osem bitov, ki jim pravimo ***bajti*** (angl. byte). Vsakemu bajtu pripada unikaten ***naslov*** (angl. address), po katerem ga ločimo od preostalih bajtov v pomnilniku. Naslov si predstavljamo preprosto kot zaporedno številko bajta. Naslednja slika kaže primer izseka iz pomnilnika, kjer vidimo štiri bajte s pripadajočimi naslovi:



Izvršilni program v pomnilniku vsebuje tako kodo (strojne ukaze), ki ustreza stavkom v izvorni kodi, kot tudi podatke, ki ustrezajo spremenljivkam. Vsaka spremenljivka, ki jo uporabimo v programu, zasede enega ali več bajtov. Velja dogovor, da je ***naslov spremenljivke*** enak naslovu prvega od bajtov, ki pripadajo tej spremenljivki. Na primer, če je spremenljivka x sestavljena iz dveh bajtov na naslovih 8000 in 8001, potem je naslov spremenljivke x enak 8000:



Doslej nas naslovi spremenljivk v pomnilniku niso zanimali, saj prevajalnik sam poskrbi za dostop do delov pomnilnika, ki so rezervirani za deklarirane spremenljivke. Z uporabo ***naslovnega operatorja*** (angl. address operator) pa lahko pridemo tudi do naslova spremenljivke. Naslovni operator (&) je unarni operator, ki ga postavimo pred ime spremenljivke:

```
&x  /* Vrne naslov spremenljivke x.  */
```

Tako smo prišli do definicije pojma kazalca: ***kazalec*** je preprosto ***pomnilniški naslov***.

Tehnično gledano je kazalec celo število, vendar območje njegovih vrednosti ne sovpada nujno s standardnimi celoštevilskimi tipi. Zato obstaja poseben ***kazalčni tip*** (angl.

pointer type), ki mu pripadajo vsi kazalci. Kadar želimo vrednost kazalca izpisati s pomočjo funkcije `printf`, jo navadno pretvorimo v nepredznačeno celoštevilsko vrednost:

```
printf("%u", (unsigned int) &x);  /* Izpis v desetiški obliki.       */
printf("%x", (unsigned int) &x);  /* Izpis v šestnajstiški obliki.   */
```

Kazalec oziroma naslov lahko seveda shranimo tudi v spremenljivko, ki pa mora biti kazalčnega tipa.

Kazalčna spremenljivka

Spremenljivki, v katero lahko shranimo kazalec oziroma naslov, pravimo ***kazalčna spremenljivka*** (angl. pointer variable). Ker imamo od kazalcev daleč najpogosteje opravka s kazalčnimi spremenljivkami, takim spremenljivkam na kratko rečemo kar kazalci. Kadar hrani kazalec naslov spremenljivke `x`, pravimo, da kazalec ***kaže*** na `x`.

Tako kot vsako spremenljivko moramo tudi kazalčno spremenljivko deklarirati. Deklaracija kazalčne spremenljivke se od običajne razlikuje po zvezdici, ki jo postavimo pred ime spremenljivke:

ime-tipa `*` *ime-kazalčne-spremenljivke* `;`

Podatkovni tip, ki ga uporabimo pri deklaraciji kazalčne spremenljivke, ne predstavlja tipa podatka, ki ga ta spremenljivka hrani (ta je vedno celo število), temveč tip podatka, na katerega ta kazalčna spremenljivka kaže. Takemu tipu pravimo tudi ***sklicevan tip*** (angl. referenced type). Zaradi enostavnosti bomo sklicevanemu tipu pogosto rekli kar tip kazalčne spremenljivke oziroma tip kazalca. Kazalčna spremenljivka lahko kaže samo na spremenljivko, katere tip je enak njenemu sklicevanemu tipu.

Na primer, naslednje tri deklaracije ustvarijo kazalčne spremenljivke tipov `float *`, `int *` in `unsigned char *`, ki lahko po vrsti kažejo na objekte tipov `float`, `int` in `unsigned char`:

```
                        /* Lahko kaže samo na objekt tipa ...  */
float *kf;              /* ... float.                          */
int *ki;                /* ... int.                            */
unsigned char *kuc;     /* ... unsigned char.                  */
```

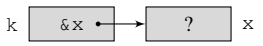
Izraz ***objekt*** v tem primeru pomeni kakršenkoli del pomnilnika, ki je rezerviran za hranjenje podatkov. Kazalec lahko namreč kaže tudi na del pomnilnika, ki ne pripada nujno deklarirani spremenljivki.

Omenimo še, da ni nobene omejitve glede tipa, ki ga lahko izberemo za sklicevani tip kazalčne spremenljivke. Tako lahko ustvarimo celo kazalčno spremenljivko, ki kaže na kazalec.

S tem ko smo deklarirali kazalčno spremenljivko, smo v pomnilniku zanjo rezervirali prostor, nismo pa je še usmerili na noben določen objekt v pomnilniku. Pomembno je, da kazalčno spremenljivko pred uporabo inicializiramo (tj. jo usmerimo na določen objekt v pomnilniku):

```
float x, *k;  /* k ne kaže na noben določen objekt.  */
k = &x;       /* k zdaj kaže na x.                   */
```

V drugi vrstici kode smo naslov spremenljivke `x` shranili v kazalčno spremenljivko `k`, s čimer smo dosegli, da `k` kaže na `x`. Naslednja slika kaže stanje v pomnilniku, ko se izvedeta gornji dve vrstici kode:



Pravokotnika na sliki predstavljata prostor v pomnilniku, ki smo ga rezervirali za spremenljivki `k` in `x`. Vanju smo vpisali vrednosti obeh spremenljivk: spremenljivka `k` hrani naslov spremenljivke `x`, vrednost spremenljivke `x` pa ni določena (označeno z vprašajem). Puščica nakazuje dejstvo, da kazalčna spremenljivka `k` kaže na spremenljivko `x`.

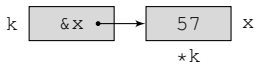
Operator posredovanja

Ko kazalčna spremenljivka enkrat kaže na določen objekt, lahko prek nje z uporabo ***operatorja posredovanja*** (angl. indirection operator) pridemo do vsebine tega objekta. Operator posredovanja zapisujemo s simbolom zvezdica (*, angl. asterisk). Tako kot naslovni operator je tudi operator posredovanja unarni operator, ki ga postavimo pred ime spremenljivke.

Vzemimo za primer kazalčno spremenljivko `k`, ki kaže na spremenljivko `x`, ki je tipa `short` in ima vrednost 57. Takšno stanje lahko dosežemo z naslednjo kodo:

```
short x, *k;
x = 57;
k = &x;
```

V gornji kodi še nismo uporabili operatorja posredovanja. Zvezdica v prvi vrstici je del deklaracije in pomeni, da je `k` kazalčna spremenljivka. Naslednja slika prikazuje stanje v pomnilniku, ko se izvede gornja koda:



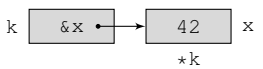
Pod pravokotnik z imenom `x` smo zapisali še izraz `*k`, ki ponazarja, da nam operator posredovanja (`*`) s kazalcem `k` omogoča dostop do spremenljivke `x`. Zato lahko vrednost spremenljivke `x` z uporabo kazalca `k` in operatorja posredovanja (`*`) izpišemo takole:

```
printf("%hd", *k);  /* Izpiše: 57  */
```

Z istim operatorjem lahko prek kazalca `k` vrednost spremenljivke `x` tudi spremenimo:

```
*k = 42;  /* Vrednost spremenljivke x je zdaj 42.  */
```

Ko se izvede zadnja vrstica kode, imamo v pomnilniku takšno stanje:



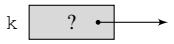
Iz zadnjih dveh primerov smo videli, da je izraz `*k` po učinku skoraj[9] popolnoma enakovreden izrazu `x` (ob predpostavki, da `k` kaže na `x`). Spomnimo se še pojma *leva vrednost* (glej stran 18), ki pomeni objekt v pomnilniku, ki se lahko nahaja na levi strani priredilnega operatorja. Ugotovimo lahko, da predstavlja kazalec z operatorjem posredovanja veljavno levo vrednost.

[9] Ker imajo nekateri operatorji višjo prednost kakor operator posredovanja, moramo včasih za enakovreden učinek uporabiti oklepaje. Na primer, če želimo prek kazalca `k` povečati vrednost spremenljivke `x` za ena, lahko to storimo z izrazom `(*k)++`.

Operatorja posredovanja nikoli ne smemo uporabiti na ***neinicializirani kazalčni spremenljivki*** (angl. uninitialized pointer variable). Posledice takšnega dejanja so nedoločene: od tega, da dobimo nesmiselne rezultate, do tega, da se program sesuje. Na primer:

```
int *k;
*k = 13;  /* Napaka: uporaba neinicializiranega kazalca.  */
```

V prvi vrstici gornje kode smo rezervirali prostor za kazalčno spremenljivko `k`, vendar ji nismo priredili nobene vrednosti. Naslednja slika kaže stanje v pomnilniku:



Ker kazalčni spremenljivki `k` nismo priredili nobene določene vrednosti, kaže `k` na nedoločen objekt v pomnilniku. V drugi vrstici gornje kode (`*k = 13;`) skušamo z operatorjem posredovanja zapisati vrednost 13 na mesto, na katero kaže `k`. Ker je to mesto nedoločeno, je tudi rezultat operacije nedoločen.

Na srečo nas prevajalnik navadno opozori, kadar skušamo na tak način uporabiti neinicializirano kazalčno spremenljivko.

Ničelni kazalec

Pred uporabo neinicializiranega kazalca se lahko zavarujemo tudi z uporabo ***ničelnega kazalca*** (angl. null pointer), ki po definiciji ne kaže nikamor. Ničelni kazalec je kazalec z vrednostjo NULL, ki jo lahko enoumno ločimo od vrednosti kakršnegakoli veljavnega kazalca. Vrednost NULL je določena z makrom v zglavni datoteki <stdlib.h>. Ker je vrednost kazalca celoštevilska vrednost, lahko kazalce med seboj primerjamo z operatorjema enakosti == in !=. Tako lahko preprečimo uporabo neinicializiranega kazalca na primer takole:

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

int main(void) {
  int *k = NULL;
  //...
  if (k == NULL) {  /* Če kazalec ne kaže nikamor.  */
    printf("Napaka: neinicializiran kazalec.");
  }
  else {
    *k = 20;
  }
  //...
  return 0;
}
```

V gornjem programu priredimo objektu, na katerega kaže kazalec `k`, vrednost 20 le v primeru, da se je kazalec pred tem primerno inicializiral. Če se to ni zgodilo, potem ima kazalec v trenutku preverjanja vrednost NULL, ki smo mu jo priredili ob njegovi deklaraciji. Na ta način smo preprečili uporabo neinicializiranega kazalca.

Vrednost NULL uporabljamo zgolj za preverjanje, ali je kazalec inicializiran. Uporaba ničelnega kazalca z operatorjem posredovanja ima prav tako nedoločene posledice kot uporaba kakršnegakoli neinicializiranega kazalca:

```
int *k = NULL;
*k = 13;  /* Nedoločeno obnašanje.  */
```

Naj opozorimo, da je zvezdica v prvi vrstici gornje kode del deklaracije, in ne operator posredovanja. Zato v prvi vrstici zapisujemo vrednost NULL v kazalčno spremenljivko. Naslednja, nekoliko daljša koda ima isti učinek:

```
int *k;
k = NULL;
*k = 13;  /* Nedoločeno obnašanje.  */
```

Prirejanje kazalcev

Vrednosti kazalcev lahko s pomočjo priredilnega operatorja kopiramo. Pri tem moramo paziti, da sta tipa kazalcev na obeh straneh priredilnega operatorja enaka. Vzemimo za primer naslednjo deklaracijo:

```
long x, *p, *q;
```

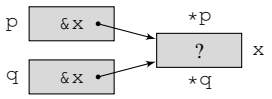
Priredilni stavek:

```
p = &x;
```

je primer prirejanja kazalcev: naslov spremenljivke `x` se priredi kazalčni spremenljivki `p`. Pri tem je pomembno, da je tip kazalca `&x` enak tipu kazalčne spremenljivke `p` (oba sta tipa `long *`). Tu je še en primer prirejanja kazalcev:

```
q = p;
```

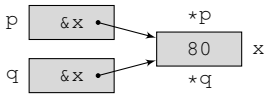
Ta stavek kopira naslov, shranjen v kazalčni spremenljivki `p`, v kazalčno spremenljivko `q`. Zdaj obe kazalčni spremenljivki kažeta na isto spremenljivko (tj. `x`), kot lahko to vidimo na naslednji sliki:



Vprašaj na gornji sliki označuje dejstvo, da spremenljivka `x` nima določene vrednosti. Ker tako `p` kot tudi `q` oba kažeta na `x`, lahko pridemo do spremenljivke `x` prek katerekoli od obeh kazalčnih spremenljivk:

```
*p = 85;      /* x postane 85.        */
*q = *q - 5;  /* Zdaj je x enak 80.   */
```

Po izvedenih zadnjih dveh vrsticah kode imamo v pomnilniku naslednjo sliko:



Bodite pozorni, da ne zamešate stavka, ki smo ga srečali v zadnjem primeru:

```
q = p;
```

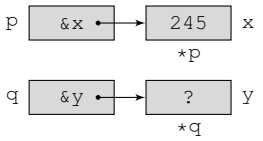
s stavkom:

```
*q = *p;
```

Prvi od zadnjih dveh stavkov predstavlja prirejanje kazalcev, drugi pa ne. Naslednji primer ilustrira, v čem je ta drugi stavek drugačen:

```
long x, y, *p, *q;
p = &x;
q = &y;
x = 245;
```

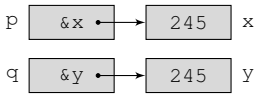
Naslednja slika kaže stanje v pomnilniku, ko se izvede gornji del kode:



Stavek:

```
*q = *p;
```

zdaj povzroči, da se vrednost 245 kopira v spremenljivko y. Zadnji stavek namreč objektu, na katerega kaže q, priredi vrednost objekta, na katerega kaže p. Zato imamo zdaj v pomnilniku naslednje stanje:



Naloga 4.11 Za vajo razmislite, kakšen učinek bo imel zadnji stavek v naslednjem delu kode. Utemeljite odgovor.

```
int *k1, *k2;
//...
k1 = k2;
*k1 = *k2;
```

Naloga 4.12 Bodite pozorni, da ne zamešate zvezdice, ki predstavlja operator posredovanja, z zvezdico, ki v času deklaracije označuje, da gre za kazalčno spremenljivko.

Pojasnite, kakšna je razlika med naslednjima dvema vrsticama kode:

```
int *k = (int *) 0xC0F22001;
*k = 0xC0F22001;
```

Razmislite še, kako se ti dve vrstici kode:

```
k1 = &x;
k2 = k1;
```

razlikujeta od naslednjih dveh vrstic:

```
int *k1 = &x;
int *k2 = k1;
```

4.6 Naloge

Naloga 4.13 Podana je naslednja koda:

```
printf ("%hd\n", (short) 0xB09F);
printf ("%hu\n", (unsigned short) 0xB09F);
```

Kaj bo izpisal vsak od gornjih klicev funkcije `printf`? Utemeljite odgovor.

Naloga 4.14 V pomnilniku je shranjena osembitna vrednost 10011011, ki predstavlja osembitno predznačeno celo število. Vrednost bi radi pretvorili v 16-bitno predznačeno celo število. Kaj moramo dodati gornjim osmim bitom, da bo desetiška vrednost tako dobljenega 16-bitnega podatka ostala nespremenjena?

Ali je nalogo mogoče rešiti brez pretvarjanja vrednosti v desetiško?

Naloga 4.15 Z uporabo spremenljivk tipa `unsigned int` napišite program, ki izračuna in izpiše prvih 80 členov Fibonaccijevega zaporedja.

Pomoč: Nalogo rešite tako, da programsko »zlepite« po dve in dve spremenljivki tipa `unsigned int` in tako dobite daljše spremenljivke dolžine 64 bitov. Najlažje bo, če omejite vrednosti v posameznih spremenljivkah na vrednosti med nič in $10^9 - 1$ (glej nalogo 5.2 na strani 86).

Naloga 4.16 Podan je naslednji program:

```
#include <stdio.h>
int main(void) {
  unsigned int x = 217;
  while (x > 0) {
    if (x > 0x7FFFFFFF) {  /* Predpostavimo, da je int 32-biten.  */
      printf("1");
    }
    else {
      printf("0");
    }
    x *= 2;
  }
  return 0;
}
```

Kaj bo program izpisal? Utemeljite odgovor.

Naloga 4.17 Napišite program, ki bo rešil naslednjo nalogo:

Štirje gusarji si morajo razdeliti poln zaboj zlatnikov. Prvi vzame najprej en zlatnik in takoj zatem še natanko četrtino preostalih zlatnikov. Nato to ponovijo še ostali trije. Na koncu ostane v skrinji še en zlatnik in še toliko, da si jih lahko razdelijo na enake dele. Najmanj koliko zlatnikov je bilo v skrinji?

Naloga 4.18 Napišite program, ki z uporabo Newtonovega postopka izračuna kvadratni koren pozitivnega realnega števila.

Primer delovanja programa:

```
Vnesi pozitivno realno število: 8
Kvadratni koren vnesenega števila: 2.828427
```

Pomoč: Predpostavimo, da je uporabnik vnesel vrednost x. Newtonov postopek potrebuje začetno vrednost, iz katere začne računati. Naj bo ta vrednost $y = 1$. Postopek v vsakem koraku izračuna povprečje vrednosti y in x/y. Dobljeno vrednost priredi spremenljivki y in postopek se ponovi. Naslednja tabela prikazuje, kako se spreminja vrednost spremenljivke y, če je $x = 8$:

y	x/y	Povprečje vrednosti y in x/y
1,000000	8,000000	4,500000
4,500000	1,777778	3,138889
3,138889	2,548673	2,843781
2,843781	2,813156	2,828469
2,828469	2,828386	2,828427
2,828427	2,828427	2,828427

Iz tabele se vidi, kako se y postopoma približuje vrednosti kvadratnega korena iz osem. Program naj preneha z računanjem takrat, ko bo absolutna vrednost razlike dveh zaporednih vrednosti spremenljivke y manjša od produkta $0{,}00001$ in y:

$$|y_{i+1} - y_i| < 0{,}00001 y_i$$

Za računanje uporabite spremenljivke z dvojno natančnostjo.

Naloga 4.19 Podan je naslednji program:

```c
#include <stdio.h>
int main(void) {
  double x = 1;
  do {
    x -= 0.1;
    printf("%.1f\n", x);
  } while (x != 0);
  return 0;
}
```

Pojasnite, zakaj se program ne ustavi, ko postane x enak nič.

Naloga 4.20 Podan je naslednji program:

```c
#include <stdio.h>
#include <math.h>
int main(void) {
  for (int i = 0; i < 5; i++) {
```

```
    printf("%5d\n", (int) pow(10, i));
  }
  return 0;
}
```

Ko program zaženemo, dobimo naslednji izpis:

```
    1
   10
   99
 1000
 9999
```

Pojasnite, zakaj so nekatere vrednosti napačne.

Namig: Standardna funkcija `pow`, ki računa potence, vrne vrednost tipa `double`.

Naloga 4.21 Napišite program, ki z vhoda bere poljubne izraze in na izhod sporoči, če se oklepaji in zaklepaji v izrazu ne ujemajo. Program naj se konča, ko vnesemo prazen izraz (tj. zgolj pritisnemo potrditveno tipko).

Primer delovanja programa:

```
Vnesi izraz (prazen izraz konča): (1 + 2) * 3
Ni napak.
Vnesi izraz (prazen izraz konča): (5 - (3 - 9))
Ni napak.
Vnesi izraz (prazen izraz konča): (13 - 2)) / 9
Napaka: zaklepaj na 9. mestu.
Vnesi izraz (prazen izraz konča): ((x + y) * (x - y)
Napaka: manjka zaklepaj.
Vnesi izraz (prazen izraz konča): 3 * (1 - y)) / (1 - x
Napaka: zaklepaj na 12. mestu.
Vnesi izraz (prazen izraz konča):
Nasvidenje!
```

Naloga 4.22 Napišite program, ki z vhoda prebere ime in priimek ter izpiše najprej priimek, potem vejico in na koncu začetnico imena s piko.

Primer delovanja programa:

```
Vnesi ime in priimek: Julio Iglesias
Iglesias, J.
```

Naloga 4.23 Napišite program, ki z vhoda prebere poljubno veliko naravno število in na izhod izpiše podatek o tem, s katerimi števili od dve do deset je to število deljivo. Deljivosti s številom sedem ni treba preverjati.

Primer delovanja programa:

```
Vnesi naravno število: 6125781625632976591276591263463246666 2
Vneseno število ...
... je deljivo z 2.
... je deljivo s 3.
... ni deljivo s 4.
... ni deljivo s 5.
... je deljivo s 6.
... ni deljivo z 8.
```

```
... ni deljivo z 9.
... ni deljivo z 10.
```

Pogoji za deljivost števila s/z:

- dve – zadnja števka mora biti deljiva z dve;
- tri – vsota števk mora biti deljiva s tri;
- štiri – dvomestno število iz zadnjih dveh števk mora biti deljivo s štiri;
- pet – zadnja števka mora biti bodisi nič bodisi pet;
- šest – število mora biti deljivo z dve in s tri;
- sedem – trimestno število iz zadnjih treh števk mora biti deljivo z osem;
- devet – vsota števk mora biti deljiva z devet;
- deset – zadnja števka mora biti nič.

Pomoč: Ker je število preveliko, da bi ga zapisali kot število tipa `unsigned long`, boste z vhoda namesto števila brali niz znakov. Paziti morate, ker je koda ASCII vsakega znaka, ki predstavlja desetiško števko, za `'0'` (oziroma 48) večja od dejanske desetiške vrednosti te iste števke. Zato morate od vsake prebrane števke to kodo odšteti, da dobite desetiško vrednost, ki jo števka predstavlja.

Naloga 4.24 Podan je naslednji del kode:

```
float x, y, *p, *q;
p = &x;
y = 10;
```

Za vsakega od naslednjih izrazov ugotovite, ali vrne vrednost ali naslov. V nekaterih izrazih se spremenijo vrednosti določenih spremenljivk. Katerih in na kakšen način? V nekaterih izrazih gre za uporabo neinicializiranega kazalca. V katerih? Utemeljite svoje odgovore.

```
(a) *p = y
(b) *p + 10
(c) *p = *q
(č) &q
(d) *q = *p
(e) q == p
(f) q = p
(g) (*p)++
```

Naloga 4.25 Z uporabo naslovnega operatorja, operatorja za pretvorbo tipa in operatorja posredovanja lahko iz pomnilnika preberemo zapis realnega števila s plavajočo vejico in ga izpišemo v šestnajstiški obliki na naslednji način:

```
float x = 12.9;
/* Predpostavimo, da sta int in float oba 32-bitna.  */
printf("%X", *(unsigned int *)&x);
```

Gornji del kode bo izpisal šestnajstiško vrednost 414E6666.

Napišite program, ki bo s tipkovnice prebral realno vrednost in v dvojiškem zapisu izpisal predznak, eksponent in mantiso števila, kakor je le-to zapisano v zapisu z 32-bitno plavajočo vejico.

Primer delovanja programa:

```
Vnesi realno vrednost: -1.25
Predznak:   1
Eksponent:  01111111
Mantisa:    01000000000000000000000
```

Razlaga: V gornjem klicu funkcije `printf` smo nad spremenljivko `x` izvedli tri operacije: Naslovni operator (&) najprej vrne naslov spremenljivke `x`, katerega tip je `float *`. Potem z operatorjem za pretvorbo tipa ta tip pretvorimo v `unsigned int *`. Zvezdica, ki je v paru oklepajev za imenom tipa, pomeni, da ne zahtevamo pretvorbe v `unsigned int`, temveč v ***kazalec*** na `unsigned int`. Na koncu z operatorjem posredovanja (*) z naslova `&x` preberemo štiri bajte podatkov in jih tolmačimo kot podatek tipa `unsigned int`. Tak je namreč zdaj sklicevani tip kazalca `&x`. To seveda deluje samo ob predpostavki, da sta tipa `float` in `int` oba dolga 32 bitov.

Opomba: Standard ne zagotavlja, da sta tipa `int` in `float` oba dolga 32 bitov, niti ne zagotavlja, da zasedata štiri bajte. Ker pa v večini sodobnih sistemov to lahko predpostavimo, smo primer uporabili, ker pomaga pri razumevanju nekaterih osnovnih pojmov, povezanih s kazalci. Kasneje bomo spoznali boljši in bolj prenosljiv način, na katerega lahko rešimo takšne naloge.

Pomoč: Pomagajte si s programom iz naloge 4.16.

Naloga 4.26 Če ste rešili nalogo 4.25, potem rešite še naslednjo nalogo:

Napišite program, ki bo s tipkovnice prebral realno vrednost neposredno v zapisu z 32-bitno plavajočo vejico (v šestnajstiškem zapisu) in na zaslon izpisal ustrezno realno vrednost. Za izpis vrednosti uporabite formatno določilo `%g`, ki povzroči, da se bo funkcija `printf` sama odločila, ali bo vrednost izpisala v običajni ali eksponentni obliki.

Primer delovanja programa:

```
Vnesi realno vrednost v 32-bitnem
zapisu s plavajočo vejico: 33D6BF95
Vneseni zapis predstavlja vrednost 1e-007.
```

5. POGLAVJE

FUNKCIJE

Podprogramu v jeziku C pravimo ***funkcija*** (angl. function)[1]. Funkcija je v resnici majhen program, ki navadno reši en problem. Če znamo iz naloge, ki je pred nami, ustrezno izolirati manjše probleme ter napisati funkcije, ki jih rešijo, bodo takšne funkcije uporabne tudi za reševanje mnogih drugih (sorodnih) nalog. Takšna ***ponovljiva uporabnost*** (angl. reusability) napisanih funkcij nam omogoča, da kasnejše naloge rešimo veliko hitreje.

5.1 Definicija in deklaracija funkcije

Preden lahko funkcijo uporabimo, jo moramo ***definirati*** (angl. define). Takole je videti splošna oblika definicije funkcije:

```
tip-vrnjene-vrednosti ime-funkcije ( parametri ) {
  deklaracije
  stavki
}
```

Tip vrnjene vrednosti (angl. return type) funkcije oziroma krajše ***tip funkcije*** je tip vrednosti, ki jo funkcija vrača s stavkom `return`. Če za tip funkcije izberemo tip `void` (slov. prazen), potem funkcija ne vrača nobene vrednosti.

[1] V nekaterih programskih jezikih ločimo pojma funkcija (podprogram, ki vrne vrednost) in procedura (podprogram, ki ne vrne vrednosti). Jezik C te razlike v poimenovanju ne pozna.)

Za imenom funkcije v paru okroglih oklepajev zapišemo ***seznam parametrov*** (angl. parameter list). Pred ime vsakega od parametrov moramo zapisati njegov tip, posamezne parametre pa ločimo z vejicami. Tudi če imamo več parametrov istega tipa, moramo ime tipa zapisati pred vsakega posebej. Na primer:

```
double povprecje(double a, double b) {
  return (a + b) / 2;
}
```

Če funkcija ne sprejme nobenega argumenta, potem uporabimo namesto seznama parametrov besedo `void`, kakor smo to že videli pri definiciji funkcije `main`.

Jezik C zahteva, da so v trenutku, ko prevajalnik naleti na klic funkcije, že znani določeni podatki o klicani funkciji: znano mora biti, kakšnega tipa je funkcija ter koliko parametrov ima. Za vsak parameter mora biti znan tudi njegov tip. To zahtevo lahko izpolnimo tako, da definicijo funkcije v izvorni kodi zapišemo pred njenim prvim klicem. Vendar včasih to ni možno ali pa je nepregledno, ker bi morale biti definicije funkcij zapisane v vrstnem redu, ki ni najbolj naraven za problem, ki ga rešujemo. Na srečo jezik C ne zahteva, da je definicija funkcije v izvorni kodi zapisana pred njenim klicem. Zahtevane podatke o tipu funkcije in njenih parametrih lahko zagotovimo tudi z ***deklaracijo funkcije*** (angl. function declaration). Deklaracija funkcije je podobna njeni definiciji, pri čemer namesto ***telesa funkcije*** (angl. function body) zapišemo podpičje:

```
tip-vrnjene-vrednosti ime-funkcije ( parametri ) ;
```

Takšni deklaraciji pravimo tudi ***prototip funkcije*** (angl. function prototype). Prototip funkcije vsebuje popolno informacijo o tem, kako je treba funkcijo klicati, nič pa ne pove o tem, kako funkcija deluje. Na primer, prototip funkcije `povprecje`, ki smo jo definirali zgoraj, je videti takole:

```
double povprecje(double a, double b);
```

Izvorna koda v več datotekah

Pri večjih projektih ni smiselno, da vso izvorno kodo hranimo v eni sami datoteki. Predpostavimo, da naš projekt obsega več sto tisoč vrstic kode. Med pisanjem kode moramo program večkrat prevesti in zagnati, da lahko preverimo njegovo delovanje. Prevajanje tako dolge kode pa je lahko izredno zamudno. Namesto ene same datoteke lahko posamezne funkcije glede na njihovo rabo razdelimo v več različnih datotek (pravimo jim tudi ***knjižnice***, angl. libraries). Tako organizirana koda je preglednejša, predvsem pa nam prihrani precej časa pri prevajanju. Če imamo kodo razdeljeno v veliko manjših datotek, nam med razvojem programa ni treba vsakokrat prevajati vse kode: prevedemo le datoteke, ki smo jih od zadnjega prevajanja spremenili. Na koncu prevedene datoteke zgolj ***povežemo*** (angl. link) z ostalimi deli programa, ki so prevedeni že od prej. Pri tem postopku nam pridejo prav prototipi funkcij.

Omenili smo že, da prevajalnik za prevajanje potrebuje zgolj prototip, ne pa tudi definicije funkcije. Definicija mora biti na voljo šele pri povezovanju kode. Zato je dovolj, da je vsaka funkcija definirana zgolj na enem mestu, njen prototip pa se mora pojaviti v vsaki od datotek, iz katerih to funkcijo kličemo. Da se izognemo večkratnemu pisanju prototipa ene in iste funkcije, prototipe zapišemo v tako imenovano ***zglavno datoteko*** (angl. header file). Zglavno datoteko potem vključimo z direktivo `#include` v vse datoteke, iz

katerih to funkcijo dejansko kličemo. Direktiva `#include` ima dve osnovni obliki. Prvo uporabimo, kadar želimo vključiti katero od standardnih cejevskih knjižnic:

```
#include <ime-datoteke>
```

Drugo obliko uporabimo za vse ostale zglavne datoteke, vključno s tistimi, ki jih napišemo sami:

```
#include "ime-datoteke"
```

Direktiva `#include` preprosto prepiše vsebino celotne datoteke, katere ime je zapisano v kotnih oklepajih ali navednicah, na mesto v datoteki, kjer se direktiva pojavi.

Razlika med gornjima dvema zapisoma direktive `#include` je majhna, a pomembna: Če zapišemo ime datoteke v kotne oklepaje, potem prevajalnik datoteko išče v sistemskih mapah. Če pa je ime datoteke zapisano v navednicah, potem prevajalnik datoteko išče najprej v trenutni mapi in šele potem v sistemskih mapah.

Poglejmo si primer, v katerem ustvarimo knjižnico z imenom *orodja* z eno samo funkcijo z imenom *povprecje*, ki jo bomo klicali iz druge datoteke. V ta namen potrebujemo tri datoteke: orodja.h, orodja.c in program.c. Prvi dve datoteki vsebujeta prototip oziroma definicijo funkcije `povprecje`, tretja datoteka pa to funkcijo kliče:

```c
/*** orodja.h ***/

double povprecje(double a, double b);
```

```c
/*** orodja.c ***/

#include "orodja.h"
double povprecje(double a, double b) {
  return (a + b) / 2;
}
```

```c
/*** program.c ***/

#include <stdio.h>
#include "orodja.h"
int main(void) {
  double a = 1.2, b = 3.8;
  printf("Povprečje števil %f in %f je %f",
    a, b, povprecje(a, b));
  return 0;
}
```

Čeprav iz datoteke orodja.c ne kličemo funkcije `povprecje`, smo vanjo vseeno vključili zglavno datoteko orodja.h. Ker zglavne datoteke v praksi poleg prototipov funkcij vsebujejo tudi druge pomembne deklaracije in definicije, je to običajen postopek.

5.2 Parametri in argumenti

Cejevske funkcije si podajajo informacijo preko argumentov in parametrov. Pojem ***parameter*** (angl. parameter) predstavlja kakršnokoli deklaracijo znotraj oklepajev za imenom

funkcije v njeni deklaraciji ali definiciji. Po drugi strani predstavlja pojem ***argument*** (angl. argument) kakršenkoli izraz v oklepaju za imenom funkcije pri njenem klicu.

Na primer, v definiciji:

```
int maks(int x, int y) {
  return x > y ? x : y;
}
```

sta `x` in `y` parametra funkcije `maks`. V klicu:

```
vec = maks(10, a + b);
```

sta konstantna vrednost `10` in izraz `a + b` argumenta funkcije.

V Ceju se ob klicu funkcije ***vsi argumenti*** prenašajo ***po vrednosti*** (angl. by value). To pomeni, da se najprej izračunajo njihove vrednosti, ki se potem kopirajo v ustrezne parametre klicane funkcije. Parametri predstavljajo lokalne spremenljivke funkcije, katerih vrednosti se nastavijo ob njenem klicu. Parametrom zato včasih pravimo tudi ***formalni parametri***, argumentom pa ***dejanski parametri***. Ker se vrednost argumenta vedno ***kopira*** v parameter, nobena sprememba vrednosti parametra znotraj funkcije nikoli ne more vplivati na vrednost originalnega argumenta.

Pri kopiranju vrednosti argumentov v ustrezne parametre moramo biti pozorni na njihove tipe. Kadar se tipi argumentov ne ujemajo s tipi parametrov, se uporabijo enake pretvorbe tipov kot v običajnih priredilnih stavkih. Prav tako lahko seveda uporabimo operator za pretvorbo tipa. Če vrednosti kakšnega argumenta ni mogoče pretvoriti v podatkovni tip ustreznega parametra (npr., če je vrednost argumenta prevelika), potem je obnašanje takšne kode nedoločeno. Obnašanje kode je nedoločeno tudi, če se število argumentov ne ujema s številom parametrov.

Vrstni red računanja argumentov

Vrstni red, v katerem se računajo vrednosti argumentov, je nepredpisan. Vzemimo naslednji primer:

```
void f(int x, int y, int z) {
  printf("x:%d, y:%d, z:%d\n", x, y, z);
}
//...
q = 10;
f(q++, q++, q);
printf("q: %d\n", q);  /* Izpiše: 12  */
q = 10;
f(q++, q++, q++);
printf("q: %d\n", q);  /* Izpiše: 13  */
```

Vse, kar lahko povemo o delovanju gornje kode, je to, da bo vrednost spremenljivke `q` po prvem klicu funkcije `f` enaka 12, po drugem klicu funkcije `f` pa 13. Ko smo gornjo kodo preizkusili, smo dobili naslednji izpis:

```
x:11, y:10, z:12
q: 12
x:12, y:11, z:10
q: 13
```

Iz izpisa se lepo vidi, da je vrstni red računanja argumentov vsakokrat drugačen. Zato se moramo v svoji kodi izogibati primerom, v katerih bi računanje vrednosti enega argumenta

vplivalo na vrednost kakšnega drugega argumenta. Če argumenti drug na drugega nimajo vpliva, potem vrstni red njihovega računanja seveda ni pomemben.

5.3 Stavek return

Funkcija, ki ni tipa void, mora vsebovati stavek return, s katerim določimo, kakšno vrednost funkcija vrača:

```
return izraz ;
```

Za izraz pogosto uporabimo zgolj konstantno vrednost ali spremenljivko:

```
return 0;
return status;
```

Uporabimo lahko tudi kompleksnejše izraze. Na primer:

```
return x > y ? x : y;
```

V zadnjem primeru se najprej izračuna vrednost izraza, ki jo funkcija nato vrne. Tip vrednosti izraza, ki stoji za stavkom return, mora ustrezati tipu funkcije, v kateri se ta stavek nahaja. V nasprotnem primeru se izvrši avtomatska pretvorba tipa. Na primer, če je tip funkcije int, izraz v stavku return pa je tipa float, se bo realna vrednost avtomatično pretvorila v celoštevilsko. Kot že vemo, takšna pretvorba enostavno odreže decimalni del realne vrednosti.

Stavek return lahko uporabimo tudi v funkciji tipa void, vendar brez pripadajočega izraza:

```
return ;
```

Takšen stavek preprosto konča izvajanje funkcije. Na primer:

```
void izpisiCifro(char zn) {
  if (zn < '0' || zn > '9') return;
  printf("%c", zn);
}
```

Funkcija izpisiCifro bo izpisala podani znak samo, če je le-ta desetiška številka. Če podani znak ni številka, potem s stavkom return predčasno zapustimo funkcijo in znak se ne izpiše. Stavek return je lahko tudi na koncu funkcije tipa void, vendar to ni potrebno.

Kadar funkcija ni tipa void, je stavek return obvezen. Še več: vsaka od možnih poti izvajanja funkcije mora imeti na koncu stavek return z izrazom ustreznega tipa. V nasprotnem primeru je – kadar vrnjeno vrednost uporabimo v programu – obnašanje takšnega programa nedoločeno. Na primer:

```
int predznak(double x) {
  if (x > 0) return 1;
  if (x < 0) return -1;
}
```

Funkcija predznak v primeru, ko je x enak nič, ne vrne nobene vrednosti. Tako bo na primer klic:

```
s = predznak(0);
```

povzročil nedoločeno obnašanje programa. Z ustreznimi nastavitvami lahko dosežemo, da nas prevajalnik opozori na napako takšne vrste.

5.4 Prekinitev izvajanja programa

Ker je izvajanje cejevskega programa povezano z izvajanjem funkcije `main`, lahko izvajanje programa prekinemo s stavkom `return` znotraj funkcije `main`. Povedali smo že, da je vrednost, ki jo vrne funkcija `main`, lahko sporočilo operacijskemu sistemu o tem, na kakšen način se je program končal.

Vendar stavek `return` ni vedno najboljši način, da končamo program. Če na primer želimo končati izvajanje programa iz kakšne druge funkcije, potem stavek `return` konča izvajanje te funkcije, ne pa tudi celega programa. Takrat lahko uporabimo funkcijo `exit` (slov. izhod), ki jo najdemo v standardni knjižnici <stdlib.h>. Funkciji `exit` podamo argument, ki ima enak pomen kot vrednost, ki jo vrne funkcija `main`:

```
exit(0); /* Konča program brez posebnosti.  */
```

Če funkcijo `exit` uporabimo v funkciji `main`, potem je njen učinek enak, kot če bi uporabili stavek `return`:

```
int main(void) {
  //...
  exit(izraz); /* Enako kot return izraz;  */
  //...
}
```

5.5 Območje in obstoj spremenljivk

Kakor hitro imamo v programu več funkcij, naletimo na nekaj pomembnih vprašanj v zvezi s spremenljivkami. Spremenljivke ločimo glede na njihovo območje in obstoj:

- ***območje*** (angl. scope) spremenljivke predstavlja del programske kode, iz katerega imamo dostop do te spremenljivke;
- ***obstoj*** (angl. storage duration) spremenljivke predstavlja del časa med izvajanjem programa, ko spremenljivka obstaja v pomnilniku.

Lokalne spremenljivke

Za spremenljivke, ki jih deklariramo znotraj katere od funkcij, pravimo, da so ***lokalne*** (angl. local) spremenljivke funkcije. Parametri so prav tako lokalne spremenljivke funkcije. Edina prava razlika med parametri in deklariranimi lokalnimi spremenljivkami je ta, da se parametrom ob klicu funkcije priredijo vrednosti ustreznih argumentov. Lokalne spremenljivke imajo naslednji dve lastnosti:

- ***Območje bloka***[2] (angl. block scope) Območje lokalnih spremenljivk je omejeno na del programske kode od njene deklaracije do konca definicije funkcije, v kateri se deklaracija nahaja. Lokalne spremenljivke niso dostopne od nikoder drugod, zato lahko v drugih funkcijah uporabimo druge spremenljivke z istim imenom.

[2] V Ceju blok pomeni kodo med dvema zavitima oklepajema. Potemtakem je telo funkcije blok.

- ***Avtomatičen obstoj*** (angl. automatic storage duration) Ob klicu funkcije se prostor za lokalne spremenljivke avtomatično ustvari (na skladu) in obstaja samo do konca izvajanja funkcije. Ko funkcijo kličemo znova, ni nobenega zagotovila, da bodo imele lokalne spremenljivke vrednosti, ki so jih imele ob zadnjem izhodu iz te funkcije.

Osvetlimo gornja dva pojma z naslednjim primerom:

```
int povecaj(int x) {
  x++;
  return x;
}

int main(void) {
  int x = 1, y = 1;
  y = povecaj(y);
  printf("%d %d", x, y);  /* Izpiše: 1 2  */
  return 0;
}
```

Program vsebuje tri spremenljivke: funkcija `main` ima dve lokalni spremenljivki (`x` in `y`), funkcija `povecaj` pa eno lokalno spremenljivko (`x`), ki je hkrati njen parameter. Ker je območje vsake od lokalnih spremenljivk z imenom `x` omejeno na svojo funkcijo, ni nobenega dvoma o tem, na katero spremenljivko se sklicujejo posamezni deli kode.

Nastopi pa vprašanje, kaj se zgodi s spremenljivko `x` v funkciji `main` v času izvajanja funkcije `povecaj`. Spremenljivka iz funkcije `povecaj` nedvomno ni dostopna, obstaja pa. Ko se namreč izvaja funkcija `povecaj`, se izvajanje funkcije `main` še ni končalo. Zato lokalna spremenljivka `x` funkcije `main` ves ta čas obstaja v pomnilniku in seveda tudi hrani svojo vrednost. Ko se izvajanje funkcije `povecaj` konča, se zato za vrednost spremenljivke `x` izpiše ena.

Globalne spremenljivke

Poleg parametrov lahko za komunikacijo med funkcijami uporabimo tudi ***globalne*** (angl. global) spremenljivke. To so spremenljivke, ki so deklarirane zunaj katerekoli funkcije. Globalne spremenljivke imajo naslednji dve lastnosti:

- ***Območje datoteke*** (angl. file scope) Globalne spremenljivke so vidne in dostopne od mesta njihove deklaracije do konca datoteke. Posledica tega dejstva je, da lahko vse funkcije, katerih definicije so zapisane za deklaracijami globalnih spremenljivk, berejo in spreminjajo vrednosti teh spremenljivk.
- ***Statičen obstoj*** (angl. static storage duration) Spremenljivka, ki ima statičen obstoj, obstaja v pomnilniku ves čas izvajanja programa.

Globalnim spremenljivkam se skušamo v kar največji meri izogniti. Njihova uporaba je na primer smiselna, če se lahko z njimi izognemo uporabi pretiranega števila parametrov. Pomembno je, da globalne spremenljivke poimenujemo s pomenljivimi in/ali dolgimi imeni. Globalne spremenljivke lahko označimo tudi tako, da njihovim imenom dodamo določeno predpono (npr. `g_`). S takšnimi ukrepi se izognemo nenamernemu spreminjanju vrednosti globalnih spremenljivk.

Bloki

Sestavljeni stavki, ki smo jih spoznali v 3. poglavju, lahko vsebujejo tudi deklaracije:

```
{ deklaracije stavki }
```

Sestavljenemu stavku pravimo tudi ***blok***. Naslednji del kode kaže primer takšnega bloka:

```
if (x < y) {
  int pom = x;
  x = y;
  y = pom;
}
```

Gornja koda zamenja med seboj vrednosti spremenljivk `x` in `y`, če je `x` manjši od `y`. Pri tem uporablja pomožno spremenljivko `pom`, ki smo jo deklarirali znotraj bloka. Tako kot lokalne spremenljivke imajo tudi blokovske spremenljivke avtomatičen obstoj (obstajajo v času izvajanja bloka) ter območje bloka (zunaj bloka niso dostopne). Deklaracija pomožnih spremenljivk znotraj bloka je smiselna, ker tako zmanjšamo navlako in s tem hkrati možnost napak.

V razdelku 3.6 (na strani 35) smo videli, da lahko prvi izraz v zanki `for` nadomestimo z deklaracijo spremenljivke. Tudi ta primer predstavlja deklaracijo spremenljivke v bloku. Takšna spremenljivka namreč zunaj zanke `for` ni dostopna.

Območna pravila

Spoznali smo, da imajo lokalne in blokovske spremenljivke območje bloka, kar omejuje njihovo dostopnost na blok, v katerem so deklarirane. Zastavi se vprašanje, kaj se zgodi, če imamo dve spremenljivki z istima imenoma, ki sta deklarirani v gnezdenih blokih. Zanima nas tudi, kaj se zgodi, če ima globalna spremenljivka enako ime kot ena izmed lokalnih spremenljivk.

Na srečo obstaja enostavno pravilo, ki razreši potencialne spore, ki nastanejo v zgoraj opisanih primerih: kadarkoli v funkciji ali bloku deklariramo spremenljivko in zanjo uporabimo isto ime, kot ga ima kaka druga spremenljivka, ki je trenutno dostopna, se ta druga spremenljivka začasno skrije. Naslednji program kaže (relativno skrajen) primer, v katerem ima identifikator `x` štiri različne pomene:

```
#include <stdio.h>

int x = 1;

void test1(void) {
  int x = 2;
  if (x > 0) {
    int x = 3;
    x++;
    printf("%d", x);  /* Izpiše: 4  */
  }
  printf("%d", x);    /* Izpiše: 2  */
}

void test2(int x) {
  printf("%d", x);    /* Izpiše: 3  */
}

int main(void) {
  test1();
  test2(3);
```

```
  printf("%d", x);     /* Izpiše: 1  */
  return 0;
}
```

V funkciji `test1` imamo dve spremenljivki `x`, od katerih je ena omejena na blok stavka `if`. V tem bloku druga spremenljivka `x` ni dostopna, zato prvi stavek `printf` izpiše vrednost štiri. Naslednji stavek `printf` izpiše vrednost dve, kar je vrednost lokalne spremenljivke `x` funkcije `test1`. V funkciji `test2` je dostopen edino njen parameter, njegova vrednost pa je po klicu iz funkcije `main` enaka tri. Funkcija `main` nima svoje lokalne spremenljivke z imenom `x`, zato uporablja globalno, katere vrednost je ena. Ta vrednost se tudi izpiše z zadnjim stavkom `printf`.

Statične lokalne in blokovske spremenljivke

Z uporabo besede `static` pred deklaracijo lokalne ali blokovske spremenljivke lahko spremenimo njen obstoj iz avtomatičnega v statičen. Takšna spremenljivka bo hranila svojo vrednost ves čas izvajanja programa, še vedno pa bo nedostopna zunaj območja svojega bloka. Naslednji primer kaže, kako lahko z uporabo statične lokalne spremenljivke štejemo, kolikokrat je bila klicana neka funkcija:

```
#include <stdio.h>

int f1(void) {
  static int st = 0;
  st++;
  return st;
}

int f2(void) {
  static int st = 0;
  st++;
  return st;
}

int main(void) {
  for (int i = 0; i < 4; i++) {
    f1();
  }
  for (int i = 0; i < 8; i++) {
    f2();
  }
  printf("%d %d", f1(), f2());  /* Izpiše: 5 9  */
  return 0;
}
```

Ker sta obe lokalni spremenljivki `st` v gornjem programu statični, se ustvarita le enkrat. To hkrati pomeni, da se njuna začetna vrednost, ki smo jima jo priredili ob deklaraciji (tj. vrednost nič), zapiše v pomnilnik le na začetku izvajanja programa. V tem primeru je pomembno, da spremenljivko inicializiramo hkrati z njeno deklaracijo. V nasprotnem primeru bi se spremenljivka nastavila na nič ob vsakem klicu funkcije:

```
int f1(void) {
  static int st;
  st = 0;  /* Ta stavek se izvede ob vsakem klicu funkcije.  */
  st++;
  return st;
}
```

Naloga 5.1 Za vajo napišite program, ki izpisuje vse pozitivne delitelje vsakokrat vnesenega naravnega števila. Program naj med izpisane delitelje postavi vejice.

Primer delovanja programa:

```
Vnesi naravno število (0 konča): 15
1, 3, 5, 15
Vnesi naravno število (0 konča): 16
1, 2, 4, 8, 16
Vnesi naravno število (0 konča): 0
Nasvidenje!
```

Nalogo rešite tako, da napišete in uporabite funkcijo `dodajStevilo`, ki naj na izhod izpiše številsko vrednost, ki ji jo podamo kot argument. Ko funkcijo kličemo prvič, naj izpiše zgolj podano vrednost, ob vsakem naslednjem klicu pa naj pred izpis vrednosti doda še vejico in presledek. Če funkciji podamo argument z vrednostjo nič, naj se ponastavi – njen naslednji klic naj spet izpiše zgolj številko brez vejice.

Primeri zaporednih klicev funkcije:

```
dodajStevilo(4);   /* Izpiše: 4                  */
dodajStevilo(4);   /* Izpiše: , 4                */
dodajStevilo(51);  /* Izpiše: , 51               */
dodajStevilo(0);   /* Ne izpiše ničesar,         */
                   /* temveč ponastavi funkcijo. */
dodajStevilo(19);  /* Izpiše: 19                 */
```

5.6 Podajanje in vračanje kazalcev

Spoznali smo že, da se v jeziku C argumenti funkcijam podajajo ***po vrednosti*** (angl. by value). Takšno podajanje nas omejuje, saj znotraj funkcije ni mogoče spremeniti vrednosti parametrov na način, da bi se sprememba poznala tudi zunaj funkcije na podanih argumentih. Na srečo lahko to težavo zaobidemo tako, da za argument uporabimo ***naslov*** podatka v pomnilniku. Poleg tega moramo seveda ustreznemu parametru dodeliti kazalčni tip. Vzemimo za primer funkcijo, ki med seboj zamenja vrednosti dveh spremenljivk:

```
void menjaj(int *x, int *y) {
  int pom = *x;
  *x = *y;
  *y = pom;
}
```

Zvezdici pred imenoma `x` in `y` v seznamu parametrov funkcije `menjaj` pomenita, da sta ta dva parametra kazalca. Omogočata nam dostop do dveh spremenljivk tipa `int`, katerih naslova bomo podali kot argumenta funkcije. Pomembno je, da v telesu funkcije nad kazalcema `x` in `y` uporabimo operator posredovanja (*), saj moramo operacijo menjave vrednosti izvesti nad ***podatkoma***, na katera kazalca kažeta.

Funkcijo `menjaj` kličemo tako, da ji kot argumenta podamo naslova spremenljivk, katerih vrednosti želimo med seboj zamenjati:

```
int a = 155, b = 1953;
menjaj(&a, &b);
printf("%d %d", a, b);  /* Izpiše: 1953 155  */
```

Lahko bi tudi rekli, da smo v gornji kodi spremenljivki `a` in `b` podali ***po sklicu*** (angl. by reference). Čeprav gre pri tem za popolnoma isti mehanizem (tj. podajanje naslova objekta

v pomnilniku), pa se v jeziku C temu uradno ne reče tako. Beseda *sklic* je rezervirana za jezik C++, kjer obstajajo določene razlike med kazalcem in sklicem.

Pomembno je, da razumemo, da se argumenti v gornjem primeru v resnici še vedno podajajo po vrednosti. Razlika je le v tem, da je vrednost, ki se podaja, naslov. Ob gornjem klicu funkcije `menjaj` se implicitno izvedeta naslednja dva priredilna stavka:

```
x = &a;
y = &b;
```

Parametra `x` in `y` v funkciji `menjaj` vsebujeta torej kopiji naslovov spremenljivk `a` in `b`. To pomeni, da bi v funkciji `menjaj` brez uporabe operatorja posredovanja zamenjali zgolj kopiji naslovov podanih argumentov. To seveda ne bi imelo nikakršnega učinka, ki bi se poznal zunaj funkcije `menjaj`:

```
void menjaj(int *x, int *y) {
  int pom = x;  /* Te tri vrstice kode med seboj zamenjajo vrednosti  */
  x = y;        /* kazalčnih spremenljivk x in y, ki sta lokalni      */
  y = pom;      /* spremenljivki funkcije. Zunaj funkcije ta menjava  */
                /* zato nima nikakršnega učinka.                      */
}
```

Poleg tega bi nas prevajalnik najverjetneje opozoril, da v prvi vrstici funkcije `menjaj` prirejamo kazalčno vrednost spremenljivki tipa `int` (v zadnji vrstici počnemo obratno), ne da bi pri tem uporabili operator za pretvorbo tipa.

Zdaj lahko pojasnimo tudi vlogo naslovnega operatorja pred imenom spremenljivke v funkciji `scanf`: če želimo, da funkcija zapiše vrednost, ki jo prebere iz vhodnega toka, v določeno spremenljivko, potem ji moramo kot argument podati naslov te spremenljivke.

Naredimo za ilustracijo še en primer. Napišimo funkcijo, ki ji kot argument (vhodni podatek) podamo število sekund, ki so pretekle od polnoči, 1. januarja 1970[3]. Funkcija naj vrne uro v obliki treh celoštevilskih vrednosti, ki predstavljajo ure, minute in sekunde. Te tri podatke bomo iz funkcije dobili preko dodatnih treh argumentov, ki jih bomo podajali kot kazalce (izhodni podatki). Dopustimo še možnost, da nas bodo včasih zanimale samo ure in minute, sekunde pa ne. V tem primeru bomo kot zadnji argument funkcije podali ničelni kazalec. Takole je videti napisana funkcija:

```
void pretvoriVUro(time_t sekunde, int *h, int *m, int *s) {
  sekunde %= (24 * 3600);  /* Izločimo informacijo o datumu.  */
  *h = sekunde / 3600;
  *m = (sekunde / 60) % 60;
  if (s != NULL) {
    *s = sekunde % 60;
  }
}
```

Parameter `sekunde` je tipa `time_t`[4], ki predstavlja osnovni aritmetični podatkovni tip za hranjenje časa in je določen v standardni knjižnici <time.h>. Gornja funkcija sicer ni nič posebnega, opozorimo naj le na stavek `if`, ki pred uporabo kazalca `s` preveri, ali je kazalec inicializiran. Namreč, če nas sekunde ne bodo zanimale, bomo ob klicu funkcije

[3] Čas je v večini računalniških sistemov zapisan kot število sekund, ki so pretekle od 1. januarja 1970 ob 00:00:00 UTC (polnoč po ***univerzalnem koordiniranem času***, angl. coordinated universal time, UTC). Takemu zapisu časa pravimo tudi ***Unixov čas*** (angl. Unix time).

[4] Standard ne pove natančno, kakšen tip je `time_t`. Ne govori niti o tem, ali je celoštevilski ali realen. Vendar boste v praksi težko naleteli na sistem, kjer ta tip ne bi bil 32- ali 64-bitno predznačeno celo število.

namesto veljavnega naslova spremenljivke kot argument podali ničelni kazalec. Takole je videti del programa, ki uporabi gornjo funkcijo:

```
int ure, minute;
pretvoriVUro(97477, &ure, &minute, NULL);
printf("%02d:%02d", ure, minute);  /* Izpiše: 03:04  */
```

V gornjem programu smo kot prvi argument funkcije `pretvoriVUro` uporabili konstantno število sekund. Namesto tega lahko uporabimo tudi funkcijo `time`, ki jo najdemo v standardni knjižnici <time.h>. Funkcija vrne število sekund, ki so pretekle od polnoči (tj. 00:00:00 in ne 24:00:00), 1. januarja 1970, in sicer kot podatek tipa `time_t`:

```
int ure, minute, sekunde;
pretvoriVUro(time(NULL), &ure, &minute, &sekunde);
printf("%02d:%02d.%02d", ure, minute, sekunde);
```

Iz zgodovinskih razlogov funkcija `time` kot argument sprejme tudi naslov spremenljivke, v katero vpiše sistemski čas. Ker funkcija isto vrednost tudi vrne, navadno kot argument uporabimo kar ničelni kazalec. Omenimo še, da funkcija `time` vrne število sekund po univerzalnem koordiniranem času. Zato bo ura, ki jo dobimo, eno uro za našim lokalnim časom.

Naloga 5.2 Za vajo napišite funkcijo za seštevanje dveh nenegativnih celih števil med nič do vključno $10^9 - 1$. Rezultat takšnega seštevanja je v splošnem desetmesten. Pri tem je lahko prva (skrajno leva) števka le nič ali ena in v resnici predstavlja prenos pri seštevanju. Funkcija naj ima tri parametre, od katerih naj bosta drugi in tretji parameter vhodna seštevanca. Prvi parameter uporabite za vračanje zadnjih devetih števk vsote. Prvo števko vsote (tj. prenos) vrnite s stavkom `return`. Za boljšo predstavo je tu primer klica takšne funkcije:

```
long vsotaH, vsotaL;  /* Višje (High) in nižje (Low)
                         števke vsote.                    */
vsotaH = sestej(&vsotaL, 999999999, 1);
printf("%d%09d", vsotaH, vsotaL);  /* Izpiše: 1000000000  */
```

Namig: Napisano funkcijo lahko uporabite za reševanje naloge 4.15 na strani 70.

Vračanje kazalca

Funkcija lahko kazalec tudi vrne (s stavkom `return`). To moramo upoštevati že pri njeni definiciji, kjer moramo za tip funkcije izbrati kazalčni tip. Na primer, naslednja funkcija od dveh spremenljivk, katerih naslova podamo kot argumenta funkcije, vrne naslov spremenljivke, ki ima večjo vrednost:

```
double *maks(double *a, double *b) {
  if (*a > *b) return a;
  else return b;
}
```

Zvezdica pred imenom funkcije nakazuje, da funkcija ne vrača vrednosti tipa `double`, temveč ***kazalec*** tipa `double *`. Ko funkcijo kličemo, njen klic uporabimo tako, kakor bi uporabili katerikoli kazalec tipa `double *`. Na primer:

```
double x = 10.2, y = 15.7, *k;
k = maks(&x, &y);
printf("%.1f", *k);  /* Izpiše: 15.7  */
```

Operator posredovanja lahko uporabimo tudi neposredno na klicu funkcije:

```
double x = 10.2, y = 15.7;
printf("%.1f", *maks(&x, &y));  /* Izpiše: 15.7  */
```

Vrednost podatka, na katerega kaže vrnjen kazalec, lahko seveda tudi spremenimo:

```
double x = 10.2, y = 15.7;
*maks(&x, &y) = 42;          /* Funkcija maks vrne kazalec na y,
                                zato 42 vpišemo v y.              */
printf("%.1f %.1f", x, y));  /* Izpiše: 10.2 42.0                 */
```

Morda se zdi nekoliko nenavadno, da je lahko na levi strani priredilnega operatorja klic funkcije. Vendar če funkcija vrača kazalec, potem je njen klic skupaj z operatorjem posredovanja popolnoma veljavna leva vrednost (tj. izraz, ki lahko stoji na levi strani priredilnega operatorja).

Paziti moramo, da iz funkcije ne vračamo naslova lokalne spremenljivke. Ta namreč po izhodu iz funkcije ne obstaja več in ostali bi z naslovom neobstoječe spremenljivke. Uporaba kazalca, ki kaže na objekt, ki je prenehal obstajati, povzroči v programu nedoločeno obnašanje. Na srečo nas na takšno napako največkrat opozori že prevajalnik:

```
int *test(void) {
  int x;
  //...
  return &x;  /* Opozorilo: funkcija vrača
                 naslov lokalne spremenljivke.  */
}
```

Lahko pa seveda vrnemo naslov statične lokalne spremenljivke:

```
int *test(void) {
  static int x;
  //...
  return &x;  /* To je v redu: statična spremenljivka
                 obstaja tudi po izhodu iz funkcije.   */
}
```

5.7 Kazalec na funkcijo

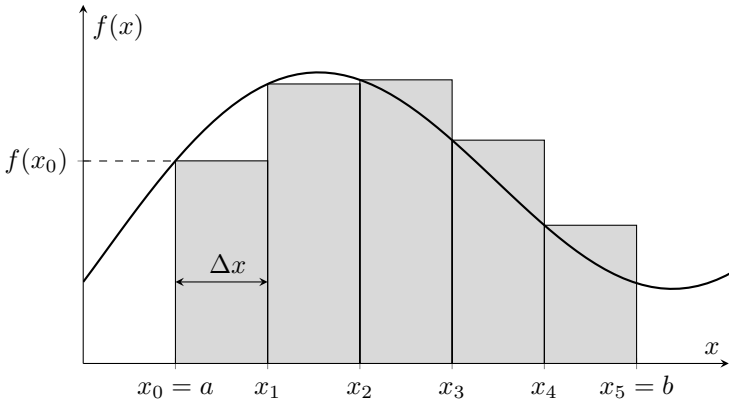
Tako kot podatki ima tudi koda svoj naslov v pomnilniku. Naslednji program izpiše naslov funkcije `main`:

```
#include <stdio.h>
int main(void) {
  printf("%x", (unsigned int) main);
  return 0;
}
```

Ime funkcije predstavlja njen naslov v pomnilniku[5]. V jeziku C ni neobičajno, da za parameter funkcije uporabimo kazalec na neko drugo funkcijo, s čimer lahko naredimo prvo funkcijo veliko splošnejšo.

[5] Če smo čisto natančni, v Ceju funkcijo kličemo tako, da nad njenim naslovom izvedemo operacijo klica funkcije. To operacijo predstavlja par okroglih oklepajev z morebitnimi argumenti, ki ga postavimo takoj za naslovom funkcije.

Za primer bomo napisali funkcijo, ki izračuna določeni integral funkcije, ki jo podamo kot parameter. Določeni integral funkcije $f(x)$ na intervalu $[a, b]$ lahko izračunamo v obliki Riemannove vsote, ki predstavlja način približevanja določenega integrala s končno vsoto, kakor kaže naslednja slika:



Površina pod krivuljo, ki jo integriramo, se nadomesti z ozkimi pravokotniki. Površina vsakega od pravokotnikov je enaka vrednosti funkcije v začetni točki intervala, ki ga pravokotnik pokrije, pomnoženi s širino intervala. Na primer, površina prvega pravokotnika na gornji sliki je enaka $f(x_0)\Delta x$. Vrednost določenega integrala funkcije $f(x)$ na intervalu $[a, b]$ je potem približno enaka vsoti površin vseh tako dobljenih pravokotnikov:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f(x_i)\Delta x,$$

pri čemer je n število intervalov, $x_i = a + i\Delta x$ in $\Delta x = (b - a)/n$.

Napišimo zdaj definicijo funkcije `integral`, ki integrira matematično funkcijo `f` med točkama `a` in `b`. Funkcija `f` naj sprejme argument tipa `double` in vrne vrednost, ki je prav tako tipa `double`. Kot prvi parameter funkcije `integral` bomo zato določili kazalec na funkcijo, ki sprejme in vrne vrednost tipa `double`. Takole je videti definicija funkcije `integral`:

```
double integral(double (*f)(double), double a, double b) {
/*******************************************************
* Izračuna določeni integral funkcije f                *
* na intervalu [a, b] kot Riemannovo vsoto.            *
*******************************************************/
  unsigned long n = 1e7;
  double vsota = 0;
  double dx = (b - a) / n;
  for (unsigned long i = 0; i < n; i++) {
    vsota += f(a + i * dx) * dx;
  }
  return vsota;
}
```

Prvi parameter funkcije predstavlja kazalec na funkcijo:

```
double (*f)(double)
```

Ime kazalca `f` z zvezdico mora biti v oklepaju, sicer bi se deklaracija tolmačila kot funkcija, ki vrne kazalec. Znotraj funkcije `integral` potem funkcijo `f` kličemo tako, kot smo navajeni: s parom oklepajev, v katerega zapišemo potrebne argumente.

Ko kličemo funkcijo `integral`, moramo za njen prvi argument uporabiti ***naslov*** funkcije, ki jo želimo integrirati. To pomeni, da moramo uporabiti samo njeno ime brez oklepajev in parametrov. Za primer vzemimo funkciji `sin` in `sqrt` (square root, slov. kvadratni koren), ki ju najdemo v standardni knjižnici <math.h>. Takole je videti klic funkcije `integral`:

```
printf("%.5f", integral(sin, 0, PI));  /* Izpiše: 2.00000  */
printf("%.5f", integral(sqrt, 1, 6));  /* Izpiše: 9.13129  */
```

Določeni integral sinusne funkcije na intervalu od nič do π je enak dve, določeni integral kvadratnega korena na intervalu od ena do štiri pa je enak $9{,}13129$.

Naloga 5.3 Za vajo si izberite in napišite svojo funkcijo, ki jo boste integrirali z uporabo pravkar napisane funkcije `integral`. Na primer, napišete lahko funkcijo `polinom`, ki vrne vrednost polinoma:

$$p(x) = 2x^2 - x + 3.$$

Če ta polinom integriramo od nič do tri, dobimo vrednost $22{,}5$, kakor kaže naslednji klic:

```
printf("%.5f", integral(polinom, 0, 3));  /* Izpiše: 22.50000  */
```

Uporaba kazalca na funkcijo v jeziku C seveda ni omejena zgolj na parametre funkcij. Deklariramo lahko tudi običajno kazalčno spremenljivko in ji priredimo naslov funkcije. Na primer, tako lahko deklariramo kazalčno spremenljivko `kf`, ki kaže na funkcijo tipa `void`, ki sprejme dva celoštevilska argumenta:

```
void (*kf)(int, int);
```

Predpostavimo, da imamo funkcijo s takšnimi lastnostmi, ki se imenuje `funkc`. Potem lahko usmerimo kazalec `kf` na funkcijo `funkc` z naslednjim priredilnim stavkom:

```
kf = funkc;
```

Ko `kf` enkrat kaže na `funkc`, lahko kličemo funkcijo `funkc` z uporabo kazalca `kf` takole:

```
int x, y;
//...
kf(x, y);
```

Par okroglih oklepajev lahko tolmačimo tudi kot neke vrste operator posredovanja, ki kliče funkcijo preko njenega naslova.

5.8 Naloge

Naloga 5.4 Podane so naslednje deklaracije:

```
void f(int a);
int g(int a, int b);
double h(double x);
int m(void);
void n(void (*kf)(int));
double x;
```

Kateri od naslednjih stavkov vsebujejo napako? Utemeljite odgovor.

```
x = f(x);
g(12.3, 7);
x = h();
m();
n(f, x);
n(f);
g(g(1, 2), 3);
```

Naloga 5.5 Kaj bo na izhod izpisal naslednji program? Utemeljite odgovor.

```
#include<stdio.h>
int x = 10;

void f(int x) {
  printf("%d", ++x);
}

int main(void) {
  f(x + 1);
  if (x == 10) {
    int x = 4;
  }
  printf("%d", x);
  return 0;
}
```

Naloga 5.6 Podan je naslednji program:

```
#include <stdio.h>

int i;

void pisi(int x, int y) {
  for (i = x; i <= y; i++) {
    printf("%3d", i);
  }
  printf("\n");
}

int main(void) {
  for (i = 0; i < 5; i++) {
    pisi(i, i + 5);
  }
```

```
  return 0;
}
```

Popravite program tako, da bo izpisal naslednji vzorec števil (trenutno izpiše le prvo vrstico tega vzorca):

```
0  1  2  3  4  5
1  2  3  4  5  6
2  3  4  5  6  7
3  4  5  6  7  8
4  5  6  7  8  9
```

Naloga 5.7 Napišite funkcijo `varnoDeljenje`, ki ima dva celoštevilska parametra in en parameter tipa `double`. Preko slednjega naj vrne rezultat deljenja. V primeru, da je deljenje uspelo, naj funkcija s stavkom `return` vrne neničelno vrednost, v primeru deljenja z nič pa naj vrne vrednost nič.

Primer programa, ki uporablja funkcijo `varnoDeljenje`:

```
printf("Vpiši števec: ");
scanf("%d", &x);
printf("Vpiši imenovalec: ");
scanf("%d", &y);
if (varnoDeljenje(x, y, &rezultat)) {
  printf("%d / %d = %.3f", x, y, rezultat);
}
else {
  printf("Napaka: deljenje z nič.");
}
```

Primer delovanja programa:

```
Vpiši števec: 0
Vpiši imenovalec: 3
0 / 3 = 0.000

Vpiši števec: 3
Vpiši imenovalec: 0
Napaka: deljenje z nič.
```

Naloga 5.8 Napišite funkcijo `beriCelo`, ki z vhoda prebere celoštevilsko vrednost in jo vrne preko svojega parametra. S stavkom `return` naj funkcija vrne podatek o izidu operacije kot eno od naslednjih treh vrednosti:

- 0: operacija je uspela,
- 1: opozorilo – vneseni celoštevilski vrednosti sledijo znaki, ki jih ni mogoče tolmačiti kot del celoštevilske vrednosti in
- 2: napaka – vhodnega podatka ni mogoče tolmačiti kot celo število.

V prvih dveh primerih naj funkcija v parameter zapiše vrednost, ki jo je prebrala, v tretjem primeru pa naj vanj vpiše vrednost nič. Za branje vrednosti z vhoda uporabite funkcijo `scanf`. Dodajte tudi čiščenje vhodnega toka podatkov, da odstranite vse neprebrane znake, ki jih funkcija `scanf` pusti za sabo (glej primer na strani 61).

Primer uporabe napisane funkcije:

```
int x;
while (1) {
  printf("Vnesi celo število (-1 konča): ");
  switch (beriCelo(&x)) {
    case 1: printf("Branje delno uspelo. ");
    case 0: printf("Prebrana vrednost: %d\n", x);
            if (x == -1) {
              printf("Nasvidenje!");
              exit(0);
            }
            break;
    case 2: printf("Napačen vnos.\n"); break;
  }
}
```

In še primer delovanja gornjega programa:

```
Vnesi celo število (-1 konča): 13
Prebrana vrednost: 13
Vnesi celo število (-1 konča): 5.8
Branje delno uspelo. Prebrana vrednost: 5
Vnesi celo število (-1 konča): #9$
Napačen vnos.
Vnesi celo število (-1 konča): -1x
Branje delno uspelo. Prebrana vrednost: -1
Nasvidenje!
```

Naloga 5.9 Sestavite knjižnico funkcij za računanje z ulomki. Knjižnica naj vsebuje funkcije za seštevanje, množenje in krajšanje ulomkov. Izhajate lahko iz kode, ki ste jo napisali kot rešitev naloge 1.8 na strani 14.

Za preizkus knjižnice lahko uporabite naslednjo kodo:

```
#include <stdio.h>
#include "ulomki.h"  /* Knjižnica funkcij za računanje z ulomki.    */
int main(void) {
  int st1 = 3, im1 = 4, st2 = 9, im2 = 6;
  printf("%d / %d + %d / %d = ", st1, im1, st2, im2);
  sestej(&st1, &im1, st2, im2);   /* Vsota se nahaja v st1 in im1.  */
  printf("%d / %d = ", st1, im1); /* Izpiše: 54 / 24                */
  krajsaj(&st1, &im1);
  printf("%d / %d\n", st1, im1);  /* Izpiše: 9 / 4                  */
  return 0;
}
```

Naloga 5.10 Ugotovite, kaj počne naslednji program. Utemeljite odgovor.

```
double minus(double a, double b) {
  return a - b;
}

double plus(double a, double b) {
  return a + b;
}

double g(double (*f)(double, double), double x, double y) {
```

```
  return f(x, y) * f(x, y);
}

int main(void) {
  double a, b;
  a = g(minus, 30, 25);
  b = g(plus, 30, 25);
  return 0;
}
```

Naloga 5.11 Napišite in preizkusite definicijo funkcije `tabeliraj`. Funkcija naj kot prvi parameter sprejme poljubno funkcijo `f`, ki sprejme in vrne vrednost tipa `double`. Poleg tega naj funkcija `tabeliraj` sprejme še tri parametre `a`, `b` in `dx`, prav tako tipa `double`. Funkcija `tabeliraj` naj izpiše tabelo vrednosti podane funkcije `f` za vrednosti parametrov od `a` do `b` s korakom `dx`.

Na primer, klic:

```
tabeliraj(log, 1, 4, .5);
```

naj izpiše naslednjo tabelo:

```
  x      f(x)
-----  -------
1.000  0.00000
1.500  0.40547
2.000  0.69315
2.500  0.91629
3.000  1.09861
3.500  1.25276
4.000  1.38629
```

Za preizkušanje funkcije `tabeliraj` uporabite različne matematične funkcije iz standardne knjižnice <math.h>, kjer najdete tudi funkcijo `log`.

Opomba: Funkcija `log` v jeziku C predstavlja naravni logaritem, to je logaritem z osnovo e.

PRAZNA STRAN

6. POGLAVJE

ENORAZSEŽNOSTNE TABELE

Poleg skalarnih pozna jezik C tudi dva ***sestavljena*** (angl. aggregate) podatkovna tipa: tabele in strukture. V tem poglavju bomo spoznali enorazsežnostne tabele.

6.1 Deklaracija tabele in indeksni operator

Tabela (angl. array) je podatkovna struktura, ki lahko vsebuje mnogo različnih podatkov, ki pa morajo biti vsi istega tipa. Posameznim podatkom v tabeli pravimo ***elementi***, vsak element pa lahko izberemo neodvisno od drugih z ***indeksnim operatorjem***. Tabelo deklariramo tako, da določimo njeno ime, tip njenih elementov in njeno dimenzijo (tj. število elementov, ki jih lahko tabela hrani). Na primer, takole deklariramo tabelo z imenom `t`, v kateri je prostora za pet podatkov tipa `int`:

```
int t[5];
```

Ker se med pisanjem programa pogosto pojavi potreba po tem, da se velikost tabele spremeni, je praktično, če velikost tabele določimo z uporabo makra:

```
#define N 5
//...
int t[N];
```

Podobno kot zvezdica pri kazalcih ima tudi par oglatih oklepajev pri deklaraciji tabele drugačno vlogo kot pri njeni uporabi. Pri deklaraciji v par oglatih oklepajev zapišemo dimenzijo tabele, pri njeni uporabi pa v par oglatih oklepajev zapišemo indeks elementa, ki

nas zanima. Iz razlogov, ki jih bomo spoznali kmalu, imajo elementi tabele indekse od nič do $N - 1$, pri čemer je N število elementov v tabeli.

Posamezne elemente tabele uporabljamo natanko tako, kot lahko uporabljamo kakršenkoli podatek istega tipa. Na primer, če je `t` tabela petih elementov tipa `int`, lahko njeno vsebino najprej preberemo, potem pa še izpišemo takole:

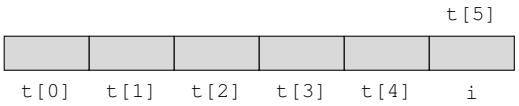
```
for (int i = 0; i < 5; i++) {
  scanf("%d", &t[i]);
}
for (int i = 0; i < 5; i++) {
  printf("%d ", t[i]);
}
```

Izraz `t[i]` pomeni `i`-ti element tabele `t` in se obnaša popolnoma enako kot katerakoli spremenljivka tipa `int`. Zato za branje in pisanje elementov tabele `t` v gornjem programu uporabimo formatno določilo `%d`.

Pozorni moramo biti, da z indeksom ne prekoračimo meje tabele. Jezik C takšne prekoračitve ne preverja, če pa se zgodi, je obnašanje programa nedoločeno. Na primer, v naslednjem delu kode smo prekoračili meje tabele `t` za ena:

```
int t[5];
for (int i = 1; i <= 5; i++) {
  t[i] = 0;
}
```

Z mnogimi prevajalniki povzroči ta na videz nedolžna koda neskončno zanko. V zadnji iteraciji skušamo prirediti vrednost nič elementu `t[5]`, ki pa v tabeli ne obstaja. Izraz `t[5]` nam kljub temu omogoči dostop do pomnilnika, in sicer do mesta, ki se nahaja takoj za elementom `t[4]`. Lahko se zgodi, da prevajalnik ravno tam izbere prostor za spremenljivko `i`. To sploh ni tako nemogoče. Nobenega posebnega razloga namreč ni, da bi bili dve lokalni spremenljivki iste funkcije shranjeni na dveh povsem različnih mestih v pomnilniku. V pomnilniku imamo lahko potemtakem takšno stanje:



Vidimo, da položaj spremenljivke `i` v pomnilniku sovpada z neobstoječim elementom tabele `t[5]`. Na to mesto se v gornjem programu preko spremenljivke `i` najprej po vrsti vpisujejo vrednosti od ena do pet. Takoj ko `i` postane enak pet, pa se zaradi izraza `t[i] = 0` na mesto `i`-ja vpiše vrednost nič. Zaradi tega postane `i` enak nič. To se seveda zgodi, preden se preveri pogoj `i <= 5`, zato se izvajanje zanke v resnici nikoli ne konča.

Standard C99 omogoča uporabo ***tabel s spremenljivo dolžino*** (angl. variable length array). To pomeni, da velikosti tabele ni treba vedeti že ob pisanju kode, temveč jo lahko določimo šele med izvajanjem programa. To storimo tako, da pri deklaraciji za dimenzijo tabele uporabimo spremenljivko. Na primer:

```
int n;
printf("Vnesi dolžino tabele: ");
scanf("%d", &n);
int t[n];    /* Deklaracija tabele z n elementi.  */
for (int i = 0; i < n; i++) {
  t[i] = 0;  /* Vse elemente postavimo na nič.    */
}
```

Inicializacija tabele

Tabelo lahko ob njeni deklaraciji tudi inicializiramo s ***seznamom začetnih vrednosti*** (angl. array initializer). Seznam začetnih vrednosti je omejen s parom zavitih oklepajev, vanj pa po vrsti zapišemo začetne vrednosti posameznih elementov, ločene z vejicami. Na primer:

```
int t[5] = {9, 6, 12, 879, 8};
```

Če je seznam začetnih vrednosti krajši od velikosti tabele, se preostali elementi tabele postavijo na vrednost nič:

```
int t[5] = {9, 6};  /* Začetne vrednosti tabele t so {9, 6, 0, 0, 0}.  */
```

Ta funkcionalnost nam omogoča, da na zelo preprost način postavimo na nič vse elemente tabele hkrati. Dovolj je, da postavimo na nič vrednost prvega elementa, v skladu s prejšnjim pravilom pa se bodo na nič postavili tudi vsi preostali elementi tabele:

```
int t[5] = {0};  /* Začetne vrednosti tabele t so {0, 0, 0, 0, 0}.  */
```

Seznam začetnih vrednosti ne sme biti prazen, zato mora vsebovati vsaj eno vrednost.

Prav tako seznam ne sme biti daljši od podane dolžine tabele:

```
int t[5] = {1, 2, 3, 4, 5, 6};  /* Napaka: preveč inicializatorjev.  */
```

V primeru, ko podamo seznam začetnih vrednosti, lahko dolžino tabele opustimo. V takem primeru določi prevajalnik dolžino tabele glede na število podanih začetnih vrednosti:

```
int t[] = {1, 2, 3, 4, 5, 6};  /* Dolžina tabele je zdaj 6.  */
```

Prevajalnik mora ob deklaraciji tabele v vsakem primeru vedeti, koliko prostora je treba zanjo rezervirati. V naslednjem primeru velikosti tabele ni mogoče določiti:

```
int t[];  /* Dolžine tabele ni mogoče določiti.  */
```

Naloga 6.1 Za vajo razmislite o naslednjih deklaracijah:

```
float t2[4] = {};
int t3[] = {0};
float t4[20];
float t1[];
int t5[2] = {4, 88, 132};
int t5[] = {4, 88, 132, 0};
```

Označite deklaracije, ki vsebujejo napako, in napako komentirajte. Za preostale deklaracije navedite dolžino tabele in vrednosti posameznih elementov takoj po deklaraciji.

Tabel s spremenljivo dolžino, ki jih podpira standard C99, ob deklaraciji seveda ni mogoče inicializirati, saj v času pisanja kode ne poznamo njihove dejanske dolžine.

6.2 Kazalec na tabelo

V jeziku C obstaja zelo tesna povezava med tabelami in kazalci, zato je razumevanje te povezave ključno. Kako tesno so tabele povezane s kazalci, nam pove že dejstvo, da je ime tabele kazalec na njen prvi element. Vzemimo za primer naslednjo deklaracijo:

```
int t[6];
```

Ker je `t` kazalec na prvi element gornje tabele, lahko do vrednosti tega elementa pridemo z operatorjem posredovanja:

```
*t = 42;  /* t[0] ima zdaj vrednost 42.  */
```

Čeprav je ime tabele kazalec, pa to ime ne predstavlja kazalčne spremenljivke. Njegove vrednosti namreč ne moremo spreminjati:

```
int t1[6], t2[6];
t1 = t2;  /* Napaka: t1 ni kazalčna spremenljivka.  */
```

Po krajšem razmisleku nas to niti ne bo presenetilo. Prevajalnik namreč ob deklaraciji tabele rezervira prostor za njene elemente, in ta prostor se v času izvajanja programa ne spreminja. Če bi spremenili vrednost kazalca, ki je ime tabele, bi tako izgubili dostop do elementov te tabele v pomnilniku.

6.3 Kazalčna aritmetika

Vemo že, da lahko kazalčnim spremenljivkam prirejamo vrednosti in da lahko dva kazalca primerjamo z operatorjema enakosti ali različnosti (ko smo na primer kazalec primerjali z ničelnim kazalcem). Kadar pa imamo opravka s kazalcem na tabelo, postanejo smiselne še nekatere aritmetične operacije: kazalcu lahko prištejemo ali od njega odštejemo celoštevilsko vrednost, lahko pa tudi odštejemo dva kazalca med seboj. Dva kazalca lahko tudi primerjamo po velikosti z uporabo relacijskih operatorjev.

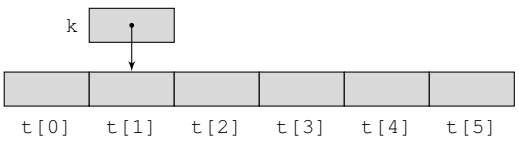
Prištevanje celoštevilske vrednosti

Če kazalcu `k` prištejemo celoštevilsko vrednost n, potem dobimo kazalec, ki kaže na element, ki je v tabeli n mest za elementom, na katerega kaže kazalec `k`.

Vzemimo za primer, da kazalec `k` usmerimo na drugi element tabele `t`:

```
int t[6], *k;
k = &t[1];  /* Kazalcu k priredimo naslov elementa t[1].  */
```

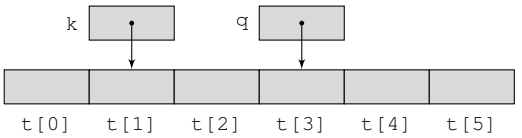
Ko se izvede gornja koda, imamo v pomnilniku naslednje stanje:



Dodajmo zdaj gornji kodi še en kazalec in mu priredimo vrednost izraza `k + 2`:

```
int t[6], *k, *q;
k = &t[1];
q = k + 2;
```

Stanje v pomnilniku je zdaj takšno:



Iz gornjih primerov lahko razberemo, da se vrednost kazalca po prištevanju celoštevilske vrednosti n ne poveča za n. Namesto tega se vrednost kazalca poveča za n, pomnožen s številom bajtov, ki jih zaseda podatek, na katerega ta kazalec kaže. Z drugimi besedami, če vrednost kazalca povečamo za n, prestavimo ta kazalec za n mest naprej po tabeli. O tem se lahko prepričamo tudi, če preizkusimo naslednji del kode (predpostavimo, da zaseda spremenljivka tipa `int` v pomnilniku štiri bajte):

```
int t[6] = {4, 84, 3}, *k;
k = t;
printf("%u\n", (unsigned int) k++);  /* Izpiše: 2686728  */
printf("%u", (unsigned int) k);      /* Izpiše: 2686732  */
```

Ko smo v predzadnji vrstici gornje kode kazalcu `k` prišteli vrednost ena, se je njegova vrednost v resnici povečala za štiri (toliko bajtov namreč zaseda en element tabele). S tem smo ta kazalec usmerili na drugi element tabele `t`, o čemer priča naslednja vrstica kode:

```
printf("%d", *k);  /* Izpiše: 84  */
```

Odštevanje celoštevilske vrednosti

Od kazalca lahko celoštevilsko vrednost tudi odštejemo. Za razliko od prištevanja celoštevilske vrednosti dobimo v tem primeru naslov elementa, ki je ***pred*** elementom, na katerega je kazal prvotni kazalec. Na primer:

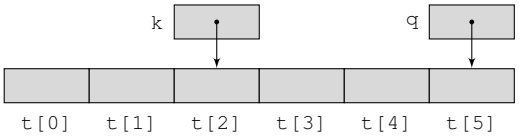
```
int t[6], *k;
k = &t[5];  /* k kaže na t[5].       */
k -= 4;     /* Zdaj k kaže na t[1].  */
```

Odštevanje in primerjava dveh kazalcev

Dva kazalca lahko med seboj tudi odštejemo ali primerjamo po velikosti. Če kazalca med seboj odštejemo, je rezultat število elementov tabele med obema kazalcema. Če kazalca primerjamo, bo večji vedno tisti kazalec, ki kaže na kasnejši element v tabeli. Vzemimo za primer naslednji del kode:

```
int t[6], *k, *q;
k = &t[2];
q = &t[5]
```

Ko se koda izvede, imamo v pomnilniku naslednje stanje:



Ker kaže kazalec `k` na element, ki je za tri elemente pred kazalcem `q`, je vrednost izraza `q - k` enaka tri. Vrednost izraza `k < q` je seveda enaka ena, vrednost izraza `k > q` pa je enaka nič, saj `k` ni večji od `q`.

Na koncu naj poudarimo, da so operacije odštevanja in primerjave kazalcev po velikosti smiselne le med kazalci, ki kažejo na isto tabelo (ali vsaj tik za njo). Če te operacije izvajamo nad kazalci, ki ne kažejo na elemente iste tabele, je njihov rezultat nedoločen.

6.4 Razlika med imenom tabele in kazalčno spremenljivko

Ker so v jeziku C tabele tako zelo povezane s kazalci, ni prav nič nenavadno, da je med imenom tabele in pravo kazalčno spremenljivko komaj kaj razlike. Kakor bomo kmalu videli, lahko kazalčno spremenljivko, ki smo jo usmerili na prvi element tabele, uporabljamo na skoraj popolnoma enak način kot samo ime tabele. Eno pomembno razliko med obema smo že spoznali: ime tabele predstavlja ***konstanten kazalec***, katerega vrednosti ni moč spreminjati. Obstaja pa še ena razlika: operator `sizeof` vrne različne vrednosti, če ga uporabimo nad imenom tabele ali nad kazalčno spremenljivko, ki kaže na isto tabelo:

```
int t[5], *k;
k = t;
printf("%u %u", sizeof t, sizeof k);  /* Izpiše: 20 4  */
```

Čeprav sta tako `t` kot tudi `k` oba kazalca istega tipa in kažeta na isto tabelo, operator `sizeof` zanju javi različni velikosti: `k` je kazalčna spremenljivka, ki v pomnilniku zaseda štiri bajte. Po drugi strani predstavlja `t` tabelo, ki v pomnilniku zaseda 20 bajtov. Ker vrednosti kazalca `t` ne moremo spreminjati – kakor tudi ne moremo spreminjati dimenzije tabele, na katero `t` kaže –, je ta podatek smiseln in dobrodošel. Operator `sizeof` ga zato vrne. Po drugi strani se vrednost kazalčne spremenljivke `k` lahko spreminja, zato sistem ne more vedeti, koliko pomnilnika je rezerviranega na naslovu, ki ga hrani kazalčna spremenljivka. Operator `sizeof` zato namesto tega sporoči velikost same kazalčne spremenljivke.

6.5 Obdelava tabel s kazalci

Vemo že, da lahko kazalec `k` usmerimo na (na primer) tretji element tabele `t` s stavkom `k = &t[2]`. Glede na to, da je `t` kazalec, lahko isto naredimo tudi s stavkom:

```
k = t + 2;
```

Če želimo priti zdaj do vrednosti tega tretjega elementa, lahko naredimo to bodisi preko kazalca `k` bodisi preko kazalca `t + 2`. Poleg tega lahko na obeh kazalcih uporabimo bodisi operator posredovanja bodisi indeksni operator. Tako naslednji štirje stavki vsi izpišejo vrednost tretjega elementa tabele `t` (`t` kaže na prvi, `k` pa na tretji element tabele):

```
printf("%d\n", *k);
printf("%d\n", k[0]);
```

```
printf("%d\n", *(t + 2));
printf("%d\n", t[2]);
```

Splošno lahko zapišemo, da je v primeru, ko kazalec `k` kaže na katerikoli element določene tabele in je `n` celo število, izraz:

```
*(k + n)
```

enakovreden izrazu[1]:

```
k[n]
```

Kadar je `n` enak nič, ne potrebujemo okroglih oklepajev. Tako velja tudi, da je izraz:

```
*k
```

enakovreden izrazu[1]:

```
k[0]
```

Zdaj lahko pojasnimo, zakaj ima indeks prvega elementa tabele vrednost nič (oziroma, zakaj se štetje začne z nič, in ne z ena, česar smo navajeni iz vsakdanjega življenja): indeks elementa v resnici predstavlja ***odmik*** od elementa, na katerega kaže kazalec, na katerem uporabimo indeksni operator. Indeks nič potemtakem pomeni element, na katerega kaže kazalec, indeks ena pomeni naslednji (drugi) element in tako dalje.

Naslednji primer kaže, kako lahko z uporabo kazalca (brez indeksnega operatorja) seštejemo vrednosti elementov tabele `a`:

```
#define N 4
//...
int a[N] = {7, 2, 9, 2};
int vsota = 0;
for (int *k = a; k < a + N; k++) {
  vsota += *k;
}
```

Seveda lahko isti učinek dosežemo tudi brez dodatne kazalčne spremenljivke in z uporabo indeksov, kar je za marsikoga tudi preglednejši način:

```
//...
int vsota = 0;
for (int i = 0; i < N; i++) {
  vsota += a[i];
}
```

[1]Zaradi različnih prednosti operatorja posredovanja in indeksnega operatorja se lahko izraza obnašata različno, če nad njima izvajamo določene dodatne operacije. Na primer, izraz `k[n]++` za ena poveča vrednost elementa v tabeli, ki je za `n` mest za elementom, na katerega kaže `k`. Po drugi strani povzroči izraz `*(k + n)++` napako pri prevajanju. Operator povečanja ima namreč prednost pred operatorjem posredovanja, zato v zadnjem izrazu v resnici skušamo povečati (spremeniti) vrednost izraza `k + n`, ki pa ni leva vrednost. Za isti učinek bi morali uporabiti dodaten par oklepajev, da spremenimo vrstni red izvajanja operatorja posredovanja in povečanja: `(*(k + n))++`.

Razlika je tudi med izrazoma `k[0]++`, ki za ena poveča vrednost elementa, na katerega kaže `k`, in `*k++`, ki za ena poveča vrednost kazalca `k`. Za isti učinek bi spet morali uporabiti oklepaje: `(*k)++`.

Razlog, ki govori v prid uporabe kazalca, je učinkovitost takšne kode. Vendar za mnogo sodobnih prevajalnikov ta trditev ne velja več: tudi če uporabimo indekse, bomo z mnogimi izvedbami prevajalnikov dobili prav tako učinkovito kodo. Na vsak način pa nam takšni primeri pomagajo k boljšemu razumevanju, kako kazalci delujejo.

6.6 Tabela kot argument funkcije

Kadar uporabimo tabelo kot argument funkcije, se v parameter funkcije vedno prenese le naslov tabele. To dejstvo ima pomemben vpliv na to, kako takšno funkcijo napišemo. Poglejmo si primer funkcije, ki vrne vsoto elementov podane tabele:

```
int sestej(int t[], int n) {
  int vsota = 0;
  for (int i = 0; i < n; i++) {
    vsota += t[i];
  }
  return vsota;
}
```

Prvi parameter funkcije (`t`) je tabela, kar vidimo iz para oglatih oklepajev, ki sledi imenu parametra. Funkcijo `sestej` lahko kličemo takole:

```
#define N 5
//...
int a[N] = {3, 8, 2, 1, 5};
printf("%d", sestej(a, N));  /* Izpiše: 19  */
```

Iz primera vidimo, da tabelo podamo tako, da podamo njen naslov (`a`). Vendar zgolj iz naslova ni mogoče ugotoviti, koliko ima tabela elementov. Zato moramo funkciji podati še drugi argument (`N`), ki predstavlja število elementov v tabeli.

Prevajalnik parameter `t` v definiciji gornje funkcije `sestej` v resnici obravnava kot kazalec. Zato ni popolnoma nobene razlike, če definicijo začnemo takole:

```
int sestej(int *t, int n) {
//...
```

Ker ima funkcija preko kazalca poln dostop do elementov tabele, lahko te elemente tudi spreminja. Če tega ne želimo – in se želimo zaščititi proti nenamerni spremembi vrednosti kakšnega elementa tabele –, lahko pred deklaracijo parametra dodamo rezervirano besedo `const`:

```
int sestej(const int *t, int n) {
//...
```

To ne pomeni, da ne moremo spremeniti vrednosti kazalca `t`. Pomeni pa, da ne moremo spremeniti vrednosti, na katero ta kazalec kaže. Vzemimo za primer, da zaključimo podano tabelo s čuvajem z vrednostjo nič. Potem lahko napišemo funkcijo `sestej` nekoliko krajše:

```
int sestej(const int *t) {
  int vsota = 0;
  while (*t != 0) {
    vsota += *t++;  /* Vrednost kazalca lahko spreminjamo.  */
  }
```

```
  return vsota;
}
```

Ker ima operator ++ prednost pred operatorjem posredovanja, izraz `*t++` poveča vrednost kazalca `t`, in ne vrednosti, na katero ta kazalec kaže. Če pa bi po pomoti želeli spremeniti element, na katerega kaže kazalec, bi prevajalnik javil napako:

```
vsota += (*t)++;  /* Napaka: preko kazalca, deklariranega
                     z besedo const, ni mogoče spreminjati
                     vsebine pomnilnika.                  */
```

Naloga 6.2 Za vajo napišite funkcijo, ki kot parameter sprejme tabelo celih števil skupaj z informacijo o njeni dolžini. Funkcija naj vrne podatek o tem, koliko različnih vrednosti je v podani tabeli.

Primer klica funkcije:

```
#define N 13
int t[N] = {5, 9, 5, 2, 6, 77, 15, 5, 5, 77, 6, 2, 8};
printf("%d", prestejRazlicne(t, N));  /* Izpiše: 7  */
```

6.7 Vračanje naslova elementa v tabeli

Včasih je koristno, če funkcija vrne naslov določenega elementa v podani tabeli. Na primer, naslednja funkcija vrne naslov najmanjšega elementa v podani tabeli:

```
int *poisciNajmanjsega(int *t, int n) {
  int min = 0;
  for (int i = 1; i < n; i++) {
    if (t[i] < t[min]) {
       min = i;
    }
  }
  return &t[min];  /* Vrne naslov elementa z indeksom min.  */
}
```

Takšno funkcijo lahko uporabimo za urejanje elementov tabele z ***navadnim vstavljanjem*** (angl. insertion sort). Po postopku navadnega vstavljanja v vsaki iteraciji poiščemo najmanjši element v neurejenem delu tabele in ga vstavimo na konec urejenega dela tabele.

Naslednji primer kaže kodo, ki uredi tabelo `a` z `N` elementi po postopku navadnega vstavljanja. Pozorni bodite, kako smo uporabili klic funkcije `poisciNajmanjsega` kot enega od argumentov funkcije `menjaj`, ki smo jo napisali na strani 84:

```
#define N 6
//...
int main(void) {
  int a[N] = {7, 2, 9, 2, -9, 3};
  for (int i = 0; i < N; i++) {
    menjaj(&a[i], poisciNajmanjsega(&a[i], N - i));
  }
  for (int i = 0; i < N; i++) {
    printf("%d ", a[i]);  /* Izpiše: -9 2 2 3 7 9  */
  }
  return 0;
}
```

V prvi od obeh zank `for` v gornjem programu je v vsaki iteraciji že urejenih `i` elementov. Z drugimi besedami, elementi z indeksi od nič do `i-1` so že urejeni, neurejeni del tabele pa predstavljajo elementi od vključno `i`-tega do zadnjega.

Klic funkcije `poisciNajmanjsega` vsakokrat vrne naslov najmanjšega od elementov iz neurejenega dela tabele. Namreč, kot izhodiščni naslov, od koder naj funkcija začne iskati najmanjši element, smo podali naslov `i`-tega elementa (tj. `&a[i]`), za število elementov, ki jih mora pregledati, pa smo podali vrednost izraza `N - i`.

Funkcija `menjaj` prejme dva argumenta: prvi je naslov `i`-tega elementa v tabeli `a`, drugi pa je naslov najmanjšega od elementov iz neurejenega dela tabele (slednjega vrne funkcija `poisciNajmanjsega`). Funkcija `menjaj` nato med seboj zamenja vrednosti elementov, ki se nahajata na podanih naslovih. Po tej menjavi se na `i`-tem mestu pojavi najmanjši element iz doslej neurejenega dela tabele, ki je zdaj postal zadnji in največji element v urejenem delu tabele.

Naj omenimo, da je gornja rešitev nekoliko neučinkovita zaradi dveh klicev funkcij v vsaki iteraciji. Vendar primer ponazarja, kako lahko na eleganten način uporabimo naslov elementa v tabeli, ki ga funkcija vrne prek stavka `return`. Standardna knjižnica <string.h>, ki vsebuje funkcije za delo z znakovnimi nizi, vsebuje polno funkcij, ki vračajo kazalec na ustrezen del znakovnega niza.

6.8 Znakovni nizi

Znakovni niz (angl. string) v programskem jeziku C je preprosto tabela vrednosti tipa `char`. Z znakovnimi nizi navadno zapisujemo besedilo ali dele besedila. Ker je obdelava besedila pomemben del mnogih uporabniških programov, pozna jezik C v zvezi z znakovnimi nizi nekaj posebnosti, ki ne veljajo splošno za tabele. Te posebnosti nam v mnogočem olajšajo delo z znakovnimi nizi.

Konstanten znakovni niz

Konstanten znakovni niz oziroma ***dobesedna navedba znakovnega niza*** (angl. string literal) v jeziku C je poljubno besedilo v ***dvojnih*** navednicah:

```
"Tako je videti konstanten znakovni niz."
```

Takšne znakovne nize smo srečali že velikokrat, zdaj pa je čas, da se vprašamo, kaj konstanten znakovni niz je in kako je shranjen v pomnilniku.

Kadar cejevski prevajalnik v programu naleti na konstanten znakovni niz dolžine n znakov, zanj najprej rezervira $n+1$ bajtov pomnilnika. V rezervirani del pomnilnika shrani vse znake znakovnega niza in na koncu doda še ***zaključni ničelni znak*** (angl. terminating null character). Ničelni znak je znak, katerega koda ASCII je 0, zapišemo pa ga tudi kot ubežno sekvenco `\0`[2].

Na primer, konstanten znakovni niz `"xyz"` je v pomnilniku shranjen kot tabela štirih znakov (`x`, `y`, `z` in `\0`):

x	y	z	\0

[2]Bodite pozorni na razliko med ničelnim znakom `\0` (poševnica nazaj in nič) in znakom `0` (nič). Koda ASCII prvega znaka je 0, drugega pa 48.

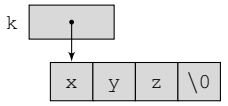
Konstanten znakovni niz je lahko prazen (`""`), vendar mora imeti tudi prazen niz zaključni ničelni znak. Niz `""` je zato videti v pomnilniku takole:

Pozorni moramo biti na razliko med enojnimi in dvojnimi navednicami: z enojnimi navednicami zapisujemo posamezen znak (ki vedno zasede en bajt pomnilnika), z dvojnimi navednicami pa znakovni niz (ki ima lahko poljubno število znakov in vedno vsebuje zaključni ničelni znak). Tako na primer izraz `'a'` v pomnilnik zapiše en bajt (vrednost 97, ki je koda ASCII črke a), izraz `"a"` pa v pomnilnik zapiše dva bajta (vrednosti 97 in 0).

Ker je znakovni niz shranjen v obliki tabele, ga prevajalnik obravnava kot kazalec tipa `char *`. Zato lahko uporabimo konstanten znakovni niz (tj. ***dobesedno navedbo*** znakovnega niza) povsod, kjer se pričakuje kazalec na podatek tipa `char`. Na primer:

```
char *k;
k = "xyz";
```

Ta priredilni stavek ne kopira niza `"xyz"`, temveč preprosto usmeri `k` na prvi znak tega niza v pomnilniku:



V zadnjem primeru smo na znakovni niz usmerili kazalčno spremenljivko `k`, kar nas lahko pripelje v skušnjavo, da uporabimo `k` za spreminjanje tega znakovnega niza:

```
char *k = "xyz";
*(k + 1) = 'w';  /* Nedoločeno obnašanje.  */
k[0] = 'q';      /* Nedoločeno obnašanje.  */
```

Takšen poskus vodi do nedoločenega obnašanja, saj konstantnega znakovnega niza ni dovoljeno spreminjati. Da bi se izognili težavam, lahko v takšnem primeru uporabimo rezervirano besedo `const`. Tako bo že prevajalnik preprečil poskus spreminjanja konstantnega znakovnega niza:

```
const char *k = "xyz";
*(k + 1) = 'w';  /* Napaka: prostor je namenjen samo branju.  */
k[0] = 'q';      /* Napaka: prostor je namenjen samo branju.  */
```

Eden izmed razlogov za to, da konstantnih znakovnih nizov ne smemo spreminjati, so določene optimizacije, ki jih uporabljajo prevajalniki za povečanje učinkovitosti kode. Na primer, če v kodi uporabimo dva popolnoma enaka konstantna niza, se lahko prevajalnik odloči, da bo v pomnilniku ustvaril en sam takšen znakovni niz (obnašanje je v tem primeru nepredpisano). Posledica tega je, da ne moremo vedeti, kakšno vrednost bo izpisal naslednji stavek:

```
printf("%d", "xyz" == "xyz");  /* Izpiše bodisi 0 bodisi 1.  */
```

Izpiše se lahko bodisi vrednost nič bodisi ena, odvisno od tega, ali bo prevajalnik shranil vsakega od dveh nizov `"xyz"` v svoj del pomnilnika, ali pa bo oba niza shranil kot en niz. Morda se bo kdo čudil, kako lahko izraz `"xyz" == "xyz"` vrne vrednost nič, saj sta

vendar vrednosti na obeh straneh primerjalnega operatorja enaki. Ne smemo pozabiti, da je konstanten znakovni niz v resnici kazalec. V tem primeru zato med seboj primerjamo dva naslova, ki sta lahko tudi različna, če je vsak od obeh (sicer enakih) znakovnih nizov shranjen na svojem naslovu v pomnilniku.

Ker je konstanten znakovni niz kazalec, lahko nad njim uporabimo indeksni operator. Na primer, izraz `"xyz"[2]` vrne črko z. To dejstvo lahko izkoristimo, če na primer potrebujemo funkcijo, ki desetiško vrednost med nič in 15 pretvori v šestnajstiško številko:

```
char desetiskoVSestnajstiskiZnak(int des) {
  return "0123456789ABCDEF"[des];
}
//...
printf("%c", desetiskoVSestnajstiskiZnak(12));  /* Izpiše: C  */
```

Deklarirani znakovni nizi

V zadnjem razdelku smo videli, da je konstanten znakovni niz kazalec na tabelo znakov v pomnilniku. Spoznali smo tudi, da vsebine te tabele ne moremo spreminjati. To je posebnost, ki si jo bomo zapomnili v zvezi z znakovnimi nizi. Znakovni niz pa lahko deklariramo tudi kot običajno tabelo in posamezne znake obravnavamo kot elemente tipa `char`. Paziti moramo le, da nikoli ne pozabimo na ničelni znak, ki mora zaključiti veljavni znakovni niz. Tako lahko na primer deklariramo tabelo s prostorom za 80 znakov:

```
#define DOLZINA_NIZA 80
//...
char niz[DOLZINA_NIZA + 1];
```

S tem smo ustvarili prostor za 80 znakov in dodaten ničelni znak. Če bo znakovni niz, ki ga bomo shranili v tabelo `niz`, krajši, nič hudega: konec znakovnega niza bo vedno označen z ničelnim znakom, ki se bo zaradi krajšega niza pojavil ustrezno prej v tabeli. Preostali bajti rezerviranega pomnilnika (za ničelnim znakom) bodo še vedno ostali rezervirani, le da bodo podatki v njih brez pomena.

Znakovni niz lahko ob deklaraciji tudi inicializiramo:

```
char niz[DOLZINA_NIZA + 1] = "Enega pa še lahko.";
```

Pomembno je dejstvo, da pri tem ne gre za konstanten znakovni niz, temveč za seznam začetnih vrednosti, kakršnega smo spoznali pri inicializaciji običajne tabele. V resnici je gornji zapis samo okrajšana različica naslednjega zapisa:

```
char niz[DOLZINA_NIZA + 1] = {'E', 'n', 'e', 'g', 'a', ' ',
                              'p', 'a', ' ', 'š', 'e', ' ',
                              'l', 'a', 'h', 'k', 'o', '.', '\0'};
```

To dejstvo ima pomembno posledico, da lahko tak znakovni niz spreminjamo:

```
char niz[DOLZINA_NIZA + 1] = "abc";
*niz = 'x';    /* Niz postane "xbc".  */
niz[1] = 'y';  /* Niz postane "xyc".  */
```

Pri inicializaciji znakovnega niza veljajo ista pravila kot za običajne tabele:

```
char niz1[] = "abc";    /* niz1 zaseda 4 bajte.      */
char niz2[3] = "defg";  /* Preveč inicializatorjev.  */
```

```
char niz3[9] = "hij";    /* Zadnjih 6 bajtov v nizu
                            niz3 se postavi na 0.     */
```

Pozorni moramo biti pri naslednji inicializaciji:

```
char niz[3] = "xyz";
```

Čeprav smo pri tem pozabili na prostor za ničelni znak, nas nekateri prevajalniki na takšno napako ne opozorijo.

Naloga 6.3 Za vajo pojasnite, kakšna je razlika med naslednjima dvema deklaracijama:

```
char sporocilo[] = "Sistem se posodablja.";
char *sporocilo = "Sistem se posodablja.";
```

Branje in pisanje znakovnih nizov

Znakovni nizi predstavljajo besedilo, besedilo pa se v programih mnogokrat pojavi kot vhodni ali izhodni podatek. Zato ni nič nenavadnega, da obstaja za njihovo branje in pisanje s funkcijama `scanf` in `printf` posebno formatno določilo (`%s`). Če imamo v programu deklariran znakovni niz z imenom `niz`, potem lahko ta niz izpišemo na izhod takole:

```
#define D 80
//...
char niz[D+1];
//...
printf("%s", niz);
```

Naj povemo, da v primeru formatnega določila `%s` funkcija `printf` pričakuje argument, ki je kazalec tipa `char *`, kar `niz` v gornjem primeru je. Seveda lahko na enak način uporabimo tudi deklariran kazalec tipa `char *`, pri čemer mora tak kazalec kazati na veljaven znakovni niz. Na primer:

```
char *k = "Pomembno sporočilo.";
printf("%s", k);
```

Na obliko izpisa lahko v določeni meri vplivamo. Na primer, drugi od naslednjih dveh klicev funkcije `printf` izpiše vsebino znakovnega niza tako, da za izpis porabi deset mest in ga poravna desno:

