UNIVERZA V LJUBLJANI
FAKULTETA ZA ELEKTROTEHNIKO

UVOD V PROGRAMSKI JEZIK C

ZAPISKI PREDAVANJ ZA PREDMETA PROGRAMIRANJE 2 IN PROGRAMIRANJE MIKROKRMILNIKOV

IZTOK FAJFAR

1. izdaja 2005

2. popravljena in dopolnjena izdaja 2014
(zadnji popravki 24. 2. 2016)

Prošnja: sodoben čas preganja in vse se hitro spreminja. Zato so tudi ti zapiski nastali v naglici in zelo bom vesel vseh najdenih napak in predlogov, ki mi jih boste sporočili na moj naslov iztok.fajfar@fe.uni-lj.si

Orodjarna

Pri predmetu bomo uporabljali dva razvojna sistema. Začeli bomo z okoljem Microsoft Visual Studio 2012 Express, v drugi polovici semestra pa se bomo lotili razvojne plošče Arduino.

Microsoft Visual Studio 2012 Express

Okolje je za študente Fakultete za elektrotehniko Univerze v Ljubljani brezplačno. Dobite ga na naslovu http://msdnaa.fe.uni-lj.si, kjer so tudi vsa navodila za namestitev.

Ko zaženete okolje, morate najprej ustvariti nov projekt:
-V meniju izberite `File->New->Project`.
-V oknu, ki se odpre, na desni izberete `Visual C++` in v srednjem stolpcu `Win32 Console Application`.
-Spodaj vpišite ime (`Name`) prokejta in določite mapo (`Location`), v kateri ga želite ustvariti.
-Kliknite `OK`.
-Odpre se vam čarovnik in takoj kliknite končaj (`Finish`).
-Avtomatično se vam odpre datoteka `<ime projekta>.cpp`, ki vsebuje med drugim naslednje:

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
  return 0;
}
```

-Spremenite zapis v takšnega:

```
#include "stdafx.h"
int main()
{
  return 0;
}
```

Vgrajeni razhroščevalnik (debugger) lahko krmilimo z naslednjimi bližnjicami:
`F9` - prekinitvena točka (breakpoint)
`F5` - poženi (run) v razhroščevalnem načinu
`Shift+F5` - ustavi (stop)
`F10` - korak preko funkcije (step over)
`F11` - korak v funkcijo (step into)
`Alt+3` - opazuj (Watch)

Program poženemo v izvajalnem načinu s kombinacijo tipk `Ctrl+F5`

Arduino

Vse o sistemu Arduino, ki ga bomo uporabljali v drugem delu semestra, najdete na uradni strani `www.arduino.cc`. Povzetek obravnavane snovi najdete v dodatku na koncu teh zapiskov.

Razlike med jezikoma C in JavaScript

V prvem semestru smo spoznali jezik JavaScript in pomembno je, da na začetku opozorimo na poglavitne razlike med obema jezikoma:

-knjižnice: v Ceju moramo na začetku z ukazom `#include` vključiti ustrezne knjižnice, odvisno od tega, katere funkcije bomo v programu uporabljali.
-podatkovni tipi: v Ceju je potrebno pri deklaracijah funkcij in spremenljivk določiti tip podatka, ki ga funkcija vrne oz spremenljivka hrani. To storimo tako, da namesto ključnih besed function oziroma var vstavimo ime ustreznega tipa (npr `int`, `float`, `void`).
-funkcija `main()`: vsak cejevski program mora imeti vsaj eno funkcijo, ki se imenuje `main()`. Program se začne izvajati v tej funkciji.
-logični Boolov tip: jezik C v osnovi ne pozna boolovega podatkovnega tipa. Vsi operatorji in stavki, ki pričakujejo boolove vrednosti, vračajo vrednosti 0 ali 1, odvisno od tega ali je pogoj izpolnjen (1) ali ne (0). Kot vhodne vrednosti sprejmejo poljubno številsko vrednost, pri čemer 0 velja za napačno (`false`) in različno od 0 za pravilno (`true`).
V resnici to ni prav posebna novost, saj so se v jeziku JavaScript pretvorbe med številskim in Boolovim tipom dogajale po natančno istih pravilih. Vrednost `false` se je tolmačila kot 0 in `true` kot 1. V obratno smer se je vrednost 0 tolmačila kot `false` in vrednost različna od 0 se je tolmačila kot `true`.
-operatorja stroge enakosti in neenakosti: C ne pozna primerjalnih operatorjev stroge enakosti (`===`) in neenakosti (`!==`).
-operator deljenja: v Ceju postane operator deljenja v primeru, da sta oba operanda celoštevilskega tipa, operator celoštevilskega deljenja. To pomeni, da se rezultatu odreže del za decimalno piko.

Kaj mi je tega treba bilo...

Takoj bomo začeli z dvema cejevskima programoma, kjer najprej ne bo jasno praktično nič. Vendar, brez panike. Vse in še več bomo postopoma in sistematično obdelali v poglavjih, ki sledijo.

Dobrodošel v Ceju!

Naslednji program s tipkovnice prebere jezikovno varianto (0 za slovensko in 1 za italijansko) in na zaslon izpiše pozdrav v ustreznem jeziku:

IZPIS IZVORNE KODE pozdrav.c

```
/* moj čisto prvi program v ceju - in to prijazen*/
#include "stdafx.h"
#include <stdio.h>

int jezik;    //0 - slovenščina, 1 - italijanščina

int main()
{
   printf("Vpiši jezikovno varianto (0-si/1-it): ");
   scanf("%d", &jezik);
   if (0 == jezik)
   {
      printf("Dobrodošel v Ceju!\n");
   }
   else
   {
```