```
char niz[D+1] = "ABCD";
//...
printf("1234567890\n");
printf("%10s", niz);
```

Dobimo naslednji izpis:

```
1234567890
      ABCD
```

Seveda lahko formatno določilo `%s` uporabimo skupaj z drugimi elementi formatnega niza:

```
char niz[D+1] = "ABCD";
int x = 13;
printf("%s-%d", niz, x);  /* Izpiše: ABCD-13  */
```

Formatni niz funkcije `printf` navadno podajamo kot konstanten znakovni niz. Ker vemo, da je konstanten znakovni niz v resnici kazalec tipa `char *`, nas ne bo presenetilo, da lahko kot prvi argument funkcije `printf` podamo kakršenkoli kazalec na znakovni niz:

```
char niz[D+1] = "ABCD";
char *k = "EFGH";
printf(niz);     /* Izpiše: ABCD  */
printf(k);       /* Izpiše: EFGH  */
printf("IJKL");  /* Izpiše: IJKL  */
```

Branje s funkcijo `scanf` poteka nekoliko drugače. Formatno določilo `%s` prebere besedilo od prvega znaka, ki je različen od presledka, do zadnjega znaka pred naslednjim presledkom. Na primer, če na vhod vtipkamo `2. april`, bo naslednja koda v `niz1` shranila vtipkano besedilo do prvega presledka (tj. `2.`), v `niz2` pa `april`:

```
char niz1[D+1], niz2[D+1];
scanf("%s%s", niz1, niz2);       /* Vtipkamo: 2. april  */
printf("-%s-%s-", niz2 , niz1);  /* Izpiše: -april-2.-  */
```

Opozorimo naj na dejstvo, da v tem primeru pred argumenti funkcije `scanf` ne potrebujemo naslovnega operatorja, saj sta `niz1` in `niz2` kazalca.

S funkcijo `scanf` in formatnim določilom `%s` torej ne moremo brati presledkov[3]. Še hujša težava te funkcije je, da se ne moremo zaščititi pred vnosom, ki vsebuje več znakov, kot imamo na voljo rezerviranega pomnilnika. V primeru, da vnesemo preveč znakov, bo funkcija `scanf` odvečne znake vpisala v nerezerviran del pomnilnika, posledice tega pa so nepredvidljive. Obe težavi lahko rešimo tako, da napišemo svojo funkcijo za branje znakovnega niza, kakor kaže naslednji primer.

Naslednja funkcija prebere celotno besedilo, ki ga vtipkamo, vključno s presledki. Če vtipkamo več znakov, kot je zanje prostora v pomnilniku, odvečne znake funkcija enostavno zavrže:

```
int beriVrstico(char *s, int n) {
  int i = 0;
  char znak;
  while (1) {
    scanf("%c", &znak);
    if (znak == '\n') break;
    if (i < n) {  /* Shrani le prvih n znakov.  */
      s[i++] = znak;
    }
  }
  s[i] = 0;  /* Zaključni ničelni znak.          */
  return i;  /* Število prebranih znakov.        */
}
```

Funkcija ni nič posebnega. Kot prvi parameter (`s`) sprejme naslov, kamor želimo shraniti prebrani znakovni niz. Drugi parameter (`n`) predstavlja največje število znakov, ki jih

[3] Alternativa funkciji `scanf` za branje znakovnih nizov je funkcija `gets`, ki prebere celoten niz skupaj z vsemi presledki. Tako kot `scanf` je tudi funkcija `gets` del standardne knjižnice <stdio.h>. Kličemo jo tako, da ji kot edini argument podamo kazalec na znakovni niz, v katerega želimo shraniti vneseno besedilo.

lahko shranimo na podani naslov. Iz kode je razvidno, da `n` pomeni število znakov brez zaključnega znaka. V pomnilniku na naslovu `s` mora biti torej prostora za `n + 1` znakov.

Napisano funkcijo lahko uporabimo na naslednji način:

```
#define N 16
//...
int main(void) {
  char niz[N+1];
  printf("Vtipkaj besedilo: ");
  printf("Uspešno prebranih %d znakov.\n", beriVrstico(niz, N));
  printf("Prebrano besedilo: ");
  printf(niz);
  return 0;
}
```

Če program zaženemo, bo deloval takole:

```
Vtipkaj besedilo: Program bo verjetno nekaj tega odrezal.
Uspešno prebranih 16 znakov.
Prebrano besedilo: Program bo verje
```

Če pa vtipkamo 16 znakov ali manj, bo funkcija uspešno prebrala vse vtipkane znake:

```
Vtipkaj besedilo: Tole bo v redu.
Uspešno prebranih 15 znakov.
Prebrano besedilo: Tole bo v redu.
```

Standardna knjižnica <stdio.h> vsebuje funkcijo `fgets`, ki deluje na podoben način kot naša funkcija `beriVrstico`. Funkcija `fgets` prebere eno vrstico besedila iz podanega vhodnega toka, pri čemer lahko omejimo število prebranih znakov. Na primer:

```
#define N 16
//...
char niz[N+1];
printf("Vtipkaj besedilo: ");
fgets(niz, N + 1, stdin);  /* Besedilo bere s standardnega
                   vhoda (stdin). Prebere največ N znakov,
                   na koncu doda še ničelni znak.          */
printf(niz);
```

Funkcija `fgets` se od funkcije `beriVrstico` med drugim razlikuje po tem, da vrne naslov prebranega niza. Zato lahko klic funkcije `fgets` uporabimo neposredno kot argument kakšne druge funkcije, katere parameter je znakovni niz. Na primer:

```
printf(fgets(niz, N + 1, stdin));  /* Prebrani niz takoj
                                      izpiše na izhod.    */
```

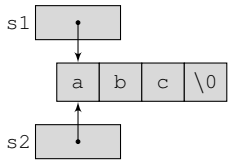
Kopiranje in primerjava znakovnih nizov

Za konec tega poglavja si oglejmo še dve pomembni operaciji nad znakovnimi nizi: kopiranje in primerjavo. Ugotovili smo že, da lahko z uporabo klasičnih operatorjev za prirejanje in primerjavo kopiramo in primerjamo zgolj naslove znakovnih nizov. Če želimo izvajati omenjene operacije nad njihovo vsebino, moramo poskrbeti, da se bodo ustrezne operacije izvedle nad vsakim znakom posebej.

Na primer, v drugi vrstici naslednjega dela kode ne kopiramo znakovnega niza, temveč zgolj usmerimo kazalčno spremenljivko `s2` na znakovni niz, na katerega že kaže kazalčna spremenljivka `s1`:

```
char *s1 = "abc", *s2;
s2 = s1;
```

Priredilni stavek `s2 = s1` kopira naslov, ki je v spremenljivki `s1`, v spremenljivko `s2`. Ko se koda izvede, imamo v pomnilniku takšno stanje:



Če želimo kopirati znakovni niz, moramo kopirati vsakega od njegovih znakov posebej. Takole je videti funkcija, ki kopira znakovni niz, ki se začne na naslovu `s2`, v del pomnilnika, ki se začne na naslovu `s1`:

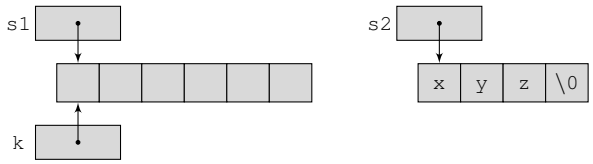
```
char *kopirajNiz(char *s1, const char *s2) {
  char *k;
  for (k = s1; *s2 != 0; k++, s2++) {
    *k = *s2;
  }
  *k = 0;
  return s1;
}
```

Na naslovu `s1` mora biti seveda dovolj prostora za vse znake niza `s2` vključno z ničelnim znakom. Parameter `s2` smo deklarirali z uporabo besede `const`. Spomnimo se, da to pomeni, da zaradi tega ni dovoljeno spreminjati podatkov, na katere kaže `s2`. To storimo zaradi varnosti, da v funkciji `kopirajNiz` ne bi po pomoti spremenili izvornega znakovnega niza.

Funkcijo `kopirajNiz` lahko uporabimo takole:

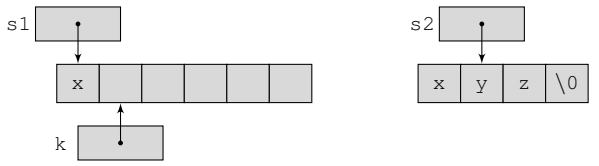
```
char niz[6];
kopirajNiz(niz, "xyz");
printf(niz);  /* Izpiše: xyz  */
```

Ko funkcijo kličemo, je v trenutku, ko se začne v njeni definiciji prvič izvajati telo zanke `for`, stanje v pomnilniku takšno:

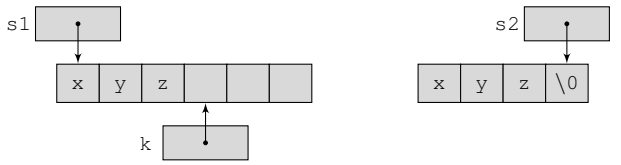


Ob klicu funkcije `kopirajNiz` se vrednost kazalca `niz` kopira v parameter `s1`. Zato kaže v tem trenutku kazalec `s1` na tabelo šestih znakov, ki je v resnici znakovni niz `niz`. Poleg tega se vrednost konstantnega kazalca `"xyz"` kopira v parameter `s2`, ki zato kaže na ta konstantni niz. Na sliki vidimo še, da na začetek tabele s šestimi znaki kaže tudi kazalec `k`. Tega smo tja usmerili v prvem izrazu zanke `for` (tj. `k = s1`).

V telesu zanke `for` kopiramo znak, na katerega kaže `s2`, na mesto, na katero kaže `k`. Takoj za tem povečamo tako `k` kot tudi `s2` za ena. Po prvi iteraciji zanke imamo zato v pomnilniku naslednje stanje:



Zanka se konča, ko `s2` kaže na ničelni znak. Stanje v pomnilniku je takrat takšno:



Kopija niza na naslovu `s1` v tem trenutku še ni popolna. Stavek `*k = 0`, ki sledi zanki `for`, vpiše na konec niza `s1` še zaključni ničelni znak.

Čeprav to ni potrebno, pa funkcija `kopirajNiz` vrne naslov ciljnega znakovnega niza, kar nam včasih prihrani nekaj tipkanja. Na primer, če bi želeli niz `"xyz"` kopirati na dve mesti, bi to lahko naredili takole:

```
char niz1[10], niz2[10];
kopirajNiz(niz1, kopirajNiz(niz2, "xyz"));  /* niz1 in niz2 zdaj
                                               hranita vsak svojo
                                               kopijo niza "xyz".  */
```

Naloga 6.4 Gornjo funkcijo `kopirajNiz` lahko napišemo tudi precej krajše:

```
char *kopirajNiz(char *s1, const char *s2) {
  char *k = s1;
  while (*k++ = *s2++) {}
  return s1;
}
```

Razmislite in pojasnite, zakaj ta koda deluje enako kot njena daljša različica.

Naloga 6.5 Predelajte obe različici funkcije `kopirajNiz` (tj. primer na strani 110 in primer iz naloge 6.4), tako da namesto operatorjev posredovanja uporabite indeksne operatorje.

Naloga 6.6 Za vajo razmislite o naslednji kodi, v kateri dvakrat kličemo gornjo funkcijo `kopirajNiz`:

```
char niz[] = "123456789";
char *s = "abcdefghi";

kopirajNiz(niz, s);
kopirajNiz(s, niz);
```

Eden od gornjih dveh klicev funkcije `kopirajNiz` je napačen. Kateri in zakaj?

Na strani 105 smo spoznali, da lahko primerjava dveh enakih konstantnih nizov z operatorjem enakosti (==) vrne vrednost bodisi nič bodisi ena, odvisno od tega, ali shrani

prevajalnik dva enaka niza na dve različni lokaciji ali pa za oba niza uporabi eno in isto kopijo. V primeru deklariranih znakovnih nizov se seveda nikoli ne bo zgodilo, da sta naslova enaka:

```
char niz1[] = "xyz";
char niz2[] = "xyz";
printf("%d", niz1 == niz2);  /* V vsakem primeru izpiše
                                vrednost nič.            */
```

V obeh gornjih primerih smo primerjali zgolj naslova, ne pa ***vsebin*** znakovnih nizov, kar nam redko koristi. Če želimo med seboj primerjati vsebini dveh znakovnih nizov, moramo primerjati posamezne znake, podobno kot smo morali posamezne znake tudi kopirati. Po vrsti primerjamo po dva in dva znaka, dokler ne naletimo bodisi na dva znaka, ki sta različna, bodisi na konec enega od obeh nizov. Spodnja funkcija primerja znakovna niza `s1` in `s2` ter vrne vrednost, ki je manjša, enaka ali večja od nič, odvisno od tega, ali je `s1` manjši, enak ali večji od `s2`. Niza primerjamo tako, kot bi primerjali dve besedi po abecedi, pri čemer upoštevamo vrednosti kod ASCII posameznih znakov:

```
int primerjajNiza(const char *s1, const char *s2) {
  for (; *s1 == *s2 && *s1; s1++, s2++) {}
  return *s1 - *s2;
}
```

Zanka `for` v gornji funkciji se konča, kakor hitro naletimo bodisi na dva znaka, ki sta med seboj različna, bodisi na ničelni znak v prvem nizu. Ni pa nam treba posebej preverjati, ali smo naleteli na ničelni znak v drugem nizu. Namreč, če v prvem nizu še nismo prišli do ničelnega znaka, se pa ničelni znak pojavi v drugem nizu, potem zagotovo ne bo izpolnjen pogoj `*s1 == *s2`, zaradi česar se zanka ustavi.

Funkcija na koncu vrne razliko dveh znakov, pri katerih se je zanka `for` ustavila.

Funkcijo `primerjajNiza` lahko kličemo takole:

```
if (primerjajNiza("slon", "muha") > 0) {  /* Če je slon večji od muhe.  */
//...
```

Naloga 6.7 Za vajo predelajte gornjo definicijo funkcije `primerjajNiza`, tako da namesto operatorjev posredovanja uporabite indeksne operatorje.

Funkciji za kopiranje in primerjavo nizov, ki smo ju napisali za vajo v tem razdelku, najdemo v standardni knjižnici <string.h> pod imenoma `strcpy` (angl. string copy) in `strcmp` (angl. string compare). V tej knjižnici je še cela vrsta uporabnih funkcij za delo z znakovnimi nizi. Z znanjem, ki ste si ga pridobili, jih boste znali sami poiskati v ustrezni literaturi ter jih uporabiti pri svojem delu, nekaj pa jih bomo spoznali še skupaj.

6.9 Naloge

Naloga 6.8 Napišite funkcijo `stVecjihOdSred`, ki kot parametra sprejme tabelo realnih števil in podatek o dolžini tabele. Funkcija naj vrne število elementov v tabeli, ki so večji od srednje vrednosti vseh elementov tabele.

Primer klica funkcije:

```
#define N 6
//...
```

```
double t[N] = {1.5, 2.1, 1.8, 2.3, 16.9, 0.9};
printf("%d", stVecjihOdSred(t, N));  /* Izpiše: 1  */
```

Naloga 6.9 Napišite funkcijo, ki izračuna drseče povprečje (angl. simple moving average) podanega zaporedja. Drseče povprečje je zaporedje povprečnih vrednosti različnih podmnožic originalnega zaporedja. Računali bomo najenostavnejše možno drseče povprečje: i-to vrednost drsečega povprečja (DP_i) bomo dobili preprosto kot povprečje n vrednosti originalnega zaporedja od z_i do z_{i+n-1}:

$$DP_i = \frac{1}{n} \sum_{k=0}^{n-1} z_{i+k}$$

Na primer, če se odločimo za $n = 3$, potem dobimo iz zaporedja:

$$1, 2, 3, 5, 9, 14, 66$$

takšno drseče povprečje:

$$2{,}0,\ 3{,}3,\ 5{,}7,\ 9{,}3,\ 29{,}7.$$

Dobljeno drseče povprečje bo za $n - 1$ vrednosti krajše od originalnega zaporedja.

Pomoč: Izhajate lahko iz naslednjega prototipa funkcije:

```
void drsPovp(double dp[], double t[], int dolz, int n);
```

Pri tem je `t` tabela z originalnim zaporedjem, `dolz` je dolžina tabele `t`, `n` pa število vrednosti, ki jih uporabimo za računanje drsečega povprečja. Rezultat naj se na koncu nahaja v tabeli `dp`.

Opomba: Drseča povprečja se uporabljajo v statistični analizi podatkov. Drseče povprečje navadno primerjamo z originalnim zaporedjem, pri čemer je pomembno, kako obe zaporedji poravnamo. V finančnih aplikacijah, kjer napovedujemo bodoče dogodke na podlagi preteklih podatkov, poravnamo zadnji element drsečega povprečja z zadnjim elementom originalnega zaporedja. Tako predstavlja vsak element drsečega povprečja povprečje zadnjih n originalnih vrednosti. To pomeni, da je povprečje časovno zakasnjeno glede na originalne vrednosti. V znanosti in inženirstvu takšne časovne zakasnitve običajno ne želimo, zato drseče povprečje osredinimo glede na originalno zaporedje.

Naloga 6.10 Podan je naslednji program:

```
#include <stdio.h>

int *poisci(int t[], int x) {
  for (; *t != -1; t++) {
    if (*t == x) {
      return t;
    }
  }
  return NULL;
}
```

```
int main(void) {
  int t[] = {1, 8, 2, 6, 2, 8, 1, 9, 6, 1, -1};
  int *k = poisci(t, 2);
  while (k) {
    *k = 0;
    k = poisci(k, 2);
  }
  //...
  return 0;
}
```

Kakšna bo vsebina tabele `t`, takoj potem ko se zaključi zanka `while`? Utemeljite odgovor.

Naloga 6.11 Predelajte funkcijo, ki ste jo napisali za rešitev naloge 6.8, tako da bo funkcija vrnila naslov prvega elementa v tabeli, ki je večji od srednje vrednosti vseh elementov tabele.

Primer klica funkcije:

```
#define N 6
//...
double *k, t[N] = {1.5, 7.1, 1.8, 2.3, 16.9, 0.9};
k = prviVecjiOdSred(t, N)
printf("%d", *k);  /* Izpiše: 7.1  */
```

Naloga 6.12 Predelajte rešitev naloge 6.2 na strani 103, tako da bo vaša funkcija iz tabele odstranila vse podvojene vrednosti.

Primer klica funkcije:

```
#define N 12
//...
int t[N] = {5, 9, 5, 2, 6, 77, 15, 5, 5, 77, 6, 2};
int *k, *zadnji;
k = t;
zadnji = izlociDuplikate(t, N);
for (;k < zadnji; k++) {
  printf("%d ", *k);  /* Izpiše: 5 9 2 6 77 15  */
}
```

Naloga 6.13 Podana je naslednja funkcija, ki naj bi med seboj zamenjala vrednosti podanih parametrov:

```
void menjaj(int *a, int *b) {
   *b = *a + *b;
   *a = *b - *a;
   *b = *b - *a;
}
```

Z uporabo te funkcije lahko obrnemo vrstni red elementov v tabeli `t` takole:

```
#include <stdio.h>
#define N 8
//...
int main(void) {
  int t[N] = {1, 2, 3, 4, 5, 6, 7, 8};
```

```
  for (int i = 0; i <= (N - 1) / 2; i++) {
    menjaj(&t[i], &t[N - i - 1]);
  }
  for (int i = 0; i < N; i++) {
    printf("%d ", t[i]);  /* Izpiše: 8 7 6 5 4 3 2 1  */
  }
  return 0;
}
```

Vidimo, da program deluje tako, kot je treba. Če pa spremenimo vrednost makra `N` na 7, dobimo takšen izpis:

```
7 6 5 0 3 2 1
```

Izkaže se, da program v primeru lihega števila elementov srednjemu elementu tabele vedno priredi vrednost nič:

```
6 5 4 3 2 1
5 4 0 2 1
4 3 2 1
3 0 1
```

Pojasnite, zakaj se to zgodi.

Naloga 6.14 Kaj se bo izpisalo na izhod, ko se izvede naslednji del programa?

```
char str[] = "abcd";
printf("%d\n", str == "abcd");
printf("%s\n", &str[1]);
printf("%c\n", str[1]);
printf("%u\n", (unsigned int) str);
printf("%u\n", (unsigned int) &str[2]);
str[3] = 0;
printf("%s\n", &str[1]);
```

Naloga 6.15 Podan je naslednji program:

```
#include <stdio.h>
#include <string.h>
int main(void) {
  char s[] = "123456789";
  strcpy(s, "abc");
  printf (s);
  printf (&s[4]);
  return 0;
}
```

Kaj se bo izpisalo na izhod, ko program zaženemo? Utemeljite odgovor.

Naloga 6.16 Napišite funkcijo `poisciZnak`, ki kot parametra sprejme znakovni niz in znak. Funkcija naj vrne naslov prvega znaka v podanem nizu, ki je enak podanemu znaku. Če takšnega znaka ni, naj funkcija vrne vrednost NULL.

Primeri klicev funkcije:

```
char niz[] = "Kdor isce, ta najde.";
char *k = poisciZnak(niz, 'n');
if (k != NULL) {
  printf("%c %s", *k, k);  /* Izpiše: n najde.                    */
}
k = poisciZnak(niz, 0);
if (k != NULL) {
  printf("%d", k - niz);  /* Izpiše: 20 (število znakov v nizu)  */
}
k = poisciZnak(niz, 'z');
if (k != NULL) {
  printf("%c", *k);  /* Ta stavek se ne izvede.                  */
}
```

Opomba: V standardni knjižnici <string.h> obstaja funkcija `strchr`, ki deluje na enak način.

Naloga 6.17 Napišite funkcijo `prilepiNiz`, ki kot parametra sprejme dva znakovna niza. Funkcija naj drugi niz kopira na konec prvega (vključno z ničelnim znakom) in vrne naslov prvega niza. Primer klica funkcije:

```
char pregovor[100] = "Tudi slepa kura";
prilepiNiz(prilepiNiz(pregovor, " zrno"), " najde.");
printf(pregovor);  /* Izpiše: Tudi slepa kura zrno najde.  */
```

Opozorilo: Niz, ki mu dodajate drugi niz, mora imeti rezerviranega dovolj prostora, da lahko hrani vse znake združenega niza.

Opomba: V standardni knjižnici <string.h> najdemo funkcijo `strcat` (angl. string concatenate), ki deluje na enak način.

Naloga 6.18 Napišite funkcijo `razlomiNiz`, ki kot parametra sprejme znakovni niz in znak. Funkcija naj v podanem nizu poišče prvi znak, ki je enak podanemu znaku, in ga nadomesti z ničelnim znakom. Funkcija naj začne iskati od začetka podanega niza. Če funkciji namesto znakovnega niza podamo argument z vrednostjo NULL, potem naj začne iskati, od koder je končala v prejšnjem klicu (uporabite statično lokalno spremenljivko). Funkcija naj vsakokrat vrne naslov znaka, pri katerem je začela iskati. Če smo prišli do konca znakovnega niza (tj. znak, pri katerem naj funkcija začne iskati, je ničelni znak), naj funkcija vrne vrednost NULL.

Takšno funkcijo lahko na primer uporabimo, da iz seznama besed, ločenega z vejicami, izločimo posamezne besede:

```
#include <stdio.h>
#include <string.h>
int main(void) {
  char seznam[] = "ena,dve,tri,štiri,pet";
  char *k;
  k = razlomiNiz(seznam, ',');
  while (k) {
    printf("%s\n", k);  /* Vsakokrat izpiše eno
                           besedo iz seznama.    */
    k = razlomiNiz(NULL, ',');
  }
```

```
  return 0;
}
```

Opomba: V standardni knjižnici <string.h> najdemo funkcijo `strtok` (angl. string tokenize), ki deluje na podoben način.

PRAZNA STRAN

7. POGLAVJE

DVORAZSEŽNOSTNE TABELE IN DINAMIČNO DODELJEVANJE POMNILNIKA

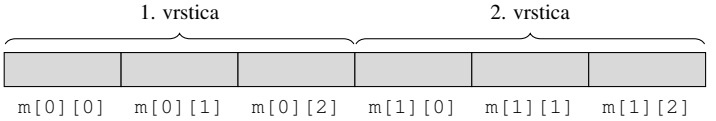
V tem poglavju bomo spoznali še dvorazsežnostne tabele. Poleg tega se bomo naučili, kako rezervirati pomnilnik v primeru, ko v času pisanja programa še ne vemo, koliko ga bomo potrebovali.

7.1 Dvorazsežnostne tabele

Tabele v jeziku C imajo lahko poljubno število razsežnosti, a mi se bomo ustavili pri dveh. Na primer, naslednja deklaracija ustvari tabelo celih števil z dvema vrsticama in s tremi stolpci:

```
int m[2][3];
```

Elementi dvorazsežnostne tabele so v pomnilniku shranjeni tako, da ena vrstica neposredno sledi drugi. V večini primerov ta podatek sicer ni pomemben, včasih pa le vpliva na delovanje kode. Gornja tabela je v pomnilniku shranjena takole:



Na gornji sliki je pod vsakim elementom zapisan izraz, ki ta element predstavlja. Na primer, do prvega elementa druge vrstice pridemo z izrazom `m[1][0]`. Tako kot pri enorazsežnostni tabeli začnemo tudi v tem primeru šteti indekse pri nič.

Inicializacija dvorazsežnostne tabele

Dvorazsežnostno tabelo inicializiramo s seznamom enorazsežnostnih seznamov začetnih vrednosti:

```
int m[3][3] = {{1, 4, 2},
               {7, 2, 5},
               {1, 1, 9}};
```

Kadar dvorazsežnostno tabelo ob deklaraciji tudi inicializiramo, lahko podatek o številu vrstic izpustimo. Število stolpcev moramo obvezno podati:

```
int m[][3] = {{9, 1, 0},
              {8, 2, 9},
              {5, 1, 1},
              {6, 8, 0}};  /* Tabela m ima 4 vrstice.  */
```

Če pri inicializaciji ne navedemo vseh vrstic – ali v kateri izmed vrstic ne navedemo vseh elementov –, se preostali elementi nastavijo na nič. Na primer, v naslednji inicializaciji dobijo tako element `m[0][2]` kot tudi vsi elementi zadnje vrstice (tj. elementi `m[2][0]`, `m[2][1]` in `m[2][2]`) vsi vrednost nič:

```
int m[3][3] = {{1, 4},
               {7, 2, 5}};
```

Kazalec na dvorazsežnostno tabelo

Tako kot ime enorazsežnostne tabele je tudi ime dvorazsežnostne tabele kazalec. Vendar to ni kazalec na element, temveč na ***vrstico*** tabele. Vzemimo za primer naslednjo tabelo:

```
#define VRSTIC 6
#define STOLPCEV 4
int m[VRSTIC][STOLPCEV];
```

Ime tabele `m` ni kazalec na `m[0][0]` (tj. na prvi element tabele), temveč kazalec na `m[0]` (tj. na prvo vrstico tabele), kar je zelo pomembno dejstvo. Če namreč povečamo vrednost takega kazalca za ena, ga pomaknemo po pomnilniku naprej za eno celo vrstico, kakor kaže naslednja koda:

```
printf("%u", (unsigned int) m);         /* Izpiše npr.: 4237824  */
printf("%u", (unsigned int) (m + 1));   /* Izpiše:      4237840  */
```

Vidimo, da je vrednost izraza `m + 1` v resnici za 16 večja od `m` (ob predpostavki, da zasede podatkovni tip `int` štiri bajte). To pomeni, da vrne izraz `m + 1` naslov, na katerem se začne druga vrstica tabele `m`. Oklepaji okrog izraza `m + 1` so potrebni, sicer bi operator pretvorbe tipa pretvoril tip `m`-ja v nepredznačeno celo število ***pred*** prištevanjem enke. Seveda bi v takem primeru prištevanje enke povečalo `m` samo za ena.

Dejstvo, da je `m` v gornjem programu kazalec na vrstico, ima za posledico, da dobimo z uporabo enega indeksnega operatorja ali operatorja posredovanja šele kazalec na prvi element določene vrstice. Na primer:

Izraz	Kazalec na prvi element ...
`m[0]`	... prve vrstice.
`*m`	... prve vrstice.
`m[2]`	... tretje vrstice.
`*(m + 2)`	... tretje vrstice.

Ko imamo enkrat kazalec na prvi element določene vrstice, izbiramo posamezne elemente v določeni vrstici na način, ki smo ga vajeni pri enorazsežnostnih tabelah. Na primer:

Izraz	Četrti element ...
`m[0][3]`	... prve vrstice.
`(*m)[3]`	... prve vrstice.
`*(m[2] + 3)`	... tretje vrstice.
`(*(m + 2))[3]`	... tretje vrstice.

Dodaten par okroglih oklepajev v drugi in četrti vrstici tabele je potreben zato, ker ima indeksni operator prednost pred operatorjem posredovanja. Izraz `*m[2]` bi nam zato vrnil prvi element tretje vrstice, medtem ko vrne izraz `(*m)[2]` tretji element prve vrstice.

Naloga 7.1 Za vajo razmislite, kaj vrne zadnji izraz v gornji tabeli, če iz njega odstranimo zunanji par okroglih oklepajev (tj. kaj vrne izraz `*(m + 2)[3]`).

Tako kot ime enorazsežnostne tabele je tudi ime dvorazsežnostne tabele konstanten kazalec – njegove vrednosti ni mogoče spreminjati Naslednji primer kaže, kako lahko deklariramo kazalčno spremenljivko, ki bo kazala na vrstico tabele. Vzemimo za primer, da želimo na nič postaviti vse elemente v `i`-tem stolpcu dvorazsežnostne tabele `m`. To lahko storimo takole:

```
int m[VRSTIC][STOLPCEV], i;
int (*k)[STOLPCEV];
//...
for (k = m; k < m + VRSTIC; k++) {
  (*k)[i] = 0;
}
```

Deklaracija `int (*k)[STOLPCEV]` pomeni, da je `k` kazalec na (enorazsežnostno) tabelo elementov tipa `int` dolžine `STOLPCEV` (tj. kazalec na vrstico tabele s STOLPCEV stolpci). Okrogla oklepaja sta potrebna, saj bi deklaracija brez njiju pomenila tabelo kazalcev tipa `int *`. Ker je `k` kazalec na vrstico tabele, pomeni izraz `*k` v telesu zanke kazalec na prvi element v tej vrstici, izraz `(*k)[i]` pa nam vrne `i`-ti element te vrstice.

Nekoliko očitnejša je koda, ki na nič postavi vse elemente v `i`-ti ***vrstici*** dvorazsežnostne tabele `m`:

```
int m[VRSTIC][STOLPCEV], i;
//...
for (int *k = m[i]; k < m[i] + STOLPCEV; k++) {
  *k = 0;
}
```

Vedeti moramo le, da predstavlja izraz `m[i]` kazalec na prvi element `i`-te vrstice tabele `m`. Natančneje, izraz:

```
m[i]
```

je krajša oblika izraza:

```
&m[i][0]
```

Seveda lahko isto dosežemo tudi brez uporabe kazalcev in enostavno uporabimo dva indeksa. Tako lahko postavimo na nič elemente `i`-tega stolpca gornje tabele `m` na naslednji način:

```
int m[VRSTIC][STOLPCEV], i;
//...
for (int k = 0; k < VRSTIC; k++) {
  m[k][i] = 0;
}
```

Elemente `i`-te vrstice pa postavimo na nič takole:

```
int m[VRSTIC][STOLPCEV], i;
//...
for (int k = 0; k < STOLPCEV; k++) {
  m[i][k] = 0;
}
```

Dvorazsežnostna tabela kot parameter funkcije

Dvorazsežnostno tabelo lahko funkciji vedno podamo v obliki kazalca na prvi element tabele. Ker so elementi dvorazsežnostne tabele v pomnilniku shranjeni drug za drugim, funkcija tako ne bo razlikovala eno- od dvorazsežnostne tabele. Na primer, takole lahko uporabimo funkcijo `sestej`, ki smo jo definirali na strani 102, da seštejemo elemente dvorazsežnostne tabele `m`:

```
int m[VRSTIC][STOLPCEV] = {{1, 4},
                           {7, 2, 5}};
printf("%d", sestej(m[0], VRSTIC * STOLPCEV));  /* Izpiše: 19  */
```

Z uporabo indeksnega operatorja smo dosegli, da je dobila funkcija `sestej` kot prvi argument kazalec na element tabele, in ne na vrstico tabele. Podobno lahko dosežemo z uporabo operatorja pretvorbe tipa:

```
sestej((int *) m, VRSTIC * STOLPCEV)
```

Če pa bi klicali funkcijo `sestej` takole:

```
sestej(m, VRSTIC * STOLPCEV)
```

bi se prevajalnik pritožil zaradi nezdružljivih tipov kazalcev. Funkcija `sestej` namreč pričakuje kazalec na podatek tipa `int`, `m` pa je kazalec na vrstico tabele.

Če želimo, da bo funkcija kot parameter sprejela dvorazsežnostno tabelo, potem moramo pri deklaraciji parametra uporabiti dva para oglatih oklepajev, število stolpcev pa moramo nujno podati. Ker bomo funkciji podali kazalec na vrstico, mora prevajalnik namreč poznati njeno dolžino. Naslednji primer kaže funkcijo, ki vrne indeks vrstice, v kateri se nahaja največji element podane dvorazsežnostne tabele:

```
int maksVrstica(int t[][STOLPCEV], int vrstic) {
  int imaks = 0, jmaks = 0;
  for (int i = 0; i < vrstic; i++) {
    for (int j = 0; j < STOLPCEV; j++) {
      if (t[i][j] > t[imaks][jmaks]) {
        imaks = i;
        jmaks = j;
      }
    }
  }
  return imaks;
}
```

Parameter `t` v gornji funkciji prevajalnik obravnava kot kazalec na vrstico tabele. Zato bi lahko definicijo funkcije `maksVrstica` začeli tudi takole:

```
int maksVrstica(int (*t)[STOLPCEV], int vrstic) {
//...
```

Funkcijo `maksVrstica` v obeh primerih kličemo takole:

```
int m[VRSTIC][STOLPCEV] = {{-3, 6, 12},
                           {1, 771},
                           {713, 32, -5, 9}};
printf("%d", maksVrstica(m, VRSTIC));  /* Izpiše: 1  */
```

Če želimo v funkciji uporabiti dva indeksna operatorja za izbiro vrstice in stolpca podane dvorazsežnostne tabele, smo – kot smo pravkar videli – omejeni na konstantno število stolpcev. Če kot parameter funkcije deklariramo kazalec na element tabele (namesto kazalca na vrstico tabele), potem te omejitve ni, imamo pa zato nekoliko več dela pri izbiri posameznih elementov.

Vzemimo za primer prototip funkcije, ki je namenjena delu z dvorazsežnostno tabelo, a ima za parameter enorazsežnostno tabelo:

```
void test(int t[], int vrst, int stolp);
```

Že na začetku tega razdelka smo videli, da lahko takšni funkciji podamo dvorazsežnostno tabelo `m` (ki ima na primer tri vrstice in štiri stolpce) z uporabo operatorja pretvorbe tipa:

```
test((int *) m, 3, 4);
```

V definiciji takšne funkcije pa žal ne moremo uporabljati indeksa vrstice in stolpca na običajen način. Na primer, element tabele v `i`-ti vrstici in `j`-tem stolpcu lahko izberemo takole (upoštevajoč dejstvo, da si elementi dvorazsežnostne tabele v pomnilniku sledijo po vrsti):

```
t[i * stolp + j]
```

Standard C99 je omenjeno omejitev na konstantno število stolpcev odpravil (glej primer tabele s spremenljivo dolžino na strani 96) in nam omogoča, da kot parameter uporabimo dvorazsežnostno tabelo s spremenljivim številom stolpcev, kot bomo spoznali v naslednji nalogi:

Naloga 7.2 Standard C99 podpira tabele spremenljivih dolžin (angl. variable sized arrays). Če vaš prevajalnik podpira ta standard, potem lahko to s pridom izkoristite za deklaracijo parametra, ki je lahko dvorazsežnostna tabela s spremenljivim številom stolpcev. Takole je videti deklaracija funkcije `izpisi`, ki kot parameter sprejme tabelo z `vrst` vrsticami in s `stolp` stolpci:

```
/* Parameter stolp mora biti deklariran pred tabelo tab,
   kateri s tem parametrom določimo število stolpcev:     */
void izpisi(int vrst, int stolp, int tab[][stolp]);
```

Napišite definicijo funkcije `izpisi`, ki bo na izhod izpisala vsebino podane tabele, ter dopolnite naslednji program, da bo deloval:

```
int vr, st;
//...
int t[vr][st];
//...
izpisi(vr, st, t);
```

Program naj izpiše tabelo tako, da bodo elementi v stolpcih drug pod drugim in poravnani desno. Na primer:

```
 -13    2    3
   4   35 -631
1239  -99    0
   4    7   42
```

Tabela znakovnih nizov

Pogosto naletimo na probleme, kjer potrebujemo tabelo znakovnih nizov. Ker je znakovni niz sam po sebi tabela, gre pri tem v resnici za dvorazsežnostno tabelo. Na primer, če potrebujemo seznam slovenskih imen dni v tednu, ustvarimo takšno tabelo:

```
char dnevi[][11] = {"ponedeljek", "torek", "sreda", "cetrtek",
                    "petek", "sobota", "nedelja"};
```

Kot že vemo, lahko v deklaraciji izpustimo število vrstic v tabeli, saj lahko to število prevajalnik ugotovi sam. Tudi število stolpcev bi bilo v gornjem primeru možno ugotoviti iz seznama začetnih vrednosti, vendar C zahteva, da število stolpcev zapišemo v vsakem primeru. Opazimo, da v gornji tabeli edino `ponedeljek` zasede polno vrstico, ki je dolga 11 znakov (deset vidnih znakov in dodaten zaključni ničelni znak). Zato je veliko prostora v tabeli neizkoriščenega in zapolnjenega z ničlami (kot že vemo, se neinicializirani elementi postavijo na nič). Takole je videti v pomnilniku tabela `dnevi`:

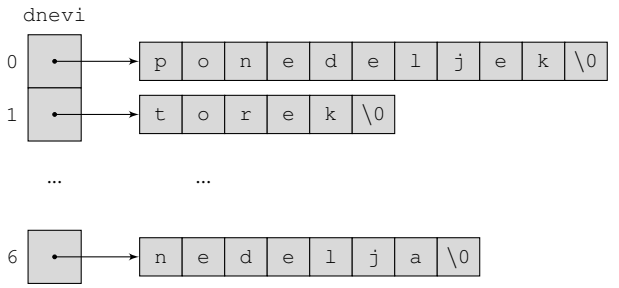
	0	1	2	3	4	5	6	7	8	9	10
0	p	o	n	e	d	e	l	j	e	k	\0
1	t	o	r	e	k	\0	\0	\0	\0	\0	\0
	...										
6	n	e	d	e	l	j	a	\0	\0	\0	\0

Indeksi na levi strani in nad tabelo predstavljajo vrstice in stolpce tabele `dnevi`. Tako na primer vrne izraz `dnevi[1][2]` črko r.

Neučinkovita izraba pomnilnika, kakor jo kaže gornji primer, je značilna za tabele znakovnih nizov. Redko se namreč zgodi, da so vsi znakovni nizi v tabeli enako dolgi. Težavo lahko rešimo tako, da namesto dvorazsežnostne tabele znakov deklariramo tabelo kazalcev na znakovne nize:

```
char *dnevi[] = {"ponedeljek", "torek", "sreda", "cetrtek",
                 "petek", "sobota", "nedelja"};
```

Ta na videz nepomembna sprememba v deklaraciji povzroči v pomnilniku povsem drugačno stanje. Zdaj predstavlja tabela `dnevi` enorazsežnostno tabelo kazalcev, od katerih vsak kaže na svoj znakovni niz v pomnilniku. Stanje je prikazano na naslednji sliki:



Iz slike vidimo, da imamo zdaj opravka s tabelo sedmih kazalčnih spremenljivk, od katerih vsaka kaže na svoj znakovni niz. Tako na primer spremenljivka `dnevi[1]` hrani naslov znakovnega niza `"torek"`.

Čeprav smo na ta način prihranili prostor za nepotrebno shranjevanje ničel na koncu krajših znakovnih nizov, pa smo zato porabili nekaj dodatnega pomnilnika za tabelo kazalcev, ki je v prejšnjem primeru nismo imeli. Vendar doseženi prihranek v mnogih primerih vseeno pretehta dodatno porabo pomnilnika za kazalčne spremenljivke.

Pri pisanju kode večinoma ni pomembno, ali smo tabelo deklarirali kot dvorazsežnostno tabelo ali kot tabelo kazalcev na enorazsežnostne tabele. Paziti moramo le, kadar želimo podatke obravnavati kot enorazsežnostno tabelo. Pri drugi različici (s tabelo kazalcev) namreč ni nobenega zagotovila, da si posamezni nizi v pomnilniku sledijo tesno drug za drugim, kakor nam to zagotavlja običajna deklaracija dvorazsežnostne tabele[1].

Obstajata pa še dve pomembni razliki: Prvič, v drugem od gornjih dveh primerov se nizi obnašajo kot ***konstantni znakovni nizi***. Poskus njihovega spreminjanja bo imel nedoločene posledice, kar smo spoznali že na strani 105:

```
dnevi[0][0] = 'P';  /* V drugem primeru je posledica
                       takšnega prirejanja nedoločena.  */
```

Drugič, v prvem primeru imamo opravka izključno s konstantnimi kazalci, v drugem primeru pa gre za tabelo kazalčnih spremenljivk. Izraz `dnevi[i]` predstavlja sicer v obeh primerih naslov `i`-tega znakovnega niza v tabeli:

```
printf(dnevi[2]);  /* V obeh primerih izpiše: sreda  */
```

[1]Pri tem seveda ne smemo pozabiti, da so vsi nizi v pomnilniku enako dolgi. Kakor smo videli, se nizom, ki so krajši, na koncu doda ustrezno število ničel.

vendar je izraz `dnevi[2]` v prvem primeru konstanten kazalec. V drugem primeru je to kazalčna spremenljivka, ki jo lahko kadarkoli usmerimo na kakšen drug znakovni niz. Na primer:

```
dnevi[2] = "Wednesday";  /* V prvem primeru to ne gre.  */
```

Naslednji primer z vhoda prebere zaporedno številko dneva v tednu (med nič in šest) in na izhod izpiše prve tri črke njegovega imena:

```
int dan;
printf("Vnesi dan v tednu (0 - 6): ");
scanf("%d", &dan);

for (int i = 0; i < 3; i++) {
  printf("%c", dnevi[dan][i]);
}
```

Koda deluje takole (ne glede na to, katero od obeh gornjih deklaracij tabele `dnevi` uporabimo):

```
Vnesi dan v tednu (0 - 6): 1
tor
```

7.2 Dinamično dodeljevanje pomnilnika

Za vse tabele, ki smo jih uporabljali doslej, smo morali že v času pisanja kode vedeti, kako velike morajo biti. Včasih pa naletimo na primere, ko tega ne moremo vedeti vnaprej. Koliko prostora naj torej rezerviramo, da ga bo dovolj? Težava je v tem, da – ne glede na to, koliko prostora rezerviramo – se lahko vedno izkaže, da ga potrebujemo še nekoliko več. Težavo rešimo z uporabo posebnih ***funkcij za dodeljevanje pomnilnika*** (angl. memory allocation functions).

Kazalec tipa `void *`

Splošne funkcije za dodeljevanje pomnilnika rezervirajo želeno količino pomnilnika in vrnejo kazalec na rezervirani blok. Ker pa ni mogoče vnaprej vedeti, kakšnega tipa bodo podatki, ki jih bomo hranili v rezerviranem delu pomnilnika, te funkcije vračajo kazalec tipa `void *`. To je splošen tip kazalca, ki lahko kaže na kakršenkoli podatek. Lahko ga uporabimo v priredilnih operatorjih z drugimi tipi kazalcev, ne da bi morali pri tem uporabiti operator pretvorbe tipa. Na primer:

```
void *vk;
int x = 42, *k;
vk = &x;
k = vk;
```

V gornji kodi smo vrednost kazalca tipa `int *` (tj. izraz `&x`) najprej priredili kazalčni spremenljivki `vk`, ki je tipa `void *`. V zadnjem stavku smo vrednost te spremenljivke priredili kazalčni spremenljivki `k`, ki je tipa `int *`. Vidimo, da za pretvorbo vrednosti iz kazalčnega tipa `void *` v `int *` in obratno ne potrebujemo operatorja pretvorbe tipa, kakor ga potrebujemo za pretvorbe med drugimi kazalčnimi tipi, ki niso tipa `void *`.

Ker kazalec tipa `void *` ne ve, na kakšen podatek kaže, nad njim ne moremo uporabiti operatorja posredovanja:

```
printf("%d", *vk);  /* Napaka: operator posredovanja
                       s kazalcem tipa void *         */
```

Operator posredovanja lahko uporabimo samo, če pred tem s pomočjo operatorja pretvorbe tipa pretvorimo tip kazalca v kazalec na konkreten podatkovni tip:

```
printf("%d", *(int *)vk);  /* Izpiše: 42  */
```

Bodite pozorni, da predstavlja zvezdica v oklepajih pred kazalcem `vk` del operatorja pretvorbe podatkovnega tipa. Šele ko kazalec `vk` postane kazalec tipa `int *`, lahko nad njim uporabimo operator posredovanja, ki ga predstavlja zvezdica pred oklepajem.

Funkciji `malloc` in `free`

Funkcija `malloc` (angl. memory allocation) rezervira želeno število bajtov pomnilnika in vrne naslov rezerviranega bloka. Če ni na voljo dovolj pomnilnika, funkcija vrne vrednost NULL. Ko pomnilnika ne potrebujemo več, ga sprostimo s funkcijo `free`, ki ji kot argument podamo naslov bloka pomnilnika, ki smo ga pred tem rezervirali s klicem funkcije `malloc`. Obnašanje funkcije `free` je nedoločeno, če se kazalec, ki ji ga podamo kot argument, ne ujema s kazalcem, ki ga je vrnil predhodni klic funkcije `malloc`. Funkcija `malloc` vrne, funkcija `free` pa sprejme kazalec tipa `void *`, zato ju lahko brez težav uporabimo s kazalci kakršnegakoli tipa.

Uporaba funkcije `malloc` za rezervacijo prostora za znakovni niz je enostavna. Ker posamezen znak vedno zasede en bajt pomnilnika, podamo kot argument funkcije zgolj število znakov z upoštevanjem zaključnega ničelnega znaka. Na primer, če želimo rezervirati prostor za `D` znakov, lahko to storimo takole:

```
p = malloc(D + 1);  /* D znakov plus zaključni ničelni znak.  */
```

pri čemer je `p` kazalec tipa `char *`. Ker vrne funkcija `malloc` kazalec tipa `void *`, nam pri tem ni treba izrecno pretvarjati tipa kazalca. Mnogi programerji kljub temu raje uporabijo operator pretvorbe tipa:

```
p = (char *) malloc(D + 1);
```

Naslednji program prikazuje praktičen primer uporabe funkcij `malloc` in `free`. Program uporabnika najprej vpraša, koliko celih števil bo vnesel. Potem vsa vnesena števila prebere in shrani ter jih v obrnjenem vrstnem redu izpiše na izhod. Program na koncu sprosti pomnilnik, ki ga je uporabljal za hranjenje prebranih števil:

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

int main(void) {
  int n;
  int *k;
  printf("Koliko števil želiš vnesti: ");
  scanf("%d", &n);
  k = malloc(n * sizeof(int));
  if (k == NULL) {
    printf("Napaka: ni dovolj pomnilnika.");
    return 0;
  }
  printf("Vnesi %d števil: ", n);
```

```
  for (int i = 0; i < n; i++) {
    scanf("%d", &k[i]);
  }
  printf("Vnesena števila v obrnjenem vrstnem redu:\n");
  for (int i = n - 1; i >= 0; i--) {
    printf("%d ", k[i]);
  }
  free(k);
  return 0;
}
```

Klic funkcije `malloc` nam dodeli pomnilnik za `n` elementov tabele tipa `int`. Funkcija za argument zahteva število potrebnih bajtov, zato moramo število elementov (`n`) pomnožiti z velikostjo podatka tipa `int` (tj. `sizeof(int)`). Če funkcija `malloc` želenega pomnilnika ne more dodeliti, vrne vrednost NULL, program pa izpiše sporočilo o napaki in se konča. V nasprotnem primeru se izvajanje programa nadaljuje. Vidimo, da kazalec `k` uporabljamo na popolnoma enak način, kot če bi ga usmerili na deklarirano tabelo tipa `int`. Tik pred izhodom iz programa s klicem funkcije `free` sprostimo dodeljeni pomnilnik. Če tega ne storimo, se bo pomnilnik po izhodu iz programa sprostil sam. Vseeno se je dobro navaditi, da vsak kos dinamično dodeljenega pomnilnika tudi sprostimo s klicem funkcije `free`. Če namreč program teče dolgo časa in pri tem neprestano rezervira bloke pomnilnika, ne da bi jih po uporabi tudi sprostil, lahko pomnilnika kaj hitro zmanjka.

Gornji program deluje takole:

```
Koliko števil želiš vnesti: 5
Vnesi 5 števil: 5 88 1 7 -2
Vnesena števila v obrnjenem vrstnem redu:
-2 7 1 88 5
```

Dinamično dodeljevanje pomnilnika za dvorazsežnostno tabelo

Z uporabo funkcije `malloc` lahko med izvajanjem programa rezerviramo prostor za dvorazsežnostno tabelo tako, da najprej rezerviramo prostor za tabelo kazalcev na posamezne vrstice, potem pa še prostor za vsako vrstico posebej. Kazalčna spremenljivka, ki bo kazala na tabelo kazalcev, mora biti kazalec na kazalec, zato jo deklariramo z dvema zvezdicama:

```
int **k;
```

Vzemimo za primer, da moramo rezervirati prostor za tabelo z `vr` vrsticami in s `st` stolpci. V prvem koraku potrebujemo prostor za `vr` kazalcev na posamezne vrstice:

```
k = malloc(vr * sizeof(int *));
```

Z gornjim klicem smo rezervirali prostor za `vr` kazalčnih spremenljivk tipa `int *` in nanj usmerili kazalec `k`. Mimogrede, izraz `sizeof(int *)` vrne število bajtov, ki jih potrebujemo za hranjenje enega kazalca tipa `int *`.

V drugem koraku moramo za vsako od `vr` vrstic rezervirati še prostor za `st` elementov tipa `int` in nanj usmeriti enega od kazalcev v zgoraj rezervirani tabeli kazalcev:

```
k[i] = malloc(st * sizeof(int));
```

Ko je prostor za tabelo enkrat rezerviran, lahko pridemo do elementa v `i`-ti vrstici in `j`-tem stolpcu na običajen način:

```
k[i][j]
```

Naj opozorimo, da se tako ustvarjena tabela razlikuje od tabele, ki jo že v osnovi deklariramo kot dvorazsežnostno tabelo (in ne kot kazalec na kazalec). Kot smo že omenili, si elementi deklarirane tabele v pomnilniku sledijo po vrsti, pri čemer prvi element vsake naslednje vrstice sledi zadnjemu elementu prejšnje vrstice. V primeru dinamično ustvarjene dvorazsežnostne tabele imamo namesto tega opravka s tabelo kazalcev, od katerih vsak kaže na svojo vrstico, ki se lahko nahaja kjerkoli v pomnilniku. Prav tako ni treba, da so pri tem vse vrstice tabele enako dolge. Razmere so prav takšne, kot smo jih videli v primeru tabele kazalcev na znakovne nize na strani 125.

Naloga 7.3 Za vajo napišite program, ki bo uporabnika vprašal za velikost dvorazsežnostne tabele, rezerviral ustrezen prostor zanjo ter na koncu vanjo vpisal elemente z naključnimi vrednostmi med nič in deset. Program naj potem na izhod izpiše vrednosti vseh elementov, ki so večji od srednje vrednosti vseh elementov v svoji vrstici tabele.

Na koncu ne pozabite sprostiti vsega rezerviranega pomnilnika s funkcijo `free`. Najprej morate sprostiti prostor za vsako vrstico tabele posebej (preko posameznih kazalcev `k[i]`), potem pa še prostor za tabelo kazalcev (preko kazalca `k`).

Naloga 7.4 Videli smo, da nam obe naslednji deklaraciji kazalca `k` omogočata, da na njem uporabimo dva indeksna operatorja za dostop do elementov dvorazsežnostne tabele:

```
int **k;
int (*k)[ST];
```

Poskusite pojasniti, kakšne so razlike med obema deklaracijama.

7.3 Naloge

Naloga 7.5 Podane so naslednje deklaracije:

```
#define M 2
#define N 3

double (*k)[M];
double *q[M];
double m[N][M];
```

Koliko pomnilnika zasede vsaka od deklaracij ob predpostavki, da kazalčna spremenljivka zasede štiri, spremenljivka tipa `double` pa osem bajtov?

Naloga 7.6 Podana je koda:

```
int t[5][10];
int (*k)[10];
k = t + 1;
```

Kakšna je vrednost izrazov `t[4] - t[1]` ob predpostavki, da podatek tipa `int` v pomnilniku zasede štiri bajte? In kakšna je vrednost izraza `k - t`? Utemeljite odgovor.

Naloga 7.7 Napišite funkcijo, ki množi podano matriko reda $M \times N$ s skalarjem in zapiše rezultat množenja v originalno matriko.

Primer klica funkcije:

```
#define M 2
#define N 3
//...
double m[M][N] = {{ 2.3,  1.9,  2.0},
                  { 3.1,  0.4, 11.7}};
skalarnoMnozenje(m, 2);  /* m je zdaj enak {{ 4.6,  3.8,  4.0},
                                              { 6.2,  0.8, 23.4}}  */
```

Naloga 7.8 Podana je naslednja nepopolna koda:

```
#include <stdio.h>
#define M 3
#define N 4
int maks(int *t, int dolzina) {
//...
}

int main(void) {
  int a[M][N] = {{ 1,   8, 11},
                 { 9, -34},
                 {55,  92,  1, 13}};
  printf("%d", maks(/* ... */));  /* Izpiše: 92  */
  return 0;
}
```

Funkcija `maks` naj bi vrnila vrednost največjega elementa v enorazsežnostni tabeli `t`, ki vsebuje `dolzina` elementov. Napišite definicijo funkcije `maks` ter vpišite manjkajoče argumente pri njenem klicu, tako da bo program na izhod izpisal vrednost največjega elementa v tabeli `a`.

Naloga 7.9 Napišite funkcijo, ki kot parametra sprejme dve kvadratni matriki ter izračuna njun matrični produkt. Produkt naj zapiše v prvo od podanih matrik.

Primer klica funkcije:

```
#define N 2
//...
int m1[N][N] = {{ 1,  2},
                { 5,  4}};
int m2[N][N] = {{-3,  5},
                { 1, -2}};
matricniProdukt(m1, m2);  /* m1 je zdaj {{-1, 1}, {-11, 17}}  */
```

Pomoč: Element i-te vrstice in j-tega stolpca produkta matrik $\mathbf{A}$ in $\mathbf{B}$ izračunamo po naslednji enačbi:

$$(\mathbf{AB})_{i,j} = \sum_{k=1}^{N} A_{i,k} B_{k,j},$$

kjer je N število stolpcev matrike $\mathbf{A}$ in število vrstic matrike $\mathbf{B}$. Produkt matrik $\mathbf{A}$ in $\mathbf{B}$ ima toliko vrstic kot matrika $\mathbf{A}$ in toliko stolpcev kot matrika $\mathbf{B}$.

Naloga 7.10 Napišite program, ki prešteje število znakov (brez zaključnih ničelnih znakov), ki jih vsebuje naslednja tabela znakovnih nizov:

```
char besede[][13] = {"ena", "enajst", "enaindvajset", "konec"};
```

Pomoč: Za štetje znakov v posameznem znakovnem nizu uporabite standardno funkcijo `strlen` (angl. string length) iz knjižnice <string.h>, ki vrne število znakov (brez ničelnega znaka) podanega niza.

Naloga 7.11 Napišite funkcijo, ki kot parameter sprejme tabelo kazalcev na znakovne nize, od katerih vsak predstavlja eno samo besedo iz malih črk angleške abecede. Zadnji od kazalcev v tabeli naj ima vrednost NULL. Funkcija naj vrne naslov niza, ki je v tabeli prvi po abecedi.

Primer klica funkcije:

```
char *besede[] = {"pig", "cow", "dragon", NULL};
printf(prviPoAbecedi(besede));  /* Izpiše: cow  */
```

Naloga 7.12 Podani sta naslednji deklaraciji:

```
char b1[][10] = {"jabolko", "banana", "borovnica"};
char *b2[] = {"jabolko", "banana", "borovnica"};
```

Iz deklaracij je razvidno, da je `b1` kazalec na prvo vrstico tabele znakov s tremi vrsticami in z desetimi stolpci, `b2` pa je kazalec na tabelo treh kazalcev na znakovne nize. Vsak od kazalcev `b1[0]` in `b2[0]` kaže na prvega od znakovnih nizov v svoji tabeli.

Odgovorite na naslednja vprašanja:

- Ali ima kazalec `b1` enako vrednost kot kazalec `b1[0]`?
- Ali ima kazalec `b2` enako vrednost kot kazalec `b2[0]`?
- Kakšna je vrednost izraza `(unsigned int) (b1 + 1) - (unsigned int) b1`?
- Kakšna je vrednost izraza `(unsigned int) (b2 + 1) - (unsigned int) b2`?

Utemeljite svoje odgovore.

Naloga 7.13 Predelajte funkcijo `beriVrstico` na strani 108, tako da bo funkcija sama rezervirala dovolj prostora za prebrano besedilo. Funkcija naj na začetku s pomočjo funkcije `malloc` rezervira prostor za (na primer) deset znakov. Takoj ko

vtipkamo enajsti znak, naj funkcija poveča rezervirani prostor za deset znakov. Isto naj naredi, ko vtipkamo enaindvajseti znak, in tako dalje.

Poleg tega napišite še funkcijo `sprostiVrstico`, ki s klicem funkcije `free` sprosti ves rezervirani pomnilnik.

Primer klica obeh funkcij:

```
char *buff;
buff = beriVrstico();
if (buff != NULL) {
  printf(buff);
  sprostiPomnilnik(buff);
}
```

Opomba: Velikost dodeljenega pomnilnika lahko povečate s funkcijo `realloc`. Ta funkcija sprejme dva parametra, od katerih je prvi kazalec na kos pomnilnika, ki smo ga predhodno rezervirali bodisi s funkcijo `malloc` bodisi s funkcijo `realloc`. Drugi parameter funkcije `realloc` je nova želena velikost rezerviranega pomnilnika v bajtih. Ta je lahko bodisi večja bodisi manjša od originalne velikosti. Funkcija `realloc` poskuša spremeniti velikost originalno dodeljenega pomnilnika tako, da ohrani njegov začetni naslov. Če to ni mogoče, rezervira prostor drugje, vanj prekopira obstoječe podatke in sprosti prvotno dodeljeni pomnilnik. Na koncu funkcija vrne naslov, na katerem se nahaja na novo dodeljeni kos pomnilnika. Če operacija ne uspe, potem vrne vrednost NULL.

8. POGLAVJE

STRUKTURE

Podobno kot tabela je tudi ***struktura*** (angl. structure) zbirka več vrednosti, ki pa lahko – za razliko od tabel – pripadajo različnim tipom. Druga razlika med tabelo in strukturo je ta, da do elementov tabele pridemo preko indeksnega operatorja, elemente strukture pa poimenujemo z imeni. V jeziku C elementom strukture pravimo tudi ***člani*** (angl. member).

8.1 Strukturne spremenljivke

Kadar želimo skupaj shraniti več različnih vrst podatkov[1], ki so med seboj pomensko povezani, je struktura primerna izbira. Vzemimo za primer podatke o hotelu iz turističnega kataloga:

```
#define DOLZINA_IMENA 20
//...
struct {
  char ime[DOLZINA_IMENA + 1];
  int stZvezdic;
  float ocenaGostov;
} hotel1, hotel2;
```

[1]Tudi če so podatki istega tipa, se lahko pomensko razlikujejo. Na primer, tlak in temperatura sta lahko oba tipa `float`, in vendar ju je bolj smiselno združiti v strukturni spremenljivki kot v tabeli. Z uporabo strukture ju lahko namreč poimenujemo z imenoma *tlak* in *temperatura*, v tabeli pa ju lahko med seboj ločimo zgolj z indeksoma.

S tem smo deklarirali dve strukturni spremenljivki `hotel1` in `hotel2`, ki imata vsaka po tri člane: `ime`, `stZvezdic` in `ocenaGostov`. Člani strukture so v pomnilniku vedno shranjeni drug za drugim, v istem vrstnem redu, kakor se pojavijo v deklaraciji. Naj omenimo, da ima deklaracija strukturne spremenljivke enako obliko kot vsaka druga deklaracija v jeziku C: zapis `struct {...}` predstavlja podatkovni tip, medtem ko sta `hotel1` in `hotel2` spremenljivki tega tipa.

Inicializacija strukturnih spremenljivk

Podobno kot tabelo lahko ob deklaraciji inicializiramo tudi strukturne spremenljivke. Seznam začetnih vrednosti naštejemo v paru zavitih oklepajev:

```
struct {
  char ime[DOLZINA_IMENA + 1];
  int stZvezdic;
  float ocenaGostov;
} hotel1 = {"Villa del Sol", 5, 9.2},
  hotel2 = {"Hotel Playa Azul", 4, 9.5},
  mojHotel;  /* Spremenljivka mojHotel ostane neinicializirana.  */
```

Vrednosti v seznamu so lahko edino konstante, naštete pa morajo biti v istem vrstnem redu, kot so v deklaraciji strukture navedeni njeni člani.

Operacije nad strukturnimi spremenljivkami

Posameznega člana strukturne spremenljivke izberemo tako, da najprej navedemo ime strukturne spremenljivke, potem piko in na koncu ime člana, ki nas zanima. Tako zapisan izraz uporabljamo popolnoma enako kot katerokoli spremenljivko, ki je istega tipa kot izbrani član. Tu je nekaj primerov:

```
scanf("%d", &hotel1.stZvezdic);     /* hotel1.stZvezdic je tipa int.  */
printf("Ime hotela: %s", hotel1.ime);  /* hotel1.ime je tipa char *.  */
mojHotel.ocenaGostov = 6.5;   /* mojHotel.ocenaGostov je tipa float.  */
hotel1.stZvezdic++;                 /* hotel1.stZvezdic je tipa int.  */
strcpy(mojHotel.ime, "Hotel Slon");  /* mojHotel.ime je tipa char *.  */
```

Za razliko od tabele ime strukturne spremenljivke ni kazalec, temveč običajna spremenljivka. Zato lahko celotno vsebino strukturne spremenljivke (tj. vrednosti vseh njenih članov) kopiramo v enem samem priredilnem stavku. Na primer, vse podatke, shranjene v spremenljivki `hotel2`, lahko kopiramo v spremenljivko `mojHotel` takole:

```
mojHotel = hotel2;  /* Kopira vse člane strukturne spremenljivke.  */
```

Za mnoge programerje je nekoliko presenetljivo, da lahko s priredilnim operatorjem kopiramo znakovni niz ali kakršnokoli drugo tabelo. Vendar če je tabela član strukturne spremenljivke, se bodo v takšnem primeru kopirali tudi vsi elementi tabele. Včasih lahko to dejstvo izkoristimo kot »trik« za kopiranje tabel s priredilnim operatorjem:

```
#define D 10
struct { int t[D]; } s1, s2;
//...
s1 = s2;  /* Kopira vse elemente tabele s2.t v tabelo s1.t.  */
```

Seveda pa same tabele še vedno ne moremo kopirati s priredilnim operatorjem:

```
s1.t = s2.t;  /* To ne gre: s1.t je konstanten kazalec.  */
```

Pomembno je tudi dejstvo, da se na tak način kopirajo zgolj deklarirane tabele, ne pa tudi podatki, na katere kažejo člani, ki so kazalčne spremenljivke:

```
struct { int *t; } s1, s2;
//...
s1 = s2;  /* Kopira zgolj naslov, ki je shranjen v s2.t.  */
```

8.2 Strukturni tip

Deklaracija strukturne spremenljivke, kakršno smo pravkar srečali, je nekoliko nepraktična. Zamislimo si, da želimo deklarirati parameter funkcije, ki je strukturna spremenljivka. V takih in podobnih primerih potrebujemo strukturni tip, ki ga lahko določimo z uporabo določila `typedef` (angl. type definition).

Določilo `typedef`

V cejevskih standardnih knjižnicah pogosto naletimo na definicije podatkovnih tipov, ki jih standard ne določa popolnoma natančno, ali pa jih želimo zaradi preglednosti in prenosljivosti določiti sami. Ko takšen tip uporabimo v svojem programu, ga lahko po potrebi enostavno zamenjamo z drugačnim, saj se njegova definicija pojavi le na enem mestu v programu.

Nov tip lahko določimo z uporabo določila `typedef`:

```
typedef obstoječ-tip nov-tip ;
```

Tako je na primer lahko videti definicija tipa `time_t`, ki smo ga srečali v povezavi s standardno funkcijo `time`:

```
typedef long time_t;
```

Podobno lahko definiramo tip `bool`:

```
typedef int bool;
```

Strukturni podatkovni tip določimo na enak način:

```
typedef struct {
  char ime[DOLZINA_IMENA + 1];
  int stZvezdic;
  float ocenaGostov;
} hotel_t;
```

Zdaj `hotel_t` ni spremenljivka, temveč podatkovni tip, ki ga lahko uporabimo za deklaracijo spremenljivk:

```
hotel_t h1, h2;      /* Dve spremenljivki tipa hotel_t             */
hotel_t hoteli[10];  /* Tabela desetih elementov tipa hotel_t      */
hotel_t *izberiNajboljsiHotel(hotel_t h[], int n); /* Deklaracija
            funkcije, ki sprejme tabelo z n elementi tipa hotel_t
            in vrne naslov elementa, ki vsebuje informacijo
            o najboljšem hotelu.                                   */
```

Imenu tipa smo dodali podčrtaj in črko t, da je že iz imena razvidno, da gre za podatkovni tip.

Primer: množenje kompleksnih števil

Za ilustracijo sledi preprost primer programa, ki množi dve kompleksni števili:

```
#include <stdio.h>

typedef struct {
  double re, im;
} kompl_t;

kompl_t mnozi(kompl_t x, kompl_t y) {
  kompl_t pom;
  pom.re = x.re * y.re - x.im * y.im;
  pom.im = x.re * y.im + x.im * y.re;
  return pom;
}

int main(void) {
  kompl_t a = {3, 2};
  kompl_t b = {1, -7};
  a = mnozi(a, b);
  if (a.im < 0) {
    printf("%.1f%.1fi", a.re, a.im);
  }
  else {
    printf("%.1f+%.1fi", a.re, a.im);
  }
  return 0;
}
```

Na začetku programa smo določili strukturni tip `kompl_t`, ki ga uporabimo pri definiciji funkcije `mnozi` ter spremenljivk `a` in `b`. Bodite pozorni, da je tako tip funkcije `mnozi` kot tudi tip obeh njenih parametrov `kompl_t`. Parametra `x` in `y` se zato v telesu funkcije uporabljata kot običajni strukturni spremenljivki, funkcija pa vrne vrednost spremenljivke `pom`, ki mora biti seveda prav tako tipa `kompl_t`. Ne smemo pozabiti, da se strukturne spremenljivke kopirajo po vrednosti, zato sta parametra `x` in `y` v resnici ***kopiji*** argumentov `a` in `b`.

Zadnji stavek `if...else` v gornjem programu poskrbi za pravilen izpis predznaka imaginarnega dela rezultata: če je ta del negativen, se bo minus izpisal avtomatsko, če pa je pozitiven, moramo plus dodati, sicer se ne bo izpisal. Ko program zaženemo, na izhod izpiše `17.0-19.0i`.

Strukturna značka

Namesto tipa, ki ga določimo z določilom `typedef`, lahko uporabimo tudi ***strukturno značko*** (angl. structure tag). Strukturna značka je ime, ki ga zapišemo takoj za besedo `struct`. Prejšnji strukturni tip, ki opisuje kompleksno število, lahko zapišemo tudi takole:

```
struct kompl {
  double re, im;
};
```

Beseda `kompl` je v tem primeru strukturna značka. Če hočemo deklarirati spremenljivko z uporabo strukturne značke, moramo pred ime značke zapisati tudi besedo `struct`. Strukturno značko za deklaracijo spremenljivke uporabimo takole:

```
struct kompl z;  /* Spremenljivka z je tipa struct kompl.  */
```

8.3 Kazalec na strukturno spremenljivko

Kot na vsak objekt lahko tudi na strukturno spremenljivko usmerimo kazalec. Na primer:

```
struct kompl *k, x;
k = &x;
```

Če želimo zdaj priti do katerega od članov strukturne spremenljivke `x` preko kazalca `k`, moramo najprej uporabiti operator posredovanja, da pridemo do strukturne spremenljivke. Šele potem lahko s selektorjem izberemo ustreznega člana. Ker ima selektor (pika) prednost pred operatorjem posredovanja, moramo uporabiti tudi oklepaje:

```
(*k).re = 12.9;  /* Realni del spremenljivke x postane enak 12.9.  */
```

Ker kazalce na strukturne spremenljivke pogosto uporabljamo, je tak zapis nekoliko neprikladen. Operator posredovanja in piko lahko nadomestimo z izbirnim operatorjem v obliki puščice (->):

```
k->re = 0.2;  /* Realni del spremenljivke x postane enak 0.2.  */
```

Primer: urejanje tabele študentov

Osvetlimo vse skupaj z nekoliko kompleksnejšim primerom, ki uredi tabelo študentov po abecedi in vpisni številki. Izhajali bomo iz naslednjega strukturnega tipa z imenom `student_t`:

```
typedef struct {
  char ime[DOLZINA_IMENA + 1];
  int vpisnaStev;
} student_t;
```

S pomočjo tega tipa lahko ustvarimo tabelo štirih študentov:

```
student_t letnik[ST_VPISANIH] = {{"Lea",  6402102}, {"Ula", 6402001},
                                 {"Miha", 6402813}, {"Gal", 6401627}};
```

Iz primera vidimo, da tabelo strukturnih spremenljivk inicializiramo tako kot vsako drugo tabelo: v par zavitih oklepajev zapišemo z vejicami ločen seznam začetnih vrednosti posameznih elementov. Ker so elementi tabele `letnik` strukturne spremenljivke, za vsakega od njih uporabimo nov par zavitih oklepajev z ustreznima začetnima vrednostma.

Tabelo `letnik` bomo uredili po abecedi in po vpisnih številkah, pri čemer bomo za urejanje uporabili funkcijo `qsort`, ki jo najdemo v standardni knjižnici <stdlib.h>. Ta funkcija uredi podano tabelo po postopku ***hitrega urejanja*** (angl. quicksort). Poleg tabele, ki jo želimo urediti, moramo funkciji podati tudi kriterij, po katerem naj primerja posamezne elemente. Slednjega podamo v obliki kazalca na funkcijo z naslednjim prototipom:

```
int primerjaj(const void *a, const void *b);
```

Ta funkcija sprejme dva kazalca tipa `void *`, ki predstavljata naslova dveh elementov, ki ju je treba med seboj primerjati[2]. Funkcija `primerjaj` se bo v postopku urejanja

[2]Če bo treba elementa med seboj zamenjati, bo to storila funkcija `qsort`. Funkcija `primerjaj` elementov tabele ne sme spreminjati, ker bi to vodilo k nedoločenemu obnašanju kode. Da do takšnega spreminjanja elementov ne pride, sta parametra deklarirana z besedo `const`.

klicala iz funkcije `qsort`. Slednja pričakuje, da funkcija `primerjaj` vrne vrednost, ki je manjša, večja ali enaka nič, odvisno od tega, ali je prvi element manjši, večji ali enak[3] drugemu od dveh elementov, ki se primerjata med seboj. Funkcija `qsort` v skladu z algoritmom hitrega urejanja s pomočjo funkcije `primerjaj` primerja po dva in dva elementa iz podane tabele in ju po potrebi med seboj zamenja.

Preden nadaljujemo z našo tabelo študentov, si oglejmo enostavnejši primer urejanja tabele celih števil. Vzemimo, da želimo s funkcijo `qsort` urediti naslednjo tabelo:

```
#define D 10
//...
int tab[D] = {12, 77, -3, 913, 13, -91, 102, 38, 1476, -100};
```

Funkcija `primerjaj` je v tem primeru zelo preprosta. Ker urejamo številske vrednosti, je dovolj, da funkcija vrne razliko med vrednostma, na kateri kažeta oba njena parametra (tj. `a` in `b`). Če želimo, da bo tabela urejena po naraščajočih vrednostih, potem moramo drugo vrednost odšteti od prve. Ker sta parametra funkcije kazalca tipa `void *`, moramo paziti, da ju pred uporabo operatorja posredovanja pretvorimo v ustrezen tip (v našem primeru `int *`). Takole je videti definicija funkcije:

```
int primerjaj(const void *a, const void *b) {
  return *(int *)a - *(int *)b;
}
```

Zdaj imamo vse potrebno, da lahko uporabimo funkcijo `qsort` za urejanje tabele `tab`. Funkciji moramo podati po vrsti naslov[4] in število elementov tabele, ki jo želimo urediti, velikost posameznega elementa te tabele ter naslov funkcije, ki vsebuje kriterij za primerjavo dveh elementov:

```
qsort(tab, D, sizeof(int), primerjaj);
```

Kot zadnji argument funkcije `qsort` smo podali naslov funkcije `primerjaj`. S pomočjo tega naslova lahko funkcija `qsort` za primerjavo elementov tabele `tab` kliče funkcijo `primerjaj` na enak način, kot smo to videli na primeru funkcije `integral` na strani 88.

Naloga 8.1 Funkcija `primerjaj`, ki smo jo pravkar napisali, vsebuje zelo zoprno napako, ki jo s površnim testiranjem zelo težko odkrijemo. Na primer, naslednji program bo najverjetneje uredil elemente tabele `tab` po naraščajočih vrednostih:

```
#include <stdio.h>
#include <stdlib.h>
#define D 10
int tab[D] = {3, -3700, -3650, 13, 9, 2147480000, 102, 38, 1476, -100};

int primerjaj(const void *a, const void *b) {
     return *(int *)a - *(int *)b;
}
```

[3] Oziroma večji, manjši ali enak, odvisno od tega, v katero smer želimo tabelo urediti (naraščajoče ali padajoče). Povedano drugače: Če funkcija `primerjaj`, ki primerja vrednosti dveh elementov tabele, vrne vrednost, ki je večja od nič, bo funkcija `qsort` vrednosti teh dveh elementov med seboj zamenjala. Če pa funkcija `primerjaj` vrne vrednost, ki je manjša od nič, funkcija `qsort` vrednosti teh dveh elementov ne bo zamenjala. Če funkcija `primerjaj` vrne vrednost nič, potem sta vrednosti elementov enaki in ju tudi ni treba zamenjati, čeprav standard vrstnega reda elementov, ki so med seboj enaki, ne predpisuje.

[4] Glede na to, da je funkcija `qsort` namenjena splošni uporabi za urejanje kakršnekoli tabele, nas ne bo presenetilo, če bomo v njeni deklaraciji opazili, da je njen prvi parameter deklariran kot kazalec tipa `void *`.

```
int main(void) {
  qsort(tab, D, sizeof(int), primerjaj);
  for (int i = 0; i < D; i++) {
    printf("%d\n", tab[i]);
  }
  return 0;
}
```

Zamenjajmo zdaj med seboj prva dva elementa tabele `tab`:

```
int tab[D] = {-3700, 3, -3650, 13, 9, 2147480000, 102, 38, 1476, -100};
```

Ko smo program zagnali s takšno tabelo, smo dobili naslednji izpis:

```
-100
3
9
13
38
102
1476
2147480000
-3700
-3650
```

Poiščite in komentirajte napako v funkciji `primerjaj`. Popravite definicijo funkcije, tako da bo delovala pravilno.

Namig: Če zamenjamo vrednost 2 147 480 000 z nekoliko manjšo vrednostjo (npr. 2 147 479 947), bo program deloval pravilno ne glede na začetni vrstni red elementov v tabeli `tab`.

Opomba: Napaka se pokaže na način, ki je opisan v nalogi, ob predpostavki, da imajo podatki tipa `int` bitno dolžino 32 in da so negativne vrednosti zapisane v dvojiškem komplementu.

Vrnimo se zdaj k naši tabeli študentov. V tem primeru bomo urejali tabelo strukturnih spremenljivk tipa `student_t`, zato bosta parametra `a` in `b` funkcije za primerjavo kazalca na strukturni spremenljivki tipa `student_t`, se pravi kazalca tipa `student_t *`. Funkcija, ki primerja dva študenta po vpisnih številkah, je zato videti takole:

```
int primerjajVpisStev (const void *a, const void *b) {
  return ((student_t *) a)->vpisnaStev - ((student_t *) b)->vpisnaStev;
}
```

V funkciji moramo najprej pretvoriti tipa obeh kazalcev iz `void *` v `student_t *`, sicer prek njiju ne bo mogoče priti do članov obeh strukturnih spremenljivk, na kateri ta dva kazalca kažeta:

```
(student_t *) a
(student_t *) b
```

Ko smo enkrat pretvorili kazalca v ustrezen tip, lahko pridemo prek njiju do članov `vpisnaStev` obeh strukturnih spremenljivk z operatorjem izbire člana (`->`). Ker ima ta operator višjo prednost kot operator pretvorbe tipa, moramo uporabiti še en par oklepajev:

```
((student_t *) a)->vpisnaStev
((student_t *) b)->vpisnaStev
```

Razliko obeh vpisnih številk, ki jo vrne funkcija `primerjajVpisStev`[5], bo uporabila funkcija `qsort` kot kriterij, ali mora elementa med seboj zamenjati ali ne.

Tabelo `letnik` zdaj uredimo po vpisnih številkah takole:

```
qsort(letnik, ST_VPISANIH, sizeof(student_t), primerjajVpisStev);
for (int i = 0; i < ST_VPISANIH; i++) {
  printf("%d %s\n", letnik[i].vpisnaStev, letnik[i].ime);
}
```

Gornja koda izpiše takšen seznam:

```
6401627 Gal
6402001 Ula
6402102 Lea
6402813 Miha
```

Če želimo urediti študente po abecedi, potrebujemo samo drugačno funkcijo za primerjavo elementov:

```
int primerjajAbeceda (const void *a, const void *b) {
  return strcmp(((student_t *) a)->ime, ((student_t *) b)->ime);
}
```

Pri tem smo uporabili funkcijo `strcmp` iz standardne knjižnice <string.h>, ki med seboj primerja dva znakovna niza. Ta funkcija vrne vrednost, ki je manjša, večja ali enaka nič, odvisno od tega, ali je prvi niz manjši, večji ali enak prvemu.

Z uporabo funkcije `primerjajAbeceda` uredimo tabelo `letnik` po abecedi takole:

```
qsort(letnik, ST_VPISANIH, sizeof(student_t), primerjajAbeceda);
for (int i = 0; i < ST_VPISANIH; i++) {
  printf("%d %s\n", letnik[i].vpisnaStev, letnik[i].ime);
}
```

Zdaj se izpiše takšen seznam:

```
6401627 Gal
6402102 Lea
6402813 Miha
6402001 Ula
```

Primer: povezan seznam

Član strukturne spremenljivke je lahko seveda tudi kazalec. Če uporabimo kazalec na strukturno spremenljivko istega tipa, potem lahko ustvarimo povezan seznam[6]. Vzemimo, da želimo v povezanem seznamu shranjevati zgolj celoštevilske vrednosti. Naslednji strukturni tip opisuje vozlišče takšnega povezanega seznama:

```
struct vozlisce {
  int podatek;
  struct vozlisce *naslednji;
};
```

[5]Ob predpostavki, da so vse vpisne številke pozitivne in v območju vrednosti podatkovnega tipa `int`, bo funkcija `primerjajVpisStev` delovala pravilno (glej nalogo 8.1 na strani 138 ter razdelek *Prekoračitev* na strani 20).

[6]Spomnimo se, da je ***povezan seznam*** (angl. linked list) seznam ***vozlišč*** (angl. node). Vsako od vozlišč poleg podatkov vsebuje tudi kazalec na naslednje vozlišče.

Vozlišče poleg podatka vsebuje še kazalec na naslednje vozlišče. Naj omenimo, da v takem primeru ne moremo določiti strukturnega tipa z uporabo določila `typedef`. Takrat ko namreč potrebujemo ta tip, da bi deklarirali kazalec `naslednji`, tip še ni do konca določen. V tem primeru je edina možnost, da uporabimo strukturno značko.

Napišimo funkcijo, ki doda novo vozlišče na začetek povezanega seznama in vrne kazalec na to novo vozlišče:

```
struct vozlisce *dodajVozlisce(struct vozlisce *seznam, int n) {
  struct vozlisce *novoVozlisce = malloc(sizeof(struct vozlisce));
  if (novoVozlisce == NULL) {
    printf("Napaka: premalo prostora v pomnilniku.");
    exit(EXIT_FAILURE);
  }
  novoVozlisce->podatek = n;
  novoVozlisce->naslednji = seznam;
  return novoVozlisce;
}
```

Funkcija `dodajVozlisce` poskuša najprej s klicem funkcije `malloc` rezervirati pomnilnik za novo vozlišče. Če ji to ne uspe, se program konča, pri čemer vrne kodo napake `EXIT_FAILURE`, kar je makro, določen v knjižnici <stdlib.h>. Če funkciji uspe rezervirati pomnilnik, potem samo še nastavi vrednosti obeh članov na novo ustvarjenega vozlišča in vrne kazalec na to novo vozlišče. Na novo ustvarjeno vozlišče je namreč zdaj prvo vozlišče v povezanem seznamu.

S to funkcijo že lahko napolnimo seznam s tremi številkami, kakor kaže naslednji program:

```
#include <stdlib.h>
//...
int main(void) {
  struct vozlisce *prvi = NULL;
  for (int i = 0; i < 3; i++) {
    prvi = dodajVozlisce(prvi, i + 1);
  }
}
```

Naloga 8.2 Za vajo napišite program, ki na izhod izpiše vse podatke, shranjene v gornjem povezanem seznamu.

Pomoč: Kot je razvidno iz gornje kode, ima kazalec `naslednji` zadnjega vozlišča v povezanem seznamu vrednost NULL. To vrednost lahko uporabite, da preverite, kdaj ste dosegli konec seznama.

Naloga 8.3 Za vajo napišite še funkcijo `brisiVozlisce`, ki iz gornjega povezanega seznama odstrani prvo vozlišče. Funkcija naj vrne kazalec na vozlišče, ki po brisanju vozlišča postane prvo vozlišče skrajšanega seznama. Če smo izbrisali zadnje vozlišče iz seznama, naj funkcija vrne vrednost NULL. V enem od parametrov naj funkcija vrne tudi vrednost, ki je bila shranjena v izbrisanem vozlišču. Ne pozabite sprostiti pomnilnika s funkcijo `free`.

Primer klica funkcije:

```
prvi = brisiVozlisce(prvi, &n); /* Argument n zdaj hrani vrednost,
                                   ki je bila shranjena v izbrisanem vozlišču.  */
```

Primer: indeksiranje podatkovne zbirke

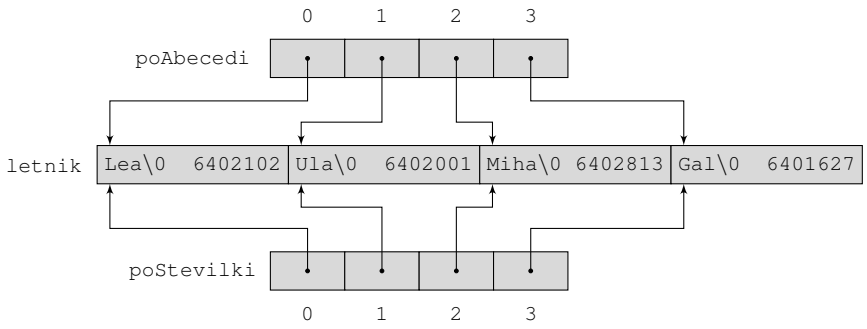
Ko smo v enem od prejšnjih primerov urejali seznam študentov po vpisnih številkah ali po abecedi, smo to počeli tako, da smo razvrščali elemente tabele s podatki o študentih neposredno v pomnilniku. Urejanje tabele na tak način pa je neučinkovito iz dveh razlogov: Prvič, če elementi tabele zasedajo veliko prostora, porabimo za njihovo kopiranje ogromno časa. Drugič, če želimo tabelo kasneje urediti po kakšnem drugem kriteriju[7], moramo postopek urejanja ponoviti, tudi če je bila tabela po tem kriteriju urejena že kdaj prej. Tabela je lahko namreč v pomnilniku naenkrat urejena le na en način. Težava je še toliko večja, ker so algoritmi za urejanje že sami po sebi časovno precej zahtevnejši od algoritmov za iskanje (podatke namreč najpogosteje urejamo prav zaradi kasnejšega lažjega iskanja). Če moramo vsakokrat, ko zamenjamo iskalni kriterij, pred iskanjem tabelo še urediti, potem je tak postopek zares neučinkovit.

Vse te težave lahko rešimo tako, da za vsak kriterij, po katerem želimo urediti seznam podatkov, ustvarimo svojo tabelo kazalcev na ta seznam. Potem ne urejamo podatkov, temveč tabele kazalcev na te podatke. Pri tem niti ni pomembno, ali so podatki shranjeni v tabeli ali povezanem seznamu.

Kot primer vzemimo še enkrat tabelo `letnik` z imeni in vpisnimi številkami študentov s strani 137 ter na vsakega od študentov usmerimo po en kazalec iz vsake od dveh tabel kazalcev:

```
#define ST_VPISANIH 4
//...
/* Dve tabeli s po štirimi kazalci tipa student_t *:     */
student_t *poAbecedi[ST_VPISANIH];
student_t *poStevilki[ST_VPISANIH];
/* Na vsak element tabele letnik usmerimo po en kazalec
   iz vsake od obeh tabel kazalcev:                      */
for (int i = 0; i < ST_VPISANIH; i++) {
  poAbecedi[i] = poStevilki[i] = &letnik[i];
}
```

Najprej smo deklarirali dve tabeli kazalcev tipa `student_t *`, ki imata isto dolžino kakor tabela `letnik`. Potem smo kazalce obeh tabel po vrsti usmerili na posamezne elemente tabele `letnik`. Ko se izvede gornja koda, imamo v pomnilniku takšno stanje:



S slike se vidi, da kazalca `poAbecedi[0]` in `poStevilki[0]` kažeta na prvi element tabele `letnik`, kazalca `poAbecedi[1]` in `poStevilki[1]` kažeta na drugi

[7]S pojmom *kriterij* tu označujemo posameznega člana strukture, po katerem izvajamo urejanje ali iskanje. V terminologiji podatkovnih zbirk govorimo namesto tega o ***polju*** (angl. field).

element tabele `letnik`, in tako dalje. Tabeli kazalcev `poAbecedi` in `poStevilki` bomo uporabili, da se izognemo urejanju tabele samih strukturnih spremenljivk. Namesto tega bomo uredili obe tabeli kazalcev. V ta namen bomo spet uporabili funkcijo `qsort`, ki pa ji bomo tokrat podali naslov ustrezne tabele kazalcev. Pri tem moramo popraviti tudi podatek o velikosti posameznega elementa tabele, ki zdaj ni strukturna spremenljivka, temveč kazalec na strukturno spremenljivko. Tretji parameter funkcije `qsort` bo zato namesto `sizeof(student_t)` zdaj `sizeof(student_t *)`. Na primer, takole uredimo študente najprej po abecedi, potem pa še po vpisnih številkah:

```
qsort(poAbecedi, ST_VPISANIH, sizeof(student_t *), primerjajAbeceda);
qsort(poStevilki, ST_VPISANIH, sizeof(student_t *), primerjajVpisStev);
```

Ker so elementi tabele, ki jo urejamo, zdaj kazalci na strukturne spremenljivke, moramo popraviti tudi definiciji obeh funkcij za primerjavo elementov `primerjajAbeceda` in `primerjajVpisStev`. Vsak od parametrov obeh funkcij bo zdaj kazalec ***na kazalec***[8] na strukturno spremenljivko, kar moramo upoštevati pri pisanju kode. Najprej moramo kazalec tipa `void *` pretvoriti v kazalec tipa `student_t **`:

```
(student_t **) a
```

Dve zvezdici v oznaki podatkovnega tipa govorita o tem, da gre za kazalec na kazalec na spremenljivko tipa `student_t`. Da pridemo prek takšnega kazalca do članov strukturne spremenljivke, moramo uporabiti najprej operator posredovanja, s čimer dobimo kazalec na strukturno spremenljivko:

```
*(student_t **)a  /* Ta izraz je kazalec tipa student_t *.  */
```

S takšnim kazalcem lahko zdaj pridemo do posameznih članov z uporabo selektorja v obliki puščice (`->`):

```
(*(student_t **)a)->ime
(*(student_t **)a)->vpisnaStev
```

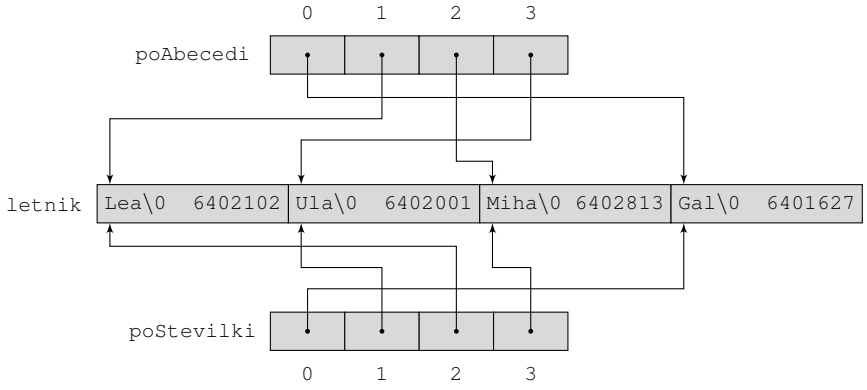
Funkciji `primerjajAbeceda` in `primerjajVpisStev` bosta tako podobni tistima s strani 139 z nekaj dodanimi zvezdicami. Tu je kompleten program, ki uredi tabelo študentov tako po abecedi kot tudi po vpisnih številkah:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define DOLZINA_IMENA 20
#define ST_VPISANIH 4

typedef struct {
  char ime[DOLZINA_IMENA + 1];
  int vpisnaStev;
} student_t;

int primerjajAbeceda (const void *a, const void *b) {
  return strcmp((*(student_t **)a)->ime, (*(student_t **)b)->ime);
}
```

[8] Parameter funkcije za primerjavo mora biti kazalec na element tabele. Element tabele je kazalec na strukturno spremenljivko, zato mora biti parameter te funkcije kazalec na kazalec na strukturno spremenljivko.

```
int primerjajVpisStev (const void *a, const void *b) {
  return (*(student_t **)a)->vpisnaStev - (*(student_t **)b)->vpisnaStev;
}

int main(void) {
  student_t letnik[ST_VPISANIH] = {{"Lea",  6402102}, {"Ula", 6402001},
                                   {"Miha", 6402813}, {"Gal", 6401627}};
  student_t *poAbecedi[ST_VPISANIH];
  student_t *poStevilki[ST_VPISANIH];

  for (int i = 0; i < ST_VPISANIH; i++) {
    poAbecedi[i] = poStevilki[i] = &letnik[i];
  }

  qsort(poAbecedi, ST_VPISANIH, sizeof(student_t *), primerjajAbeceda);
  qsort(poStevilki, ST_VPISANIH, sizeof(student_t *), primerjajVpisStev);

  /* Izpiše tabelo v originalnem vrstnem redu:    */
  for (int i = 0; i < ST_VPISANIH; i++) {
    printf("%d %s\n", letnik[i].vpisnaStev, letnik[i].ime);
  }

  /* Izpiše tabelo urejeno po abecedi:            */
  for (int i = 0; i < ST_VPISANIH; i++) {
    printf("%d %s\n", poAbecedi[i]->vpisnaStev, poAbecedi[i]->ime);
  }

  /* Izpiše tabelo urejeno po vpisnih številkah:  */
  for (int i = 0; i < ST_VPISANIH; i++) {
    printf("%d %s\n", poStevilki[i]->vpisnaStev, poStevilki[i]->ime);
  }
  return 0;
}
```

Po obeh klicih funkcije `qsort` v gornjem programu imamo v pomnilniku takšno stanje:



Vidimo, da se originalna tabela ni spremenila, obe tabeli kazalcev pa zdaj kažeta na elemente tako, da lahko preko njiju pridemo do elementov v urejenem vrstnem redu: Prvi kazalec tabele `poAbecedi` kaže na element tabele `letnik`, ki je prvi po abecedi, drugi kazalec kaže na element, ki je drugi po abecedi, in tako dalje. Na enak način kaže prvi kazalec tabele `poStevilki` na element tabele `letnik`, ki je prvi po vpisni številki, drugi kazalec kaže na element, ki je drugi po vpisni številki, in tako dalje.

Ko gornji program zaženemo, dobimo na izhodu tri različne vrstne rede izpisa: originalnega, urejenega po abecedi in urejenega po vpisnih številkah:

```
6402102 Lea      6401627 Gal      6401627 Gal
6402001 Ula      6402102 Lea      6402001 Ula
6402813 Miha     6402813 Miha     6402102 Lea
6401627 Gal      6402001 Ula      6402813 Miha
```

Čas, ki ga pri urejanju tabele pridobimo zaradi manjše količine podatkov, ki jih moramo pri tem kopirati, navadno odtehta dodaten pomnilniški prostor, ki ga porabimo za pomožne tabele kazalcev. Zlasti kadar zasedejo elementi tabele veliko pomnilnika, je njihovo kopiranje v primerjavi s kopiranjem kazalcev zelo zamudno. Še pomembneje je, da lahko z več tabelami kazalcev eno in isto tabelo hkrati uredimo po različnih kriterijih. Iskanje po takšni tabeli je zelo učinkovito iz dveh razlogov: Prvič, ker so podatki urejeni, lahko za iskanje uporabimo na primer ***dvojiško iskanje*** (angl. binary search). Za razliko od linearne časovne zahtevnosti ($\mathcal{O}(n)$), ki jo ima linearno iskanje po neurejenih podatkih, je časovna zahtevnost dvojiškega iskanja logaritemska ($\mathcal{O}(\log n)$). Drugič, ker je lahko tabela hkrati urejena po poljubno mnogo kriterijih, je dovolj, da izvedemo urejanje po vsakem od kriterijev le enkrat. To je zelo pomembno, kajti celo najučinkovitejši algoritmi, med katere spada tudi algoritem hitrega urejanja s funkcijo `qsort`, izkazujejo skoraj linearno časovno zahtevnost ($\mathcal{O}(n \log n)$), kar je slabše celo od linearnega iskanja po neurejeni tabeli. Zato bi bilo časovno zelo potratno, če bi morali pred vsako menjavo iskalnega kriterija znova zagnati urejanje tabele.

Podobno tehniko uporabljajo ***podatkovne zbirke*** (angl. database), kjer lahko nad polji, po katerih bomo pogosteje iskali podatke, ustvarimo ***indeks*** (angl. index). Indeks ni nič drugega kot tabela kazalcev na zapise v podatkovni zbirki. Indeksirana podatkovna zbirka omogoča veliko hitrejše iskanje.

Naloga 8.4 Funkcija `qsort` je prikladna za urejanje podatkov, ki so shranjeni v tabeli, ne moremo pa je uporabiti neposredno za urejanje povezanega seznama. Na srečo lahko tudi nad povezanim seznamom ustvarimo indeks, preko katerega lahko uredimo seznam s funkcijo `qsort`.

Za vajo napišite program, ki po abecednem redu in vpisni številki uredi seznam študentov, ki je shranjen v povezanem seznamu.

Primer: zmanjševanje števila parametrov funkcije

Vzemimo za primer, da imamo v programu definirano funkcijo, ki iz tega ali onega razloga potrebuje zelo veliko parametrov:

```
void narediNekaj(char *ePosta, char *uporIme, int status, char spol,
                 int steviloNeuspelihPoskusov, time_t zadnjaPrijava,
                 time_t zadnjaNeuspelaPrijava, float lokacija[]) {
//...
}
```

Koda, ki vsebuje takšne vrste funkcije, je nepregledna in zahtevna za vzdrževanje. Pri tolikšni količini parametrov obstaja precejšnja verjetnost, da se v času razvoja programa pojavi potreba po dodatnih parametrih. To prinese precej zamudnega dela, ko moramo na več mestih predelovati obstoječo kodo. Težavo predstavlja tudi vrstni red parametrov, ki mnogokrat ni logičen in si ga je zato težko zapomniti. Vsemu temu se lahko izognemo

tako, da parametre zložimo v ustrezen strukturni tip, ki ga potem uporabimo kot edini parameter funkcije:

```
typedef struct {
  char *ePosta;
  char *uporIme;
  int status;
  char spol;
  int steviloNeuspelihPoskusov;
  time_t zadnjaPrijava;
  time_t zadnjaNeuspelaPrijava;
  float lokacija[3];
} parametri_t;

void narediNekaj(parametri_t *par) {
//...
}
```

Funkcijo `narediNekaj` kličemo zdaj z enim samim parametrom:

```
parametri_t mojiParametri;
//...
narediNekaj(&mojiParametri);
```

Naloga 8.5 Predelajte funkcijo `pretvoriVUro` na strani 85, tako da bo funkcija sprejela en sam parameter tipa `ura_t`, ki je določen takole:

```
typedef struct {
  time_t sekunde;  /* Vhodni podatek.   */
  int h, m, s;     /* Izhodni podatki.  */
} ura_t;
```

Prototip funkcije bo potemtakem videti takole:

```
void pretvoriVUro(ura_t *u);
```

8.4 Naloge

Naloga 8.6 Določite strukturni tip `mojNiz` z dvema članoma. Prvi član naj bo znakovni niz (tabela znakov velikosti `D + 1` bajtov), drugi član pa naj bo celo število, ki hrani dejansko dolžino shranjenega znakovnega niza (tj. število vidnih znakov brez ničelnega znaka).

Napišite tudi definiciji funkcij `nastaviNiz` in `izlociPresledke`. Prvi parameter funkcije `nastaviNiz` naj bo kazalec na spremenljivko tipa `mojNiz`, drugi parameter pa naj bo znakovni niz. Funkcija naj v podano strukturno spremenljivko shrani podani znakovni niz in njegovo dolžino. Če je znakovni niz predolg (daljši od `D` zakov), naj ga ustrezno odreže.

Funkcija `izlociPresledke` naj kot parameter sprejme kazalec na spremenljivko tipa `mojNiz` in iz znakovnega niza, ki je v njej, odstrani vse presledke. Ustrezno naj popravi tudi podatek o dolžini niza. Funkcija naj vrne število odstranjenih presledkov.

Primer kode, ki uporablja strukturni podatkovni tip `mojNiz` in obe zahtevani funkciji:

```
#define D 9
//...
mojNiz s1;
nastaviNiz(&s1, "a b c d e f");
printf("%s - %d", s1.niz, s1.dolz);  /* Izpiše: a b c d e - 9  */
printf("%d", izlociPresledke(&s1));  /* Izpiše: 4              */
printf("%s - %d", s1.niz, s1.dolz);  /* Izpiše: abcde - 5      */
```

Naloga 8.7 Spremenite strukturni tip `mojNiz` in definicijo funkcije `nastaviNiz` iz naloge 8.6. Prvi član strukturnega tipa `mojNiz` naj bo namesto znakovnega niza kazalčna spremenljivka tipa `char *`, ki jo ob inicializaciji strukturne spremenljivke tipa `mojNiz` postavite na vrednost NULL.

Funkcija `nastaviNiz` naj z uporabo funkcije `malloc` rezervira dovolj pomnilnika za podani niz in na rezervirani blok usmeri prvega člana podane strukturne spremenljivke tipa `mojNiz`. Če podana strukturna spremenljivka že vsebuje znakovni niz (tj., če ima njen prvi član vrednost, različno od NULL), naj funkcija `nastaviNiz` najprej preveri, ali je rezerviranega dovolj pomnilnika za vpis novega znakovnega niza. Če ga ni, naj najprej ustrezno spremeni količino rezerviranega pomnilnika s funkcijo `realloc` (glej nalogo 7.13 na strani 131).

Primer kode, ki uporablja tip `mojNiz` in funkcijo `nastaviNiz`:

```
mojNiz s1 = {NULL, 0};
nastaviNiz(&s1, "ab c d");
printf("%s - %d", s1.niz, s1.dolz);  /* Izpiše: ab c d - 6    */
nastaviNiz(&s1, "EF..GH.I");
printf("%s - %d", s1.niz, s1.dolz);  /* Izpiše: EF..GH.I - 8  */
```

Naloga 8.8 Napišite funkcijo `absKompl`, katere parameter naj bo strukturna spremenljivka, ki predstavlja kompleksno število. Funkcija naj vrne absolutno vrednost podanega parametra.

Naloga 8.9 Napišite funkcijo `deli`, ki kot parametra sprejme dve strukturni spremenljivki, ki predstavljata kompleksni števili. Funkcija naj vrne kvocient podanih parametrov.

Pomoč: Kvocient w/z kompleksnih števil $w = u + vi$ in $z = x + yi$ izračunamo tako, da števec in imenovalec najprej množimo s kompleksno konjugirano vrednostjo imenovalca:

$$\frac{w}{z} = \frac{w\bar{z}}{z\bar{z}} = \frac{w\bar{z}}{|z|^2}$$

Naloga 8.10 Napišite program, ki iz telefonskega imenika izpiše vse vnose, ki se začnejo s kombinacijo črk, ki jo vpiše uporabnik. V programu uporabite tabelo, ki jo kaže naslednji del kode:

```
#define ST_ZNAKOV 20
//...
typedef struct {
  char ime[ST_ZNAKOV + 1];
  unsigned long tel;
```

```
} vnos_t;
//...
vnos_t imenik[] = {{"Janez",    41555666},
                   {"Tristan",  30652853},
                   {"Jasminka", 40761845},
                   {"Janko",    51764981},
                   {"Barbara",  51158592},
                   {"Terezija", 41888656}};
```

Primer delovanja programa:

```
Išči (0 konča): Jan
  Janez 041555666
  Janko 051764981
Išči (0 konča): Fr
  Ni zadetkov.
Išči (0 konča): 0
Nasvidenje!
```

Naloga 8.11 Napišite program, ki z vhoda prebere največ deset angleških besed in jih urejene po abecedi izpiše na izhod. Za urejanje po abecedi uporabite knjižnično funkcijo `qsort`.

Naloga 8.12 Napišite program, ki uporabniku omogoča vnos, iskanje in spreminjanje podatkov v telefonskem imeniku.

Primer delovanja programa:

```
Ukaz (1-išči, 2-dodaj/spremeni, 3-izhod): 2
Vnesi ime: Klavdija
Vnesi številko: 041222888
Ukaz (1-išči, 2-dodaj/spremeni, 3-izhod): 2
Vnesi ime: Teodor
Vnesi številko: 051123456
Ukaz (1-išči, 2-dodaj/spremeni, 3-izhod): 1
Vnesi ime: Klavdija
Telefonska številka: 041222888
Ukaz (1-išči, 2-dodaj/spremeni, 3-izhod): 2
Vnesi ime: Klavdija
Trenutna številka: 041222888
Vnesi novo številko: 051657981
Ukaz (1-išči, 2-dodaj/spremeni, 3-izhod): 3
Nasvidenje!
```

Program napišite z uporabo povezanega seznama.

9. POGLAVJE

NIZKONIVOJSKO PROGRAMIRANJE

Lastnosti jezika C, ki smo jih spoznali doslej, niso vezane na sistem ali hardver, na katerem program teče. Tehnike, ki jih bomo srečali v tem poglavju, pa terjajo, da natančno vemo, kako so podatki zapisani v pomnilniku, kar je odvisno od računalnika in prevajalnika. Ker je programiranje z uporabo teh tehnik vezano na najnižji nivo v hierarhiji programske kode, takšnemu programiranju pravimo ***nizkonivojsko programiranje*** (angl. low-level programming). Programi, ki jih pišemo z uporabo nizkonivojskih tehnik, zaradi tesnega stika z operacijskim sistemom in s hardverom niso zlahka prenosljivi. Uporabnost nizkonivojskih tehnik se pokaže zlasti pri pisanju sistemskih programov (kot so na primer operacijski sistemi in prevajalniki), grafičnih programov in programov za šifriranje ali programov, pri katerih je ključna učinkovita izraba pomnilnika in časa.

9.1 Bitni operatorji

Jezik C pozna šest ***bitnih*** (angl. bitwise) operatorjev, ki izvajajo operacije na nivoju posameznih bitov ***celoštevilskih*** podatkov. Ker uporabljamo bitne operacije večinoma v situacijah, v katerih imajo za nas pomen vrednosti posameznih bitov (ne pa vrednosti celotnih spremenljivk), se pri tem navadno omejimo na ***nepredznačena cela števila***. Namreč, določene operacije, kot je na primer pomik bitov v desno, imajo lahko na predznačenih številih drugačen učinek kot na nepredznačenih številih, rezultat operacije pa je lahko celo odvisen od izvedbe prevajalnika. Poleg tega bomo iz istega razloga (ker nas bodo zanimali

posamezni biti) zapisovali konstantne vrednosti v tem poglavju večinoma v šestnajstiškem zapisu.

Pomik bitov

Operacija ***pomika bitov*** (angl. bitwise shift) pomakne bite v dvojiškem zapisu celega števila v levo ali desno za določeno število bitov. C pozna dva operatorja za pomik bitov, ki ju prikazuje naslednja tabela:

Operator	Opis
>>	pomik v desno
<<	pomik v levo

Operandi obeh operatorjev so lahko podatki kakršnegakoli celoštevilskega tipa vključno s tipom `char`. Vrednost, ki jo vrne izraz `x << n`, je vrednost, ki jo dobimo, če bite v spremenljivki `x` pomaknemo za `n` mest v levo. Za vsak bit, ki na levi strani »izpade« iz vrednosti, se na desni strani vstavi ničla. Podobno je vrednost, ki jo vrne izraz `x >> n`, enaka vrednosti, ki jo dobimo, če bite v spremenljivki `x` pomaknemo za `n` mest v desno. Če je `x` nepredznačeno število, potem se z leve strani doda ustrezno število ničel. Če pa je `x` predznačeno število, potem se z desne dodajo enice ali ničle, odvisno od izvedbe prevajalnika. Smisel dodajanja enic je v tem, da se ohranja predznak podatka, katerega bite pomikamo v desno[1].

Kot smo omenili že uvodoma, se bomo nepotrebnim težavam, ki izvirajo iz različnih možnih izidov pomikanja bitov v desno, najlaže izognili tako, da bomo te operacije izvajali nad nepredznačenimi števili.

Poleg tega moramo paziti, da je desni operand operatorjev pomika bitov nenegativen ter manjši od končne bitne dolžine podatkovnega tipa izraza (po morebitnih pretvorbah tipa). V nasprotnem primeru je obnašanje operatorjev pomika bitov nedoločeno. Na primer:

```
unsigned long x;
unsigned char c;
//...
x = c >> 40;  /* Nedoločeno: pomik večji od bitne dolžine izraza.  */
x = c << 32;  /* Nedoločeno: pomik enak bitni dolžini izraza.      */
x = c << 10;  /* V redu: bitna dolžina izraza je 32.               */
x = c >> -2;  /* Nedoločeno: pomik za negativno število bitov.     */
```

Naslednji primeri kažejo učinek pomikanja bitov na spremenljivko tipa `unsigned short`, ki v pomnilniku navadno zasede 16 bitov:

```
unsigned short x, y;
x = 0x2A;          /* x je 2A (dvojiško 0000000000101010).  */
y = x << 2;        /* y je A8 (dvojiško 0000000010101000).  */
y = x >> 3;  /* y je zdaj  5 (dvojiško 0000000000000101).  */
y = x >> 7;  /* y je zdaj  0 (dvojiško 0000000000000000).  */
```

Tako kot smo navajeni pri večini cejevskih operatorjev, tudi operatorja pomikanja bitov ne spremenita vrednosti nobenega od svojih operandov. Če želimo rezultat operacije shra-

[1] Pomik bitov za n mest v levo je enakovreden množenju, pomik za n mest v desno pa deljenju z 2^n. Zato je smiselno, da se pri pomiku bitov v desno predznak števila ohranja. Pri pomiku v levo se predznak avtomatično ohranja. Kadar pa se ne, to pomeni, da je prišlo do prekoračitve.

niti, potrebujemo zato priredilni operator, lahko pa uporabimo tudi sestavljena priredilna operatorja >>= in <<=. Na primer:

```
unsigned short x;
x = 0x2A;          /* x je 2A (dvojiško 0000000000101010).  */
x <<= 14;  /* x je zdaj 8000 (dvojiško 1000000000000000).  */
x >>= 15;  /* x je zdaj    1 (dvojiško 0000000000000001).  */
```

Eniški komplement, bitni IN, ALI ter izključni ALI

V naslednji tabeli so zbrani preostali štirje bitni operatorji:

Operator	Opis
~	eniški komplement
&	bitni IN
|	bitni ALI
^	bitni izključni ALI

Od operatorjev v tabeli je ***eniški komplement*** (angl. one's complement) edini unarni operator. Eniški komplement enostavno negira vse bite v podanem operandu, kakor kaže naslednji primer:

```
unsigned short x = 0x42;  /* x je 42 (dvojiško 0000000001000010).  */
x = ~x;             /* x je zdaj FFBD (dvojiško 1111111110111101).  */
```

Ostali trije operatorji izvajajo bitne operacije IN, ALI in ***izključni ALI*** (angl. exclusive or). Naslednja tabela prikazuje, kako ti trije operatorji delujejo:

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Operator IN vrne vrednost ena samo takrat, ko sta oba bita, ki nastopata v operaciji, enaka ena. Operator ALI vrne vrednost ena, kakor hitro je vsaj en od obeh bitov enak ena. Operator izključni ALI vrne vrednost ena natanko takrat, ko sta bita med seboj različna. Naslednja koda prikazuje, kako lahko te tri operatorje uporabimo v programu:

```
unsigned short x, y, m;
x = 0x2E;          /* x je 2E (dvojiško 0000000000101110).  */
y = 0x2B;          /* y je 2B (dvojiško 0000000000101011).  */
                   /* ------------------------------------  */
m = x & y;         /* m je 2A (dvojiško 0000000000101010).  */
m = x | y;  /* m je zdaj 2F (dvojiško 0000000000101111).  */
m = x ^ y;  /* m je zdaj  3 (dvojiško 0000000000000101).  */
```

Iz primera se vidi, da se operacije izvajajo nad istoležnimi biti. To pomeni, da je na primer skrajno desni bit vrnjene vrednosti posledica ustrezne operacije nad skrajno desnima bitoma obeh operandov. Prav tako je drugi bit z desne posledica ustrezne operacije nad drugim bitom prvega in drugim bitom drugega operanda, in tako dalje.

Naloga 9.1 Za vajo razmislite, kakšna je razlika med bitnima in logičnima operatorjema IN in ALI. Podana je naslednja deklaracija:

```
unsigned short x = 1, y = 2, z = 3;
```

Izračunajte vrednosti naslednjih izrazov:

```
(a) x && y
(b) x && z
(c) x & y
(č) x & z
(d) x || y
(e) x | y
(f) x || 1
(g) x | 1
```

Bodite pozorni na razliko med logičnima operatorjema `&&` in `||` ter bitnima operatorjema `&` in `|`. Čeprav vrneta operatorja `&&` in `&` včasih isto vrednost, sta to dva povsem različna operatorja. Isto velja za operatorja `||` in `|`.

Za vsakega od treh binarnih bitnih operatorjev `&`, `|` in `^` obstajajo tudi ustrezni sestavljeni bitni operatorji, kot lahko vidimo na naslednjem primeru:

```
unsigned short x, y;
x = 0x0D;      /* x je  D (dvojiško 0000000000001101).  */
y = 0x19;      /* y je 19 (dvojiško 0000000000011001).  */
               /* ------------------------------------  */
y &= x;  /* y je zdaj  9 (dvojiško 0000000000001001).  */
y |= x;  /* y je zdaj  D (dvojiško 0000000000001101).  */
y ^= x;  /* y je zdaj  0 (dvojiško 0000000000000000).  */
```

Dostop do bitov s pomočjo bitnih operatorjev

Kadar se lotimo nizkonivojskega programiranja, se nam pogosto pojavi potreba po shranjevanju ali branju informacije v obliki posameznih bitov ali skupkov bitov. V tem razdelku si bomo ogledali, kako lahko ***postavimo***[2] (angl. set), ***pobrišemo***[2] (angl. clear) ali ***negiramo*** določen bit v spremenljivki, ne da bi pri tem spremenili vrednosti preostalih bitov. Spoznali bomo tudi, kako lahko ***beremo*** vrednost posameznega bita v spremenljivki. Tako kot doslej bomo za primere uporabljali spremenljivke tipa `unsigned short`, za katere predpostavimo, da imajo dolžino 16 bitov.

Postavljanje bita Vzemimo, da je spremenljivka x logična spremenljivka, ki lahko zavzame vrednost bodisi nič bodisi ena. Operacija ALI ima naslednji dve pomembni lastnosti:

$$\begin{aligned} x \,|\, 0 &= x \\ x \,|\, 1 &= 1 \end{aligned}$$

Z drugimi besedami: operacija ALI z ničlo vrne nespremenjeno vrednost bita, medtem ko vrne operacija ALI z enko v vsakem primeru vrednost ena.

[2]Kadar priredimo bitu vrednost ena, pravimo, da bit ***postavimo***, kadar mu priredimo vrednost nič, pa pravimo, da ga ***pobrišemo***.

Z upoštevanjem teh dveh ugotovitev lahko postavimo bit na določenem mestu v spremenljivki x tako, da uporabimo bitno operacijo ALI ter tako imenovano ***masko*** iz samih ničel in enko na mestu bita, ki ga želimo postaviti. Na primer, tako lahko postavimo bit številka pet v spremenljivki x:

```
unsigned short x = 0xFF00;  /* Dvojiško 1111111100000000.          */
unsigned short m = 0x0020;  /* Dvojiško 0000000000100000 (maska).  */
x |= m;                     /* x je zdaj 1111111100100000.         */
```

Če želimo dejansko spremeniti vrednost bita v spremenljivki x, moramo seveda uporabiti bodisi priredilni bodisi sestavljeni priredilni operator, kajti bitne operacije ne spreminjajo vrednosti svojih operandov.

Z uporabo operatorja za pomik bitov lahko isti učinek dosežemo na preglednejši način. Naslednji del kode postavi bit spremenljivke x, katerega položaj je shranjen v spremenljivki m:

```
unsigned short x = 0xFF00, m = 5;
x |= 1 << m;
```

Izraz `1 << m` pomakne enko za m mest v levo. Ker je m enak pet, nam ta izraz vrne dvojiško vrednost 10000, kar je šestnajstiško 20. To pa je vrednost, ki smo jo uporabili že v prejšnjem primeru.

Brisanje bita Za brisanje bita uporabimo operacijo IN, ki ima naslednji pomembni lastnosti:

$$\begin{aligned} x \,\&\, 0 &= 0 \\ x \,\&\, 1 &= x \end{aligned}$$

Z drugimi besedami: operacija IN z enko vrne nespremenjeno vrednost bita, medtem ko vrne operacija IN z ničlo v vsakem primeru vrednost nič.

Z upoštevanjem teh dveh lastnosti lahko pobrišemo bit številka pet v spremenljivki x tako, da uporabimo bitno operacijo IN ter masko iz samih enk in ničlo na mestu bita številka pet:

```
unsigned short x = 0x00FF;  /* Dvojiško 0000000011111111.          */
unsigned short m = 0xFFDF;  /* Dvojiško 1111111111011111 (maska).  */
x &= m;                     /* x je zdaj 0000000011011111.         */
```

Z uporabo operatorja za pomik bitov lahko isti učinek dosežemo na elegantnejši način. Naslednji del kode pobriše bit spremenljivke x, katerega položaj je shranjen v spremenljivki m:

```
unsigned short x = 0x00FF, m = 5;
x &= ~(1 << m);
```

Na desni strani sestavljenega operatorja `&=` pomaknemo enko za m mest proti levi in nad tako dobljeno vrednostjo naredimo eniški komplement. Tako dobimo vrednost iz samih enk in ničle na mestu, ki ga določa vrednost spremenljivke m. V našem konkretnem primeru je to šestnajstiška vrednost FFDF, ki smo jo uporabili za masko že v prejšnjem primeru.

Negacija bita Za negacijo bita uporabimo operacijo izključni ALI, ki ima naslednji pomembni lastnosti:

$$\begin{aligned} x \hat{} 0 &= x \\ x \hat{} 1 &= \tilde{} x \end{aligned}$$

Z drugimi besedami: operacija izključni ALI z ničlo vrne nespremenjeno vrednost bita, medtem ko vrne operacija izključni ALI z enko njeno negirano vrednost.

Z upoštevanjem teh dveh lastnosti lahko negiramo bit številka pet v spremenljivki `x` tako, da uporabimo bitno operacijo izključni ALI ter masko iz samih ničel in enko na mestu bita številka pet:

```
unsigned short x = 0xFFFF;  /* Dvojiško 1111111111111111.          */
unsigned short m = 0x0020;  /* Dvojiško 0000000000100000 (maska).  */
x ^= m;                     /* x je zdaj 1111111111011111.         */
```

Podobno kot pri postavljanju in brisanju bita lahko tudi v tem primeru sestavimo masko s pomočjo operatorja za pomik bitov. Naslednji del kode negira bit spremenljivke `x`, katerega položaj je shranjen v spremenljivki `m`:

```
unsigned short x = 0xFFFF, m = 5;
x ^= 1 << m;
```

Preverjanje vrednosti bita Z masko s samimi ničlami in eno samo enko ter operacijo IN lahko preverimo, kakšno vrednost ima določen bit v nekem podatku. Trik je v tem, da vrne operacija IN z masko, ki vsebuje eno samo enko, neničelno vrednost takrat in samo takrat, ko je na mestu enke v maski enka tudi v podatku. Naslednji primer kaže, kako preverimo vrednost bita številka pet v spremenljivki `x`:

```
if (x & 0x0020) printf("Bit je postavljen.");
else printf("Bit je pobrisan.");
```

Isti učinek lahko dosežemo tudi z naslednjim stavkom, ki preveri stanje bita na položaju `m`:

```
if (x & 1 << m) printf("Bit %d je postavljen.", m);
else printf("Bit %d je pobrisan.", m);
```

Bodite pozorni, ker imajo bitni operatorji nižjo prednost kot primerjalni in aritmetični operatorji. Pri preverjanju vrednosti bita se pogosto prikrade naslednja napaka:

```
if (x & 0x0020 != 0) //...
```

Primerjalni operator ima prednost pred bitnim operatorjem IN. Zato je pogojni izraz, ki smo ga uporabili v gornjem stavku `if`, enakovreden izrazu `x & (0x0020 != 0)`. Ker je konstanta `0x0020` vedno različna od nič, je videti zadnji stavek `if` v resnici takole:

```
if (x & 1) //...
```

Ta pogoj je izpolnjen samo takrat, kadar je različen od nič skrajno desni bit spremenljivke `x`.

Če želimo preglednejšo programsko kodo, lahko posamezne bite poimenujemo z uporabo makrov. Vzemimo za primer, do so prvi štirje biti (tj. biti s številkami od nič do tri)

v spremenljivki `indikatorji` povezani s stanji štirih signalnih svetlečih diod. Te bite lahko poimenujemo takole:

```
#define LED_VKLOP 1
#define LED_PRIPRAVLJENOST 2
#define LED_ALARM_TEMPERATURA 4
#define LED_ALARM_TLAK 8
```

Bodite pozorni, da ti makri ne hranijo zaporednih številk bitov, temveč so že kar maske, ki jih bomo uporabljali neposredno za upravljanje z ustreznimi biti.

Naslednji primeri kažejo, kako lahko prižgemo, ugasnemo ali preverimo stanje posamezne diode:

```
/* Prižge diodo, ki signalizira stanje vklopa:           */
indikatorji |= LED_VKLOP;
/* Ugasne diodo, ki signalizira stanje pripravljenosti:  */
indikatorji &= ~LED_PRIPRAVLJENOST;
/* Preveri stanje diode za alarm temperature:            */
if (indikatorji & LED_ALARM_TEMPERATURA) //...
```

Na podoben način lahko postavimo, brišemo ali preverimo več bitov hkrati:

```
/* Prižge diodi, ki signalizirata alarma temperature in tlaka:         */
indikatorji |= LED_ALARM_TEMPERATURA | LED_ALARM_TLAK;
/* Ugasne diodi, ki signalizirata alarma temperature in tlaka:         */
indikatorji &= ~(LED_ALARM_TEMPERATURA | LED_ALARM_TLAK);
/* Preveri stanji diod, ki signalizirata alarma temperature in tlaka:  */
if (indikatorji & (LED_ALARM_TEMPERATURA | LED_ALARM_TLAK)) //...
```

Velja omeniti, da zadnji stavek v gornji kodi preverja, ali je prižgana vsaj ena od obeh diod.

Polja bitov

Skupini več zaporednih bitov pravimo ***polje bitov*** (angl. bit-field). Kot bomo videli v nadaljevanju, je delo s polji bitov nekoliko bolj zapleteno kot delo s posameznimi biti. Ogledali si bomo dve najpogostejši operaciji, ki sta zapisovanje in branje podatka.

Zapisovanje podatka v polje bitov Če želimo v polje bitov zapisati določen podatek, potem moramo to polje najprej pobrisati (z operacijo IN), potem pa vanj zapisati želeno vrednost (z operacijo ALI). Na primer, če želimo vpisati desetiško vrednost 12 v polje štirih bitov spremenljivke `x` med bitoma štiri in sedem, lahko to storimo takole:

```
unsigned short x = 0xFFFF, n = 4, vr = 12;
x = x & ~(0xF << n) | (vr << n);  /* x je zdaj 0xFFCF.  */
```

Izraz `~(0xF << n)` ustvari masko `0xFF0F`, s pomočjo katere v spremenljivki `x` pobrišemo bite med bitoma štiri in sedem. Izraz `vr << n` pa ustvari masko `0x00C0`, s pomočjo katere vpišemo vrednost 12 (šestnajstiško C) v polje štirih bitov med bitoma štiri in sedem.

Branje podatka iz polja bitov Če želimo iz polja bitov prebrati podatek, moramo ta podatek najprej pomakniti na skrajno desno. V premaknjenem podatku potem enostavno pobrišemo vse bite, ki niso del polja bitov. Na primer, če želimo prebrati podatek, ki je vpisan v polju bitov med bitoma štiri in sedem spremenljivke `x`, potem to storimo takole:

```
unsigned short x = 0xFFCF, n = 4, vr;
vr = (x >> n) & 0x000F;  /* vr je zdaj 12.  */
```

Polja bitov v strukturah

Čeprav nam tehnika, ki smo jo spoznali v prejšnjem razdelku, omogoča kompaktno shranjevanje podatkov, pa zna biti begajoča. Na srečo ponuja jezik C alternativno možnost: deklariramo lahko strukturo, katere člani so polja bitov. Kot primer vzemimo strukturo, ki hrani datum kot dan v mesecu, mesec in leto. Pri tem nam za zapis dneva v mesecu zadostuje pet bitov, za mesec štirje biti in – če se zadovoljimo z letnicami med nič in 4095 – za leto 12 bitov. Takole lahko deklariramo strukturni tip z omenjenimi bitnimi dolžinami njegovih članov `dan`, `mesec` in `leto`:

```
typedef struct {
  unsigned int dan: 5;
  unsigned int mesec: 4;
  unsigned int leto: 12;
} datum_t;
```

Številka za vsakim članom pomeni njegovo dolžino v bitih. Koliko pomnilnika zasede spremenljivka strukturnega tipa, ki vsebuje polja bitov, je odvisno od izvedbe prevajalnika. Polja bitov so shranjena tesno drugo za drugim znotraj tako imenovane ***shranjevalne enote*** (angl. storage unit), ki je lahko dolga bodisi osem bodisi 16 bodisi 32 bitov. Če ostane na koncu shranjevalne enote še nekaj prostora, vendar premalo za naslednje polje bitov, potem pustijo nekateri prevajalniki vrzel in začnejo shranjevati preostala polja bitov v naslednjo shranjevalno enoto. Drugi prevajalniki razlomijo polje bitov in ga razdelijo med dve shranjevalni enoti, tako da vmes ni nobenih vrzeli.

Tip polja bitov je lahko bodisi `unsigned int` bodisi `signed int`. Standard dovoljuje tudi tip `int`, ki pa lahko povzroči težave s prenosljivostjo, saj v tem primeru nekateri prevajalniki najpomembnejšega bita ne obravnavajo kot predznak.

Polja bitov uporabljamo kot običajne člane strukturnih spremenljivk:

```
datum_t danes = {27, 4, 2021};
//...
printf("%u", danes.dan);
```

Ker se lahko polje bitov začne kjerkoli znotraj posameznega bajta, ne more imeti naslova v klasičnem pomenu besede. Zato na članih strukturne spremenljivke, ki so polja bitov, ne moremo uporabiti naslovnega operatorja. Na primer, polja bitov ne moremo uporabljati za branje podatkov neposredno s funkcijo `scanf`:

```
scanf("%u", &danes.dan);  /* Napaka: naslov polja bitov ne obstaja.  */
```

Seveda pa lahko funkcijo `scanf` še vedno uporabimo za branje vrednosti v običajno spremenljivko, ki jo potem priredimo polju bitov `danes.dan`.

9.2 Izpis pomnilnika

V različnih primerih je uporabno, če si lahko izpišemo vsebino določenih delov pomnilnika v najbolj surovi obliki – kot zaporedje bajtov brez podatka o tem, ali gre za kodo ali podatke. V jeziku C lahko z uporabo kazalcev to brez težav storimo. Zavedati se moramo le, da večina sistemov poganja programe v tako imenovanem zaščitenem načinu (angl. protected mode). To pomeni, da program brez posebnega dovoljenja sistema nima dostopa do delov pomnilnika, ki ne pripadajo neposredno programu. Naslednji program izpiše 80 bajtov vsebine pomnilnika, ki se začne na naslovu, ki ga vnese uporabnik:

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

#define VRSTICA 8
typedef unsigned char BYTE;

int main(void) {
  unsigned int naslov;
  BYTE *k;
  char t[] = "To so podatki";
  char *niz = "To je konstanten niz";
  short x = -128;
  printf("Naslov konstantnega niza: %X\n", (unsigned int) niz);
  printf("Naslov spremenljivke x: %X\n", (unsigned int) &x);
  printf("Vnesi naslov: ");
  scanf("%x", &naslov);
  k = (BYTE *) naslov;
  printf("Naslov           Bajti            Znaki\n");
  printf("-----------------------------------------\n");
  for (int i = 0; i < 10; i++) {
    printf("%X ", (unsigned int) k);
    for (int j = 0; j < VRSTICA; j++) {
      printf(" %02X", *(k + j));
    }
    printf("  ");
    for (int j = 0; j < VRSTICA; j++) {
      if (isprint(*(k + j))) {
        printf("%c", *(k + j));
      }
      else {
        printf(".");
      }
    }
    printf("\n");
    k += VRSTICA;
  }
  return 0;
}
```

Prva stvar v gornjem programu, ki je vredna omembe, je definicija tipa `BYTE`, ki smo ga določili s pomočjo tipa `char`. Kot že vemo, podatkovni tip `char` vedno zasede natanko en bajt, lahko pa je predznačen ali nepredznačen. Kadar se spustimo na nivo vsebine pomnilnika, obravnavamo bajt kot najmanjšo enoto, ki jo lahko naslavljamo, ne zanima pa nas njegov konkretni pomen – zanima nas zgolj kombinacija bitov, ki je v njem zapisana. Kot že vemo, za takšne vrste podatkov uporabimo nepredznačen tip. Lahko bi torej uporabili podatkovni tip `unsigned char`. Koda pa je veliko bolj čitljiva, če ta tip preimenujemo v tip `BYTE`, ki že sam po sebi govori o tem, da nas zanimajo zgolj surovi bajti.

Nekaj spremenljivk v funkciji `main` smo vključili zgolj zato, da bomo opazovali njihov zapis v pomnilniku. Gornji program najprej izpiše naslov, na katerem se nahaja konstanten znakovni niz, potem pa še naslov spremenljivke `x`. Program potem z vhoda prebere nepredznačeno celo število, ki ga vnese uporabnik kot naslov, od katerega naprej si želi ogledati vsebino pomnilnika. To nepredznačeno celo število se potem s pomočjo operatorja za pretvorbo tipa pretvori v kazalec tipa `BYTE *`, njegova vrednost pa se priredi kazalčni spremenljivki `k` (ki je tipa `BYTE *`).

Program na koncu izpiše 80 bajtov vsebine pomnilnika od vnesenega naslova naprej. Vsaka vrstica vsebuje osem bajtov in se začne z naslovom prvega od njih. Sledijo šestnajstiške kode vseh osmih bajtov in še istih osem bajtov, izpisanih v obliki znakov ASCII.

S funkcijo `isprint`, ki jo najdemo v standardni knjižnici <ctype.h>, preverimo, ali določena koda predstavlja znak, ki ga je mogoče izpisati. Namesto znakov, ki jih ni mogoče izpisati (kot so na primer kontrolni ali ničelni znaki), izpišemo piko.

Program zaženemo in dobimo takšen izpis:

```
Naslov konstantnega niza: 408024
Naslov spremenljivke x: 28FEFC
Vnesi naslov: 408024

Naslov           Bajti             Znaki
-----------------------------------------
408024  54 6F 20 6A 65 20 6B 6F  To je ko
40802C  6E 73 74 61 6E 74 65 6E  nstanten
408034  20 6E 69 7A 00 4E 61 73   niz.Nas
40803C  6C 6F 76 20 6B 6F 6E 73  lov kons
408044  74 61 6E 74 6E 65 67 61  tantnega
40804C  20 6E 69 7A 61 3A 20 25   niza: %
408054  58 0A 00 4E 61 73 6C 6F  X..Naslo
40805C  76 20 73 70 72 65 6D 65  v spreme
408064  6E 6C 6A 69 76 6B 65 20  nljivke
40806C  78 3A 20 25 58 0A 00 56  x: %X..V
```

Vpisali smo naslov konstantnega znakovnega niza. Po pričakovanjih smo v pomnilniku naleteli na ta konstantni znakovni niz skupaj z ostalimi konstantnimi znakovnimi nizi, ki smo jih uporabili kot parametre funkcij `printf`, ki sledijo.

Iz gornjega izpisa vidimo tudi, da so konstantni znakovni nizi shranjeni v povsem drugem delu pomnilnika kot pa lokalne spremenljivke funkcije `main` (kar vidimo iz naslova lokalne spremenljivke `x`). Poženimo program še enkrat, le da tokrat vpišimo naslov spremenljivke `x`:

```
Naslov konstantnega niza: 408024
Naslov spremenljivke x: 28FEFC
Vnesi naslov: 28FEFC

Naslov           Bajti             Znaki
-----------------------------------------
28FEFC  80 FF 54 6F 20 73 6F 20  ..To so
28FF04  70 6F 64 61 74 6B 69 00  podatki.
28FF0C  FC FE 28 00 24 80 40 00  ..(.$.@.
28FF14  00 00 00 00 03 00 00 00  ........
28FF1C  1C FF 28 00 28 FF 28 00  ..(.(.(.
28FF24  34 9E 9D 75 94 FF 28 00  4..u..(.
28FF2C  FD 10 40 00 01 00 00 00  ..@.....
28FF34  78 2F 39 00 60 1B 39 00  x/9.`.9.
28FF3C  FF FF FF FF 5C FF 28 00  ....\.(.
28FF44  D5 8C 9E 75 3C A3 BC 51  ...u<..Q
```

Spremenljivka `x` je tipa `short` in ima vrednost -128. Z znanjem, ki ga imamo o kodiranju predznačenih celih števil, ugotovimo, da je 16-bitni dvojiški zapis vrednosti -128 (v dvojiškem komplementu) enak 11111111 10000000. V šestnajstiškem zapisu je to FF80. Če pogledamo prav na začetek gornjega izpisa (na naslov 28FEFC), vidimo, da je tam v

resnici vpisana vrednost FF80, le da sta bajta med seboj zamenjana. To je posledica zapisa z ***manjšim koncem*** (angl. little-endian)[3].

Spremenljivki x v pomnilniku sledi znakovni niz t, za katerega vidimo, da zaseda 14 bajtov skupaj z ničelnim znakom, ki ga vidimo na zadnjem mestu druge vrstice. Prepoznamo lahko tudi naslednje štiri bajte, ki hranijo vrednost spremenljivke naslov. Samo obrniti moramo vrstni red teh štirih bajtov in dobimo vrednost 0028FEFC. To je naslov, ki smo ga vtipkali in ga je funkcija scanf shranila v spremenljivko naslov. Na koncu opozorimo še na zadnje štiri bajte v tretji vrstici, ki predstavljajo vrednost 00408024. To je naslov konstantnega niza, ki je shranjen v kazalčni spremenljivki niz.

Naloga 9.2 Za vajo poskusite ugotoviti, kje v pomnilniku je shranjena kazalčna spremenljivka k iz zadnjega programa, in pojasnite, zakaj je njena vrednost v izpisu takšna, kot je.

9.3 Opredeljevalec volatile

Na nekaterih sistemih so lahko določena pomnilniška mesta ***nestanovitna*** (angl. volatile) – vrednost, ki je shranjena na takšnem mestu, se lahko spremeni, tudi če je ni spremenil sam program. Vzemimo za primer, da se na mesto, na katero kaže kazalec p, shrani koda tipke, ki jo pritisne uporabnik. To mesto je nestanovitno: vrednost, ki je na njem shranjena, se spremeni vsakokrat, ko uporabnik pritisne tipko. Kot bomo videli v drugem delu učbenika, se takšne reči zgodijo kot posledica prekinitev, ki niso časovno usklajene z glavnim programom in prevajalnik zanje ne more vedeti.

Iz glavnega programa lahko na primer beremo kode pritisnjenih tipk takole:

```
while (Medpomnilnik ni poln) {
  Čakaj na podatek;
  medpomnilnik[i] = *p;
  if (medpomnilnik[i++] == '\n') break;
}
```

Prevajalnik lahko ugotovi, da se nikjer v gornji zanki while niti kazalec p niti podatek, na katerega kaže kazalec p, ne spreminjata. Zato optimizira kodo tako, da podatek, na katerega kaže kazalec p, prebere samo enkrat in ga shrani v poseben register:

```
Shrani podatek, na katerega kaže p, v register;
while (Medpomnilnik ni poln) {
  Čakaj na podatek;
  medpomnilnik[i] = Vrednost, shranjena v registru;
  if (medpomnilnik[i++] == '\n') break;
}
```

Optimiziran program bo zdaj medpomnilnik napolnil z več kopijami enega in istega podatka, ki je shranjen v registru.

Težavo lahko preprečimo tako, da ob deklaraciji kazalca p uporabimo opredeljevalec volatile:

[3] Vrstnega reda posameznih bajtov v spremenljivki ne predpisuje jezik C, temveč je ta odvisen od sistema. Zapisu, ki ima nižje bajte zapisane na nižjih naslovih, pravimo zapis z ***manjšim koncem*** (angl. little-endian). Obstaja tudi zapis, kjer so na nižjih naslovih zapisani višji bajti, čemur pravimo zapis z ***večjim koncem*** (angl. big-endian).

```
volatile BYTE *p;
```

Če tak kazalec uporabimo za branje iz pomnilnika, potem bo program v vsakem primeru vedno znova prebral podatek neposredno z naslova, ki je shranjen v `p`. To bo storil tudi, če iz kode ni razvidno, da bi se lahko ta podatek spremenil.

9.4 Naloge

Naloga 9.3 Napišite funkcijo z imenom `prestejBite`, ki prešteje, koliko bitov v podatku tipa `unsigned short` ima vrednost ena.

Primer klica funkcije:

```
printf("%d", prestejBite(0x06F1));  /* Izpiše: 7  */
```

Naloga 9.4 Podana je naslednja definicija funkcije:

```
unsigned short f(unsigned short x, int m, int n) {
  return (x >> m) & ~(~0 << (n - m + 1));
}
```

Kaj naredi ta funkcija? Utemeljite odgovor.

Naloga 9.5 Napišite funkcijo z imenom `zasukaj`, ki zasuka (angl. rotate) bite v podatku tipa `unsigned short` za določeno število mest v levo ali desno. Zasuk bitov je operacija, ki je podobna pomiku bitov. Od pomika se razlikuje po tem, da bite, ki na eni strani izpadejo iz podatka, na drugi strani znova vstavi vanj.

Primer klica funkcije:

```
#define LEVO -1
#define DESNO 1
//...
unsigned short x = zasukaj(0xF824, 2, LEVO);
printf("%X", x);  /* Izpiše: E093  */
x = zasukaj(0x1A6D, 4, DESNO);
printf("%X", x);  /* Izpiše: D1A6  */
```

Naloga 9.6 Napišite program, ki s tipkovnice prebere najprej 16-bitno dvojiško vrednost, potem pa še informacijo o tem, ali je vrednost predznačena ali nepredznačena. Program naj vneseno vrednost izpiše v desetiškem zapisu. Predznačene vrednosti, ki so negativne, naj bodo zapisane v dvojiškem komplementu. Če uporabnik vnese manj kot 16 bitov, naj program predpostavi, da je spredaj ustrezno število ničel.

Primer delovanja programa:

```
Vpiši 16-bitno dvojiško vrednost (0 konča): 111010100
Predznačeno/nepredznačeno (p/n): p
Pretvorjeno v desetiško vrednost: 468
Vpiši 16-bitno dvojiško vrednost (0 konča): 1111111110000101
Predznačeno/nepredznačeno (p/n): n
Pretvorjeno v desetiško vrednost: 65413
Vpiši 16-bitno dvojiško vrednost (0 konča): 1111111110000101
Predznačeno/nepredznačeno (p/n): p
Pretvorjeno v desetiško vrednost: -123
```

```
Vpiši 16-bitno dvojiško vrednost (0 konča): 0
Nasvidenje!
```

Naloga 9.7 V računalniku so barve pogosto shranjene kot kombinacije treh številk, ki predstavljajo jakost rdeče, zelene in modre komponente svetlobe. Napišite funkcijo `kodirajBarvo`, ki sprejme tri parametre z vrednostmi med nič in 255 ter zapiše sprejete vrednosti v en sam podatek tipa `unsigned int`. Napišite še funkcije `rdeca`, `zelena` in `modra`, ki iz tako zapisane barve izluščijo vrednost ustrezne barvne komponente.

Primer klicev funkcij:

```
unsigned int barva;
barva = kodirajBarvo(12, 55, 255);
printf("%d", rdeca(barva));   /* Izpiše: 12   */
printf("%d", zelena(barva));  /* Izpiše: 55   */
printf("%d", modra(barva));   /* Izpiše: 255  */
```

Opomba: Nalogo rešite najprej z uporabo polj bitov v strukturah, potem pa še brez uporabe struktur (tj. z bitnimi operatorji).

Naloga 9.8 ***Dvojiško kodiranje desetiških števil*** (angl. binary-coded decimal, BCD) je način dvojiškega zapisa, kjer je vsaka desetiška števka zapisana s fiksnim številom bitov. Na primer, če uporabimo za vsako števko štiri bite, potem je zapis BCD desetiške vrednosti 74 enak 01110100. Števka sedem je namreč zapisana kot 0111, števka štiri pa kot 0100.

Napišite funkcijo `ushort2BCD`, ki kot parameter sprejme celoštevilsko vrednost med nič in 9999. Funkcija naj vrne isto vrednost, zapisano v zapisu BCD s štirimi biti za vsako števko.

Primer klica funkcije:

```
unsigned short bcd, des = 6149;
bcd = ushort2BCD(des);
printf("%X", bcd);  /* Izpiše: 6149  */
```

Naloga 9.9 Šifriranje z izključnim ALI (angl. XOR encryption) deluje tako, da med bajti originalnega sporočila in šifrirnim ključem izvede operacijo izključni ALI. Označimo i-ti bajt originalnega sporočila z $b_{orig,i}$ in i-ti bajt ključa s k_i. Potem dobimo i-ti bajt šifriranega sporočila $b_{šif,i}$ takole:

$$b_{šif,i} = b_{orig,i} \hat{} k_i.$$

Pri dešifriranju sporočila se opremo na pomembno lastnost operacije izključni ALI, po kateri dvakratna operacija z enakim operandom ohranja originalno vrednost:

$$a \hat{} b \hat{} b = a.$$

Tako lahko vsak bajt sporočila dešifriramo takole:

$$b_{orig,i} = b_{šif,i} \hat{} k_i.$$

Če je ključ izbran ***resnično naključno*** (angl. truly random) in je vsaj tako dolg kot sporočilo, ki ga šifriramo, potem je takšno šifro teoretično nemogoče razbiti. Seveda ob predpostavki, da si pošiljatelj in prejemnik predhodno izmenjata ključ in ga uporabita ***samo enkrat*** (angl. one-time pad).

Napišite program, ki z vhoda prebere kratko sporočilo in ključ enake dolžine kot sporočilo. Sporočilo naj šifrira po postopku šifriranja z operacijo izključni ALI. Za branje sporočila uporabite standardno funkcijo `gets`, ki smo jo omenili na strani 108.

Primer delovanja programa:

```
Vnesi sporočilo:
Samo brez panike!
Vnesi ključ:
09 45 92 AC 3A 7A 11 C6 21 6B E3 94 52 BA 99 82 86
Šifrirano sporočilo:
5A 24 FF C3 1A 18 63 A3 5B 4B 93 F5 3C D3 F2 E7 A7
```

Pomoč: Zaporedje resnično naključnih števil lahko dobite na primer na spletni strani random.org, ki med drugim ponuja brezplačno storitev ustvarjanja naključnih vrednosti iz atmosferskega šuma.

Napišite še program, ki dešifrira vneseno sporočilo:

```
Vnesi sporočilo:
5A 24 FF C3 1A 18 63 A3 5B 4B 93 F5 3C D3 F2 E7 A7
Vnesi ključ:
09 45 92 AC 3A 7A 11 C6 21 6B E3 94 52 BA 99 82 86
Dešifrirano sporočilo:
Samo brez panike!
```

A

TABELA CEJEVSKIH OPERATORJEV

Naslednja tabela vsebuje vse operatorje, ki smo jih obravnavali v tem dokumentu. Operatorji, ki so višje v tabeli, imajo prednost pred operatorji, ki so nižje v tabeli (tj. operatorji z nižjo številko imajo prednost pred tistimi z višjo številko). Operatorji, ki so v isti vrstici tabele, imajo enako prednost in se izvajajo z leve proti desni (leva asociativnost) oziroma z desne proti levi (desna asociativnost).

Prednost	Operator(ji)	Opis	Asociativnost
1	`[]`	indeks elementa v tabeli	leva
	`()`	klic funkcije	leva
	`. ->`	član strukture	leva
	`++ --`	povečanje in zmanjšanje po vračanju vrednosti (angl. postfix)	leva
2	`++ --`	povečanje in zmanjšanje pred vračanjem vrednosti (angl. prefix)	desna
	`&`	naslov	desna
	`*`	posredovanje (angl. indirection)	desna

	`+ -`	unarni plus in minus (predznak)	desna
	`~`	eniški komplement	desna
	`!`	logična negacija	desna
	`sizeof`	velikost podatka	desna
3	`()`	zahtevana pretvorba tipa	desna
4	`* / %`	množenje in deljenje	leva
5	`+ -`	seštevanje in odštevanje	leva
6	`<< >>`	pomik bitov	leva
7	`> < >= <=`	relacija	leva
8	`== !=`	enakost	leva
9	`&`	bitni IN	leva
10	`^`	bitni izključni ALI	leva
11	`|`	bitni ALI	leva
12	`&&`	logični IN	leva
13	`||`	logični ALI	leva
14	`?:`	pogojni	desna
15	`= += -= *= /= %= <<= >>= &= ^= |=`	prirejanje	desna
16	`,`	vejica	leva

B

TABELA ZNAKOV ASCII

0 (NUL)	16 (DLE)	32 (SP)	48 0
1 (SOH)	17 (DC1)	33 !	49 1
2 (STX)	18 (DC2)	34 "	50 2
3 (ETX)	19 (DC3)	35 #	51 3
4 (EOT)	20 (DC4)	36 $	52 4
5 (ENQ)	21 (NAK)	37 %	53 5
6 (ACK)	22 (SYN)	38 &	54 6
7 \a (BEL)	23 (ETB)	39 '	55 7
8 \b (BS)	24 (CAN)	40 (	56 8
9 \t (HT)	25 (EM)	41)	57 9
10 \n (LF)	26 (SUB)	42 *	58 :
11 \v (VT)	27 (ESC)	43 +	59 ;
12 \f (FF)	28 (FS)	44 ,	60 <
13 \r (CR)	29 (GS)	45 -	61 =
14 (SO)	30 (RS)	46 .	62 >
15 (SI)	31 (US)	47 /	63 ?

64 @	80 P	96 `	112 p
65 A	81 Q	97 a	113 q
66 B	82 R	98 b	114 r
67 C	83 S	99 c	115 s
68 D	84 T	100 d	116 t
69 E	85 U	101 e	117 u
70 F	86 V	102 f	118 v
71 G	87 W	103 g	119 w
72 H	88 X	104 h	120 x
73 I	89 Y	105 i	121 y
74 J	90 Z	106 j	122 z
75 K	91 [	107 k	123 {
76 L	92 \	108 l	124 |
77 M	93]	109 m	125 }
78 N	94 ^	110 n	126 ~
79 O	95 _	111 o	127 (DEL)

C
TEŽAVE S PRENOSLJIVOSTJO

Čeprav je prenosljivost kode ena močnejših odlik jezika C, pa standard pušča kar nekaj primerov, ko obnašanje kode na tak ali drugačen način ni določeno. Zaradi tega moramo biti previdni, kadar želimo že delujočo in preizkušeno kodo kasneje prevesti s kakšnim drugim prevajalnikom. Pri tem standard opredeljuje tri različne kategorije:

- ***Nepredpisano obnašanje*** (angl. unspecified behavior) V tem primeru gre za obnašanje, za katero standard predvideva dve ali več možnosti, nič pa ne govori o tem, katero od predpisanih možnosti je treba upoštevati v določenih okoliščinah. Primer nepredpisanega obnašanje je vrstni red, v katerem se računajo vrednosti argumentov funkcij.
- ***Nedoločeno obnašanje*** (angl. undefined behavior) Do takšnega obnašanja pride v primeru napačne uporabe elementov jezika, pri čemer standard ne postavlja popolnoma nobenih zahtev, kako naj se sistem odzove na napako. Primer nedoločenega obnašanja je prekoračitev območja vrednosti pri računanju s predznačenimi celimi števili.
- ***Obnašanje, odvisno od izvedbe prevajalnika*** (angl. implementation defined behavior) V tem primeru gre za obnašanje, ki ga ne določa standard, temveč ga določa posamezna izvedba prevajalnika. Način obnašanja mora biti tudi jasno zapisan v dokumentaciji prevajalnika. Primer takšnega obnašanja je vrednost bita MSB pri pomikanju predznačenega števila v desno.

Ta dodatek predstavlja povzetek primerov, ki jih obravnavamo v tem učbeniku. Popoln seznam lahko najdete v dodatku G izvornega besedila standarda C89 [2] oziroma v dodatku J izvornega besedila standarda C99 [3].

C.1 Nepredpisano obnašanje

Obnašanje kode je nepredpisano v naslednjih okoliščinah:

- Dobesedni navedbi dveh enakih znakovnih nizov sta v pomnilniku predstavljeni bodisi z enim in istim nizom bodisi z dvema enakima nizoma (stran 105).
- Vrstni red, v katerem se računajo vrednosti posameznih izrazov, ki so del večjega izraza, kadar vrstni red računanja ni posebej določen (kot na primer pri kratkostičnem računanju). Na primer, pri računanju vsote dveh ali več izrazov (stran 20).
- Vrstni red, v katerem se računajo argumenti funkcije (stran 78).
- Vrstni red dveh enakih elementov v tabeli, ki jo urejamo po velikosti s funkcijo `qsort` (stran 138).

C.2 Nedoločeno obnašanje

Obnašanje kode je nedoločeno v naslednjih okoliščinah:

- Uporaba kazalca na objekt, ki je v pomnilniku prenehal obstajati (stran 87).
- Prekoračitev območja vrednosti pri računanju s predznačenimi celimi števili (stran 49).
- Izid pretvorbe realne vrednosti iz zapisa z večjo natančnostjo v zapis z manjšo natančnostjo, če vrednost presega območje, ki ga je mogoče zapisati (stran 62).
- Izid pretvorbe realne vrednosti v celoštevilsko vrednost, če celi del realne vrednosti presega območje, ki ga je mogoče zapisati (stran 62).
- Program poskuša spremeniti znakovni niz, ki je v programu zapisan z dobesedno navedbo (stran 105).
- Operand operatorja posredovanja (`*`) ima neveljavno vrednost ali vrednost NULL (stran 67).
- Vrednost desnega operanda operatorjev deljenja (`/`) ali ostanka pri deljenju (`%`) je nič (stran 16).
- Odštevanje dveh kazalcev, ki ne kažeta na isto tabelo (oziroma vsaj tik za koncem iste tabele) (stran 100).
- Indeks tabele ima vrednost zunaj območja (stran 96). To velja celo v primeru, ko je element tabele na videz dostopen (na primer, uporaba izraza `a[1][7]` ob deklaraciji `int a[4][5]`).
- Pomik podatka za negativno število bitov ali za število bitov, ki je večje ali enako bitni dolžini končne vrednosti izraza (stran 150).
- Klic funkcije, katere vrnjena vrednost se uporablja na mestu njenega klica, se konča brez stavka `return` (stran 79).

- Število argumentov funkcije se ne ujema s številom njenih parametrov ali pa je podatkovni tip kakšnega argumenta nezdružljiv s podatkovnim tipom ustreznega parametra (stran 78).
- Kazalec, uporabljen kot argument funkcije `free` ali `realloc`, je različen od kazalca, ki ga je vrnil predhodni klic funkcije za dodeljevanje pomnilnika (tj. klic funkcije `malloc` ali `realloc`) (strani 127 in 131).
- Funkcija za primerjavo elementov, ki jo uporablja funkcija `qsort`, spremeni vrednost elementov tabele, ki jo funkcija `qsort` ureja (stran 137).

C.3 Obnašanje, odvisno od izvedbe prevajalnika

V naslednjih primerih je obnašanje kode odvisno od izvedbe prevajalnika:

- Vrednosti, s katerimi so kodirani posamezni znaki (stran 58).
- Način, na katerega so zapisana predznačena cela števila (eniški komplement, dvojiški komplement ali predznak in absolutna vrednost) (stran 47).
- Izid pretvorbe celega števila v predznačeno celo število, kadar vrednosti ni mogoče zapisati v tem podatkovnem tipu (stran 62).
- Izidi nekaterih bitnih operacij nad predznačenimi celimi števili (stran 149).
- Rezultat operacije deljenja (`/`) in ostanka pri deljenju (`%`) celih števil, kadar je vsaj en od operandov negativno število (samo standard C89, stran 16).
- Način zaokroževanja vrednosti pri pretvorbi v zapis s plavajočo vejico, v katerem ne moremo natančno zapisati izvorne vrednosti (strani 53 in 63).

PRAZNA STRAN

II. DEL

PROGRAMIRANJE MIKROKRMILNIKOV

PRAZNA STRAN

10. POGLAVJE

STROJNA IN PROGRAMSKA OPREMA

10.1 Mikrokrmilnik

Mikrokrmilnik (angl. microcontroller) je majhen računalnik na enem samem čipu. Poleg ene ali več ***osrednjih računskih enot*** (angl. central processing unit, CPU) vsebuje mikrokrmilnik še pomnilnik in programljive vhodno-izhodne enote, ki služijo komunikaciji z okolico. Za razliko od mikroprocesorjev (angl. microprocessor), ki se uporabljajo v osebnih računalnikih in ostalih splošnonamenskih napravah, služijo mikrokrmilniki gradnji vgrajenih sistemov, ki opravljajo točno določene funkcije znotraj večjih elektronskih ali mehanskih sistemov.

10.2 Razvojni učni sistem

Za programiranje mikrokrmilnikov potrebujemo najprej določeno strojno, predvsem pa programsko opremo, ki je lahko za začetnika sorazmerno zahtevna za uporabo. Zaradi časovnih omejitev, ki nam ne dopuščajo spoznavanja takšne opreme, bomo pri našem delu uporabili razvojni učni sistem Arduino. Programska oprema tega sistema je zgrajena tako, da uporabniku skrije precej podrobnosti, s čimer lahko tudi začetniki brez kakršnihkoli izkušenj zelo hitro začnejo z delom. Te skrite podrobnosti pa na srečo niso nedostopne, zato bomo marsikatero od njih razkrili.

Arduino je podjetje, ki načrtuje in izdeluje odprtokodno strojno in programsko opremo. Njihovi izdelki predstavljajo mikrokrmilnike na matičnih ploščah (angl. single-board mi-

crocontrollers) in mikrokrmilniške komplete za enostavno gradnjo digitalnih sistemov. Na spletni strani podjetja (arduino.cc) je mogoče dobiti tudi načrte in datoteke za izdelavo tiskanih vezij za nekatere od njihovih izdelkov.

Večina Arduinovih plošč je zgrajenih na osnovi Atmelovih osembitnih mikrokrmilnikov AVR, leta 2012 pa je podjetje predstavilo 32-bitni Arduino Due, zgrajen z Atmelovim mikrokrmilnikom SAM3X8E, ki ga bomo uporabljali za večino primerov v tem učbeniku. Arduinovi sistemi so opremljeni z zagonskim nalagalnikom (angl. boot loader), ki omogoča enostavno nalaganje programov z osebnega računalnika v bliskovni pomnilnik mikrokrmilnika (angl. on-chip flash memory) preko zaporedne povezave USB.

Strojno opremo Arduino lahko programiramo v kateremkoli jeziku s prevajalniki, ki prevedejo programe v strojno kodo za ustrezen ciljni krmilnik. Podjetje Microchip ponuja razvojno okolje Atmel Studio za njihove osem- in 32-bitne krmilnike. Obstaja tudi Arduinovo integrirano razvojno okolje (angl. integrated development environment, IDE) za sisteme Windows, macOS in Linux, v katerem lahko pišemo programe v jezikih C in C++. Uporaba tega okolja je zelo enostavna, od uporabnika pa zahteva, da sledi posebnim pravilom pisanja kode. Koda, ki jo napiše uporabnik, je sestavljena iz dveh delov: prvi del služi zagonskim nastavitvam, drugi del pa je programska zanka.

10.3 Skica

Program, ki ga napišemo v Arduinovem integriranem razvojnem okolju, se imenuje ***skica*** (angl. sketch). Skice se na računalniku shranjujejo kot besedilne datoteke s končnico .ino. Najmanjši cejevski program v okolju Arduino je sestavljen iz dveh funkcij:

- `setup` Ta funkcija se kliče le enkrat ob zagonu ali vnovičnem zagonu sistema. Uporablja se za nastavitev začetnih vrednosti spremenljivk, načinov komunikacije z zunanjimi napravami ter inicializaciji programskih objektov, ki jih bomo potrebovali v skici.
- `loop` Ko se funkcija `setup` konča, se začne izvajati funkcija `loop`, ki se kliče znova in znova, dokler sistema ne izklopimo ali znova zaženemo.

Med prevajanjem se koda iz skice poveže s funkcijo `main`, ki se projektu doda avtomatično. Funkcijo najdemo v datoteki okolja Arduino z imenom main.c, ki je nikoli ne spreminjamo. Datoteka main.c vsebuje takšno definicijo funkcije `main`:

```
int main(void) {
  /* Nekaj sistemske inicializacije.        */
  setup();
  for (;;) {  /* Neskončna zanka.           */
    loop();
    /* Preverjanje sistemske komunikacije.  */
  }
  return 0;
}
```

Na mestih prve in tretje opombe je v originalni kodi še nekaj klicev funkcij, ki skrbijo za delovanje sistema in smo jih zaradi preglednosti izpustili iz prikazane kode.

Ker je funkcija `main` edina funkcija, ki se izvaja na sistemu, je ne smemo nikoli zapustiti. Takšnemu načinu izvajanja programa, ki se ponavlja v neskončni zanki, pravimo ***ciklično izvajanje*** (angl. cyclic executive), uporablja pa se kot nadomestek operacijskega sistema.

Utripanje svetleče diode

Večina plošč Arduino vsebuje svetlečo diodo (angl. light-emitting diode, LED), ki je preko omejevalnega upora priključena na sponko (angl. pin) 13. Ta dioda je prikladna za izvajanje različnih preizkusov, pogosto pa se uporablja tudi pri najosnovnejšem prikazu delovanja sistema. Začetniki delo s sistemom Arduino najpogosteje začnejo s programom, ki povzroči utripanje (angl. blink) svetleče diode. Program *blink* v tem primeru nadomešča znani program *Hello, Word!*, ki je za marsikaterega začetnika navadno prvi program na sistemih z besedilnim prikazovalnikom.

Takole je videti program, napisan v okolju Arduino:

```
#define LED_PIN 13  /* Številka sponke, na katero je priključena
                       svetleča dioda.                          */
void setup(void) {
  pinMode(LED_PIN, OUTPUT);  /* Sponka 13 naj bo digitalni izhod.  */
}

void loop(void) {
  digitalWrite(LED_PIN, HIGH);  /* Vklopi svetlečo diodo.          */
  delay(1000);                  /* Počakaj eno sekundo (1000 ms).  */
  digitalWrite(LED_PIN, LOW);   /* Izklopi svetlečo diodo.         */
  delay(1000);                  /* Počakaj eno sekundo (1000 ms).  */
}
```

Identifikatorji `OUTPUT`, `HIGH` in `LOW` so makri, določeni v sistemski zglavni datoteki <Arduino.h>, ki pa je ni treba vključiti z direktivo `#include`, saj okolje Arduino vključi svoje sistemske datoteke avtomatično. Program ta hip ne potrebuje dodatne razlage, k njemu pa se bomo še vrnili, ko bomo natančneje spoznali pojem splošno namenskih vhodno-izhodnih sponk.

Naloga 10.1 Če imate pri roki svoj sistem Arduino, potem poskusite nanj naložiti in zagnati gornji program, ki povzroči utripanje diode LED. Če te možnosti nimate, poiščite na spletu kakšen emulator, kakršen je na voljo na primer na spletni strani tinkercad.com.

PRAZNA STRAN

11. POGLAVJE

SPLOŠNO NAMENSKE VHODNO-IZHODNE SPONKE

Vsak računalniški sistem mora komunicirati z okolico. Verjetno najpomembnejši in najpogosteje uporabljen način komunikacije je z uporabo tako imenovanih ***splošno namenskih vhodno-izhodnih sponk*** (angl. general-purpose input/output, GPIO). Splošno namenske sponke so namenjene prenosu digitalne informacije (tj. bodisi logične ničle bodisi logične enke) in nimajo nobenega vnaprej določenega namena. Njihovo obnašanje – vključno z izbiro, ali delujejo kot vhod ali izhod – določa program v času izvajanja.

Sodobni mikrokrmilniki ponujajo veliko različnih načinov, na katere lahko nastavimo splošno namenske sponke. Pri tem je pomembno, da ločimo naslednje tri pojme:

- ***Plavajoča sponka*** (angl. floating pin) Če nastavimo sponko v ***stanje visoke impedance***[1] (angl. high-impedance state, High-Z), potem takšna sponka teoretično ni priklopljena nikamor in njen električni potencial ni določen. Pravimo, da sponka ***plava***. Pomembno je, da takšna sponka zunanjega vezja, ki ga priklopimo nanjo, tokovno praktično ne obremenjuje. Zato lahko sponko, ki je v stanju visoke impedance, uporabimo kot ***vhodno sponko*** mikrokrmilnika: njen potencial je določen z zunanjim vezjem.
- ***Logična ničla*** Sponko lahko nastavimo tako, da jo povežemo z ozemljitvijo, s čimer sponka postane ***tokovni ponor*** (angl. current sink). Takšna sponka deluje kot ***izhodna***

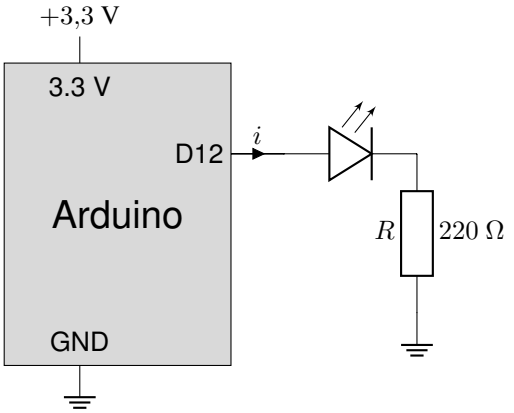
[1] Impedanca plavajoče sponke je lahko od nekaj 100 kΩ pa do 100 MΩ in več. Sicer pa v mikrokrmilniških priročnikih tega podatka običajno ne boste našli v obliki impedance, temveč se podaja kot ***tok puščanja*** (angl. leakage current).

sponka mikrokrmilnika: če jo priključimo na zunanje vezje, jo vezje čuti kot povezavo na ozemljitev.