```
        printf("Benvenuti alla programmazione in C!\n");
    }
    return 0;
}
```

Sestavine programa

Opombe: Prva vrstica programa `pozdrav.c` je opomba. Opombe so mišljene kot pomoč programerju in jih prevajalnik popolnoma ignorira. Opombe lahko pišemo med parom simbolov `/*` in `*/`, ali pa uporabimo dve poševni črti `//`, ki povesta prevajalniku, naj ignorira preostanek vrstice.

#include: S tem ukazom vključimo knjižnice funkcij, ki jih bomo potrebovali v programu. V našem primeru smo med drugim vključili standardno knjižnico `stdio.h`, ki vsebuje definiciji funkcij `printf()` in `scanf()`. Standardne knjižnice se nahajajo v sistemskih mapah, zato moramo njihova imena obdati s kotnimi oklepaji (< in >), da jih prevajalnik najde. Če se knjižica nahaja v trenutni mapi, potem namesto kotnih oklepajev uporabimo dvojne navednice, kakor to vidimo pri knjižnici `stdafx.h`. Mimogrede: knjižnica `stdafx.h` se ustvari samodejno, vanjo pa lahko dodajamo druge knjižnice, ki jih uporabljamo in se med razvojem projekta ne bodo spreminjale.

Spremenljivke: Spremenljivke v programih uporabljamo za začasno hranjenje podatkov. Spremenljivko ustvarimo tako, da navedemo najprej njen tip, potem pa še ime. Na koncu ne smemo pozabiti na podpičje:

SKLADNJA	deklaracija spremenljivke
`tip ime_spremenljivke;`	

Skladnja oziroma *sintaksa* (angl. syntax) določa osnovna pravila, po katerih združujemo posamezne elemente jezika v delujočo celoto. V naših zapisih skladenjskih pravil bo veljalo, da moramo besede v angleščini in ločila pisati tako kot je navedeno, besede v slovenščini pa nadomeščamo s katerim od ustreznih elementov.

Z zapisom

```
int jezik;
```

smo v programu `pozdrav.c` ustvarili spremenljivko `jezik`, ki je predznačenega celoštevilskega tipa (`int`). Takšnemu zapisu pravimo strokovno *deklaracija* spremenljivke.

main(): Glavni del vsakega cejevskega programa se imenuje `main()`. `main()` je, kot bomo kasneje spoznali, funkcija. K funkciji `main()` spada še imenovanje podatkovnega tipa funkcije `int`, o njegovem pomenu bomo še govorili, ter prvi in zadnji zaviti oklepaj s stavkom `return`. Glavni del programa ima torej takšno minimalno obliko:

```
int main()
{
        return 0;
}
```

To mora imeti vsak cejevski program. Hkrati je to tudi že program, ki ga lahko prevedemo in zaženemo, čeprav ne naredi absolutno ničesar uporabnega.

Kadar program zaženemo, se izvajanje začne v prvi vrstici funkcije `main()`. V programu `pozdrav.c` se tako prva izvede funkcija `printf()`, ki na zaslon izpiše poziv uporabniku, naj izbere jezikovno varianto.

Zamiki: Pozoren bralec bo opazil, da koda ni poravnana ob levi rob, ampak je od njega različno zamaknjena. Ti zamiki sicer niso pomembni za delovanje programa (kot večinoma tudi presledki niso), so pa izjemno pomembni za programerja. Koda je na ta način bolj berljiva in obvladljiva. Pomembna posledica tega je, da je vzdrževanje in nadgrajevanje takšne kode enostavnejše in mnogo cenejše. Način zamikanja kode ni na noben način predpisan, je stvar osebnega okusa in udobja. Na mnogih sistemih obstajajo celo softverska orodja, ki cejevsko kodo uredijo in dodajo ustrezne zamike.

Naslednji program je sicer povsem pravilen (čeprav bodo imeli nekateri novejši prevajalniki z njim manjše težave) in celo deluje, vendar je popolnoma neberljiv:

```
IZPIS IZVORNE KODE            vlakec.c
                                                        extern int
                                                            errno
                                                              ;char
                                                                 grrr
                                ;main(                             r,
  argv, argc )            int    argc                                ,
   r        ;            char *argv[];{int                        P( );
#define x  int i,        j,cc[4];printf("      choo choo\n"      ) ;
x  ;if    (P(  !         i              )        |  cc[  !      j ]
&  P(j    )>2  ?         j              :        i  ){*  argv[i++ +!-i]
;              for    (i=              0;0    ;i++                    );
_exit(argv[argc- 2    / cc[1*argc]|-1<<4 ]    ) ;printf("%d",P(""));}}
  P  (    a  )   char a   ;  {    a  ;   while(    a  >      "  B   "
  /* -    by E            ricM    arsh             all-      */);    }
```

Koda je vzeta s spletne strani, posvečene zmagovalcem med najbolj obupno napisanimi cejevskim programi. Gre za zanimivo mednarodno tekmovanje v pisanju nemogoče cejevske kode na strani http://www.ioccc.org/.

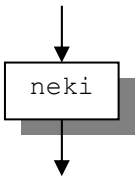
Stavki: Vse dogajanje v cejevskih programih se odvija znotraj funkcij. O funkcijah bomo podrobno govorili kasneje, zaenkrat nam bo pomembno le to, da telo funkcije vsebuje niz stavkov, ki se izvajajo eden za drugim. Da bomo lahko gradili programe, moramo razumeti, kaj je stavek, zato si to poglejmo kar takoj.

Stavki v Ceju

Stavke v ceju lahko gradimo bodisi iz izrazov bodisi lahko uporabimo katerega od krmilnih stavkov. Da dobimo stavek iz izraza, mu moramo dodati podpičje:

```
SKLADNJA              stavek
izraz;
```

Tudi na koncu vsakega drugega stavka mora biti podpičje. Izjema so le stavki, ki se zaključijo z zavitim oklepajem. Takrat podpičja ne pišemo. Narišimo si diagram poteka za poljuben stavek:



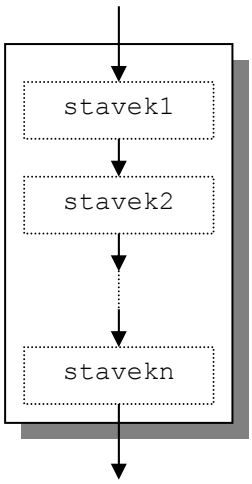
Kot vidimo iz diagrama, bo imel v splošnem vsak stavek en sam vhod in en sam izhod.

Stavke lahko tudi sestavljamo v bloke. To storimo tako, da več zaporednih stavkov zapremo v par zavitih oklepajev. Zelo pomembna lastnost takega bloka je, da se navzven kaže kot en sam stavek. To lastnost bomo kasneje s pridom uporabili pri gradnji odločitvenih stavkov. Takšen blok, pravimo mu tudi *sestavljen stavek*, zapišemo takole:

SKLADNJA	sestavljen stavek

```
{
   stavek1
   stavek2
    ...
   stavekn
}
```

In diagram poteka:



Funkcija `main()` v programu `pozdrav.c` vsebuje štiri stavke. Prva dva stavka sta klica knjižničnih funkcij `printf()` oziroma `scanf()` (kot bomo videli kasneje, klici funkcij spadajo med izraze), sledi jima pogojni stavek `if` (ta stavek spada med krmilne stavke), ki ga bomo spoznali malenkost kasneje. Na koncu je stavek `return`, ki klicatelju funkcije (operacijskemu sistemu) vrne informacijo o tem, kako se je program zaključil. Najprej si oglejmo funkciji `printf()` in `scanf()`.

Komunikacija z zunanjim svetom

Program lahko komunicira z zunanjim svetom na ogromno različnih načinov. Mi se bomo na začetku omejili na komunikacijo prek tipkovnice in monitorja.

Funkcijo `printf()` uporabljamo v našem programu za izpis besedila na zaslon. V prvi vrstici se bo na zaslon izpisalo besedilo dobesedno tako, kakor je navedeno v dvojnih narekovajih: `Vpiši jezikovno varianto (0-si/1-it):`. Pri naslednjih dveh klicih te funkcije vidimo poleg pozdrava, ki ga želimo izpisati, še znak `\n`. Znakom, ki se začnejo z nazaj nagnjeno poševno črto (angl. backslash), pravimo *ubežne* (angl. escape) *sekvence*. Ti znaki se ne izpisujejo na zaslon, ampak ponavadi prispevajo k oblikovanju besedila. Ubežna sekvenca `\n` predstavlja pomik kurzorja v novo vrstico (angl. newline).

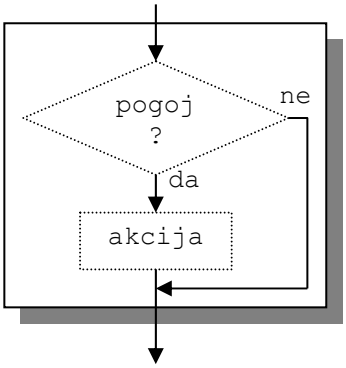
Funkcija `scanf()` ima v programu `pozdrav.c` nalogo zajemati podatke s tipkovnice. Funkciji smo v oklepaju podali dva parametra, ločena z vejico. Prvi, ki je v dvojnih narekovajih, predstavlja formatni niz, ki funkciji pove, kakšen tip podatka naj pričakuje. Formatni niz je v našem primeru sestavljen iz takoimenovanega *formatnega določila* `%d`, ki funkciji pove, naj pričakuje predznačen celoštevilski podatek. Drugi parameter je spremenljivka, v katero bi radi prebrani podatek shranili. Pred imenom spremenljivke mora biti obvezno naslovni operator (`&`). Zakaj je to potrebno, bomo razumeli kasneje, ko se bomo pogovarjali o kazalcih in podajanju parametrov funkcijam po referenci.

Odločitvena stavka if in if..else

Naloga prvih dveh stavkov programa `pozdrav.c` je ta, da od uporabnika izvesta, v katerem jeziku bi rad prebral pozdrav. Stavka se izvedeta vedno, ne glede na okoliščine. Takoj nato pridemo do dileme, kateri pozdrav izpisati. Potrebujemo stavek, ki se bo odločil glede na vnešeno vrednost (t.j. vrednost spremenljivke `jezik`). Za to imamo na voljo *odločitveni stavek* `if`, ki spada med krmilne stavke. Stavek zapišemo takole:

SKLADNJA	odločitveni stavek if
`if (pogoj) akcija`	

Stavek (ali blok stavkov) `akcija` se bo izvršil v primeru, da je `pogoj` izpolnjen oziroma resničen (angl. true). Diagram poteka izgleda takole:



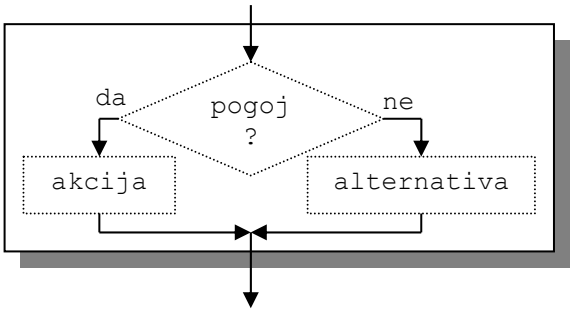
Stavek `if..else` je pravzaprav razširjena oblika stavka `if` in ima takšno obliko:

SKLADNJA	odločitveni stavek if..else
`if (pogoj) akcija` `else alternativa`	

Pri tem stavku se v primeru, ko `pogoj` ni izpoljnjen oziroma je napačen (angl. false), izvede stavek ali blok stavkov `alternativa`.

Spomnimo se, da stavek, ki se konča z zavitim oklepajem (na primer sestavljen stavek), na koncu nima podpičja.V programu `pozdrav.c` smo v pogojnem stavku uporabili dva sestavljena stavka. Oba sicer obsegata po en sam stavek, pomembno pa je, da za zavitima zaklepajema ni podpičij.

Diagram poteka stavka `if..else` izgleda takole:

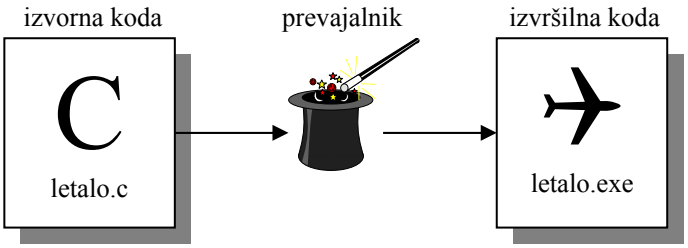


O splošni obliki zapisa pogoja bomo še govorili. V našem primeru želimo prebrano vrednost, ki je shranjena v spremenljivki `jezik`, primerjati z ničlo. V ta namen uporabimo *primerjalni operator*, ki preverja enakost. Operator je sestavljen iz dveh enačajev (==). Tako zapišemo naš pogoj kot primerjalni izraz `0 == jezik`.

Pozor! Srečali bomo še več operatorjev, ki so sestavljeni iz dveh ali treh znakov. Vse te operatorje (razen oklepajev) moramo pisati skupaj, brez presledka. Narobe bi bilo, na primer, če bi pisali operator primerjanja enakosti kot `= =`.

Pa to res deluje?

Če želimo odgovoriti na to vprašanje, moramo program v *izvorni kodi* (angl. source code) najprej prevesti v obliko, razumljivo digitalnemu računalniku (strojni jezik), čemur pravimo navadno *izvršilna koda* (angl. executable). To napravimo z ustreznim cejevskim prevajalnikom:



Mi bomo uporabljali prevajalnik, ki je del razvojnega okolja Visual Studio 2012 Express, ki ga bomo uporabljali na vajah (okolje dobite na http://msdnaa.fe.uni-lj.si/).

Ko program prevedemo in zaženemo, dobimo na zaslonu takšen izpis:

```
DELOVANJE PROGRAMA          pozdrav.exe
Vpiši jezikovno varianto (0-si/1-it): 0
Dobrodošel v Ceju!
_
```

Pri tem predstavlja običajen tisk besedilo, ki ga izpiše računalnik, mastni tisk predstavlja vnos uporabnika, podčrtaj (_) pa je položaj kurzorja na zaslonu, ko se program izteče. Ker smo v funkciji `printf()` uporabili ubežno sekvenco `\n`, se je kurzor pomaknil na začetek nove vrstice.

Študentski osebni račun

Oglejmo si še program, ki se bo pretvarjal, da je bankomat. Program najprej pozdravi uporabnika in izpiše stanje na študentskem osebnem računu. Potem čaka na dvig in po uspešnem dvigu izpiše novo stanje. Če bi z dvigom presegli dovoljeni limit, program sporoči, da denarja ne more izplačati. Izvajanje programa se konča, če na vprašanje o še kakšni storitvi odgovorimo z ničlo, sicer se vrne na ponovni dvig.

```
IZPIS IZVORNE KODE          bankomat.c
#include "stdafx.h"
#include <stdio.h>

int stanje;
int limit;
int dvig;

int main()
{
   stanje = 100;
   limit = 50;

   printf("Dobrodošli v študentski banki\n");
   printf("Trenutno stanje na računu: %d EUR\n", stanje);

   do
   {
      printf("\nDvig: ");
      scanf("%d", &dvig);
      if (dvig > stanje + limit)
      {
         printf("Zneska vam ne moremo izplačati.\n");
      }
      else
      {
         stanje = stanje - dvig;
         printf("Po opravljeni storitvi je\n");
         printf("stanje na računu %d EUR.\n", stanje);
      }
      printf("Želite opraviti še kakšno storitev?\n");
      printf("(1 - da / 0 - ne): ");
      scanf("%d", &dvig);
   } while (0 != dvig);
   printf("\nHvala, ker ste osrečili naš bankomat.\n");
   return 0;
}
```

Sestavine programa

Inicializacija spremenljivke: Vsako spremenljivko moramo pred uporabo postaviti na določeno začetno vrednost (temu pravimo s tujko, da spremenljivke *inicializiramo*). Eden izmed načinov, kako to storimo, je s pomočjo *priredilnega*

operatorja (=). V gornjem programu smo postavili začetno stanje na računu in dovoljeni limit v prvih dveh vrsticah funkcije `main()`:

```
stanje = 100;
limit = 50;
```

Spremenljivke lahko inicializiramo tudi ob njihovi deklaraciji:

SKLADNJA	inicializacija ob deklaraciji
tip ime_spremenljivke = zacetna_vrednost;	

Spremenljivki `stanje` in `limit` bi lahko postavili na želene začetne vrednosti ob njuni deklaraciji takole:

```
int stanje = 100;
int limit = 50;
```

Formatna določila pri izpisu: Poleg ubežnih sekvenc lahko za uravnavanje oblike izpisa, ki ga proizvede funkcija `printf()` uporabimo tudi formatna določila, ki smo jih srečali že pri branju podatkov s funkcijo `scanf()`. V programu `bankomat.c` vidimo primer takšne uporabe formatnega določila `%d` pri klicu funkcije `printf()`, ki izpisuje trenutno stanje na računu. Na mestu formatnega določila se bo izpisala vrednost celoštevilskega izraza, ki ga podamo kot drugi parameter funkcije. V našem primeru se bo na zaslon izpisala vrednost spremenljivke `stanje`.

Z enim klicem funkcije `printf()` lahko izpišemo poljubno različnih vrednosti z uporabo večih formatnih določil. Vrednosti, ki jih želimo izpisovati, dodajamo na koncu funkcije. Za vsako vrednost moramo v besedilo, ki ga funkcija izpisuje, na ustrezno mesto dodati ustrezno formatno določilo. Tako lahko skupni znesek in dovoljeni limit izpišemo na zaslon takole:

```
printf("Stanje: %d, Dovoljeni limit: %d\n", stanje, limit);
```

Vrednost spremenljivke `stanje` se bo izpisala na mestu prvega formatnega določila in vrednost spremenljivke `limit` na mestu drugega formatnega določila.

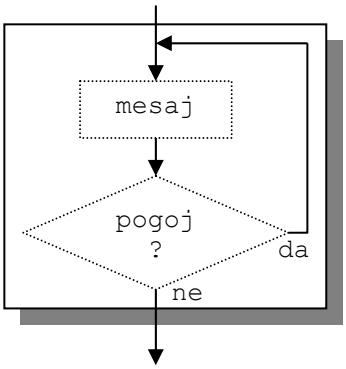
Ponavljalni stavek do..while

Program `bankomat.c` se od `pozdrav.c` razlikuje po tem, da se v njem ena in ista operacija večkrat ponovi. Program nam sporoča stanje na računu, s tipkovnice prebira želene dvige in ustrezno spreminja stanje na računu, dokler ne vnesemo ničle. Takšno ponavljanje dosežemo s katerim od treh ponavljalnih stavkov (pravimo jim tudi *zanke*), ki jih pozna C. Tako kot odločitvena stavka `if` in `if..else`, tudi ponavljalni stavki spadajo med krmilne stavke. V našem programu smo uporabili stavek `do..while`, ki ima takšno obliko zapisa:

SKLADNJA	ponavljalni stavek do..while
do mesaj while(pogoj);	

Stavek `mesaj` se izvaja, dokler je `pogoj` izpolnjen, pri čemer se `mesaj` v vsakem primeru izvrši vsaj enkrat, kajti pogoj se preverja šele na koncu stavka. Na koncu stavka `do..while` mora biti vedno podpičje, ker se konča z okroglim oklepajem.

Ilustrirajmo delovanje stavka `do..while` še z diagramom poteka:



Stavku (ali bloku stavkov), ki se ponavlja znotraj kakšne zanke (v našem primeru je to stavek `mesaj`), pravimo tudi *telo* (angl. body) zanke.

V programu `bankomat.c` smo za pogoj vstavili primerjalni izraz s primerjalnim operatorjem, ki preverja različnost. Operator je sestavljen iz klicaja in enačaja (`!=`). Naš primerjalni izraz se glasi `0 != dvig`, nazaj na ponovni dvig se namreč vračamo, dokler je odločitev o ponovni storitvi različna od nič.

Ko program prevedemo in zaženemo, dobimo tole:

```
DELOVANJE PROGRAMA          bankomat.exe
Dobrodošli v študentski banki.
Trenutno stanje na računu: 100 EUR

Dvig: 120
Po opravljeni storitvi je
stanje na računu -20 EUR.
Želite opraviti še kakšno storitev?
(1 - da / 0 - ne): 1

Dvig: 50
Zneska vam ne moremo izplačati.
Želite opraviti še kakšno storitev?
(1 - da / 0 - ne): 0

Hvala, ker ste osrečili naš bankomat.
_
```

Saj ni tako hudo, gremo zdaj od začetka...

Povej tako, da bom razumel

Preden nadaljujemo s programiranjem digitalnega računalnika, za trenutek pozabimo na to, kar smo že slišali o Ceju ali kateremkoli drugem programskem jeziku in se pozabavajmo s starim vicem o hladilniku, žirafi in slonu.

Vprašanje: *Kako spraviš žirafo v treh potezah v hladilnik?*
Odgovor: *Odpreš hladilnik, daš žirafo v hladilnik in zapreš hladilnik.*

Vprašanje: *Kako spraviš slona v štirih potezah v hladilnik?*
Odgovor: *Odpreš hladilnik, vzameš žirafo iz hladilnika, daš slona v hladilnik in zapreš hladilnik.*

Tako kot vsak dober vic, nas tudi ta precej nauči. Zamislimo si, da imamo hladilnik, v katerem lahko naenkrat hranimo le eno žival. Poleg tega si zamislimo še robota, ki razume in zna izvesti naslednje osnovne operacije nad hladilnikom:

- Odpri vrata.
- Zapri vrata.
- Daj *žival* v hladilnik.
- Vzemi *žival* iz hladilnika.

Pri tem je *žival* poljubna žival. Obenem ima robot dve tipali, s katerimi lahko ugotovi:

- če je hladilnik poln
- če so vrata hladilnika zaprta

Zamislimo si sedaj, da imamo krokodila in bi od robota radi, da ga spravi v hladilnik. Kako bi robotu to dopovedali? Glede na to, da robot razume ukaz "Daj *žival* v hladilnik", ali mu lahko preprosto velimo naj "Da krokodila v hladilnik"? Seveda lahko, vendar je verjetnost, da bo operacija uspela, zelo majhna. Če so, na primer, vrata zaprta, robot ne bo mogel spraviti krokodila v hladilnik, ker mu nismo ukazali, naj prej odpre vrata. Prav tako sam od sebe ne bo zaznal, da so vrata zaprta oziroma, da je v hladilniku že slon, če mu izrecno ne velimo, naj potipa vrata oziroma notranjost hladilnika. Pravilno bi robotu problem predočili takole:

- Če so vrata hladilnika zaprta:
  Odpri vrata.
- Če je hladilnik poln:
  Vzemi slona iz hladilnika.
- Daj krokodila v hladilnik.
- Zapri vrata.

Tako smo sestavili *algoritem*. Algoritem je niz navodil, ki opisujejo, kako pridemo do rešitve problema. Algoritme pogosto zapisujemo v nekakšni mešanici računalniškega jezika in slovenščine, čemur pravimo tudi *psevdo koda*. Algoritme bomo zapisovali v takšni obliki:

```
ALGORITEM                    spravi krokodila
if vrata hladilnika zaptra
   odpri vrata
if hladilnik poln
   sprazni hladilnik
spravi krokodila v hladilnik
zapri vrata
```

Načrtovanje z vrha navzdol

Pogost problem pri programiranju je, kako nalogo pravilno razčleniti na zaporedje opravil, ki jih lahko neposredno predočimo računalniku. Problem ponavadi členimo na vedno drobnejše (pod)probleme, dokler ne pridemo do tako elementarnih opravil, da jih lahko zapišemo direktno v programskem jeziku, v katerem želimo programirati.

Oglejmo si primer, ko dobimo za nalogo organizirati poroko. Problem v prvem koraku razčlenimo na naslednja opravila:

- Rezerviraj matičarja, duhovnika, restavracijo, ...
- Povabi goste
- Organiziraj kupovanje daril
- Sestavi primeren meni
- Poskrbi za glasbo
- Poskrbi za hrano in pijačo

Vsakega od teh opravil lahko razčlenimo na preprostejše komponente. Na primer:

Povabi goste:
- Vzemi ženinov seznam
- Vzemi nevestin seznam
- Preveri neskladja
  -med obema seznamoma
  -med končnim seznamom in starši
- Razpošlji vabila

Takšno splošno strategijo drobljenja na enostavnejša opravila pogosto uporabljamo pri programiranju. Postopku pravimo *načrtovanje z vrha navzdol*. Postopek je najbolj učinkovit, kadar so opravila med seboj neodvisna. Če so opravila močno odvisna drug od drugega, potem je postopek manj učinkovit. Na primer:

Seznam gostov je dolg ⇒ Rezerviraj velik prostor za žur
⇒ Preveliki stroški
⇒ Skrajšaj seznam gostov
⇒ Rezerviraj manjši prostor...

Proti koncu semestra, ko bomo vedeli malo več o Ceju, si bomo ta postopek ogledali podrobneje na primeru računalniškega problema.

Začnimo že enkrat s Cejem

Spremenljivke

Vemo že, da program med svojim izvajanjem potrebuje prostor za začasno hranjenje podatkov, ki se ustvari v pomnilniku. Podatkom pravimo spremenljivke in te ustvarimo (deklariramo) takole:

SKLADNJA deklaracija spremenljivke
`tip ime_spremenljivke;`

Spremenljivki dodelimo poljubno ime in katerega od vgrajenih ali uporabniško definiranih tipov. Pri izbiri imena veljajo določene omejitve. Ime lahko vsebuje velike in male črke angleške abecede, desetiške cifre in podčrtaj (_), ki ga navadno uporabljamo namesto presledka, kadar bi si želeli, da je ime spremenljivke sestavljeno iz več besed. Presledka v imenu namreč ne smemo uporabiti. Velja tudi omejitev, da prvi znak imena ne sme biti cifra. Za ime ne smemo izbrati katere od rezerviranih besed, kot sta na primer `while` ali `if`.

Pri izbiri imen ne smemo pozabiti, da **C loči velike in male črke**. Tako je `miha` nekaj drugega kot `Miha`, in `MIHA` je spet nekaj popolnoma tretjega.

Tipi spremenljivk

Od tipa spremenljivke bo odvisno, koliko pomnilniških celic bo zasedla in na kakšen način bo njena vrednost zapisana v pomnilniku. Za potrebe našega predmeta se bomo zadovoljili z naslednjimi tipi:

opis	oznaka	dolžina (bajtov)	formatno določilo
najmanjša naslovljiva enota računalnika. Gre za celo število, ki je lahko bodisi predznačeno ali nepredznačeno (odvisno od izvedbe)	`char`	običajno 1	%d, %c
predznačeno celo število	`short`	vsaj 2	%d
osnovno predznačeno celo število	`int`	vsaj 2	%d
predznačeno celo število	`long`	vsaj 4	%ld
realno št. v plavajoči vejici, enojna natančnost	`float`	običajno 4	%f
realno št. v plavajoči vejici, dvojna natančnost	`double`	običajno 8	%lf
podatkovni tip brez vrednosti (prazen)	`void`		

Pred deklaracijo celoštevilskih tipov iz gornje tabele lahko dodamo še *modifikator* `signed` (predznačen) ali `unsigned` (nepredznačen), s katerim eksplicitno zahtevamo, ali naj bo tip predznačen ali nepredznačen. Razlika med enim in drugim je v možnem predznaku, ki je pri nepredznačenih številih lahko zgolj pozitiven, pri predznačenih pa je laho tudi negativen. Če se odločimo za nepredznačen tip, potem moramo črko `d` v formatnem določilu zamenjati s črko `u`. Tako bo imelo, na primer, formatno določilo za tip `unsigned long` oblikio `%lu`.

Zastavi se vprašanje, zakaj imamo toliko različnih tipov. Različni tipi nam omogočajo, da se odločamo med porabo pomnilnika in *območjem* (pri celih številih) oz. *natančnostjo* (pri realnih številih) zapisa števil. Iz matematike dvojiških števil se boste spomnili, da lahko z n dvojiškimi mesti zapišemo 2^n različnih števil. Velja naslednje:

-v nepredznačeno celoštevilsko spremenljivko dolžine n bitov lahko spravimo katerokoli vrednost med vključno 0 in 2^n-1. Največja vrednost, ki jo lahko spravimo je za ena manjša od števila različnih možnih vrednosti, ker je v to število všteta tudi nula. Z osmimi biti, na primer, dobimo 256 različnih vrednosti med vključno 0 in 255.

-v predznačeno celoštevilsko spremenljivko dolžine n bitov lahko spravimo katerokoli vrednost med vklučno -2^{n-1} do 2^{n-1}-1. Tudi to je 2^n različnih vrednosti.

Pri realnih številih območje navadno ni problematično, omejeni pa smo z *natančnostjo*, s katero so števila zapisana. Vrednosti so zapisane bodisi z enojno (`float`) bodisi z dvojno (`double`) natančnostjo: pri enojni natančnosti je točnih prvih 7, pri dvojni pa prvih 15 desetiških mest, pri čemer se začne šteti pri prvi cifri, ki je različna od nič. Omenimo naj še, da vsake desetiške vrednosti ni mogoče brez napake pretvoriti v zapis s plavajočo vejico. Če vrednosti ne moremo sestaviti kot vsote (pozitivnih in negativnih) potenc števila 2, potem vrednosti ne moremo zapisati brez napake. Vrednost 0.625 (= 1/2 + 1/8), na primer, lahko zapišemo brez napake, vrednosti 0.2 pa ne. Ker so konstante zapisane z dvojno natančnostjo, nas bo prevajalnik opozoril na zaokroževanje v naslednjem primeru:

```
float x = 0.2; //Warning: truncation from double to float
```

V tem primeru pa zaokroževanja ni:

```
float x = 0.625;
```

Modifikatorji formatnih določil

Kot smo že spoznali, moramo za izpisovanje vrednosti s funkcijo `printf()` uporabiti pravilno formatno določilo. Poleg tipa podatka pa lahko z ustreznimi modifikatorji določimo tudi obliko izpisa. Osnovni trik je, da predpišemo število mest, ki naj se uporabijo za izpis ter določimo, kako naj se zapolnijo vodilna mesta, če želimo izpis na več mest kot ima izpisana vrednost cifer. Pri celih številih z modifikatorjem povemo, koliko mest želimo porabiti za izpis vrednosti. Izpisana vrednost se bo poravnala ob desni rob predpisanega prostora. Manjkajoča mesta na levi se bodo zapolnila s presledki oz. z ničlami, če pred modifikator dodamo ničlo. Na primer:

```
int D = 15;
printf("D=%5d", D );   //izpiše D=   15
printf("D=%05d", D );  //izpiše D=00015
```

Pri realnih številih lahko poleg števila mest (šteje tudi decimalna pika) določimo še število mest, ki naj se izpišejo za decimalnio piko. Število mest v modifikatorju zapišemo pred piko, število decimalnih mest pa za piko. Na primer:

```
float F = 15.731;
printf("F=%.2f", F);     //izpiše F=15.73
printf("F=%5.2f", F);    //izpiše F=15.73
printf("F=%6.2f", F);    //izpiše F= 15.73
printf("F=%08.4f", F);   //izpiše F=015.7310
```

Deklaracija večih spremenljivk in prirejanje vrednosti

Kadar imamo v programu več spremenljivk istega tipa, jih lahko vse deklariramo v eni vrstici:

SKLADNJA	deklaracija več spremenljivk istega tipa
`tip ime_spremenljivke1, ime_spremenljivke2, ...;`	

Spremenljivka ima poleg imena in tipa še *vrednost*. Vrednost spremenljivke lahko nastavimo že ob njeni deklaraciji in jo kjerkoli kasneje v programu spremenimo. V obeh primerih nastavimo vrednost spremenljivke s *priredilnim operatorjem* (=). Priredilni operator deluje tako, da vrednost, ki mu jo podamo na desni, zapiše v spremenljivko, ki je na levi:

```
ime_spremenljivke = vrednost;
```

Zato mora biti **na levi strani priredilnega operatorja vedno spremenljivka ali izraz, ki določa pomnilniško lokacijo, kamor lahko priredilni operator vpiše vrednost**. Temu pravimo strokovno tudi *lvalue* (angl. left value = vrednost na levi). Bojda zato, ker stoji na levi strani priredilnega operatorja, menda pa tudi zato, ker je pomnilniška lokacija določena z naslovom, ki ga na skicah pomnilnika navadno označujemo levo od lokacije, ki vsebuje dejansko vrednost.

Dovolj smo nakladali, čas je že, da se pozabavamo s primerom. Naslednji program je namenjen učenju poštevanke do 100. Program naključno generira račune in jih izpisuje na zaslon. Uporabnik mora sproti vnašati rezultate, program jih preveri in šteje pravilne in nepravilne odgovore. Ko uporabnik pravilno reši 5 računov, program izpiše odstotek pravilnih izračunov in se zaključi:

IZPIS IZVORNE KODE postevanka.c

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

unsigned int prav = 0, vseh = 0;
unsigned int mnoz1, mnoz2, rezultat;
unsigned int odgovor;
unsigned int odstotek;

const int kriterij = 5;

int main()
{
   srand((unsigned)time(NULL));
   printf("Dobrodošel v poštevanki!\n");
```

```
   printf("Pravilno moraš rešiti 5 računov,\n");
   printf("potem greš lahko gledat televizijo.\n\n");
   do
   {
      mnoz1 = rand() % 10 + 1;
      mnoz2 = rand() % 10 + 1;
      printf("%u * %u = ", mnoz1, mnoz2);
      rezultat = mnoz1 * mnoz2;
      scanf("%u", &odgovor);
      if (odgovor == rezultat)
      {
             prav = prav + 1;
      }
      else
      {
             printf ("Narobe. Prav je %u\n", rezultat);
      }
      vseh = vseh + 1;
   } while (prav < kriterij);
   odstotek = 100 * prav / vseh;
   printf("Dosegel si %u%% uspeh.\nZdaj pojdi.\n", odstotek);
   return 0;
}
```

Prva novost, ki jo opazimo v tem programu, sta dve novi knjižnici (`stdlib.h` in `time.h`). Prvo rabimo zaradi funkcij `srand()` in `rand()`, ki ju potrebujemo za generiranje psevdonaključnih števil. Psevdonaključna števila niso zares naključna, ampak imajo samo statistične lastnosti podobne naključnemu procesu. Zato potrebujemo `srand()`, ki nam nastavi generator naključnih števil na primerno začetno vrednost, sicer bi se generirano zaporedje psevdonaključnih števil začelo vedno popolnoma enako. To stori tako, da prebere sistemski čas v trenutku njenega klica, in ta čas se smatra za naključno vrednost (uporabnik zažene program ob naključnem času). V ta namen kličemo funkcijo `time()`, ki je v knjižnici `time.h`.

Sledijo deklaracije spremenljivk. Ker bomo imeli opravka izključno z nenegativnimi števili, smo deklarirali spremenljivke kot nepredznačena števila. Seveda nič ne bi bilo narobe, če bi izbrali katerikoli drug celoštevilski tip, kajti nobena vrednost ne bo presegla 100.