- ***Logična enka*** Sponko lahko nastavimo tako, da jo povežemo z napajalno napetostjo, s čimer sponka postane ***napetostni vir*** (angl. voltage source). Tudi takšna sponka deluje kot ***izhodna sponka*** mikrokrmilnika: če jo priključimo na zunanje vezje, jo vezje čuti kot povezavo na napajalno napetost.

11.1 Utripanje svetleče diode, drugič

Vrnimo se k primeru z utripajočo svetlečo diodo, le da tokrat ne uporabimo vgrajene diode, temveč priključimo svojo diodo na digitalno sponko D12, kakor kaže naslednja slika:



Utripanje diode dosežemo tako, da postavljamo sponko D12 izmenično v stanje logične ničle (takrat je dioda ugasnjena) in logične enke (takrat je dioda prižgana). Ko je dioda prižgana, iz sponke teče tok i, ki pa ne sme biti večji od največjega dopustnega izhodnega toka za to sponko. Iz tabele v dodatku D je razvidno, da je največji dopustni izhodni tok za sponko D12 enak 15 mA. Izhodni tok omejimo z zaščitnim uporom R, na katerem je padec napetosti enak napajalni napetosti, zmanjšani za ***prevodno napetost*** (angl. forward voltage) svetleče diode. Slednja je običajno okrog 1,8 V (za rdečo diodo LED), zato je padec na uporu R enak 1,5 V. Iz tega izračunamo izhodni tok iz sponke $i = 1{,}5\ \mathrm{V}/220\ \Omega \approx 6{,}82$ mA. Z uporom smo tok omejili precej pod največjo dovoljeno vrednost. Eden izmed razlogov, zakaj smo to naredili, je ta, da je omejena tudi vsota vseh tokov, ki lahko hkrati tečejo skozi vse priključne sponke mikrokrmilnika. Za krmilnik SAM3X, ki ga uporablja sistem Due, znaša ta omejitev 130 mA. To pomeni, da lahko z največjim izhodnim tokom 15 mA brez skrbi krmilimo zgolj osem svetlečih diod.

Na gornjem vezju lahko zdaj povzročimo utripanje svetleče diode s programom na strani 175, pri čemer moramo popraviti makro v prvi vrstici kode, ki določa sponko, na kateri je priključena dioda:

```
#define LED_PIN 12
```

V programu potem v funkciji `setup` kličemo funkcijo `pinMode`:

```
pinMode(LED_PIN, OUTPUT);
```

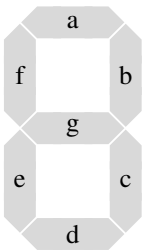
S tem digitalno sponko 12 postavimo v stanje nizke impedance, kar pomeni, da lahko vanjo ali iz nje teče tok, potreben za krmiljenje zunanjega vezja. V funkciji `loop` nato s klicema funkcije `digitalWrite` na sponko 12 izmenično pišemo logično enico (`HIGH`) in logično ničlo (`LOW`). S tem zunanje vezje z diodo izmenično priklapljamo na napajalno napetost in na ozemljitev. Vsako od obeh stanj zadržimo za 1000 ms, kar dosežemo s klicema funkcije `delay`:

```
digitalWrite(LED_PIN, HIGH);
delay(1000);
digitalWrite(LED_PIN, LOW);
delay(1000);
```

Princip, ki smo ga spoznali v tem razdelku, lahko uporabimo za krmiljenje sedemsegmentnega prikazovalnika, ki ni nič drugega kot sedem neodvisnih diod LED.

11.2 Sedemsegmentni prikazovalnik

Sedemsegmentni prikazovalnik (angl. seven-segment display) je prikazovalnik, namenjen enostavnemu prikazovanju desetiških številk. Sestavljen je iz sedmih svetlečih diod v obliki razpotegnjenih šestkotnikov, kakor prikazuje naslednja slika:

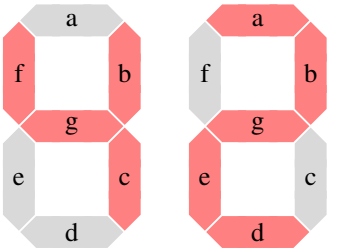


S prižiganjem ustreznih svetlečih diod lahko na prikazovalniku prikažemo različne simbole, najpogosteje pa ga uporabljamo za prikazovanje desetiških števk. Naslednja tabela prikazuje, katere od sedmih svetlečih diod moramo prižgati za prikaz določene števke (enka predstavlja prižgano, ničla pa ugasnjeno diodo):

Števka	**Stanje svetlečih diod sedemsegmentnega prikazovalnika**						
	g	**f**	**e**	**d**	**c**	**b**	**a**
0	0	1	1	1	1	1	1
1	0	0	0	0	1	1	0
2	1	0	1	1	0	1	1
3	1	0	0	1	1	1	1
4	1	1	0	0	1	1	0
5	1	1	0	1	1	0	1
6	1	1	1	1	1	0	1
7	0	0	0	0	1	1	1

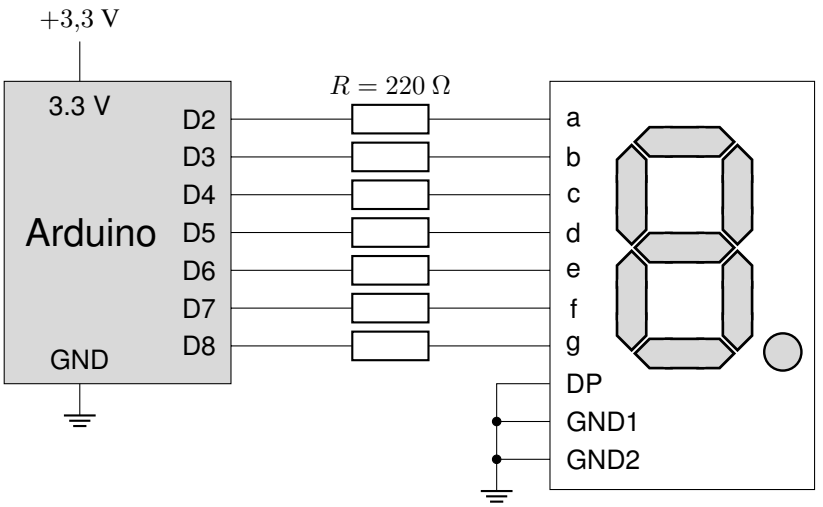
8	1	1	1	1	1	1	1
9	1	1	0	1	1	1	1

Na primer, štirico prikažemo tako, da prižgemo diode b, c, f in g, dvojko pa tako, da prižgemo vse diode razen diod c in f:



Sedemsegmentni prikazovalnik priklopimo na sistem Arduino tako, kot bi priklopili sedem neodvisnih diod LED. Vsako od diod priklopimo na svojo izhodno digitalno sponko, pri čemer ne smemo pozabiti na zaščitne upore.

Ohišje sedemsegmentnega prikazovalnika ima običajno deset sponk: sedem za posamezne segmente za prikazovanje števke, eno za decimalno piko in dva za skupno katodo (oz. anodo, odvisno od izvedbe prikazovalnika[2]). Ker decimalne pike ne bomo uporabljali, priklopimo sponko DP (decimal point, slov. decimalna pika) na ozemljitev, kamor priklopimo tudi obe sponki skupne katode (GND1 in GND2). Naslednja slika prikazuje celotno vezje:



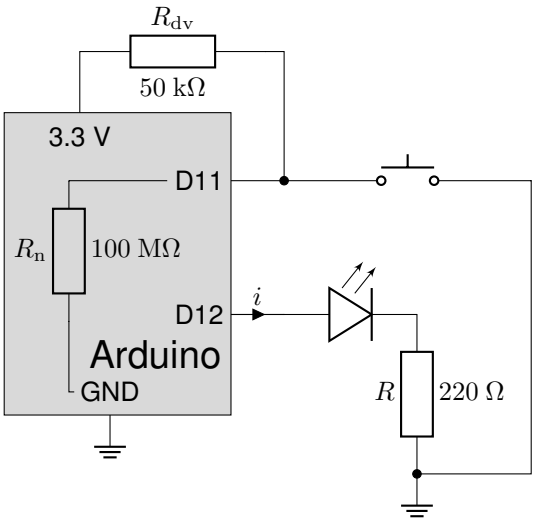
[2]Obstajata dve izvedbi sedemsegmentnega prikazovalnika: s ***skupno anodo*** (angl. common anode) in ***skupno katodo*** (angl. common cathode). V prvi različici so anode (tj. pozitivne sponke) vseh diod povezane skupaj in speljane na skupno sponko na ohišju prikazovalnika. V drugi različici so povezane skupaj katode (tj. negativne sponke) vseh diod.

Naloga 11.1 Za vajo napišite program, ki bo štel po modulu deset (tj. $0, 1, 2, \cdots, 8, 9, 0, 1, 2, \cdots$), pri čemer naj številke izpisuje na sedemsegmentnem prikazovalniku. Pri pisanju programa upoštevajte vezavo, ki je prikazana na gornji sliki.

V zadnjem primeru smo sedemsegmentni prikazovalnik priključili neposredno na sponke mikrokrmilnika. Slabost takšnega načina krmiljenja je, da zanj porabimo sedem sponk, ki nam jih lahko zaradi tega zmanjka za ostale komponente sistema, ki ga gradimo. Poleg tega moramo napisati dodatno kodo za pretvarjanje posameznih števk v sedemsegmentne kombinacije bitov, kakor jih prikazuje tabela na strani 179. Namesto tega lahko uporabimo poseben čip, ki preslika zapis BCD (glej nalogo 9.8 na strani 161) v sedemsegmentno kombinacijo bitov (angl. BCD to seven-segment decoder). Z uporabo takšnega čipa porabimo na mikrokrmilniku le štiri sponke, na katere pišemo števke neposredno v štiribitnem zapisu BCD.

11.3 Branje tipke

Dodajmo vezju na strani 178 še tipko, tako da bomo lahko napisali program, ki bo ugasnil svetlečo diodo, če bo tipka pritisnjena. Takole je videti vezje:



Tipko smo vezali na digitalno sponko 11, ki jo bomo nastavili kot vhodno sponko. Ker je vhodna sponka v stanju zelo visoke impedance[3], se zaradi električnega šuma iz okolice napetost na njej na videz naključno spreminja, kadar sponka ni nikamor priključena. Zaradi visoke impedance je namreč treba zelo malo toka, da se stanje sponke spremeni iz logične enice v logično ničlo in obratno. Temu pravimo, da sponka »plava«. Kadar je tipka pritisnjena, je sponka D11 priključena na ozemljitev in krmilnik prebere logično ničlo. Ko pa je tipka spuščena, bi sponka D11 zaplavala, če ne bi bilo upora R_{dv}. Temu uporu pravimo ***dvižni upor*** (angl. pull-up resistor), njegova naloga pa je, da »povleče« potencial na sponki proti napajalni napetosti, kadar je tipka spuščena. Upora R_{dv} in R_{n} tvorita napetostni

[3]Notranja impedanca sponke D11 je na sliki prikazana z uporom R_{n}, ki je vezan med sponko D11 in ozemljitvijo.

delilnik in padec napetosti na uporu R_{dv} je zgolj približno pol promila napajalne napetosti. Ta padec je seveda povsem zanemarljiv in krmilnik v primeru spuščene tipke na sponki D11 zanesljivo prebere logično enico.

Takole je videti program, ki povzroči, da je svetleča dioda ugasnjena, ko je tipka pritisnjena, in obratno – da je svetleča dioda prižgana, ko je tipka spuščena:

```
#define TIPKA_PIN 11
#define LED_PIN 12

void setup(void) {
  pinMode(TIPKA_PIN, INPUT);
  pinMode(LED_PIN, OUTPUT);
}

void loop(void) {
  digitalWrite(LED_PIN, digitalRead(TIPKA_PIN));
}
```

V funkciji `setup` smo dodali klic funkcije `pinMode` z argumentom `INPUT` (makro, določen v datoteki <Arduino.h>). S tem postavimo sponko, na kateri je priključena tipka, v stanje visoke impedance. V funkciji `loop` potem kličemo funkcijo `digitalRead`, ki vrne logično stanje (`LOW` oz. `HIGH`) na sponki, katere zaporedno številko podamo kot argument te funkcije. Vrednost, ki jo funkcija `digitalRead` vrne, uporabimo neposredno kot drugi argument funkcije `digitalWrite`, s čimer stanje takoj prikažemo na diodi LED.

Naloga 11.2 Z uporabo dvižnega upora na vhodni sponki dosežemo tako imenovano ***negativno logiko*** (angl. negative logic): logično ničlo dobimo, ko je tipka pritisnjena, logično enko pa, ko je tipka spuščena. Z uporabo ***poteznega upora*** (angl. pull-down resistor) dosežemo na vhodni sponki ***pozitivno logiko*** (angl. positive logic): ko je tipka pritisnjena, preberemo logično enko, ko pa je tipka spuščena, preberemo logično ničlo. Za vajo narišite vezje, s katerim to dosežemo.

Mnogi mikrokrmilniki imajo vgrajen dvižni upor, ki ga je mogoče vklopiti programsko. Sistem Arduino ima določen tudi makro `INPUT_PULLUP`, s katerim lahko vklopimo dvižni upor[4]:

```
pinMode(TIPKA_PIN, INPUT_PULLUP);
```

Zdaj seveda upora R_{dv} v zadnjem vezju ne potrebujemo več.

Zaznavanje spremembe stanja

V zadnjem primeru nas je zanimalo zgolj stanje tipke. Iz tipke pa lahko izluščimo še več informacije, če beremo tudi ***spremembo*** njenega stanja. Naslednji program uporablja naše zadnje vezje brez dvižnega upora R_{dv}, njegova naloga pa je, da preklopi stanje diode vsakokrat, ko zazna pritisk tipke:

```
#define TIPKA_PIN 11
#define LED_PIN 12
```

[4] Vrednost vgrajenega dvižnega upora je odvisna od tipa krmilnika, giblje pa se od nekaj deset kΩ do 100 kΩ in več.

```
void setup(void) {
  pinMode(TIPKA_PIN, INPUT_PULLUP);
  pinMode(LED_PIN, OUTPUT);
}

void loop(void) {
  static int staro = HIGH;
  static int led = HIGH;
  int novo;
  digitalWrite(LED_PIN, led);
  novo = digitalRead(TIPKA_PIN);
  if (novo != staro) {
    staro = novo;
    if (novo == LOW) {
      if (led == LOW) {
        led = HIGH;
      }
      else {
        led = LOW;
      }
    }
  }
}
```

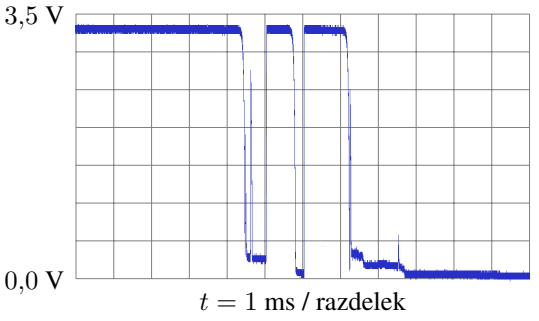
V programu smo uporabili dve statični lokalni spremenljivki, ki morata hraniti svoji vrednosti med posameznimi klici funkcije `loop`. Spremenljivka `staro` hrani stanje tipke, kakršno je bilo v času zadnjega klica funkcije `loop`. To je pomembno, da lahko ugotovimo, ali se je to stanje vmes spremenilo. Spremenljivka `led` hrani trenutno stanje svetleče diode. Tudi to je pomembno, da lahko njeno stanje spremenimo vsakokrat, ko zaznamo, da se je stanje na tipki spremenilo iz `HIGH` v `LOW`.

Program v vsakem obhodu zanke najprej osveži stanje svetleče diode, takoj zatem pa prebere stanje na vhodni sponki. Če se je to stanje od zadnjega branja spremenilo, si najprej zapomni novo stanje. Če pa se je stanje poleg tega spremenilo iz visokega v nizko (tj. novo stanje na vhodu je zdaj enako `LOW`), preklopi tudi stanje diode. Namreč, stanje svetleče diode želimo spremeniti le takrat, ko tipko pritisnemo. Ko tipko spustimo, se stanje diode ne sme spremeniti.

Ko program zaženemo, opazimo, da se ob vsakem pritisku tipke dioda prižiga in ugaša na videz naključno. V čem je težava?

Pri pritisku tipke se ne moremo izogniti pojavu, ki mu pravimo ***odskakovanje kontakta*** (angl. contact bounce). To je običajna težava, ki jo zaznavamo pri vseh mehanskih stikalih. Zaradi neidealnosti v izdelavi in zaradi prožnosti kovin, iz katerih so stikala izdelana, se ob sklenitvi (in razklenitvi) kontakta ta v resnici v kratkem času večkrat sklene in razklene, preden se dokončno ustvari stabilen stik.

Naslednja slika kaže posnetek osciloskopa ob prehodu signala na vhodu iz logične enke v logično ničlo, kjer opazimo izrazito odskakovanje:



Na sliki pojav izzveni po nekaj ms, vendar bomo za stikala različnih kakovosti izmerili zelo različne čase, tako krajše kot tudi daljše.

Obstaja več rešitev proti odskakovanju, od strojnih nizkopasovnih sit do programskih števcev oziroma zakasnitev. Najenostavnejša rešitev je ta, da takoj po tem, ko zaznamo prehod iz enega stanja v drugo, počakamo dovolj dolgo, da odskakovanje izzveni. Kako dolgo je treba počakati, je odvisno od kakovosti tipke, običajno pa je 20 ms dovolj.

Dodajmo v naš zadnji program tik pred koncem zunanjega stavka `if` klic funkcije `delay`, in težava z odskakovanjem bo izginila:

```
//...
if (novo != staro) {
  staro = novo;
  if (novo == LOW) {
    if (led == LOW) {
      led = HIGH;
    }
    else {
      led = LOW;
    }
  }
  delay(20);
}
```

Naloga 11.3 Za vajo napišite program, ki šteje, kolikokrat smo pritisnili tipko, in število pritiskov sproti izpisuje na Arduinov ***serijski vmesnik*** (angl. Serial Monitor).

Serijski vmesnik omogoča komunikacijo z osebnim računalnikom in je v Arduinovem okolju predstavljen s programskim objektom `Serial`. Osnovno delo z njim je preprosto: V funkciji `setup` najprej vzpostavimo komunikacijo s klicem:

```
Serial.begin(9600);
```

Argument postopka `begin` je hitrost komunikacije z vmesnikom v baudih (oz. bitih na sekundo, oznaka Bd), katere privzeta vrednost je 9600 Bd. Na vmesniku, ki ga odpremo v Arduinovem razvojnem okolju, lahko nastavimo tudi drugačne hitrosti. Po vzpostavljeni komunikaciji lahko na vmesnik pišemo poljubne podatke s postopkoma `print` ali `println`. Postopka se med seboj razlikujeta le po tem, da se pri postopku `println` na koncu izpisa kazalnik (angl. cursor) na vmesniku pomakne v

novo vrstico. Na primer, če je vrednost spremenljivke `stevec` enaka 42, potem bo koda:

```
Serial.print(stevec);
Serial.println("...");
stevec++;
Serial.print(stevec);
```

na vmesnik izpisala:

```
42...
43
```

Pisanje na serijski vmesnik je relativno počasno, zato program ne bo občutil odskakovanja tipke, če boste na vmesnik pisali prav vsako spremembo števca. Napišite program tako, da na vmesnik izpiše samo vrednosti števca, ki so deljive z deset.

Opomba: Serijski vmesnik je prikladno orodje za razhroščevanje, saj lahko nanj izpisujemo sled programa (angl. program trace).

11.4 Strojni registri

Strojni register (angl. hardware register) je del digitalnega vezja, ki ima podobne lastnosti kakor običajna pomnilniška lokacija:

- Iz registra lahko beremo ali vanj zapisujemo podatke.
- Vsak register ima določen naslov, ki nam omogoča dostop do njega, podobno kot ima naslov tudi vsaka običajna pomnilniška lokacija.

Pomembna razlika med strojnim registrom in običajno pomnilniško lokacijo je ta, da so podatki, zapisani v registru, običajno neposredno povezani z določenim fizikalnim stanjem sistema. Na primer, stanje določenega bita v izhodnem registru se odraža neposredno v napetosti na ustrezni digitalni izhodni sponki. Ali, napetost na določeni digitalni vhodni sponki se odraža neposredno v stanju ustreznega bita v vhodnem registru.

Druga pomembna razlika med registrom in običajno pomnilniško lokacijo je ta, da ima vsak register natančno določen pomnilniški naslov. Običajna pomnilniška lokacija, ki zgolj hrani podatek, nima vnaprej določenega naslova. Slednjega določi prevajalnik na podlagi podatkov o razpoložljivem pomnilniku.

Kadar imamo na voljo ustrezne uporabniške funkcije, nam ni treba delati neposredno z registri. Na primer funkcije `pinMode`, `digitalWrite` in `digitalRead` vse pišejo in berejo informacijo iz ustreznih registrov, za katere nam ni treba vedeti. Pogosto pa je poznavanje dela z registri le koristno.

Sito proti odskakovanju

Mikrokrmilnik SAM3X vsebuje posebno ***sito proti odskakovanju*** (angl. debounce filter), ki ga vklopimo in nastavimo tako, da zapišemo ustrezne podatke v tri različne registre, ki imajo vsi dolžino 32 bitov:

- ***Register, ki omogoči vhodno sito*** (angl. Input Filter Enable Register, IFER) V tem registru moramo postaviti bit, ki ustreza sponki, na kateri želimo vzpostaviti sito.

Preslikava med registri mikrokrmilnika SAM3X in sponkami sistema Arduino je nekoliko zapletena, vidimo pa jo lahko v tabeli v dodatku D. Sponke so razdeljene med štiri periferne enote (angl. peripherals) z oznakami A, B, C in D. Iz tabele vidimo, da je digitalna sponka 11, na katero smo priključili tipko, krmiljena preko bita številka sedem periferne enote D (tj. sponka D11 na sistemu Arduino Due ustreza sponki PD7 na krmilniku SAM3X).

- ***Register, ki izbere sito proti odskakovanju*** (angl. Debouncing Input Filter Select Register, DIFSR). Tudi v tem registru moramo postaviti bit, ki ustreza sponki, na kateri želimo vzpostaviti sito.
- ***Register, ki deli frekvenco počasne ure*** (angl. Slow Clock Divider Register, SCDR) Čas zakasnitve sita proti odskakovanju se meri s tako imenovano ***počasno uro*** (angl. Slow Clock), katere frekvenca je običajno 32 768 Hz. V register SCDR moramo vpisati vrednost DIV, ki jo izračunamo iz frekvence počasne ure in iz želene zakasnitve sita:

$$DIV = \frac{1}{2} f_{\mathrm{pu}} t_{\mathrm{z}} - 1.$$

Pri tem je f_{pu} frekvenca počasne ure in t_{z} želeni čas zakasnitve sita. Za čas zakasnitve 20 ms izračunamo, da je $DIV = 1/2 \times 32\,768\ \mathrm{Hz} \times 0{,}02\ \mathrm{s} - 1 = 326$. Ker lahko v register SCDR vpisujemo le celoštevilsko vrednost, je dobljena vrednost zaokrožena.

Našteti trije registri so določeni kot makri v zglavni datoteki <instance_piod.h>, ki pa je v okolju Arduino ni treba posebej vključevati, saj se definicije vključijo avtomatsko:

```
#define REG_PIOD_IFER    (*(WoReg *)0x400E1420u)
#define REG_PIOD_DIFSR   (*(WoReg *)0x400E1484u)
#define REG_PIOD_SCDR    (*(RwReg *)0x400E148Cu)
```

Ker obstajajo štiri periferne enote, je v imenih makrov dodana beseda PIOD, ki označuje, da gre za enoto D. Povsem na desni so zapisane konstantne šestnajstiške nepredznačene (pripona u na koncu šestnajstiških konstant) vrednosti, ki predstavljajo naslove, na katerih se nahajajo posamezni registri v krmilniku. Pred vsako konstantno nepredznačeno vrednostjo je še operator za pretvorbo tipa, ki pretvori tip vrednosti iz številske v kazalec tipa bodisi `WoReg *` bodisi `RwReg *`. Pred vse to je postavljen še operator posredovanja (`*`), ki nam prek tako dobljenega kazalca omogoči dostop do lokacije na navedenem naslovu. Podatkovna tipa `WoReg` in `RwReg` predstavljata register, ki je namenjen samo pisanju (angl. write-only), in register, ki je namenjen branju in pisanju (angl. read-write). Določena sta takole:

```
typedef volatile unsigned int WoReg;
typedef volatile unsigned int RwReg;
```

Vidimo, da sta oba tipa določena na enak način, različno poimenovanje je uporabljeno le zaradi preglednosti, ker prva dva registra v resnici nista namenjena, da bi ju brali. Pomembno je, da je pri obeh tipih uporabljen opredeljevalec `volatile`, saj se lahko vsebina registrov spreminja (in uporablja) neodvisno od poteka glavnega programa. Pomembno je tudi, da je tip podatka, zapisanega v registru, nepredznačeno celo število. V registrih nas namreč navadno zanimajo le posamezni biti ali skupki bitov, zaradi česar nad njimi izvajamo večinoma bitne operacije.

Način, na katerega so določeni gornji trije makri, nam omogoča, da jih v programu uporabljamo kot običajne spremenljivke. Na primer, če želimo v register SCDR vpisati vrednost, ki je shranjena v spremenljivki `div`, to naredimo z običajnim priredilnim stavkom:

```
REG_PIOD_SCDR = div;
```

Zapis na levi strani priredilnega operatorja namreč izvede operacijo posredovanja preko kazalca `(RwReg *)0x400E148Cu`:

```
(*(RwReg *)0x400E148Cu) = div;
```

V tem primeru zunanji par oklepajev sicer ni potreben, potrebujemo pa ga, kadar nad makrom izvajamo operacijo, ki ima prednost pred operatorjem posredovanja.

Dopolnimo zdaj program s strani 182, ki ob pritisku na tipko preklopi stanje svetleče diode. V funkcijo `setup` moramo dodati potrebne operacije nad registri IFER, DIFSR in SCDR, s čimer vzpostavimo sito proti odskakovanju z zakasnitvijo 20 ms na digitalni sponki 11:

```
#define TIPKA_PIN 11
#define LED_PIN 12

void setup(void) {
  unsigned int ms = 20;       /* Želena zakasnitev sita je 20 ms.  */
  unsigned int div = (unsigned int) (ms * 16.384 - 1);
  pinMode(TIPKA_PIN, INPUT_PULLUP);
  pinMode(LED_PIN, OUTPUT);
  REG_PIOD_IFER |= 1 << 7;    /* Postavi bit 7 v registru.         */
  REG_PIOD_DIFSR |= 1 << 7;   /* Postavi bit 7 v registru.         */
  REG_PIOD_SCDR = div;        /* Vpiše vrednost div v SCDR.        */
}

void loop(void) {
  static int staro = HIGH;
  static int led = HIGH;
  int novo;
  digitalWrite(LED_PIN, led);
  novo = digitalRead(TIPKA_PIN);
  if (novo != staro) {
    staro = novo;
    if (novo == LOW) {
      if (led == LOW) {
        led = HIGH;
      }
      else {
        led = LOW;
      }
    }
  }
}
```

Spomnimo se, da moramo za vklop sita na digitalni sponki postaviti bit številka sedem v registrih IFER in DIFSR, kar naredimo z ustrezno masko in bitno operacijo ALI. Podatek o času zakasnitve, ki ga vpišemo v register SCDR, je edini podatek, ki ga ta register hrani, zato ga lahko vanj vpišemo enostavno z uporabo priredilnega operatorja. Če bi bile v registru zapisane še kakšne druge informacije, ki jih pri tem ne bi smeli spreminjati, bi morali za zapisovanje uporabiti tehniko polja bitov.

Naloga 11.4 Večina sistemov Arduino (izjema je Arduino Due) uporablja Atmelove osembitne mikrokrmilnike iz družine AVR[a]. V teh mikrokrmilnikih so splošno namenske vhodno-izhodne sponke razdeljene med tri ***vrata*** (angl. port): B, C in D. Vsaka od vrat krmilimo s tremi strojni registri: DDRx, PORTx in PINx. Digitalne sponke med D0 in D7 krmilimo prek vrat D z naslednjimi registri:

- ***Smerni register*** (angl. Port D Data Direction Register, DDRD) S tem registrom določimo, ali bo posamezna sponka vhodna ali izhodna. Ničla pomeni, da je ustrezna sponka vhodna, enka pa, da je izhodna.
- ***Podatkovni register*** (angl. Port D Data Register, PORTD) Biti, ki jih pišemo v ta register, se pojavijo na izhodnih sponkah. V primeru, da je sponka določena kot vhodna, pa vpis na ustrezno mesto v tem registru vklopi oziroma izklopi dvižni upor. Vpis enice ta upor vklopi, vpis ničle pa ga izklopi.
- ***Register vhodnih sponk*** (angl. Port D Input Pins Register, PIND) Iz tega registra preberemo stanja na ustreznih vhodnih sponkah.

Vsak od bitov gornjih treh registrov ustreza eni sponki sistema. Pri tem predstavlja bit številka nič sponko D0, bit številka ena sponko D1, in tako dalje.

Za nalogo si zamislite sistem, na katerega so na sponkah D7, D6 in D5 priklopljene tri diode LED, na sponkah D4, D3 in D2 pa tri stikala[b]. Pojasnite, kaj počne naslednji program:

```
void setup(void) {
  DDRD &= 0xE3;
  PORTD |= 0x1C;
  DDRD |= 0xE0;
}

void loop(void) {
  PORTD = PORTD & 0x1F | (~PIND & 0x1C) << 3;
}
```

Opomba: Z neposrednim delom z registri lahko napišemo kodo, ki je precej krajša in hitrejša. Slaba stran takšne kode je, da je dokaj nečitljiva in slabše prenosljiva.

[a] Emulator sistema Arduino Uno na spletni strani tinkercad.com prav tako posnema tak mikrokrmilnik.
[b] Sponki D0 in D1 se uporabljata pri serijski komunikaciji, kar vključuje tudi nalaganje skice na sistem. Sponki lahko uporabljamo, če ne potrebujemo serijske komunikacije, vendar med nalaganjem skice na teh dveh sponkah ne sme biti priklopljenega nič takšnega, kar bi motilo njuno delovanje.

Naloga 11.5 Če ste rešili nalogo 11.4, potem rešite še naslednji problem:

Napišite funkcijo `prikazi7Segm`, s katero boste lahko na sedemsegmentni prikazovalnik pisali desetiške številke. Funkcija naj kot prvi parameter sprejme tabelo sedmih celih števil, ki hrani številke priključnih sponk, na katere je priključen sedemsegmentni prikazovalnik (po vrsti od a do g). Drugi parameter funkcije naj bo desetiška številka, ki jo želimo prikazati. Tako je videti primer klica funkcije:

```
int sponke[] = {2, 3, 4, 5, 6, 7, 8};
//...
prikazi7Segm(sponke, 9);  /* Prikaže številko 9 na prika-
```

```
                          zovalniku, priklopljenem na
                          sponkah od D2 do D8.          */
```

Napišite še funkcijo `inic7Segm`, ki bo vse podane sponke določila za izhode. Funkcijo boste klicali takole:

```
inic7Segm(sponke);  /* Sponke od D2 do D8 postanejo
                       izhodne sponke.              */
```

Pri pisanju kode ne smete uporabiti funkcij `pinMode` in `digitalWrite`. Namesto tega uporabite ustrezne registre. Za krmiljenje sponk do D7 se uporabljajo vrata D (glej nalogo 11.4), sponko D8 pa krmili bit LSB (skrajno desni bit) vrat B. Za nastavitev smeri sponke D8 uporabite bit LSB registra DDRB, za pisanje na to sponko pa bit LSB registra PORTB.

Delovanje kode preizkusite na vezju na strani 180.

Pomoč: V funkciji `prikazi7Segm` uporabite naslednjo tabelo šestnajstiških zapisov vseh desetih kombinacij stanja diod iz tabele na strani 179:

```
unsigned char BCD[] = {0x3F, 0x06, 0x5B, 0x4F, 0x66,
                       0x6D, 0x7D, 0x07, 0x7F, 0x6F};
```

Tabela preslikav med sponkami

Videli smo, da je preslikava med registri mikrokrmilnika SAM3X in sponkami sistema Arduino Due precej neprikladna za splošno programiranje. Zaradi tega je v Arduinovi knjižnici vzpostavljena globalna tabela `g_APinDescription` strukturnih spremenljivk, ki hrani podatke o teh preslikavah. Tabela med drugim uporablja podatkovni tip `Pio`, ki je določen v zglavni datoteki <component_pio.h>:

```
typedef struct {
  //...
  WoReg PIO_IFER;
  //...
  WoReg PIO_DIFSR;
  //...
  RwReg PIO_SCDR;
  //...
} Pio;
```

Člani tega strukturnega podatkovnega tipa so vsi dolgi 32 bitov[5] in navedeni v istem vrstnem redu, v kakršnem so razporejeni ustrezni registri v krmilniku. Za dostop do navedenih strojnih registrov potrebujemo zato le kazalec tipa `Pio *`, ki ga usmerimo na prvega od registrov, ki jih navaja strukturni tip `Pio`. Na ta način navidezno preslikamo celotno polje registrov v strukturno spremenljivko tipa `Pio`. Posledica tega je, da lahko posamezen register izberemo na enak način, kakor izberemo člana običajne strukturne spremenljivke preko kazalca (tj. z operatorjem `->`). Na primer, niz registrov, ki upravljajo periferno enoto D, se začne na šestnajstiškem naslovu 400E1400. Do registra IFER, ki pripada periferni enoti D, pridemo zato takole:

```
#define PIOD ((Pio *) 0x400E1400u)
//...
PIOD->PIO_IFER |= 1 << 7;
```

[5] Ker je mikrokrmilnik SAM3X 32-biten, so vsi njegovi registri dolgi 32 bitov.

Makro `PIOD` smo določili kot konstanten kazalec tipa `Pio *`, ki je usmerjen na blok podatkov z začetkom na šestnajstiškem naslovu 400E1400. Ker je tip `Pio` strukturni tip, je učinek isti, kot da bi se na tem naslovu začela običajna strukturna spremenljivka tega tipa. Njeni člani so vsi 32-bitna nepredznačena cela števila, ki se nahajajo na naslovih ustreznih strojnih registrov mikrokrmilnika.

Napišimo zdaj definicijo splošne funkcije za vzpostavitev sita proti odskakovanju na določeni sponki. Takole je videti funkcija `setDebounce`, ki nastavi sito z zakasnitvijo `ms` milisekund na sponki `pin`[6]:

```
void setDebounce(unsigned int pin, unsigned int ms) {
  unsigned int div = (unsigned int) (ms * 16.384 - 1);
  if (g_APinDescription[pin].ulPinType == PIO_NOT_A_PIN) {
    return;
  }
  g_APinDescription[pin].pPort->PIO_IFER |=
                         g_APinDescription[pin].ulPin;
  g_APinDescription[pin].pPort->PIO_DIFSR |=
                         g_APinDescription[pin].ulPin;
  g_APinDescription[pin].pPort->PIO_SCDR = div;
}
```

Iz kode je razvidno, da iz globalne tabele `g_APinDescription` odčitamo podatke o ustrezni sponki tako, da uporabimo številko sponke (kakor jo uporablja sistem Arduino) kot indeks tabele. Iz kode je razvidno tudi, da vsebuje posamezen element tabele vse podatke o preslikavi izbrane sponke v krmilniku SAM3X v strukturni spremenljivki s člani `ulPinType`, `ulPin` in `pPort`. Član `ulPinType` hrani podatek o vrsti sponke. Tega najprej primerjamo z identifikatorjem `PIO_NOT_A_PIN`, ki je določen v zglavni datoteki <pio.h> in predstavlja številko, ki ne označuje veljavne digitalne sponke. Če sponka ni veljavna digitalna sponka, potem funkcijo na tem mestu končamo. V nasprotnem primeru nastavimo ustrezne vrednosti v treh registrih IFER, DIFSR in SCDR. Ti so podani kot člani strukturne spremenljivke tipa `Pio`, na katero kaže kazalec `pPort`. Ustrezen bit v registrih IFER in DIFSR postavimo z bitno operacijo ALI z masko, ki je shranjena v članu `ulPin`.

Z uporabo funkcije `setDebounce` lahko zdaj napišemo funkcijo `setup` v našem zadnjem programu na strani 187 precej krajše:

```
void setup(void) {
  pinMode(TIPKA_PIN, INPUT_PULLUP);
  pinMode(LED_PIN, OUTPUT);
  setDebounce(TIPKA_PIN, 20);
}
```

Za konec pripomnimo še to, da funkcija `setDebounce`, ki smo jo pravkar napisali, ob vsakem klicu nastavi čas zakasnitve za ***vse*** sponke tiste periferne enote, ki ji pripada tudi sponka `pin`. To velja seveda le za sponke, na katerih smo že vzpostavili sito proti odskakovanju. Drugače niti ne gre, saj ima vsaka od perifernih enot le en register SCDR.

[6]Imena identifikatorjev `g_APinDescription`, `pPort`, `ulPin` in `ulPinType`, ki so določena v Arduinovi knjižnici, so izbrana v skladu s tako imenovanim ***madžarskim zapisom*** (angl. Hungarian notation). Zapis se imenuje po madžarskem programerju Charlesu Simonyju, ki je bil nekaj časa vodilni programski arhitekt pri Microsoftu. Posebnost madžarskega zapisa je ta, da se ime spremenljivke začne na način, ki namiguje na njen tip. Na primer, spremenljivka `ulPin` je tipa `unsigned long`, spremenljivka `pPort`, je kazalec (angl. pointer), spremenljivka `g_APinDescription` pa je globalna spremenljivka.

11.5 Prikazovalnik LCD

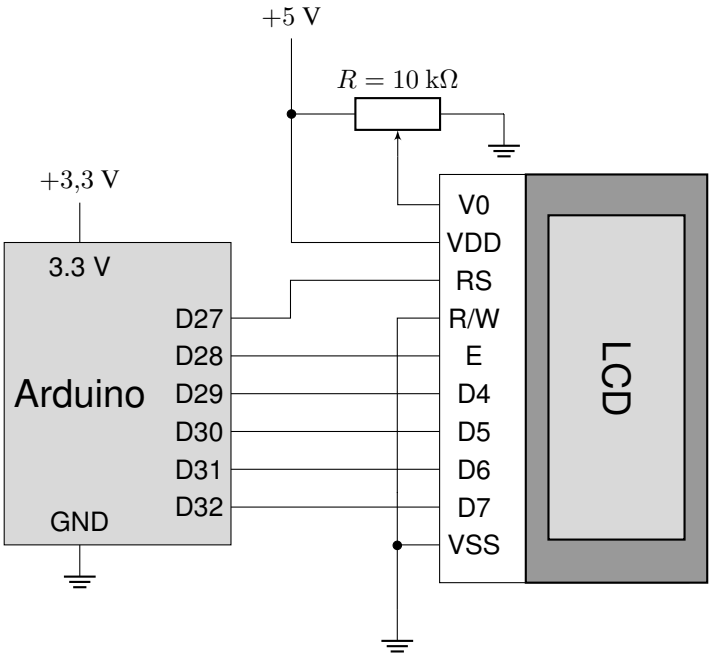
Primeri uporabe splošno namenskih sponk, ki smo jih srečali doslej, so bili dokaj preprosti. Vsi primeri so bili usmerjeni bodisi na branje bodisi pisanje enega samega bita preko ene same sponke[7]. Mnogokrat pa je komunikacija z zunanjimi napravami precej bolj zapletena. Tak primer predstavlja prikazovalnik LCD, ki tudi sam vsebuje mikrokrmilnik. Ta skrbi tako za delovanje samega prikazovalnika kot tudi za komunikacijo z mikrokrmilnikom, na katerega priklopimo prikazovalnik. Skupku pravil, ki določajo, kako takšne naprave med seboj komunicirajo, pravimo ***komunikacijski protokol*** (angl. communication protocol). Velika večina prikazovalnikov LCD uporablja protokol HD44780, ki je dobil svoje ime po Hitachijevem mikrokrmilniku, namenjenem krmiljenju prikazovalnikov LCD. Protokol je razmeroma zapleten, zato se na tem mestu ne bomo spuščali v vse njegove podrobnosti, ogledali si bomo le nekaj osnovnih principov. Tipičen prikazovalnik LCD ima med drugim naslednje priključne sponke:

- **VSS** Ta sponka se priključi na ozemljitev.
- **VDD** Ta sponka se priključi na napajalno napetost (običajno 5 V).
- **V0** To je analogna sponka, preko katere nadzorujemo kontrast prikazanega besedila. Običajno sponko V0 priklopimo na potenciometer, s čimer lahko uporabnik nastavi želeni kontrast prikaza.
- **D7 – D0** ***Podatkovne sponke*** (angl. Data Pins) Stanje na teh osmih sponkah predstavlja osembitni podatek, ki ga pošiljamo prikazovalniku. Ta podatek se lahko tolmači bodisi kot znak za prikazovanje bodisi kot ukaz, odvisno od stanja sponke RS. Podatek lahko prenesemo tudi v dveh delih, s čimer na mikrokrmilniku prihranimo štiri sponke. V tem primeru uporabimo podatkovne sponke od D4 do D7.
- **R/W** ***Sponka za izbiro smeri prenosa podatkov*** (angl. Read/Write Pin) Na prikazovalnik lahko pišemo, prav tako pa lahko z njega beremo. S sponko R/W izberemo enega od obeh možnih načinov: z logično enko izberemo branje, z logično ničlo pa pisanje na prikazovalnik. Če bomo na prikazovalnik samo pisali (kar je najobičajneje), lahko to sponko priklopimo na ozemljitev. S tem postavimo sponko R/W v trajno stanje logične ničle ter prihranimo eno priključno sponko na mikrokrmilniku.
- **RS** ***Sponka za izbiro registra*** (angl. Register Select Pin) Prikazovalnik vsebuje dva osembitna registra: ***podatkovni register*** (angl. data register, DR) služi začasnemu hranjenju podatkov, ki jih pošiljamo ali beremo s prikazovalnika, v ***ukazni register*** (angl. instruction register, IR) pa zapisujemo ukaze. Ukazi so lahko na primer *pobriši zaslon* ali *nastavi kazalnik* (angl. cursor). V primeru, ko na prikazovalnik pišemo, stanje sponke RS določa, kakšne vrste informacijo pošiljamo prikazovalniku (preko podatkovnih sponk): če postavimo sponko RS na logično ničlo, pošiljamo ukaz (tj. poslani podatek se zapiše v ukazni register), sicer pošiljamo podatek za prikaz (tj. poslani podatek – ki je običajno koda ASCII – se zapiše v podatkovni register).
- **E** ***Sponka, ki omogoči pisanje v registre*** (angl. Enable Pin) Na to sponko priklopimo signal, ki skrbi za usklajeno pisanje podatka na podatkovnih sponkah v ustrezni register prikazovalnika. Šele ko je podatek na podatkovnih sponkah pripravljen, sprožimo njegovo pisanje v register prikazovalnika s prehodom iz logične enke v logično ničlo na sponki E.

[7]Tudi krmiljenje sedemsegmentnega prikazovalnika je na koncu sestavljeno zgolj iz pisanja posameznih bitov na posamezne sponke brez kakršnihkoli časovnih ali drugih dodatnih zahtev.

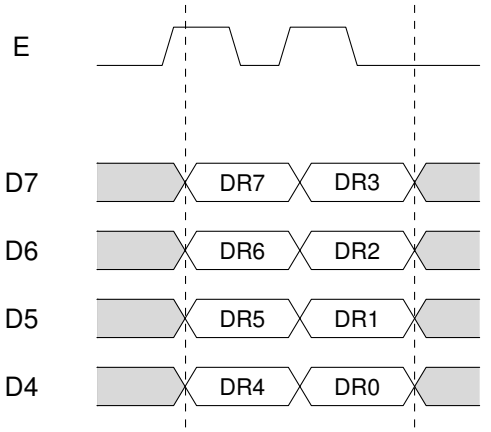
Iz gornjih opisov opazimo pomembno razliko med prikazovanjem podatkov na pasivnem prikazovalniku, kakršen je sedemsegmentni prikazovalnik, in prikazovanjem na prikazovalniku LCD, katerega delovanje nadzoruje lastni mikrokrmilnik: pri prikazovalniku LCD se stanje, ki ga povzročimo na njegovih sponkah, ne odraža neposredno na zaslonu, temveč služi kot vhodni podatek za program, ki se bo izvedel na krmilniku prikazovalnika. Zato je pomembno, da prikazovalnik »ve«, kdaj natančno je vhodni podatek pripravljen za obdelavo. Pri sedemsegmentnem prikazovalniku to ni pomembno: posamezni segmenti se lahko prižigajo s poljubnimi zakasnitvami in v poljubnem vrstnem redu. Pomembno je le, da so zakasnitve dovolj majhne, da jih človeško oko ne zazna.

Naslednji primer nam bo pomagal razumeti, kako uporabimo usklajevalni signal na sponki E, da sporočimo krmilniku prikazovalnika, da je podatek pripravljen. Takole smo priklopili prikazovalnik LCD na sistem Arduino:



Iz slike je razvidno, da smo sponko R/W priključili na ozemljitev, kar pomeni, da bomo na prikazovalnik samo pisali. Od podatkovnih sponk smo uporabili le sponke od D4 do D7, zato bomo morali izpeljati pisanje slehernega podatka v dveh korakih.

Vzemimo za primer, da je na sponki RS logična enka. Skupaj s trajno logično ničlo na sponki R/W to pomeni, da bomo prikazovalniku preko sponk od D4 do D7 poslali kodo znaka, ki ga želimo prikazati. Naslednja slika prikazuje ***časovni diagram*** (angl. timing diagram) postopka, ki je v tem primeru potreben za pisanje osembitnega podatka na prikazovalnik:



Časovni diagrami so namenjeni prikazu niza logičnih signalov v časovnem prostoru. Vsak signal je prikazan v svoji vrstici, eden od signalov pa je navadno bodisi uskladitveni signal bodisi ura (angl. clock). Časovni diagram je orodje, ki ga pogosto uporabljamo pri načrtovanju digitalnih elektronskih vezij. Poleg tega da nam tak diagram na pregleden način prikaže časovna razmerja med signali, nam pomaga tudi pri odkrivanju nevarnosti pojavitev logičnih napak zaradi časovnih zakasnitev. Zaradi fizikalnih omejitev so v resničnih digitalnih sistemih takšne zakasnitve neizogibne. V gornjem diagramu je na vrhu prikazan uskladitveni signal, pod njim pa so štirje signali na podatkovnih sponkah. Višja vrednost signala pomeni logično enko, nižja vrednost pa logično ničlo. Kjer sta prikazani dve vrednosti hkrati, pomeni, da je lahko signal bodisi v stanju logične enke bodisi ničle. Če je področje med obema stanjema belo, je stanje določeno, če pa je to območje osenčeno, potem stanje ni določeno oziroma ni pomembno.

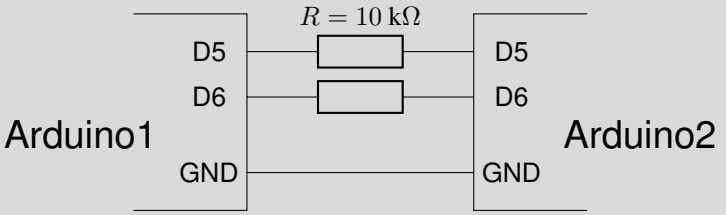
Vpisovanje osembitnega podatka v podatkovni register prikazovalnika poteka v času, ki je na gornjem diagramu označen z dvema navpičnima prekinjenima črtama. Pred vpisovanjem stanje na podatkovnih sponkah ni pomembno. Poševne črte nakazujejo dejstvo, da sprememba stanja ni hipna, temveč traja nek (zelo kratek) čas. V realnosti tudi ni nujno, da se vrednosti pojavijo na vseh sponkah hkrati. Pomembno je (kar je razvidno tudi iz diagrama), da je v času prvega spusta uskladitvenega signala na podatkovnih sponkah vzpostavljeno stanje, ki predstavlja gornje štiri bite podatka (oznake od DR7 do DR4 predstavljajo gornje štiri bite podatkovnega registra). Ko je vpis prvih štirih bitov zaključen, se ponovi še za druge štiri bite podatka.

Pri izvajanju gornjega postopka je treba upoštevati še nekaj časovnih omejitev: perioda uskladitvenega signala ne sme biti krajša od 1000 ns, od česar mora biti signal v stanju logične enke vsaj 450 ns. Ko je vpis vseh osmih bitov zaključen, je treba počakati vsaj 37 μs, da krmilnik prikazovalnika izvrši prikaz poslanega znaka[8].

Naloga 11.6 Za vajo napišite program, ki bo z enega sistema Arduino (oddajnik) pošiljal osembitne podatke na drug sistem Arduino (sprejemnik). Sprejemnik naj izpisuje sprejete osembitne vrednosti v obliki znakov na serijski vmesnik. Za napotke, kako delati s serijskim vmesnikom, poglejte nalogo 11.3 na strani 184.

Povežite dva sistema, kakor kaže naslednja slika:

[8]Ravno toliko časa traja tudi izvajanje večine ukazov. Izjema je na primer ukaz, ki pobriše zaslon, ki traja 1,52 ms.



Odločimo se, da bomo podatke pošiljali preko sponke D5, sponko D6 pa bomo uporabljali za uskladitveni signal. Napisati moramo dva programa, ki med drugim počneta naslednje:

- ***Oddajnik*** Oddajnik postavi sponko D6 v stanje logične enke, sponko D5 pa v stanje, ki ustreza vrednosti bita, ki ga želi prenesti. Oddajnik na koncu postavi sponko D6 v stanje logične ničle, kar pomeni, da je bit na sponki D5 pripravljen za prevzem. S tem oddajnik pošlje en bit podatka. Za vsak bit se mora postopek ponoviti, da pa oddajanje ne bo prehitro (sprejemnik mu mora biti sposoben slediti), pošljemo bit le vsakih 10 ms.
- ***Sprejemnik*** Sprejemnik čaka, dokler na sponki D6 ne zazna prehoda iz logične enke v logično ničlo[a], kar pomeni, da je bit na sponki D5 pripravljen za sprejem. Sprejemnik zdaj ta bit prebere in ustrezno obdela, nato pa že lahko začne čakati na naslednjega.

[a] Prehodu signala iz logične enke v logično ničlo pravimo tudi ***padajoči rob*** (angl. falling edge). Prehodu iz logične ničle v logično enko pa pravimo ***naraščajoči rob*** (angl. rising edge).

Kakor smo videli, je pisanje podatka na prikazovalnik sorazmerno zapleten postopek, ki mora poleg logičnih zakonitosti upoštevati še določene časovne zahteve. Tudi sama inicializacija prikazovalnika zahteva precej duhamornega pisanja različnih ukazov z vmesnimi obdobji čakanja. Vendar se v dodatne podrobnosti komunikacijskega protokola HD44780 ne bomo spuščali. Namesto tega bomo uporabili knjižnico <LiquidCrystal.h>, ki je del okolja Arduino in nam omogoča zelo prikladno delo s prikazovalnikom. Knjižnico vključimo v program z direktivo `#include`:

```
#include <LiquidCrystal.h>
```

Knjižnica je napisana v jeziku C++, ki je objektno usmerjen jezik. Knjižnica vsebuje razred `LiquidCrystal`, s pomočjo katerega ustvarimo objekt, ki bo v programu zastopal prikazovalnik LCD, ki je priključen na sistem. Vendar brez skrbi, ni nam treba poznati jezika C++, da uporabimo omenjeno knjižnico. Naslednja vrstica ustvari globalno dostopen objekt `g_lcd` in pripravi vse potrebno za uspešno komunikacijo mikrokrmilnika s prikazovalnikom:

```
LiquidCrystal g_lcd(27, 28, 29, 30, 31, 32);
```

Številke, ki smo jih podali v oklepaju, predstavljajo številke splošno namenskih sponk, na katere smo po vrsti priklopili sponke RS, E ter podatkovne sponke od D4 do D7. Pred začetkom uporabe prikazovalnika moramo s postopkom `begin` določiti še število stolpcev in vrstic, ki jih ima naš prikazovalnik. Zatem že lahko izpišemo pozdravno sporočilo:

```
void setup(void) {
  g_lcd.begin(16, 2);  /* Naš LCD ima 16 stolpcev in 2 vrstici.  */
```

```
  g_lcd.print("Hello World!");   /* Izpiše pozdravno sporočilo.  */
}
```

S postopkom `print` lahko na zaslon pišemo tudi vrednosti spremenljivk:

```
int x = 42;
g_lcd.print(x);  /* Izpiše: 42  */
```

Naloga 11.7 Za vajo napišite program, ki bo na prikazovalniku LCD štel v sekundnem intervalu. Vsak klic postopka `print` izpiše podatke na zaslon od mesta, na katerem se je končal prejšnji izpis. Če želite doseči, da se pri štetju številka na zaslonu spreminja (in ne dodaja), boste morali pred vsakim novim izpisom nastaviti kazalnik s postopkom `setCursor`. Postopku morate podati številko stolpca in vrstice, kamor želite postaviti kazalnik, pri čemer se štetje začne z ničlo. Na primer, tako lahko postavite kazalnik na začetek druge vrstice zaslona:

```
g_lcd.setCursor(0, 1);
```

Če je treba, lahko zaslon prikazovalnika tudi pobrišete s postopkom `clear`. S tem hkrati postavite kazalnik v gornji levi kot zaslona.

Naloga 11.8 Preden nadaljujemo, rešite še naslednji problem:

Predpostavimo, da imamo na sistemu Arduino priklopljen prikazovalnik LCD, ki je že inicializiran:

```
LiquidCrystal lcd(13, 12, 11, 10, 9, 8);
//...
lcd.begin(16, 2);
```

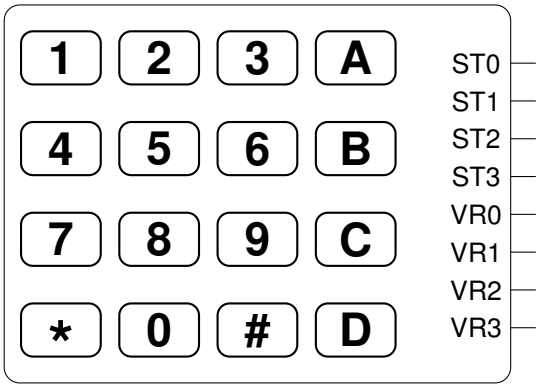
Napišite kodo, ki bo na ta prikazovalnik izpisala sporočilo *Hello World!* z uporabo funkcij `digitalWrite` in `delayMicroseconds`[a] (tj. brez postopka `print`). Pri tem upoštevajte naslednje:

- pred pisanjem morate poskrbeti, da bo na sponki RS logična enka;
- logična enka na sponki E mora pred spustom v logično ničlo trajati vsaj 450 ns;
- perioda signala na sponki E ne sme biti krajša od 1000 ns;
- ko je vpis enega osembitnega podatka zaključen, morate počakati vsaj 37 μs.

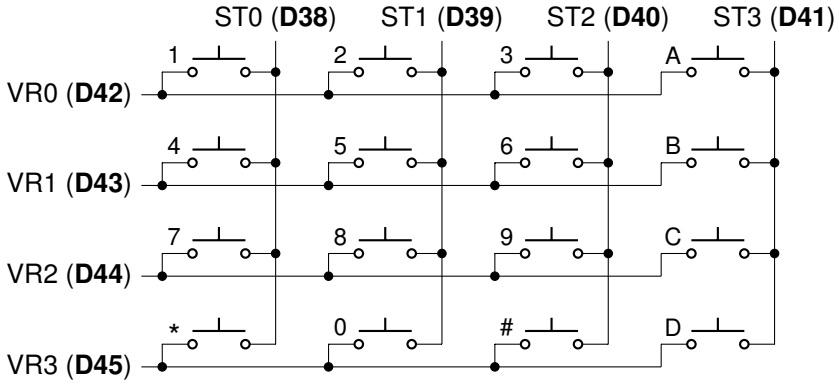
[a] Funkcija `delayMicroseconds` deluje tako kot funkcija `delay`, le da ji podamo čas zakasnitve v mikrosekundah namesto v milisekundah.

11.6 Matrična tipkovnica

Matrična tipkovnica (angl. matrix keypad) se pogosto uporablja v vgrajenih sistemih, kot so na primer telefoni, kalkulatorji ali bančni avtomati. Prednost matrične tipkovnice je ta, da z njo zmanjšamo število uporabljenih sponk na mikrokrmilniku. Na naslednji sliki vidimo primer matrične tipkovnice s 16 tipkami in z osmimi priključnimi sponkami:

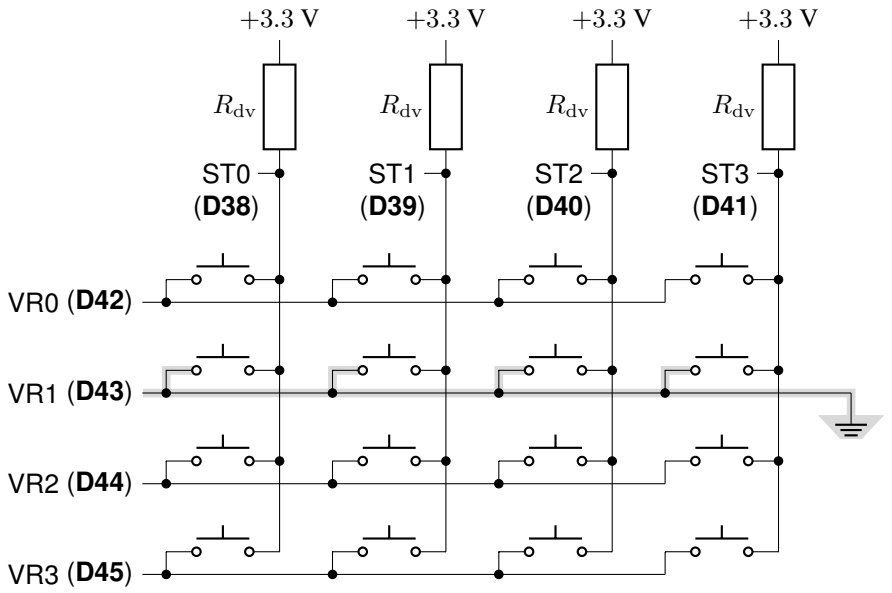


Matrična tipkovnica je pasivno vezje in je zgrajena izključno iz mehanskih tipk. Na naslednji sliki vidimo vezalno shemo gornje tipkovnice:



Od osmih priključnih sponk so sponke od ST0 do ST3 priključene na štiri stolpce, sponke od VR0 do VR3 pa na štiri vrstice. Na vsakem od 16 stičišč stolpcev in vrstic se nahaja tipka, ki povezuje en stolpec in eno vrstico. Sponke od D38 do D45, ki so navedene v oklepajih, predstavljajo sponke sistema Arduino, na katere bomo priklopili našo tipkovnico. Tudi za matrično tipkovnico obstaja knjižnica (<Keypad.h>), vendar je branje s takšne tipkovnice ravno pravšnji oreh, da se ga bomo lotili sami.

Posamezno tipko matrične tipkovnice bomo brali na enak način, kakor smo brali eno samo tipko v primeru na strani 181. Branja se lotimo tako, da beremo vsako vrstico posebej. Vrstico, ki jo beremo, priklopimo na ozemljitev, vsak stolpec posebej pa priklopimo na svojo vhodno sponko, pri čemer ne smemo pozabiti na dvižne upore. Če želimo brati na primer vrstico tipk na sponki VR1, potem moramo vzpostaviti takšno stanje:



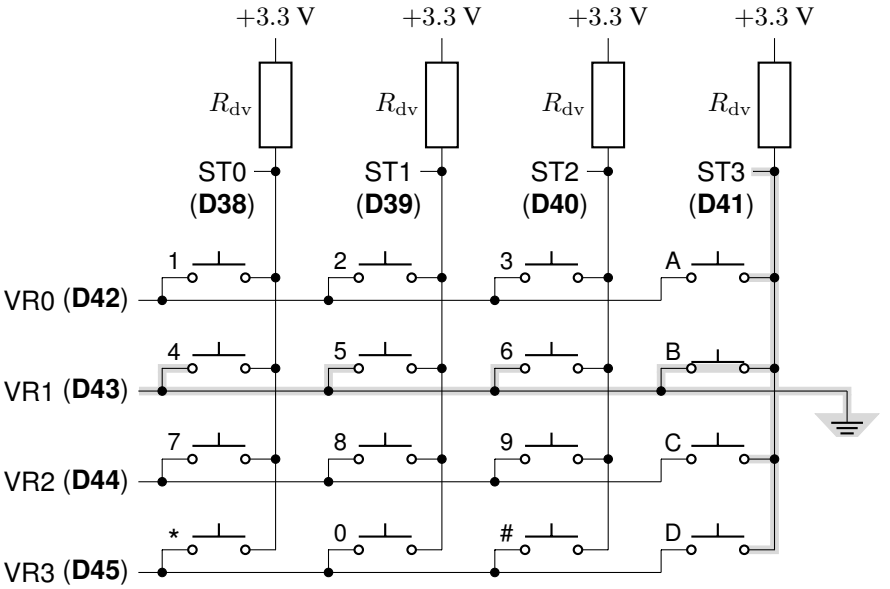
Prikazano stanje dobimo z naslednjim programom:

```
#define ST_VRSTIC 4
#define ST_STOLPCEV 4
int g_st[ST_STOLPCEV] = {38, 39, 40, 41};
int g_vr[ST_VRSTIC]   = {42, 43, 44, 45};

void izberiVrstico(int v) {
  for (int i = 0; i < ST_VRSTIC; i++) {
    if (i == v) {
      pinMode(g_vr[i], OUTPUT);
      digitalWrite(g_vr[i], LOW);
    }
    else {
      pinMode(g_vr[i],INPUT);
    }
  }
}

void setup(void) {
  for (int i = 0; i < ST_STOLPCEV; i++) {
    pinMode(g_st[i], INPUT_PULLUP);
  }
  izberiVrstico(1);
}
```

Sama koda ni nič posebnega. V funkciji `setup` najprej (v stavku `for`) določimo, da so sponke od D38 do D41 vhodne sponke s priključenimi dvižnimi upori. Zatem kličemo funkcijo `izberiVrstico` z argumentom ena. Funkcija `izberiVrstico` odklopi vse sponke od D42 do D45 (tj. jih določi za vhodne sponke, s čimer jih postavi v stanje visoke impedance), razen sponke, ki jo določa parameter `v`. Slednjo določi za izhodno sponko in nanjo zapiše logično ničlo, s čimer jo priklopi na ozemljitev. V konkretnem primeru gornjega programa se priklopi na ozemljitev sponka D43, na kateri je priključena vrstica tipkovniceVR1. Seveda bomo kasneje v programu posamezne vrstice matrične tipkovnice

odklapljali in priklapljali drugo za drugo, s čimer bomo sistematično pregledali celotno tipkovnico.

Ko ni pritisnjena nobena tipka (kot je prikazano na zadnji sliki), preberemo na vseh štirih vhodih (sponke od D38 do D41) logično enko. Ko pa pritisnemo na primer zadnjo tipko v vrstici VR1 (spodnja slika), se na vhodu D41, na katerega je priključena sponka tipkovnice ST3, pojavi logična ničla. Povezave, ki so v stanju logične ničle, so na sliki osenčene:



Da ugotovimo, katera tipka je pritisnjena, moramo pregledati celotno tipkovnico, tako da za vsako vrstico posebej preverimo, ali je na katerem od štirih vhodov logična ničla. Takole je videti funkcija, ki vrne kodo ASCII tipke, ki je pritisnjena:

```
char g_tipkovnica[ST_VRSTIC][ST_STOLPCEV + 1] = {"123A",
                                                 "456B",
                                                 "789C",
                                                 "*0#D"};

char beriZnak(void) {
  for (int v = 0; v < ST_VRSTIC; v++) {
    izberiVrstico(v);  /* Vrstico tipk, ki jo beremo, moramo
                          priklopiti na ozemljitev.          */
    for (int s = 0; s < ST_STOLPCEV; s++) {
      if (digitalRead(g_st[s]) == LOW) {
        return g_tipkovnica[v][s];  /* Vrnemo kodo pritis-
                                       njene tipke.          */
      }
    }
  }
  return 0;  /* Če ni pritisnjena nobena tipka.             */
}
```

V globalno dvorazsežnostno tabelo `g_tipkovnica`, iz katere odčitavamo kode pritisnjenih tipk, smo zapisali znake, ki so na tipkah naše tipkovnice. V funkciji `beriZnak` vsako vrstico posebej (zunanja zanka `for`) najprej priključimo na ozemljitev, potem pa v

notranji zanki `for` pregledamo stanja vseh štirih stolpcev. Če na katerem od vhodov zaznamo logično ničlo, končamo izvajanje funkcije, tako da vrnemo kodo pritisnjene tipke. Če ni pritisnjena nobena tipka, potem funkcija `beriZnak` vrne vrednost nič. Iz kode je razvidno tudi to, da v primeru, ko je pritisnjenih več tipk, funkcija vrne kodo prve tipke, ki jo zazna.

Združimo zdaj matrično tipkovnico s prikazovalnikom LCD in napišimo program, ki na prikazovalniku prikazuje znake, ki jih tipkamo na tipkovnici. Da lahko to naredimo, moramo v programu zaznavati spremembe stanj na tipkovnici. Podoben problem smo enkrat že rešili, v naslednjem programu pa je zaznavanje spremembe stanja tipk izvedeno v funkciji `loop`. Takole je videti dokončan program:

```
#include <LiquidCrystal.h>

#define ST_VRSTIC 4
#define ST_STOLPCEV 4

int g_st[ST_STOLPCEV] = {38, 39, 40, 41};
int g_vr[ST_VRSTIC]   = {42, 43, 44, 45};

char g_tipkovnica[ST_VRSTIC][ST_STOLPCEV + 1] = {"123A",
                                                 "456B",
                                                 "789C",
                                                 "*0#D"};
LiquidCrystal g_lcd(27, 28, 29, 30, 31, 32);

void izberiVrstico(int v) {
  for (int i = 0; i < ST_VRSTIC; i++) {
    if (i == v) {
      pinMode(g_vr[i], OUTPUT);
      digitalWrite(g_vr[i], LOW);
    }
    else {
      pinMode(g_vr[i], INPUT);
    }
  }
}

char beriZnak(void) {
  for (int v = 0; v < ST_VRSTIC; v++) {
    izberiVrstico(v);
    for (int s = 0; s < ST_STOLPCEV; s++) {
      if (digitalRead(g_st[s]) == LOW) {
        return g_tipkovnica[v][s];
      }
    }
  }
  return 0;
}

void setup(void) {
  g_lcd.begin(16, 2);
  for (int i = 0; i < ST_STOLPCEV; i++) {
    pinMode(g_st[i], INPUT_PULLUP);
  }
}

void loop(void) {
  static char staro = 0;
  char novo = beriZnak();
```

```
  if (staro != novo) {
    staro = novo;
    if (novo != 0) {
      g_lcd.print(novo);
    }
  }
  delay(20);  /* Zakasnitev proti odskakovanju.  */
}
```

V funkciji `loop` izpišemo znak na prikazovalnik vsakokrat, ko uporabnik pritisne tipko. Ko uporabnik tipko spusti (tj. novo stanje je enako nič), si program to zapomni, vendar na prikazovalnik ne izpiše ničesar.

Naloga 11.9 Za vajo dopolnite program na strani 199 s funkcijo `beriTipke`. Funkcija naj vrne podatek tipa `unsigned short`, v katerem naj bodo postavljeni vsi biti, ki ustrezajo trenutno pritisnjenim tipkam na matrični tipkovnici.

Na primer, če uporabnik hkrati drži tipke 1, 2 in C, naj funkcija vrne dvojiško vrednost 11000000 00010000.

Opomba: Na prikazovalniku LCD (kot tudi na serijskem vmesniku) lahko izpisujete dvojiške vrednosti tako, da kot drugi argument postopka `print` (na serijskem vmesniku tudi `println`) podate makro `BIN`:

```
g_lcd.setCursor(0, 0);
g_lcd.print(beriTipke(), BIN);
Serial.println(beriTipke(), BIN);
```

Ker se vodilne ničle pri tem ne bodo izpisale, morate na prikazovalniku LCD med posameznimi izpisi izbrisati zaslon s postopkom `clear`. V nasprotnem primeru se vam bo dogajalo, da bo v primeru, ko se dolžina izpisa skrajša, na zaslonu ostalo še del izpisa prejšnje vrednosti.

Naloga 11.10 Na sistemu Arduino imamo priklopljeno matrično tipkovnico in dvovrstični prikazovalnik LCD. Napišite program za kalkulator, ki pretvarja predznačeno desetiško celoštevilsko vrednost med -32768 in 32767 v 16-bitni dvojiški zapis. Za negativna števila naj uporabi dvojiški komplement. Kalkulator naj deluje tako, da v prvi vrstici prikazuje desetiško vrednost, ki jo vnašamo preko tipkovnice. Ob sleherni spremembi vpisane vrednosti naj popravi tudi njen dvojiški zapis, ki naj se prikazuje v drugi vrstici. Če vpisana desetiška vrednost prekorači območje 16-bitnih predznačenih števil, naj se v drugi vrstici izpiše *Napaka*. Poleg številskih tipk, ki služijo vnosu desetiškega števila, uporabite še tipki # in C. Prva naj služi menjavi predznaka vpisanega števila, druga pa brisanju zaslona.

Primer delovanja kalkulatorja:

```
 0
0000000000000000
```
(Vklop)

```
 7
0000000000000111
```
(Pritisnemo 7)

```
-7
1111111111111001
```
(Pritisnemo #)

```
-72
1111111110111000
```
(Pritisnemo 2)

```
-726
1111110100101010
```
(Pritisnemo 6)

```
 726
0000001011010110
```
(Pritisnemo #)

```
 7269
0001110001100101
```
(Pritisnemo 9)

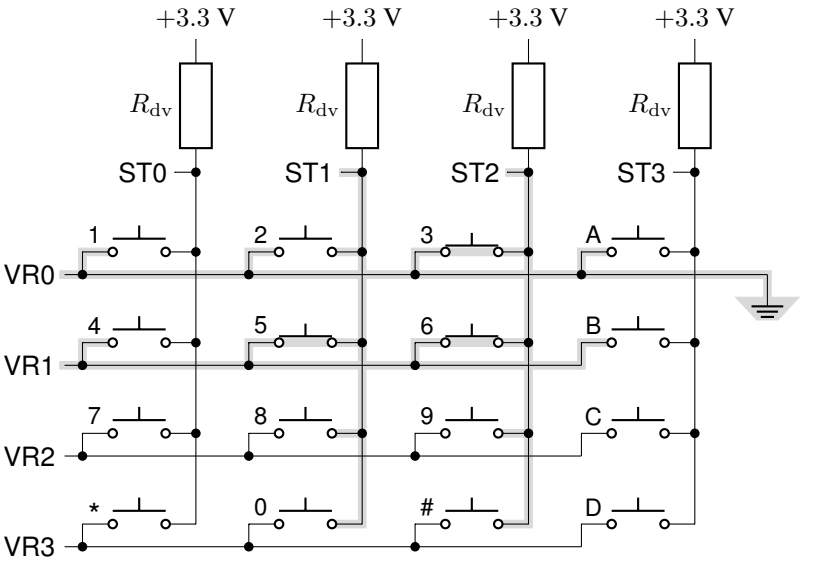
```
 72690
Napaka
```
(Pritisnemo 0)

```
 0
0000000000000000
```
(Pritisnemo C)

Navidezna tipka

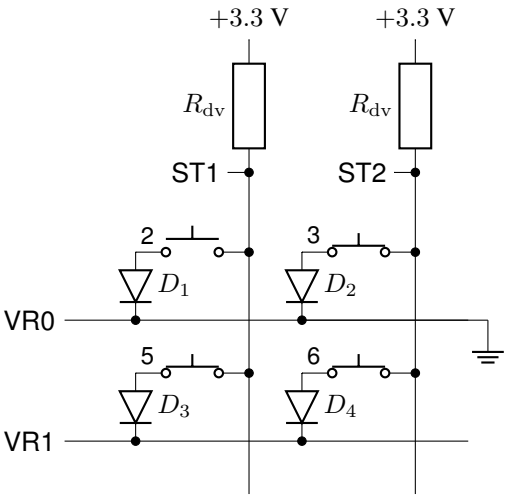
Vrnimo se še enkrat k primeru na strani 199. Če bi na primer hkrati pritisnili tipke 3, 5 in 6, bi program zaznal, da smo pritisnili tipko 2. Podoben pojav opazimo vsakokrat, ko pritisnemo tri tipke, ki ležijo v ogliščih pravokotnika: z vidika mikrokrmilnika je pritisnjena tudi tipka v četrtem oglišču pravokotnika. Takemu pojavu pravimo ***navidezna tipka*** (angl. key ghosting).

Naslednja slika prikazuje, kako pride v matrični tipkovnici do pojava navidezne tipke:



Na sliki so pritisnjene tipke 3, 5 in 6. Ko beremo tipke v vrstici VR0, je ta vrstica priklopljena na logično ničlo. Iz slike se lepo vidi, da se stanje logične ničle z vrstice VR0 preko treh pritisnjenih tipk prenese tudi na stolpec ST1. Ker pomeni logična ničla na stolpcu pritisnjeno tipko v vrstici, ki je trenutno ozemljena, bo sistem napačno zaznal, da je v prvi vrstici poleg tretje tipke pritisnjena tudi druga tipka (tj. tipka 2).

Za preprečevanje pojava navidezne tipke lahko na vsako tipko vežemo izolacijsko diodo, kot je prikazano na naslednji sliki:



Slika predstavlja izsek matrične tipkovnice, ki jo uporabljamo že ves čas, z dodanimi izolacijskimi diodami. Čeprav so tipke 3, 5 in 6 še vedno pritisnjene, pa dioda D_4 zdaj preprečuje, da bi tok s sponke ST1 stekel proti ozemljitvi, s čimer ostane ta sponka na logični enki. To je tudi pravilno stanje, saj tipka 2 ni pritisnjena.

Težavo navidezne tipke je mogoče rešiti tudi programsko, vendar ne brez določenih predpostavk in omejitev. Na primer, določene sodobne tipkovnice so programirane tako, da, namesto da bi zaznale četrto tipko, ignorirajo tretjo tipko, čemur pravimo ***zagozdenje*** (angl. jamming). Zagozdenje lahko opazimo, če uporabljamo običajno tipkovnico za igranje računalniških iger, kjer moramo pogosto pritiskati več tipk hkrati v različnih kombinacijah. Zato imajo nekatere igralne tipkovnice (angl. gaming keyboard) priklopljene izolacijske diode na določenih tipkah, ki se v igrah najpogosteje uporabljajo (npr. tipke W, A, S in D ter smerne tipke (angl. arrow keys)). Večina glasbenih klaviatur in računalniških tipkovnic višjega cenovnega razreda uporablja izolacijske diode na vseh tipkah, zaradi česar lahko pravilno berejo kakršnokoli kombinacijo tipk, ki jih lahko uporabnik pritiska ali spušča v poljubnem vrstnem redu.

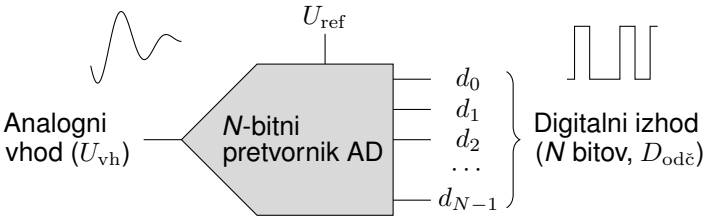
12. POGLAVJE

BRANJE IN PISANJE ANALOGNIH SIGNALOV

Fizikalni svet, s katerim računalniki komunicirajo, je običajno analogen. Zato obstajajo tudi naprave za pretvarjanje med analognimi in digitalnimi vrednostmi. Večina mikrokrmilnikov ima takšne naprave vgrajene, nekaj osnovnih principov njihovega delovanja pa bomo spoznali v tem poglavju.

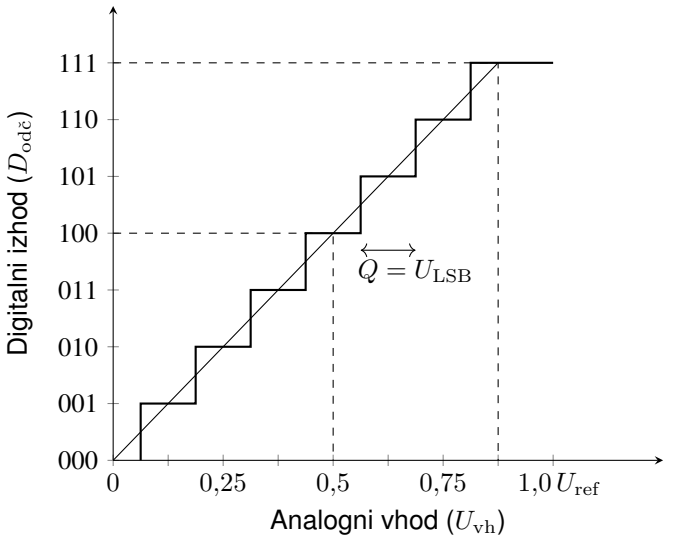
12.1 Analogno-digitalni pretvornik

Analogno-digitalni pretvornik (angl. analog-to-digital converter, ADC) je vezje, ki pretvarja analogni signal v digitalnega, kot je prikazano na naslednji sliki:



Kot je prikazano na gornji sliki, pretvornik AD vhodno napetost U_{vh} pretvori v N-bitno dvojiško vrednost $D_{\mathrm{odč}}$, sestavljeno iz bitov od d_0 do d_{N-1}. Pretvornik v postopku pretvarjanja primerja vhodno napetost z referenčno napetostjo U_{ref}, ki hkrati predstavlja

zgornjo mejo za napetost, ki jo lahko pretvarjamo. Idealni pretvornik AD pretvarja analogne vrednosti v skladu s prenosno funkcijo, ki jo prikazuje naslednja slika:



Zaradi enostavnosti je na sliki prikazan tribitni pretvornik AD. S slike najprej vidimo, da pri vhodni napetosti, ki je enaka polovici referenčne napetosti, na izhodu pretvornika odčitamo dvojiško vrednost 100, pri napetosti, ki je enaka $7/8$ referenčne napetosti, pa odčitamo dvojiško vrednost 111. V splošnem lahko iz odčitane vrednosti po naslednji enačbi izračunamo, kakšna je analogna vhodna napetost:

$$U_{\mathrm{vh}} = \frac{D_{\mathrm{odč}}}{2^N} U_{\mathrm{ref}}. \tag{12.1}$$

Pri tem je N število bitov, ki jih pretvornik AD uporablja za pretvorbo.

Zaradi kvantizacije pa ta enačba ne da vedno natančne vrednosti. To lahko vidimo tudi iz gornje slike, kjer se na primer v vrednost 100 preslikajo tudi napetosti, ki so nekoliko večje (kot tudi nekoliko manjše) od polovice referenčne napetosti. Kakor vidimo na gornjem grafu, je prenosna funkcija pretvornika AD stopničasta funkcija. Vidimo lahko, da dobimo natančne vrednosti po enačbi (12.1) samo v točkah, kjer se stopničasta funkcija seka s poševno črto.

Pomemben pojem, povezan s pretvornikom AD, je njegova ***ločljivost*** (angl. resolution). Ločljivost pretvornika AD je določena kot najmanjša sprememba vhodne napetosti, ki zagotovo povzroči spremembo na izhodu. Ker se ta sprememba odraža na vrednosti skrajno desnega bita, ločljivosti pretvornika AD pravimo tudi ***napetost bita LSB*** (angl. LSB voltage). Ločljivost izračunamo po enačbi:

$$Q = U_{\mathrm{LSB}} = \frac{U_{\mathrm{ref}}}{2^N},$$

označena pa je tudi na gornjem grafu. Včasih ločljivost podajamo tudi v bitih. V tem primeru govorimo o pretvorniku AD z ločljivostjo N bitov.

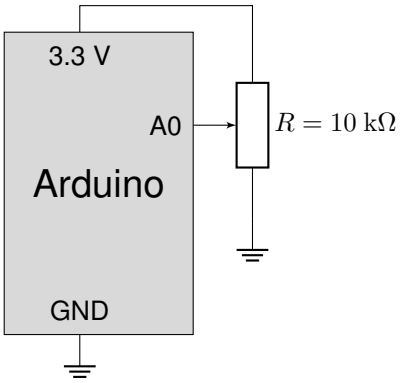
Iz gornjega grafa lahko razberemo tudi, da je največja absolutna napaka, ki jo dobimo, če računamo vhodno napetost po enačbi (12.1), enaka $Q/2$. Tej napaki pravimo ***kvantizacijska napaka*** (angl. quantization error).

Vgrajen pretvornik AD

Mikrokrmilnik SAM3X ima vgrajen 16-kanalni 12-bitni pretvornik AD, katerega uporaba pa je sorazmerno zapletena. Na tem mestu bomo za delo s pretvornikom uporabili prikladno funkcijo `analogRead`, ki jo ponuja Arduinovo integrirano razvojno okolje. Funkciji preprosto podamo oznako analogne sponke, na kateri želimo meriti napetost, vrnjeno pa dobimo pretvorjeno vrednost v obliki desetbitnega nepredznačenega celega števila (tj. vrednost med 0 in 1023)[1].

Za ilustracijo delovanja pretvornika bomo napisali program, ki bo analogno vhodno napetost, izmerjeno na vhodu A0, prikazal na prikazovalniku LCD kot stolpec osmih črtic. Za namen preizkušanja bomo napetost spreminjali s potenciometrom, v resnici pa bi lahko na vhod priklopili kakršenkoli analogen vir napetosti: na primer, primerno ojačan signal s tlačnega tipala, ki meri tlak v grelnem sistemu, ali s Hallove sonde (angl. Hall effect sensor), povezane s plovcem, ki meri količino goriva v avtomobilskem rezervoarju. V nobenem od omenjenih primerov ne potrebujemo niti velike natančnosti niti hitrosti, tako da bo naš pristop zadovoljivo rešil nalogo.

Takšno je naše vezje:



Program pa je videti takole:

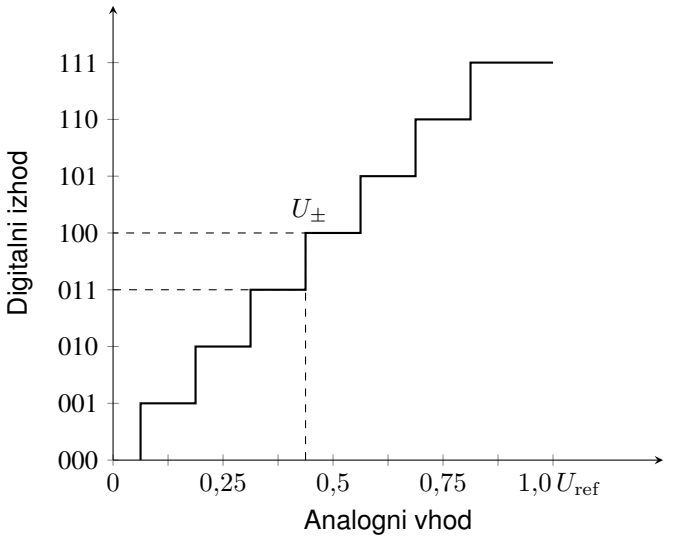
```
#include <LiquidCrystal.h>

#define ANALOG_PIN A0
#define D 8

LiquidCrystal g_lcd(27, 28, 29, 30, 31, 32);

void setup(void) {
  g_lcd.begin(16, 2);
}

void loop(void) {
  char prikaz[D + 1];
  int i, odcitek = analogRead(ANALOG_PIN);
```

[1]Funkcija `analogRead` uporablja desetbitno ločljivost zaradi združljivosti z vsemi sistemi Arduino. V Arduinovem integriranem razvojnem okolju obstaja tudi funkcija `analogReadResolution`. S to funkcijo lahko na sistemu Arduino DUE dosežemo, da bo funkcija `analogRead` izkoristila polno 12-bitno ločljivost vgrajenega pretvornika AD in bo vračala vrednosti med 0 in 4095. Funkcijo `analogReadResolution` kličemo iz funkcije `setup`.

```
  for (i = 0; i < D; i++) {
    if (i <= D * odcitek / 1024) {
      prikaz[i] = '-';
    }
    else {
      prikaz[i] = ' ';
    }
  }
  prikaz[i] = 0;  /* Zaključni ničelni znak.  */
  g_lcd.setCursor(0, 0);
  g_lcd.print(prikaz);
  delay(20);
}
```

Ker je odčitek s pretvornika AD desetbiten, ga moramo za prikaz deliti z $2^{10} = 1024$ in pomnožiti s številom želenih stopenj prikaza (v našem primeru z osem). Zanka `for` poskrbi za to, da se v znakovni niz `prikaz` vpiše ustrezno število črtic v skladu z velikostjo vhodne napetosti. Ko je znakovni niz pripravljen, ga s postopkom `print` izpišemo na prikazovalnik. Pred vsakim izpisom uporabimo postopek `setCursor`, s katerim postavimo kazalnik nazaj v gornji levi kot prikazovalnika. S tem dosežemo animacijo, ki v vsakem hipu prikazuje eno od osmih različnih stanj (od ene do osmih črtic). Na koncu smo dodali zakasnitev v dolžini 20 ms. Tako dolga zakasnitev sicer še zdaleč ni potrebna, vendar želimo s tem poudariti, da pretvornik AD za pretvorbo vrednosti potrebuje določen čas. Če bi pretvorbe zahtevali prepogosto, bi lahko dobili na izhodu neveljavne odčitke.

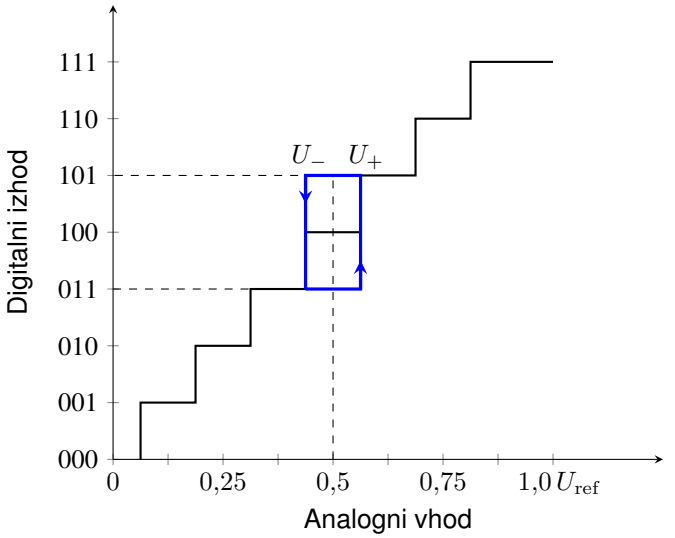
Ko gornji program poženemo, ugotovimo, da v določenih vmesnih stanjih zadnja črtica na zaslonu nekoliko migota. To je običajen pojav, kadar pretvarjamo zvezne analogne signale v diskretne vrednosti. Na naslednji sliki vidimo primer, ko je vhodna napetost enaka $U_{\pm} = U_{\mathrm{ref}}/2 - Q/2$:



V takšnem primeru lahko že najmanjši šum na vhodu povzroči nenadzorovano preklapljanje med vrednostma 100 in 011 na izhodu. Težave ne moremo popolnoma odpraviti niti s povprečenjem signala, saj lahko zadnji bit preskakuje tudi v povprečni vrednosti.

Histereza

V našem primeru je takšno migotanje zgolj moteče, v mnogih primerih pa je lahko tudi škodljivo. Predstavljajmo si termostat, ki vklaplja in izklaplja črpalko centralnega ogrevanja. Če je temperatura v prostoru enaka nastavljeni želeni temperaturi na termostatu, lahko začne termostat nenadzorovano vklapljati in izklapljati črpalko, ki jo bo zaradi tega hitro uničil. Rešitev v takšnih primerih je, da uporabimo ***histerezo*** (angl. hysteresis). Histereza je prenosna funkcija sistema, katere izhodna vrednost ni odvisna le od stanja na vhodu sistema, temveč tudi od njegove zgodovine. Naslednja slika prikazuje, kako lahko v naš sistem dodamo histerezo:



Izhodna vrednost lahko zdaj preskakuje med stanjema 101 in 011 le po modrih črtah v smeri puščic. Če odčitamo na primer na izhodu stanje 011, potem mora vhodna napetost narasti vse do napetosti $U_+ = U_{\mathrm{ref}}/2 + Q/2$, preden bomo na izhodu dobili odčitek 101. Ko se to enkrat zgodi, mora vhodna napetost pasti na vrednost $U_- = U_{\mathrm{ref}}/2 - Q/2$, da se lahko na izhodu stanje spremeni nazaj na 011.

Histerezo vgradimo v program tako, da odčitano stanje vsakokrat primerjamo s stanjem, ki se trenutno prikazuje na izhodu. Stanje na izhodu preklopimo šele, ko se odčitano stanje od njega dovolj razlikuje. Ker v našem programu prikazujemo le osem od 1024 različnih možnih stanj, si lahko privoščimo malo večjo histerezo. Vrednosti na izhodu pretvornika AD namreč kar precej nihajo, saj v sistemu nimamo zelo stabilne referenčne napetosti. Takole je videti naša funkcija `loop` z dodano histerezo:

```
void loop(void) {
  static int prikazanOdcitek = 0;
  char prikaz[ST_PRIKAZ + 1];
  int i, odcitek = analogRead(ANALOG_PIN);
  if (abs(odcitek - prikazanOdcitek) >= 10) {  /* Absolutna vrednost
                                                  razlike.            */
    prikazanOdcitek = odcitek;
    for (i = 0; i < ST_PRIKAZ; i++) {
      if (i <= ST_PRIKAZ * odcitek / 1024) {
        prikaz[i] = '-';
      }
```

```
      else {
        prikaz[i] = ' ';
      }
    }
    prikaz[i] = 0;
    g_lcd.setCursor(0,0);
    g_lcd.print(prikaz);
  }
  delay(20);
}
```

Program zdaj posodobi prikaz na prikazovalniku le, če se odčitana vrednost spremeni vsaj za deset. Širina histereze, ki smo jo tako uporabili, je enaka devetkratni napetosti bita LSB (na gornjem grafu je širina histereze enaka napetosti bita LSB).

Naloga 12.1 Za vajo napišite program za naslednji problem:

Mikrokrmilnik ima na vhodu svojega 12-bitnega analogno-digitalnega pretvornika z referenčno napetostjo 3,3 V priklopljen izhod merilnega vezja, ki temperaturno območje od 3 do 28 °C preslika (linearno) v območje napetosti od 0 do 3,3 V[a]. Poleg tega sta na dveh digitalnih sponkah mikrokrmilnika priklopljeni dve tipki (digitalni vhod), na eni digitalni sponki pa je priklopljeno stikalo grelne naprave (digitalni izhod). Napišite program, da bo sistem deloval kot termostat: pritisk na prvo tipko naj zmanjša, pritisk na drugo tipko pa poveča želeno temperaturo za desetinko stopinje. Termostat naj meri temperaturo ter po potrebi vklaplja oziroma izklaplja grelno napravo. V program dodajte še histerezo z razponom sedemkratne napetosti bita LSB.

Opomba: Sistem lahko simulirate tako, da namesto temperaturnega tipala priklopite potenciometer, namesto grelne naprave pa svetlečo diodo. Želeno in dejansko temperaturo lahko izpisujete bodisi na prikazovalnik LCD bodisi na serijski vmesnik.

[a] Pri temperaturi 3 °C imamo tako na izhodu vezja napetost 0 V, pri temperaturi 28 °C imamo na izhodu napetost 3,3 V, pri temperaturi (na primer) 10 °C pa imamo na izhodu vezja napetost $3{,}3(10-3)/(28-3) = 0{,}924$ V.

12.2 Digitalno-analogni pretvornik

Digitalno-analogni pretvornik (angl. digital-to-analog converter) opravlja obratno delo kot pretvornik AD: digitalni signal pretvarja v analognega. Mikrokrmilnik SAM3X ima vgrajen dvokanalni 12-bitni pretvornik DA. Njegova izhoda sta na sistemu Arduino DUE priklopljena na sponki DAC0 in DAC1, ki pa ju v tem učbeniku ne bomo uporabljali. Kadar nimamo zelo velikih zahtev glede pretvorbe signala v analogno obliko, lahko uporabimo veliko cenejši pristop z uporabo ***pulzno-širinske modulacije*** (angl. pulse-width modulation, PWM). Zato mnogi mikrokrmilniki ne vsebujejo pretvornika DA in tudi mi bomo pri svojem delu uporabili pristop s pulzno-širinsko modulacijo.

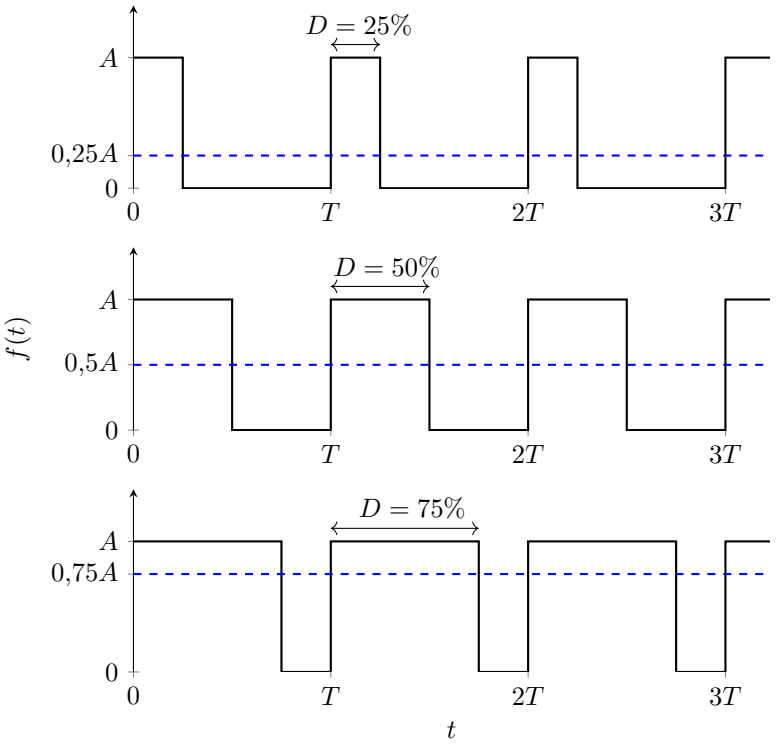
Pulzno-širinska modulacija

Osnovna ideja pulzno-širinske modulacije je ta, da analogni signal razdeli v vlak diskretnih impulzov, s čimer lahko brezizgubno krmilimo moč, ki se prenese na breme. V davni pre-

teklosti se je za krmiljenje motorjev uporabljal reostat[2], ki je bil povezan zaporedno z motorjem, ki ga je krmilil. Takšno krmiljenje je bilo neučinkovito, ker je porabljalo moč za nepotrebno segrevanje uporovnega elementa reostata. Krmiljenje s pulzi ne porablja skoraj nobene moči. Ko je stikalo izklopljeno, je vrednost toka praktično nič. Ko pa je stikalo vklopljeno, se domala vsa moč prenaša na breme, saj padca napetosti na stikalu skoraj ni. Izgubna moč na stikalu, ki je enaka produktu toka in napetosti, je zaradi tega v obeh primerih zanemarljiva.

Pri krmiljenju analognih naprav s signalom PWM je pomembno, da je frekvenca signala PWM dovolj velika v primerjavi s časovno konstanto naprave, ki jo krmilimo. Na primer, za krmiljenje električne pečice je dovolj, da se signal preklopi nekajkrat na minuto. Za zatemnilno stikalo (angl. dimmer) mora biti frekvenca signala PWM 120 Hz. Za krmiljenje električnega motorja potrebujemo signal frekvence od nekaj kHz pa do nekaj deset kHz, za avdio signal pa nekaj sto kHz.

Pri pulzno-širinski modulaciji je pomemben pojem ***delovne periode*** (angl. duty cycle), ki predstavlja odstotek časa, ko je signal vklopljen. Naslednja slika prikazuje tri signale, ki imajo delovno periodo D po vrsti enako 25, 50 in 75 odstotkov:



Z modro prekinjeno črto je prikazana povprečna vrednost signala, ki se spreminja sorazmerno z delovno periodo po naslednji enačbi:

$$\overline{f(t)} = DA,$$

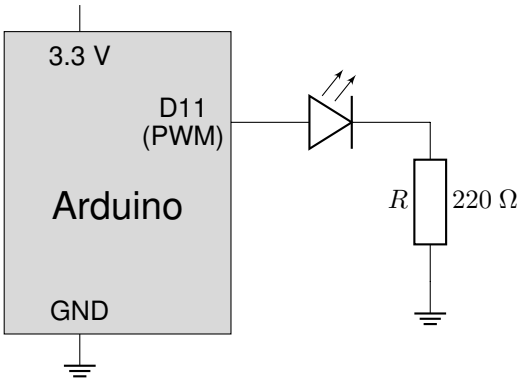
[2]Reostat je spremenljiv upor z dvema priključkoma. Beseda *reostat* se v sodobni terminologiji uporablja čedalje redkeje, nadomešča pa jo splošnejši izraz *potenciometer*.

pri čemer je A amplituda signala.

Utripanje svetleče diode, tretjič

Na različnih sistemih Arduino lahko na določenih sponkah proizvedemo signal PWM, katerega privzeta frekvenca se giblje med 490 Hz in 1 kHz, odvisno od sistema. Frekvenco lahko glede na tip uporabljenega krmilnika spreminjamo, vendar se na tem mestu v podrobnosti tega početja ne bomo spuščali. Prav tako bomo uporabili privzeto ločljivost signala, ki je enaka osem bitov. Arduinovo integrirano razvojno okolje pozna funkcijo `analogWrite`, ki sprejme dva argumenta: številko sponke in celoštevilsko vrednost med 0 in 255. Funkcija na podani sponki povzroči signal PWM z delovno periodo, sorazmerno podani vrednosti. Pri tem predstavlja vrednost nič delovno periodo $D = 0\%$, vrednost 255 pa delovno periodo $D = 100\%$.

Za primer se še enkrat vrnimo k vezju s svetlečo diodo, ki jo tokrat priklopimo na sponko D11:



Še enkrat bomo napisali program, ki bo povzročil utripanje svetleče diode, le da bo prižiganje in ugašanje diode tokrat postopno. Takšno utripanje bomo dosegli tako, da bomo v neskončni zanki na približno vsaki 2 ms povečali vrednost spremenljivke `pwm` za ena, dokler ne bomo prišli do vrednosti 255. Potem bomo na približno vsaki 2 ms zmanjšali vrednost spremenljivke `pwm` za ena, dokler ne bomo prišli do vrednosti nič. Tako bo frekvenca utripanja približno 1 Hz:

```
#define LED_PIN 11

void loop(void) {
  static int pwm = 0;
  static int dPwm = 1;         /* Smer spreminjanja spremenljivke pwm.    */
  analogWrite(LED_PIN, pwm);   /* Na sponki 11 povzroči signal PWM
                                  z delovno periodo D=pwm/255.            */
  pwm += dPwm;
  /*  Če smo na zgornji ali spodnji meji, popravimo smer spreminjanja:  */
  if (pwm == 255) {
    dPwm = -1;
  }
  if (pwm == 0) {
    dPwm =  1;
  }
  delay(2);
}
```

Naloga 12.2 Mnoge naprave s svetlečimi prikazovalniki imajo funkcijo, s katero lahko zmanjšamo njihovo svetilnost, da ne bi bile ponoči preveč moteče s svojo svetlobo. Predpostavite, da je na analognih izhodih (PWM) sistema Arduino priklopljen sedemsegmentni prikazovalnik, na eni njegovi digitalni sponki pa je priklopljena tipka. Predelajte program, ki ste ga napisali kot rešitev naloge 11.1 na strani 181, tako da se bo prikaz z vsakokratnim pritiskom na tipko nekoliko zatemnil. Vsak četrti zaporedni pritisk tipke naj namesto tega povrne največjo svetilnost prikazovalnika.

Opomba: Če ima vaš Arduino samo šest sponk PWM, potem lahko nalogo še vedno rešite, le da štejete po modulu sedem (tj. od nič do šest). Pri številkah od nič do šest imata namreč diodi a in d na sedemsegmentnem prikazovalniku popolnoma enaki stanji in ju lahko vežete na isto sponko. Pri tem bodite pozorni, da ne vežete dveh diod vzporedno, temveč naj bo vsaka od obeh diod vezana preko svojega zaporednega zaščitnega upora. Če namreč vežemo dve diodi vzporedno, bo napetost na obeh diodah enaka. Zaradi majhnih razlik v kolenskih napetostih (angl. knee voltage) diod, ki izvirajo iz neidealnosti izdelave, je lahko zaradi tega tok skozi obe diodi precej različen (zaradi zelo strme tokovno-napetostne karakteristike). To pa pomeni, da bosta diodi svetili različno močno.

PRAZNA STRAN

13. POGLAVJE

VGRAJENI SISTEMI

Vgrajen sistem (angl. embedded system) je mikrokrmilniški sklop, posvečen točno ***določenemu opravilu*** znotraj večjega mehanskega ali električnega sistema, v katerega je ***vgrajen***. Od vgrajenega sistema se običajno zahteva, da deluje v ***realnem času*** (angl. real time). Ker vgrajeni sistemi opravljajo točno določene funkcije, jih je mogoče optimizirati glede na velikost in ceno ter hkrati povečati njihovo zanesljivost in učinkovitost. Vgrajene sisteme srečamo malodane povsod: od digitalnih ur in prenosnih telefonov, preko avtomobilskih sistemov, obsežnih stacionarnih postavitev v proizvodnih linijah, do zapletenih sklopov v letalskih in vesoljskih sistemih. Po nekaterih ocenah je okrog 90 odstotkov proizvedenih mikroprocesorjev in mikrokrmilnikov uporabljenih v vgrajenih sistemih.

13.1 Sistemi v realnem času in večopravilni sistemi

Z vgrajenimi sistemi sta tesno povezana pojma ***sistem v realnem času*** (angl. real-time system) in ***večopravilnost*** (angl. multitasking), ki ju bomo na tem mestu spoznali le bežno.

Sistem v realnem času Za sistem v realnem času je pomembno, da se na določene dogodke odzove v točno predpisanih časovnih okvirih. Navadno gre pri tem za kratke čase v območju mili- ali celo mikrosekund, ni pa to nujno. Časovni okviri, ki jih postavimo na sistem v realnem času, pogosto določajo tako zgornjo kot tudi spodnjo časovno omejitev: dogodek se ne sme zgoditi prepozno, ne sme pa se zgoditi niti prezgodaj. Pravilnost delo-

vanja sistemov v realnem času je torej pogojena tako z njihovo funkcionalnostjo kot tudi s pravočasnostjo odziva.

Včasih kot sistem v realnem času razumemo tudi sistem, ki se odzove »brez občutne zakasnitve«, pri simulacijah pa govorimo o realnem času takrat, ko teče ura v simulaciji enako hitro kot resnična ura.

Večopravilni sistem Večopravilni sistem je sistem, ki je v določenem času sposoben izvajati več opravil (ali procesov) hkrati. Kadar poganjamo opravila na enem samem procesorju, lahko to dosežemo na en sam način: namesto da bi računalnik čakal, da se eno opravilo konča, preden se lahko začne naslednje, se opravila med seboj prekinjajo. Ker je izvajanje dovolj hitro, daje tak sistem navzven občutek, da se opravila izvajajo hkrati. Najenostavnejši pristop k načrtovanju večopravilnega sistema smo že spoznali: sistem Arduino izvaja opravila v neskončni zanki, čemur pravimo ***ciklično izvajanje*** (angl. cyclic executive). Tako smo na primer hkrati brali vrednost z analognega vhoda in jo prikazovali na prikazovalniku tako, da smo v vsakem obhodu zanke drugo za drugim postorili oboje.

Primer: digitalna ura

Kot primer vgrajenega sistema si bomo ogledali načrtovanje digitalne ure. Čeprav digitalna ura na videz ni zapleten sistem, pa mora vseeno izpolnjevati obe pomembni zahtevi, ki ju pričakujemo od večine vgrajenih sistemov: delovati mora v realnem času in sposobna mora biti opravljati več opravil hkrati. V prvem koraku načrtovanja digitalne ure bomo poskrbeli le za dve opravili, ki ju bo sistem izvajal ciklično: povečevati je treba števec sekund in osveževati prikazani čas. Prikazani čas lahko osvežujemo brez časovnih omejitev v vsakem obhodu zanke, števec sekund pa moramo povečati enkrat na sekundo. Slednjega iz dveh razlogov ne moremo narediti z uporabo funkcije `delay`:

- Če bi hoteli, da vsak obhod zanke traja natanko eno sekundo, bi morali najprej vedeti, koliko od tega časa se porabi za izvajanje kode. Preostali čas bi zapolnili s klicem funkcije `delay`. Vendar je praktično nemogoče natančno izračunati, koliko časa se koda izvaja. Še zlasti, ker se časi posameznih obhodov zaradi različnih drugih opravil med seboj večinoma razlikujejo.
- Uri bomo kasneje dodali tudi tipke, s katerimi bo moč uro nastavljati. Uporabnik lahko pritisne tipko kadarkoli: tudi med tem, ko s funkcijo `delay` »čakamo«, da mine ena sekunda. Med izvajanjem funkcije `delay` pa ne moremo izvajati druge kode, torej tudi ne moremo brati tipk in se odzivati na pritiske. Ker traja pritisk tipke tipično od nekaj stotink do nekaj desetink sekunde, lahko tako precej pritiskov zgrešimo.

Za merjenje časa lahko namesto funkcije `delay` uporabimo funkcijo `millis`, ki nam ob vsakem klicu vrne število milisekund, ki so minile od trenutka vklopa sistema. S pomočjo te funkcije ob vsakem obhodu zanke zgolj preverimo, ali je že minila sekunda od trenutka zadnje spremembe, ki ga bomo hranili v spremenljivki `oznakaSekunde`. Če je sekunda že minila, potem povečamo števec sekund za ena. Hkrati povečamo vrednost spremenljivke `oznakaSekunde` za 1000. Takole je videti del programa, ki vsako sekundo poveča števec sekund:

```
if (millis() - oznakaSekunde >= 1000) {
  sekunde++;
  oznakaSekunde += 1000;
}
```

Še vedno se lahko zgodi, da pridemo zaradi izvajanja preostale kode do gornjega stavka prepozno (tj. kasneje kot v 1000 milisekundah), kar pa nas ne skrbi. Zakasnitev ne bo večja od nekaj milisekund, kar se pri prikazovanju časa ne bo opazilo. Pomembno je predvsem to, da se zakasnitev ne kopiči. To dosežemo tako, da spremenljivko `oznakaSekunde` vsakokrat povečamo za 1000. Na ta način premaknemo časovno oznako vsakokrat natanko za sekundo naprej, zaradi česar se napake zaradi posameznih zakasnitev ne bodo seštevale. S tem ko smo cikličnemu izvajanju dodali merjenje časa, smo dobili tako imenovano ***časovno vodeno ciklično izvajanje*** (angl. time-driven cyclic executive).

Napisati moramo še funkcijo za prikaz časa, ki pa časa ne bo prikazovala neposredno na prikazovalnik. Namesto tega bo zapisala ustrezno oblikovan izpis v znakovni niz, ki ga bo sprejela kot prvi parameter. Na ta način bo funkcija prenosljiva, kar pomeni, da jo bomo lahko uporabili v različnih programih na različnih sistemih. Če bi funkcija izpisovala čas neposredno na prikazovalnik, bi jo morali za uporabo na kakšnem drugem sistemu predelati. Predelati bi jo morali že, če bi hoteli čas izpisovati na primer na serijski vmesnik.

Poleg znakovnega niza, kamor bomo vpisali ustrezno oblikovan čas, bo funkcija sprejela še naslednje parametre: število sekund, ki so pretekle od polnoči, 1. januarja 1970 (Unixov čas, glej primer na strani 85), način prikaza (12- ali 24-urni prikaz), ter podatka o tem, ali naj prikaže ure oziroma minute. Slednje je treba zato, da bodo lahko bodisi ure bodisi minute na prikazovalniku utripale, kar bomo potrebovali pri nastavljanju ur in minut.

Takole je videti napisana funkcija:

```
void oblikujUro(char *buff, time_t s,
                bool nacin12, bool pokaziH, bool pokaziM) {
/*************************************************************
* Funkcija sprejme čas kot število sekund, ki so pretekle od  *
*  polnoči, 1. 1. 1970 (Unixov čas) ter ga pretvori v zapis   *
*  z urami, z minutami in s sekundami.                        *
*                                                             *
* Vhodni parametri:                                          *
*  s       - Unixov čas, ki ga je treba prikazati.           *
*  nacin12 - Način prikaza (12- ali 24-urni). Na primer:     *
*            false: 14:35:08                                  *
*            true:   2:35:08 pm                               *
*  pokaziH - Določa, ali bodo prikazane ure. Na primer:      *
*            false:   :35:08                                  *
*            true:  14:35:08                                  *
*  pokaziM - Določa, ali bodo prikazane minute. Na primer:   *
*            false: 14:  :08                                  *
*            true:  14:35:08                                  *
*                                                             *
* Izhodni parameter:                                         *
*  buff    - Znakovni niz dolžine vsaj 12 bajtov.            *
*            Vanj se zapiše oblikovana ura.                   *
*************************************************************/
  char amPm[] = "  ";
  int h = ure(s);
  if (nacin12) {
    if (h < 12) {  /* 12:00 am - polnoč, 12:00 pm - poldan   */
      strcpy(amPm, "am");
    }
    else {
      strcpy(amPm, "pm");
    }
    h %= 12;
    if (h == 0) {
      h = 12;
    }
```

```
  }
  sprintf(buff, "%2d:%02d.%02d%3s", h, minute(s), sekunde(s), amPm);
  if (pokaziH == false) buff[0] = buff[1] = ' ';
  if (pokaziM == false) buff[3] = buff[4] = ' ';
}
```

V gornji definiciji smo uporabili funkcije `ure`, `minute` in `sekunde`, ki iz podanega Unixovega časa izluščijo ure, minute ali sekunde:

```
int ure(time_t s) {
  return (s / 3600) % 24;
}

int minute(time_t s) {
  return (s / 60) % 60;
}

int sekunde(time_t s) {
  return s % 60;
}
```

Funkcija `oblikujUro` v primeru, ko izberemo 24-urni prikaz, enostavno oblikuje čase od `0:00.00` do `23:59.59`. Pri 12-urnem prikazu pa mora poskrbeti, da za uro namesto ničle izpiše 12 ter da loči dopoldanski čas (am) od popoldanskega (pm). Pri tem velja dogovor, da pomeni zapis `12:00.00 am` polnoč, zapis `12:00.00 pm` pa poldan. Za oblikovanje zapisa smo uporabili standardno cejevsko funkcijo `sprintf`, ki deluje enako kot funkcija `printf`, le da zapisuje oblikovano besedilo v znakovni niz, ki ji ga podamo kot prvi argument.

Zdaj lahko sestavimo vse skupaj v naslednji program, ki prikazuje čas v 24-urnem zapisu:

```
#include <LiquidCrystal.h>
LiquidCrystal g_lcd(27, 28, 29, 30, 31, 32);

void setup(void) {
  g_lcd.begin(16, 2);
}

void loop(void) {
  static time_t sekunde = 1577836800;  /* Polnoč, 1. 1. 2020 UTC  */
  static unsigned long oznakaSekunde = 0;
  char prikaz[12];
  if (millis() - oznakaSekunde >= 1000) {
    oznakaSekunde += 1000;
    sekunde++;
  }
  lcd.setCursor(0, 0);
  oblikujUro(prikaz, sekunde, false, true, true);
  g_lcd.print(prikaz);
}
```

Ko gornji program zaženemo, začne prikazovati čas od polnoči naprej. Čeprav smo nastavili začetno vrednost spremenljivke `sekunde` na polnoč, 1. januarja 2020, pa to v našem programu ne igra nobene vloge, saj datuma pri prikazu ne upoštevamo.

Preobrat števca in težava leta 2038

Funkcija `millis` vrne število milisekund, ki so minile od vklopa sistema. Ker je ta vrednost tipa `unsigned long`, ki je na sistemu Arduino 32-biten, to pomeni, da bo števec

milisekund dosegel svojo končno vrednost po malo več kot 49 dneh. Potem se zgodi ***preobrat*** (angl. rollover) števca, kar pomeni, da začne števec znova šteti od nič naprej. Ali to pomeni, da bo naša ura takrat prenehala delovati?

Ker vemo, kako se v takšnem primeru vedejo nepredznačena cela števila (glej primer računanja z uro na strani 50), nas preobrat števca milisekund v našem konkretnem primeru ne skrbi: tudi ko se preobrat zgodi, bo vrednost izraza `millis() - oznakaSekunde` še vedno pravilna. Na primer, če se je zadnja sprememba zgodila v času 4 294 967 016 milisekund, potem bo funkcija `millis` čez eno sekundo vrnila vrednost 720. Vrednost nepredznačenega izraza $720 - 4\,294\,967\,016$ pa je enaka 1000.

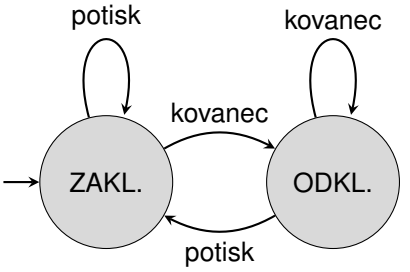
Drugo vprašanje je, kaj se bo zgodilo, ko bo svojo največjo vrednost dosegel števec sekund, ki je v mnogih sistemih zapisan kot 32-bitno predznačeno celo število. Na tak način ne moremo zapisati časov, ki so kasnejši od 19. januarja 2038 ob 3:14.07 UTC. Omenjena težava je znana pod imenom ***težava leta 2038*** (angl. year 2038 problem).

13.2 Končni avtomati

Čas je, da naši uri dodamo tipkovnico, da jo bomo lahko nastavljali. Pri tem bomo uporabili le tri tipke, s katerimi bomo nastavljali ure in minute ter preklapljali med 12- in 24-urnim prikazom. Vendar naloga, ki je pred nami, kljub temu ni povsem nedolžna. Kaj hitro se zapletemo, če ne uporabimo posebnega pristopa k načrtovanju takšnih sistemov. ***Končni avtomat*** (angl. finite automaton oz. finite-state machine oz. state machine) je matematični model abstraktnega računskega stroja, ki se s pridom uporablja tudi pri načrtovanju vgrajenih sistemov. Končni avtomat je lahko v vsakem trenutku v enem od svojih možnih ***stanj***, ki jih je končno število. Iz enega v drugo stanje lahko prehaja kot posledica določenih (zunanjih) dražljajev. Končni avtomat je tako določen s seznamom svojih stanj, z začetnim stanjem ter s pogoji za prehajanje med stanji.

Vzemimo za primer preprost končni avtomat, ki ponazarja delovanje vrtljivega križa (angl. turnstile). Vrtljivi križ je preprosta prepreka, sestavljena iz treh vrtljivih palic, ki jih s potiskom zavrtimo in tako vstopimo v stavbo ali na določeno območje. Vrtljivi križ se pogosto uporablja na vhodih v muzeje ali športne stadione.

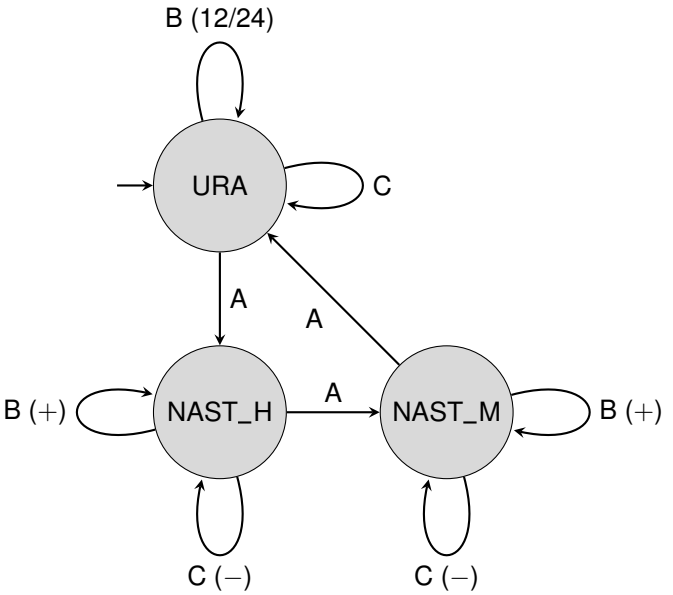
Naslednja slika prikazuje delovanje vrtljivega križa v obliki končnega avtomata:



Dve možni stanji (zaklenjeno in odklenjeno) sta prikazani z dvema krogoma, prehodi med njima pa s puščicami. Začetno stanje je zaklenjeno, kar je prikazano s puščico, ki v to stanje vstopa z leve. Ko je križ v zaklenjenem stanju, potisk palice nima učinka, kar je prikazano s puščico, ki se vrne nazaj v isto stanje. Če v režo vstavimo kovanec (ali približamo kartico), se križ odklene, kar je prikazano s puščico (z napisom *kovanec*), ki vodi iz zaklenjenega v odklenjeno stanje. Če zdaj vstavimo še en kovanec (ali spet pri-

bližamo kartico), se ne zgodi nič, saj smo za vstop že plačali. Zato se puščica z napisom *kovanec*, ki vodi iz odklenjenega stanja, vrne nazaj v isto stanje. Po drugi strani potisk palice spusti mimo eno osebo in križ spet zaklene.

Vrnimo se k naši digitalni uri, ki smo ji zdaj dodali tri tipke A, B in C, s katerimi bomo uro upravljali. Naslednji diagram prikazuje končni avtomat, ki predstavlja delovanje takšne ure:



Ura je lahko v enem od treh stanj, pri čemer je začetno stanje običajen prikaz časa (stanje URA). V tem stanju tipka C nima učinka, s tipko B pa preklapljamo med 12- in 24-urnim prikazom. Tipka A z vsakim pritiskom preklopi sistem v naslednje stanje. S pritiskom na to tipko pridemo iz stanja prikazovanja časa v stanje, kjer lahko nastavljamo ure (stanje NAST_H). V tem stanju na prikazovalniku ure utripajo, s pritiskom na tipki B in C pa lahko ure bodisi večamo bodisi manjšamo. Z vnovičnim pritiskom na tipko A pridemo v stanje, kjer lahko nastavljamo minute (stanje NAST_M). V tem stanju utripajo minute, ki jih spet lahko nastavljamo s tipkama B in C. Tretji pritisk na tipko A vrne uro v začetno stanje običajnega prikazovanja časa.

Kodo, ki uresniči delovanje gornjega končnega avtomata, najdemo v spodnjem programu v funkciji `upravljajStanja`. Funkcija kot vhodni parameter (parameter `vhod`) sprejme kodo pritisnjene tipke (`'A'`, `'B'` ali `'C'`). Ostali trije parametri (`stanje`, `sek` in `nacin12`) so hkrati vhodni in izhodni, zato so določeni v obliki kazalcev. Ti trije parametri namreč hranijo pomembne informacije o stanju sistema, ki jih funkcija `upravljajStanja` spreminja glede na vhod in stanje avtomata: poleg osnovnih treh stanj sistema, ki so prikazana v gornjem diagramu (parameter `stanje`), določata stanje sistema še trenutni čas (Unixov čas, parameter `sek`) in način prikaza (12- ali 24-urni, parameter `nacin12`).

Takole je videti celoten program za digitalno uro:

```
#include <LiquidCrystal.h>
```

```
/*****************************************************************
* Stanja:                                                        *
*****************************************************************/
#define URA 0
#define NAST_H 1
#define NAST_M 2
#define ST_STANJ 3

/*****************************************************************
* Priklop tipkovnice:                                            *
*****************************************************************/
#define ST_VRSTIC 4
#define ST_STOLPCEV 4
int g_st[ST_STOLPCEV] = {38, 39, 40, 41};
int g_vr[ST_VRSTIC]   = {42, 43, 44, 45};
char g_tipkovnica[ST_VRSTIC][ST_STOLPCEV + 1] = {"123A",
                                                 "456B",
                                                 "789C",
                                                 "*0#D"};

/*****************************************************************
* Priklop LCD-ja:                                                *
*****************************************************************/
LiquidCrystal g_lcd(27, 28, 29, 30, 31, 32);

/*****************************************************************
* Branje s tipkovnice:                                           *
*****************************************************************/
void izberiVrstico(int v) {
  for (int i = 0; i < ST_VRSTIC; i++) {
    if (i == v) {
      pinMode(g_vr[i], OUTPUT);
      digitalWrite(g_vr[i], LOW);
    }
    else {
      pinMode(g_vr[i], INPUT);
    }
  }
}

char beriZnak(void) {
  for (int v = 0; v < ST_VRSTIC; v++) {
    izberiVrstico(v);
    for (int s = 0; s < ST_STOLPCEV; s++) {
       if (digitalRead(g_st[s]) == LOW) {
         return g_tipkovnica[v][s];
       }
    }
  }
  return 0;
}

/*****************************************************************
* Oblikovanje izpisa:                                            *
*****************************************************************/
int ure(time_t s) {
  return (s / 3600) % 24;
}
```

```
int minute(time_t s) {
  return (s / 60) % 60;
}

int sekunde(time_t s) {
  return s % 60;
}

void oblikujUro(char *buff, time_t s,
                bool nacin12, bool pokaziH, bool pokaziM) {
  char amPm[] = "  ";
  int h = ure(s);
  if (nacin12) {
    if (h < 12) {  /* 12:00 am - polnoč, 12:00 pm - poldan        */
      strcpy(amPm, "am");
    }
    else {
      strcpy(amPm, "pm");
    }
    h %= 12;
    if (h == 0) {
      h = 12;
    }
  }
  sprintf(buff, "%2d:%02d.%02d%3s", h, minute(s), sekunde(s), amPm);
  if (pokaziH == false) {
    buff[0] = buff[1] = ' ';
  }
  if (pokaziM == false) {
    buff[3] = buff[4] = ' ';
  }
}

/*********************************************************************
* Končni avtomat:                                                    *
*********************************************************************/
void upravljajStanja(char vhod, int *stanje,
                     time_t *sek, bool *nacin12) {
  switch (*stanje) {
    case URA:
      switch (vhod) {
        case 'A': *stanje = (*stanje + 1) % ST_STANJ; break;
        case 'B': *nacin12 = !*nacin12; break;
        case 'C': break;
      }
      break;
    case NAST_H:
      switch (vhod) {
        case 'A': *stanje = (*stanje + 1) % ST_STANJ; break;
        case 'B': *sek += 3600; break;
        case 'C': *sek -= 3600; break;
      }
      break;
    case NAST_M: {
      int dMin = -1;  /* V katero smer spreminajmo minute.        */
      int h;
      switch (vhod) {
        case 'A': *stanje = (*stanje + 1) % ST_STANJ; break;
        case 'B': dMin = 1;
        case 'C':
        /* Če smo pritisnili B, ima dMin zdaj vrednost 1, če smo   *
```

```
        *  pritisnili C, pa ima dMin vrednost -1. Naslednji del    *
        *  kode omogoča krožno spreminjanje minut. To pomeni, da   *
        *  vrednosti 59 sledi nula (oz. nuli sledi 59, če minute   *
        *  manjšamo). Pri tem ne želimo, da se spremenijo tudi     *
        *  ure. Če se ure po nakljućju spremenijo, jih na koncu    *
        *  popravimo na prejšnjo vrednost:                         */
                h = ure(*sek);
                *sek += dMin * 60;
                if (h != ure(*sek)) {
                  *sek -= dMin * 3600;
                }
      }
      }
      break;
  }
}

/*********************************************************************
* Inicializacija:                                                    *
*********************************************************************/
void setup(void) {
  g_lcd.begin(16, 2);
  for (int i = 0; i < ST_STOLPCEV; i++) {
    pinMode(g_st[i], INPUT_PULLUP);
  }
}

/*********************************************************************
* Glavna zanka:                                                      *
*********************************************************************/
void loop(void) {
  static unsigned long oznakaSekunde = 0;
  const  unsigned int  korakSekunde = 1000;
  static unsigned long oznakaTipke = 0;
  const  unsigned int  korakTipke = 50;
  static char zadnjaTipka = 0;
  static int stanje = URA;
  static bool nacin12 = false;
  static time_t sekunde = 1577836800;  /* Polnoč, 1. 1. 2020 UTC  */

  bool kaziMinute;
  bool kaziUre;
  char prikaz[12];

  /*******************************************************************
  * Vsakih 50 ms beri tipke in ukrepaj:                              *
  *******************************************************************/
  if (millis() - oznakaTipke >= korakTipke) {
    char tipka = beriZnak();
    oznakaTipke += korakTipke;
    if (zadnjaTipka != tipka) {
      zadnjaTipka = tipka;
      upravljajStanja(tipka, &stanje, &sekunde, &nacin12);
    }
  }

  /*******************************************************************
  * Vsakih 1000 ms povečaj sekunde za ena:                           *
  *******************************************************************/
  if (millis() - oznakaSekunde >= korakSekunde) {
    oznakaSekunde += korakSekunde;
```

```
    sekunde++;
  }

  /*****************************************************************
  * Vsakič, ko je minilo več kot 500 ms od zadnjega preklopa       *
  * sekund, ugasni ure ali minute, če smo v stanju nastavljanja    *
  * ur ali minut:                                                  *
  *****************************************************************/
  kaziUre = kaziMinute = true;
  if (millis() - oznakaSekunde > 0.5 * korakSekunde) {
    switch (stanje) {
      case NAST_H: kaziUre = false; break;
      case NAST_M: kaziMinute = false; break;
    }
  }

  /*****************************************************************
  * Ob vsakem obhodu zanke osveži prikazovalnik:                   *
  *****************************************************************/
  g_lcd.setCursor(0, 0);
  oblikujUro(prikaz, sekunde, nacin12, kaziUre, kaziMinute);
  g_lcd.print(prikaz);
}
```

V glavni zanki gornjega programa opazimo poleg štetja sekund še dve opravili, vezani na čas. Prvo takšno opravilo je branje tipkovnice, ki ga izvajamo vsakih 50 ms. S tem se zagotovo izognemo odskakovanju, še vedno pa je branje dovolj pogosto, da pritiska tipke ne bomo spregledali. Tudi če ne bi bilo odskakovanja, tipkovnice ni smiselno brati pogosteje, saj s tem po nepotrebnem zapravljamo čas. Če bi programu dodali še več opravil, bi lahko zaradi prepogostega branja tipkovnice (ki je v naši izvedbi dokaj počasno) odziv sistema v določenih točkah postal nezadosten.

Drugo na čas vezano opravilo je ugašanje in prižiganje (utripanje) prikazanih ur oziroma minut v primeru, ko se sistem nahaja v stanju, ki omogoča nastavljanje ur oziroma minut. To opravilo se izvaja dvakrat na sekundo, čeprav je tehnično izvedeno drugače: utripanje dosežemo tako, da v primeru, da je pretekla več kot polovica časa od zadnjega preklopa sekund, v vsakem obhodu zanke postavimo ustrezno spremenljivko na vrednost `false`. Kasneje vrednost te spremenljivke odloči, ali se bodo ure oziroma minute prikazale ali ne.

Zadnje opravilo je osveževanje prikazanega časa na prikazovalniku. To opravilo se izvaja v vsakem obhodu zanke, da se, kolikor je mogoče hitro, pokažejo vse spremembe, ki so se zgodile v tem obhodu. Ker je obhod zanke dovolj hiter, bo uporabnik dobil občutek, da ura nemoteno teče (in utripa v primeru, ko jo nastavljamo), hkrati pa se odzove na vsak pritisk tipke praktično v hipu.

Opozorimo še na to, kako v gornjem programu uporabljamo globalne in statične spremenljivke. Vse spremenljivke, ki jih potrebujemo samo v funkciji `loop` in morajo med posameznimi klici funkcije ohraniti svojo vrednost, smo deklarirali kot statične lokalne spremenljivke. Globalnim spremenljivkam se še vedno skušamo izogniti v kar največji meri. V globalnih spremenljivkah hranimo le podatke, ki so neposredno vezani na sistem, kot so na primer podatki o priključnih sponkah ali objekti, ki zastopajo priključene naprave. Zaradi preglednosti smo v skladu z madžarskim zapisom dodali imenom globalnih spremenljivk predpono `g_`.

Preden nadaljujemo, razmislite o naslednjih izzivih:

Naloga 13.1 Dopolnite program za digitalno uro na strani 218, tako da bo mogoče nastavljati tudi sekunde. Stanje nastavljanja sekund naj sledi stanju nastavljanja minut, vanj pa naj sistem preide s pritiskom na tipko A. Ko je sistem v stanju nastavljanja sekund, naj prikaz sekund utripa, pritisk na tipko B pa naj sekunde postavi na nič. Pritisk na tipko A v stanju nastavljanja sekund naj vrne sistem v stanje prikazovanja časa.

Naloga 13.2 Program za digitalno uro na strani 218 deluje tako, da ob pritisku na tipko A v stanju prikazovanja časa že omogoči nastavljanje ure. Takšno obnašanje v praksi ni zaželeno, saj lahko že kratek nenamerni pritisk tipke A in potem še katere od tipk B ali C (npr. če imamo takšno uro v žepu) povzroči, da se bo ura nastavila na napačno vrednost. Predelajte program, tako da bo treba za prehod iz stanja URA v stanje NAST_H držati tipko A vsaj dve sekundi.

Naloga 13.3 Programu za digitalno uro na strani 218 dodajte možnost prikaza in nastavljanja datuma. Program trenutno ne deluje za negativne vrednosti spremenljivke `sekunde` (tj. za datume pred 1. januarjem 1970). Popravite program tako, da bo deloval tudi za te datume.

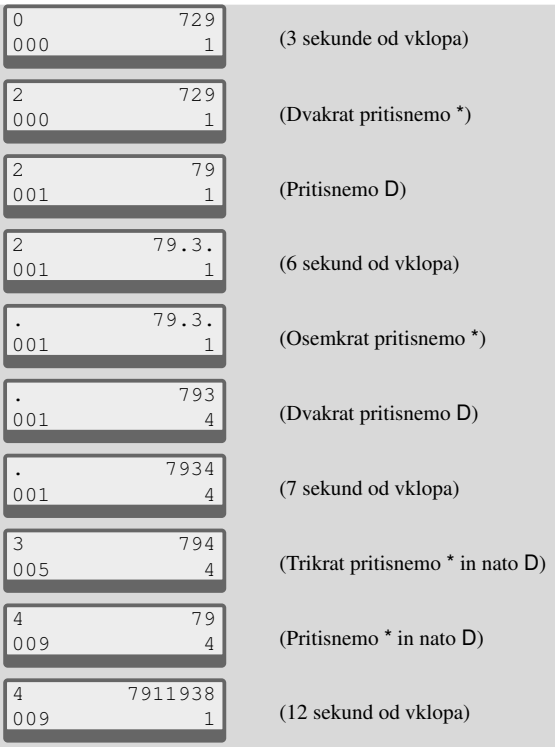
Naloga 13.4 Na sistemu Arduino imamo priklopljen prikazovalnik LCD in matrično tipkovnico. Napišite program za računalniško igrico izbijanja znakov iz naključnega znakovnega niza.

Na prikazovalniku naj se na levi strani ves čas prikazuje »ključ«, ki predstavlja bodisi desetiško številko med nič in devet bodisi piko. Igralec naj ima možnost spreminjati vrednost ključa s pritiskom na tipko *, in sicer tako, da vsak pritisk na to tipko poveča vrednost ključa za ena. Pri tem naj devetki sledi pika in potem spet ničla.

Z desne strani prikazovalnika naj se v določenih časovnih razmikih pomika niz iz naključnih števk med nič in devet ter pike. Če se ob določenem trenutku ključ ujema s katerim od znakov v naključnem nizu, lahko igralec s pritiskom na tipko D iz niza na desni odstrani ta znak. Pri tem naj se vsi znaki, ki so levo od odstranjenega znaka, pomaknejo za eno mesto proti desni. Vsaka odstranjena številka naj šteje eno točko. Če igralec odstrani piko, naj to ne prinese točk, temveč povzroči, da v naslednjih petih sekundah vsaka odstranjena številka šteje dvojno. Če v teh petih sekundah igralec znova odstrani piko, naj se število točk za vsako odstranjeno številko še podvoji, čas, ko ta podvojitev velja, pa naj spet traja pet sekund. Ko to obdobje petih sekund enkrat poteče, ne da bi igralec medtem iz niza izbil kakšno piko, naj se število točk, ki jih šteje ena izbita številka, postavi nazaj na ena. V drugi vrstici prikazovalnika naj se na levi strani prikazuje skupno število doseženih točk, na desni strani pa naj se prikazuje število točk, ki jih dobi igralec za vsako odstranjeno številko. Igra naj se konča, ko potujoči niz naključnih znakov na prikazovalniku doseže ključ. Pritisk na tipko A naj postavi sistem v začetno stanje.

Primer delovanja programa:

(Vklop / Začetno stanje)



13.3 Prekinitve

Videli smo, da je z uporabo preprostega časovno vodenega cikličnega izvajanja mogoče zgraditi večopravilni sistem, ki deluje v realnem času. Tak sistem je enostaven za programiranje, ni pa vedno učinkovit. Na primer, branje tipke smo dosegli tako, da smo stanje tipkovnice preverjali na vsakih 50 ms. Takšnemu načinu preverjanja stanja določene naprave pravimo ***poizvedovanje*** (angl. polling). Medtem ko je poizvedovanje relativno učinkovita metoda branja tipkovnice, pa postane popolnoma neuporabno pri opravilih, ki se pojavljajo pogosteje ali zahtevajo veliko hitrejše odzive. Zato delovanje malodane vsakega sodobnega vgrajenega sistema upravljamo tudi s ***prekinitvami*** (angl. interrupt). Prekinitev je zahteva po takojšnjem odzivu, ki jo krmilniku pošlje bodisi strojna bodisi programska oprema[1]. Ko krmilnik prejme takšno zahtevo, mora takoj prekiniti izvajanje glavnega programa in mora začeti izvajati tako imenovano ***prekinitveno rutino*** (angl. interrupt service routine, ISR). Prekinitvena rutina je v resnici funkcija, ki vsebuje kodo, za katero želimo, da se izvede kot posledica določenega dogodka. Preden pa sistem kliče prekinitveno rutino, mora shraniti svoje celotno stanje, da lahko po povratku iz prekinitve nemoteno nadaljuje izvajanje glavnega programa. Zaradi načina izvajanja mora prekinitvena rutina izpolnjevati vsaj naslednja dva pogoja:

[1]Možnih virov zunanjih (strojnih) prekinitev je veliko: od signala za vnovični zagon sistema (angl. reset), signala nadzornika (angl. watchdog timer), do signala ob dokončani analogno-digitalni pretvorbi. Tudi program, ki se izvaja na krmilniku, lahko proži svoje prekinitve. Na primer, za svoje izvajanje lahko program potrebuje vrednost, ki je v določenem območju. Če takšne vrednosti ne dobi in je za nadaljevanje njegovega izvajanja potrebno takojšnje ukrepanje, lahko to sporoči sistemu z zahtevo po prekinitvi.

- Ker prekinitveno rutino kliče sistem neodvisno od izvajanja glavnega programa (tj. ***asinhrono***), takšna rutina nima nobenih parametrov niti ne vrača nobene vrednosti. Komunikacija s preostalo kodo poteka preko posebnih globalnih podatkovnih struktur, ki so v najenostavnejšem primeru običajne globalne spremenljivke. Ker se lahko vrednost takšnih spremenljivk spreminja zunaj glavnega toka izvajanja programa, jih je treba deklarirati z opredeljevalcem `volatile` (glej razdelek 9.3). To je treba zato, da preprečimo morebitne optimizacije kode, ki jih izvaja prevajalnik, ki bi utegnile preprečiti pravočasno posodabljanje vrednosti takšnih spremenljivk v pomnilniku.
- Ker se lahko zgodi prekinitev kadarkoli, se lahko kadarkoli kliče tudi prekinitvena rutina. Nujno je, da se prekinitvena rutina izvede hitro (tj. njena koda mora biti kratka), tako da se lahko izvajanje prekinjenega glavnega programa nadaljuje, kakor hitro je to mogoče.

Običajno je, da sistem med izvajanjem določene prekinitvene rutine ne dovoli, da bi se ista prekinitev zgodila še enkrat, kar precej poenostavi pisanje kode. Privzeto obnašanje prekinitev v sistemu Arduino je celo takšno, da sistem med izvajanjem določene prekinitvene rutine onemogoči (maskira) ***vse*** prekinitve. Zato v prekinitvenih rutinah niso na voljo funkcionalnosti, ki temeljijo na prekinitvah, kot na primer funkciji `millis` in `delay` ali serijski vmesnik.

Upravljanje s prekinitvami

Za naš namen si bomo ogledali dva enostavna primera prekinitev. Prvi primer je prekinitev, ki jo lahko sprožimo kot posledico spremembe logičnega nivoja na določeni vhodni sponki. Če želimo uporabljati takšno prekinitev, moramo narediti dve reči: prvič, napisati moramo prekinitveno rutino, in drugič, prekinitveno rutino moramo povezati z enim od možnih dogodkov na želeni sponki:

- CHANGE (slov. sprememba) Če izberemo ta dogodek, se bo prekinitev prožila ob vsaki spremembi stanja na sponki.
- FALLING (slov. padajoč) Če izberemo ta dogodek, se bo prekinitev prožila ob vsaki spremembi stanja na sponki, ki gre iz logične enke v logično ničlo.
- RISING (slov. rastoč) Če izberemo ta dogodek, se bo prekinitev prožila ob vsaki spremembi stanja na sponki, ki gre iz logične ničle v logično enko.

V sistemu Arduino povežemo prekinitveno rutino z dogodkom na sponki z uporabo funkcije `attachInterrupt`. Ker se številka digitalne sponke ne ujema vedno s številko prekinitve, je priporočljivo uporabiti tudi funkcijo `digitalPinToInterrupt`. Ta funkcija preslika številko digitalne sponke v številko prekinitve, ki jo uporablja sistem. Takole na primer dosežemo, da se bo funkcija `ukrepaj` klicala vsakokrat, ko se bo na sponki 4 zgodila sprememba logičnega stanja:

```
attachInterrupt(digitalPinToInterrupt(4), ukrepaj, CHANGE);
```

Ker bo sistem klical funkcijo (prekinitveno rutino) `ukrepaj` kasneje, moramo sistemu v tem trenutku podati le njen naslov. Funkciji `attachInterrupt` zato podamo kazalec na našo prekinitveno rutino (tj. ime funkcije brez oklepajev).

Za drug primer prekinitve bomo uporabili ***časovnik*** (angl. timer), ki je del strojne opreme tipičnega mikrokrmilnika. Časovnik je digitalni števec, ki šteje navzgor ali navzdol

z določeno konstantno frekvenco. Parametre časovnika je mogoče nastavljati, pomembno pa je, da je časovnik sposoben prožiti prekinitev, ki se navadno proži v trenutku, ko števec doseže vrednost nič. Seveda so za sistem Arduino na voljo knjižnice za enostavno delo s časovnikom. Za sistem Arduino Due je to na primer knjižnica <DueTimer.h>[2]. Knjižnica vsebuje devet objektov za devet neodvisnih časovnikov, od katerih bomo uporabljali le enega (tj. objekt `Timer0`). Tako kot smo to storili z dogodkom na sponki, moramo tudi časovnik povezati s prekinitveno rutino. Poleg tega moramo povedati še, kako pogosto želimo, da se prekinitev časovnika proži. Vse to storimo z uporabo postopkov `attachInterrupt` in `start`:

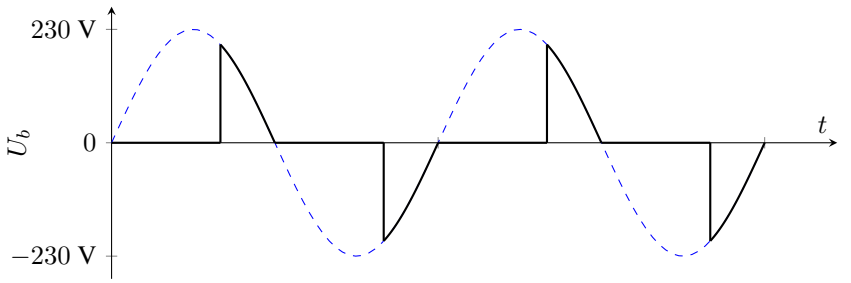
```
Timer0.attachInterrupt(ukrepaj);
Timer0.start(1000);
```

Z gornjima dvema klicema dosežemo, da se bo funkcija (prekinitvena rutina) `ukrepaj` klicala na vsakih 1000 mikrosekund. Če želimo ponavljanje proženja prekinitve ustaviti, potem uporabimo postopek `stop`:

```
Timer0.stop();
```

Primer: zatemnilnik

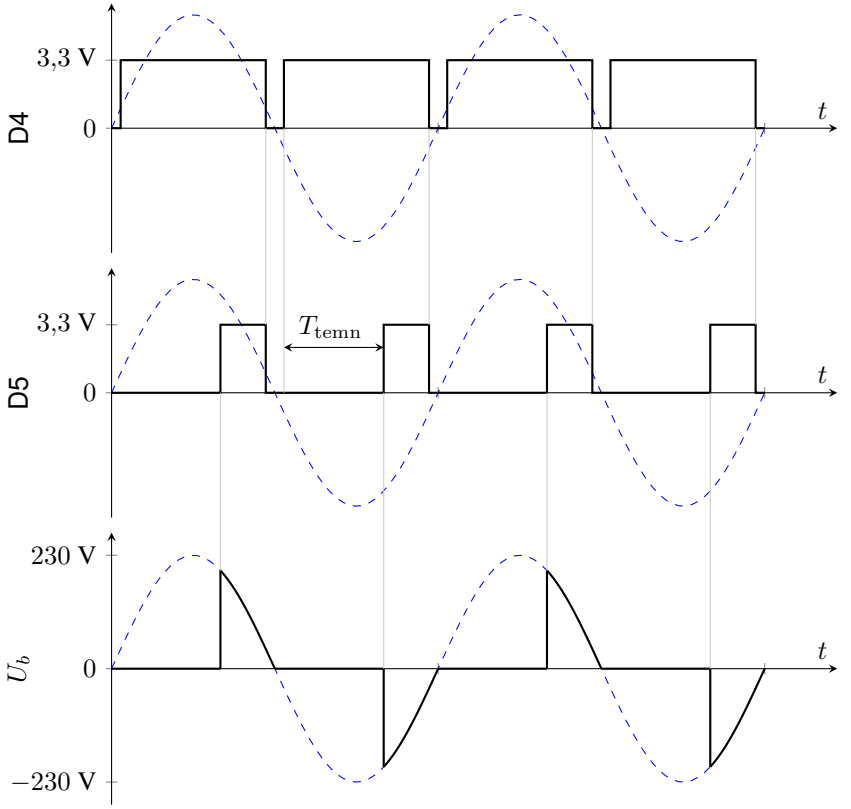
Kot primer uporabe prekinitev si bomo ogledali zatemnilnik (angl. dimmer), s katerim lahko nastavljamo svetlost luči. Stikalni načrt vezja, ki ga bomo pri tem uporabili, je prikazan na sliki v dodatku E. Vezje deluje na osnovi triaka (angl. triac), ki opravlja funkcijo hitrega močnostnega stikala. Ob vsakem prehodu izmenične napetosti skozi ničlo triak breme avtomatično izklopi. Nazaj ga lahko vklopimo z napetostnim impulzom na njegovi tretji sponki, ki se imenuje vrata (angl. gate). Če z vklopom vsakokrat nekoliko počakamo, potem dobimo na bremenu ustrezno porezan napetostni signal, ki ga prikazuje črna neprekinjena črta na naslednji sliki:



Na tak način učinkovito zmanjšamo moč in posledično svetlost luči. Princip spominja na pulzno-širinsko modulacijo, le da na breme ne pošiljamo pravokotnih impulzov, temveč namesto tega »režemo« obstoječo omrežno napetost. Ker triak breme odklopi avtomatsko ob prehodu napetosti skozi ničlo, moramo z našim mikrokrmilnikom poskrbeti le, da po določenem času proizvedemo prožilni signal, s katerim spet priklopimo breme na napetost. Ključno pri tem je, da se odzovemo časovno izjemno natančno, sicer bo luč migotala. Želene natančnosti odziva ne moremo doseči s poizvedovanjem, zato bomo uporabili prekinitve.

[2]Če uporabljate sistem, ki temelji na Atmelovem osembitnem mikrokrmilniku iz družine AVR (npr. Arduino Uno), potem lahko uporabite knjižnico <TimerOne.h>.

Vezje v dodatku E vsebuje usmernik in optični sklopnik (angl. optocoupler), ki proizvedeta signal, ki ga prikazuje črna neprekinjena črta na prvem od treh grafov na naslednji sliki:



Ta signal pripeljemo na vhodno sponko D4, služi pa nam za proženje časovnika. Ko na sponki D4 zaznamo naraščajoči rob signala, zaženemo časovnik. Ta po pretečenem času temnenja (T_{temn}) sproži prekinitev, ki takoj vklopi prožilni signal na sponki D5 (srednji graf na gornji sliki). Visok signal na sponki D5 povzroči proženje triaka, moramo pa ga ugasniti najkasneje pred naslednjim prehodom omrežne napetosti skozi ničlo, sicer triak takrat ne bo izklopil. Zato moramo ob padajočem robu signala na sponki D4 poskrbeti za nizek logični nivo na sponki D5.

Vse skupaj lahko krmilimo z naslednjim programom:

```
#include <DueTimer.h>

/*******************************************************
* Upravljanje s časom temnenja (v mikrosekundah):      *
*******************************************************/
#define MAKS_TEMN 9000
#define MIN_TEMN 1
volatile int g_temn = MIN_TEMN;

/*******************************************************
* Sponki, na kateri je priključen zatemnilnik:         *
*******************************************************/
#define PREHOD_NULA_PIN 4
```

```
#define PROZI_TRIAK_PIN 5

/*******************************************************
* Tipkovnica:                                          *
*******************************************************/
#define ST_VRSTIC 4
#define ST_STOLPCEV 4
int g_st[ST_STOLPCEV] = {38, 39, 40, 41};
int g_vr[ST_VRSTIC]   = {42, 43, 44, 45};
char g_tipkovnica[ST_VRSTIC][ST_STOLPCEV + 1] = {"123A",
                                                 "456B",
                                                 "789C",
                                                 "*0#D"};
void izberiVrstico(int v) {
  for (int i = 0; i < ST_VRSTIC; i++) {
    if (i == v) {
      pinMode(g_vr[i], OUTPUT);
      digitalWrite(g_vr[i], LOW);
    }
    else {
      pinMode(g_vr[i], INPUT);
    }
  }
}

char beriZnak(void) {
  for (int v = 0; v < ST_VRSTIC; v++) {
    izberiVrstico(v);
    for (int s = 0; s < ST_STOLPCEV; s++) {
      if (digitalRead(g_st[s]) == LOW) {
        return g_tipkovnica[v][s];
      }
    }
  }
  return 0;
}

/*******************************************************
* Prekinitvene rutine:                                 *
*******************************************************/
void prozi(void) {
  digitalWrite(PROZI_TRIAK_PIN, HIGH);
  Timer0.stop();
}

void prehodNula(void) {
  if (digitalRead(PREHOD_NULA_PIN) == HIGH) {
    /* Naraščajoči rob:  */
    Timer0.start(g_temn);
  }
  else {
    /* Padajoči rob:     */
    digitalWrite(PROZI_TRIAK_PIN, LOW);
  }
}

/*******************************************************
* Začetne nastavitve:                                  *
*******************************************************/
void setup(void) {
  for (int i = 0; i < ST_STOLPCEV; i++) {
```

```
    pinMode(g_st[i], INPUT_PULLUP);
  }
  pinMode(PREHOD_NULA_PIN, INPUT);
  pinMode(PROZI_TRIAK_PIN, OUTPUT);
  Timer0.attachInterrupt(prozi);
  attachInterrupt(digitalPinToInterrupt(PREHOD_NULA_PIN),
                  prehodNula, CHANGE);
}

/*******************************************************
* Glavna zanka - branje tipk s poizvedovanjem (polling)  *
* in nastavljanje časa temnenja:                         *
*******************************************************/
void loop(void) {
  static unsigned long oznakaTipka = 0;
  const unsigned long korakTipka = 10;
  if (millis() - oznakaTipka >= korakTipka) {
    oznakaTipka += korakTipka;
    switch (beriZnak()) {
      case '1':
        g_temn += 40;
        if (g_temn > MAKS_TEMN) {
          g_temn = MAKS_TEMN;
        }
        break;
      case '2':
        g_temn -= 40;
        if (g_temn < MIN_TEMN) {
          g_temn = MIN_TEMN;
        }
        break;
    }
  }
}
```

Iz kode je razvidno, da nastavljamo svetlost luči s spreminjanjem časa temnenja T_{temn}, ki je zapisan v globalni spremenljivki `g_temn` in je podan v mikrosekundah. Čas temnenja spreminjamo s tipkama 1 in 2, in sicer med vrednostma 1 (`MIN_TEMN`, najsvetleje) in 9000[3] (`MAKS_TEMN`, najtemneje). Vendar se program ne odziva na pritiske, temveč na stanja obeh tipk: dokler je katera od obeh tipk pritisnjena, se vsakih 10 ms bodisi poveča bodisi zmanjša vrednost spremenljivke `g_temn` za 40. Tako bo trajalo popolno prižiganje oziroma ugašanje luči 2,25 sekunde.

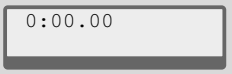
Funkciji `prozi` in `prehodNula` predstavljata prekinitveni rutini, ki smo ju z ustreznimi prekinitvami povezali v funkciji `setup`. Funkcija `prehodNula` se kliče ob vsaki spremembi signala na sponki D4. V tej funkciji najprej preverimo stanje na sponki D4, s čimer ugotovimo, ali je prekinitev sprožil naraščajoči ali padajoči rob signala. Ob naraščajočem robu moramo zagnati časovnik, da bo prožil prekinitev po natanko `g_temn` mikrosekundah. Na prekinitev časovnika se odzovemo s prekinitveno rutino `prozi`: ko poteče čas temnenja, se na sponko D5 zapiše logična enka, s čimer se proži triak. Poleg tega funkcija `prozi` ustavi časovnik, ki ga znova zaženemo šele ob naraščajočem robu signala na D4. Funkcija `prehodNula` ob vsakem padajočem robu signala na sponki D4

[3]Vrednost 9000 (= 9 ms) smo določili empirično: polovica periode omrežne napetosti znaša 10 ms, logična ničla na sponki D4 pa traja nekoliko manj kot 0,9 ms. Preostalih 0,1 ms predstavlja varnostni faktor, da ne bi prožili triaka kasneje, kot se zgodi padajoči rob signala na sponki D4.

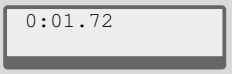
zapiše na sponko D5 logično ničlo, da bo lahko triak ob prehodu omrežne napetosti skozi ničlo odklopil breme.

Naloga 13.5 Za konec napišite še program za digitalno štoparico, ki meri čas na stotinko sekunde natančno. Pritisk na tipko A naj štoparico požene oziroma ustavi. Pritisk na tipko B naj štoparico ponastavi. Če štoparica v trenutku pritiska na tipko B teče, naj se štoparica hkrati ustavi.

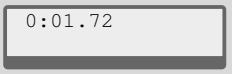
Primer delovanja štoparice:



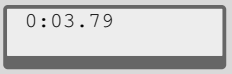
(Vklop, pritisnemo A)



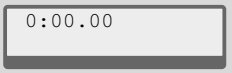
(Po sekundi in 72 stotink pritisnemo A)



(Po določenem času spet pritisnemo A)



(Dve sekundi in sedem stotink od zadnjega pritiska A)



(Pritisnemo B)

Opomba: Za pogon in ustavitev štoparice lahko namesto tipke uporabite fotoelektrični senzor, ki avtomatično zazna na primer prehod tekmovalca skozi cilj.

D

PRESLIKAVA SPONK MED SAM3X IN ARDUINO DUE

Sponka na sistemu Due	Sponka na krmilniku SAM3X	Največji izhodni tok (mA)	Največji ponorni tok (mA)
0 / RX0	PA8	3	6
1 / TX0	PA9	15	9
2 / D2	PB25	15	9
3 / D3	PC28	15	9
4 / D4	PA29 in PC26	15	9
5 / D5	PC25	15	9
6 / D6	PC24	15	9
7 / D7	PC23	15	9
8 / D8	PC22	15	9
9 / D9	PC21	15	9
10 / D10	PA28 in PC29	15	9
11 / D11	PD7	15	9
12 / D12	PD8	15	9

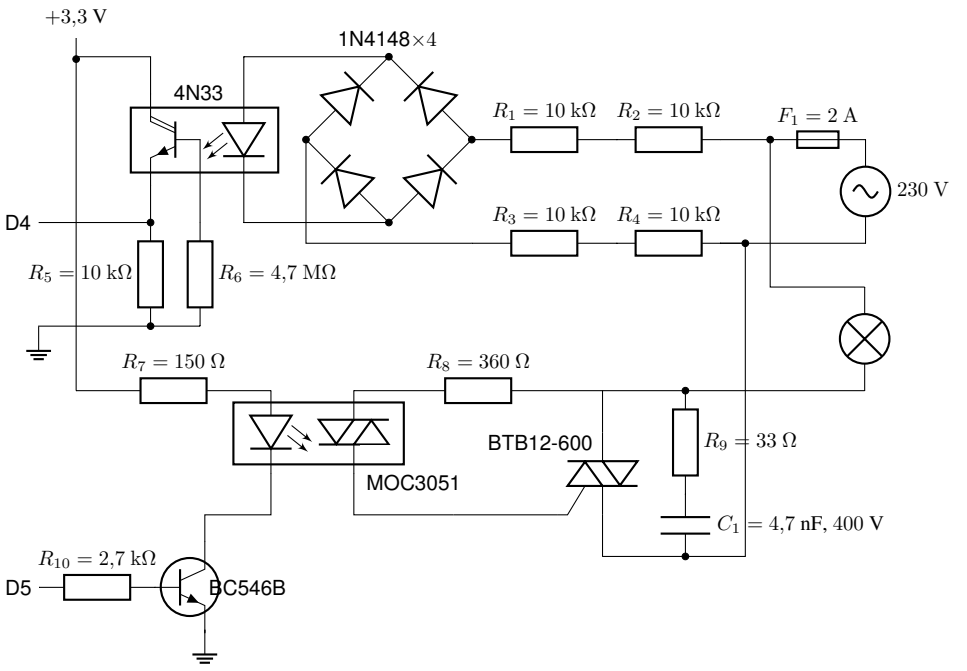
13 / D13 / LED	PB27	3	6
14 / TX3	PD4	15	9
15 / RX3	PD5	15	9
16 / TX2	PA13	3	6
17 / RX2	PA12	3	6
18 / TX1	PA11	3	6
19 / RX1	PA10	3	6
20 / SDA	PB12	3	6
21 / SCL	PB13	3	6
22 / D22	PB26	3	6
23 / D23	PA14	15	9
24 / D24	PA15	15	9
25 / D25	PD0	15	9
26 / D26	PD1	15	9
27 / D27	PD2	15	9
28 / D28	PD3	15	9
29 / D29	PD6	15	9
30 / D30	PD9	15	9
31 / D31	PA7	15	9
32 / D32	PD10	15	9
33 / D33	PC1	15	9
34 / D34	PC2	15	9
35 / D35	PC3	15	9
36 / D36	PC4	15	9
37 / D37	PC5	15	9
38 / D38	PC6	15	9
39 / D39	PC7	15	9
40 / D40	PC8	15	9
41 / D41	PC9	15	9
42 / D42	PA19	15	9
43 / D43	PA20	3	6
44 / D44	PC19	15	9
45 / D45	PC18	15	9
46 / D46	PC17	15	9
47 / D47	PC16	15	9
48 / D48	PC15	15	9
49 / D49	PC14	15	9
50 / D50	PC13	15	9
51 / D51	PC12	15	9

52 / D52	PB21	3	6
53 / D53	PB14	15	9
54 / A0	PA16	3	6
55 / A1	PA24	3	6
56 / A2	PA23	3	6
57 / A3	PA22	3	6
58 / A4	PA6	3	6
59 / A5	PA4	3	6
60 / A6	PA3	3	6
61 / A7	PA2	3	6
62 / A8	PB17	3	6
63 / A9	PB18	3	6
64 / A10	PB19	3	6
65 / A11	PB20	3	6
66 / DAC0	PB15	3	6
67 / DAC1	PB16	3	6
68 / CANRX	PA1	3	6
69 / CANTX	PA0	15	9
70 / SDA1	PA17	3	6
71 / SCL2	PA18	15	9
72 / LED »RX«	PC30	15	9
73 / LED »TX«	PA21	3	6
74 / (MISO)	PA25	15	9
75 / (MOSI)	PA26	15	9
76 / (SCLK)	PA27	15	9
77 / (NPCS0)	PA28	15	9
78 / (nepriključena)	PB23	15	9
USB / ID	PB11	15	9
USB / VBOF	PB10	15	9

PRAZNA STRAN

E

STIKALNI NAČRT ZA ZATEMNILNIK



PRAZNA STRAN

VIRI

[1] King, K. N. *C programming: a modern approach*, W. W. Norton & Company, (2008).

[2] *American National Standard for Programming Languages – C (ANSI/ISO 9899-1990)*, American National Standards Institute, (1990).

[3] *ISO/IEC 9899:1999: Programming languages – C*, ISO – International Organization for Standardization, (2007).

[4] *Atmel SAM3X / SAM3A Series: SMART ARM-based MCU*, podatkovni list, (http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf).

[5] *Atmel ATmega328P: 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash*, podatkovni list, (http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf).

[6] *Hitachi HD44780U (LCD-II): Dot Matrix Liquid Crystal Display Controller/Driver*, podatkovni list.

[7] https://www.arduino.cc/, spletna stran.

PRAZNA STRAN

O AVTORJIH

Iztok Fajfar, 1967, je dobil prvi osebni računalnik v začetku osemdesetih let prejšnjega stoletja: ZX Spectrum z neverjetnimi 48 KB RAMa. Računalniki so kmalu postali njegova strast in nepogrešljivi spremljevalci. Sodeloval je pri mnogih projektih razvoja programske opreme tako za domača kot tudi tuja podjetja. Njegovo raziskovalno delo vključuje evolucijske algoritme, zlasti genetsko programiranje. Fajfar poučuje računalniško programiranje na vseh ravneh, od strojnega do objektno usmerjenega, in na vseh stopnjah študija. Je tudi avtor več učbenikov, med drugim učbenika *Start programming using HTML, CSS, and JavaScript*, ki ga je izdala priznana mednarodna založba. Zaposlen je kot izredni profesor na Fakulteti za elektrotehniko.

Jernej Olenšek, 1980, se je že kot mladi raziskovalec iz študentskih klopi preselil pred tablo in ugotovil, da ima vloga učitelja poseben čar. Po zaključenem podiplomskem študiju se je za nekaj let preselil v industrijske vode. Začel je kot spletni analitik in kasneje soustanovil podjetje, v katerem je delal kot razvojni programer. S pridobljenim praktičnim znanjem se je vrnil na univerzo. Zaposlen je kot asistent na Fakulteti za elektrotehniko v Ljubljani, kjer poučuje širok spekter snovi: programiranje osebnih računalnikov ter večopravilnih vgrajenih sistemov in sistemov v realnem času. Prav tako linearna elektronska vezja in vezja z operacijskimi ojačevalniki. Vodi tudi praktične delavnice za učence in dijake. Raziskovalno dela na področju optimizacijskih algoritmov in njihove uporabe pri načrtovanju elektronskih vezij.

PRAZNA STRAN

IZ RECENZIJE

V učbeniku avtorja na bralcu prijazen način ponujata odgovore na vprašanja, ki se po njunih izkušnjah študentom najpogosteje porajajo. Pozoren bralec bo odkril tudi mnogo uporabnih namigov, in sicer točno tam, kjer jih potrebuje. Tudi oblikovno je učbenik zelo domišljen, pregleden in dosleden. Jezik je tekoč, stvaren in enostaven, pa vendar ne dolgočasen.

V drugem delu učbenika avtorja skrbno izbereta primere uporabe, ki so značilni za elektrotehniko in celo analogno elektrotehniko. Primeri so vzeti iz realnega življenja in širše uporabni. Na zelo posrečen način se dotakneta problemov krmiljenja v realnem času in tako postavita pomembne temelje za različne predmete višjih letnikov, ne da bi pri tem študente preobremenila.

prof. dr. Tadej Tuma, UL FE