Za spremenljivkami smo definirali konstantno vrednost `kriterij`. Konstante definiramo v naslednji obliki:

SKLADNJA	definicija konstante
`const tip ime_konstante = vrednost;`	

Konstantna vrednost se prav tako kot ostale spremenljivke zapiše v pomnilnik, vendar se njena vrednost med delovanjem programa ne more spreminjati.

Glavno dogajanje poteka znotraj ponavljalnega stavka `do..while`. Program najprej izbere dva naključna množenca, izpiše račun na zaslon, izračuna pravilen rezultat in prebere odgovor, ki ga uporabnik vnese prek tipkovnice. Nato odgovor primerja s pravilnim rezultatom in v primeru, da je odgovor pravilen, poveča števec pravilnih odgovorov za ena. V nasprotnem primeru izpiše pravilen odgovor. Na koncu še poveča števec poskusov za ena:

```
vseh = vseh + 1;
```

Če je pravilnih odgovorov manj kot smo zahtevali, se postopek ponovi, sicer se ponavljalni stavek zaključi. Na koncu se izračuna in izpiše še odstotek pravilnih odgovorov. Za izpis znaka za odstotek (`%`) moramo napisati dva takšna znaka, enega za drugim, kajti en sam znak predstavlja začetek formatnega določila.

Funkcija `rand()` deluje tako, da vrne naključno vrednost med 0 in `RAND_MAX`. Pri tem je `RAND_MAX` odvisen od knjižnice, ki jo uporabljate, vendar nikakor ne bo manjši od 32767. Če vrednost, ki jo funkcija `rand()` vrne, delimo z določeno celoštevilsko vrednostjo, manjšo od 32767 in vzamemo le ostanek deljenja, lahko omejimo nabor naključnih vrednosti. Tako dobimo s klicem

```
rand() % 10
```

naključno vrednost med 0 in 9. Ker potrebujemo vrednosti med 1 in 10, moramo temu prišteti še ena. Tako dobljeno vrednost prenesemo v ustrezen množenec s priredilnim operatorjem:

```
mnoz1 = rand() % 10 + 1;
```

Ta stavek in še nekaj ostalih stavkov v programu `postevanka.c`, ki vsebujejo priredilni operator, zahteva malo več pozornosti. V nadaljevanju bomo govorili o cejevskih izrazih (angl. expression).

Preden zares nadaljujemo, ne bo odveč, če si pogledamo še primer delovanja naše poštevanke:

```
DELOVANJE PROGRAMA          postevanka.exe
Dobrodošel v poštevanki!
Pravilno moraš rešiti 5 računov,
potem greš lahko gledat televizijo.

10 * 10 = 100
4 * 6 = 24
9 * 7 = 73
Narobe. Prav je 63
2 * 3 = 6
2 * 2 = 4
7 * 4 = 28
Dosegel si 83% uspeh.
Zdaj pojdi.
_
```

Izrazi

Izraz je sestavljen iz enega ali več operandov, ki so lahko

- konstante (npr.: 47, 'k', "hojladrajsa")
- spremenljivke (npr.: x, napetost, stevec)
- klici funkcij (npr.: rand(), printf("%d", y))

Operandi so povezani med seboj z *operatorji*. Operatorji določajo, kakšne operacije naj se izvajajo nad operandi. Izraz lahko vsebuje tudi pare okroglih oklepajev, ki določajo vrstni red izvajanja operacij. Tako kot spremenljivke ima tudi izraz svoj tip in vrednost. Tip izraza se določi avtomatično glede na tipe posameznih operandov in

vrste operatorjev. Vrednost izraza se izračuna glede na vrednosti operandov in vrsto operacije, ki se nad njimi izvaja. Poglejmo si enostaven primer.

Imejmo deklaraciji:

```
int a = 3;
int b = 8;
```

Tako bo imel izraz

```
a + b
```

predznačen celoštevilski tip, iz matematike namreč vemo, da je vsota dveh celih števil tudi celo število. Vrednost izraza je seveda 11. Pri tem ne bo odveč opozoriti, da izraz `a+b` sam zase nima nobenega učinka. Operator seštevanja (`+`) namreč ne vpliva na vrednosti sumandov. Gornji izraz pa bi lahko uporabili za gradnjo priredilnega stavka:

```
vsota = a + b;
```

Ko se izvrši ta stavek, dobi vsota vrednost izraza `a + b`. Spomnimo se namreč, ko smo govorili o tem, da priredilni operator (`=`) vrednost, ki jo dobi z desne, zapiše v spremenljivko na svoji levi strani. Zdaj se lahko bralec vrne nazaj na program `postevanka.c` in skuša ugotoviti, kako delujejo posamezni priredilni stavki, ki jih je v programu kar nekaj.

Zdaj smo že dobili nekaj občutka za gradnjo cejevskih izrazov. Cejevski izrazi imajo vsi svoj tip in vrednost. Najenostavnejši izrazi vsebujejo zgolj konstanto, spremenljivko ali funkcijo. Poljubne izraze lahko združujemo z operatorji in tako dobimo nove, kompleksnejše, izraze, ki imajo spet svoj tip in vrednost.

Da bomo lahko gradili izraze, moramo najprej spoznati nekaj operatorjev. Oglejmo si najprej priredilni operator in aritmetične operatorje.

Priredilni operator

Priredilni operator smo že velikokrat srečali. Zdaj, ko smo spoznali, kaj je to izraz, lahko zapišemo splošno obliko uporabe tega operatorja:

SKLADNJA	priredilni izraz
spremenljivka = izraz	

Tako dobljenemu izrazu, ki je sestavljen iz spremenljivke na levi (spomnimo se, da temu rečemo tudi lvalue), priredilnega operatorja in poljubnega izraza na desni, pravimo *priredilni izraz*. Če pa pristavimo na koncu še podpičje, potem dobimo *priredilni stavek*.

Enostaven primer priredilnega stavka je

```
odstotek = 100 * prav / vseh;
```

ki smo ga srečali v programu `postevanka.c`. Tu dobi spremenljivka `odstotek` vrednost izraza, ki je na desni. V izrazu nastopa konstanta `100`, spremenljivki `prav` in

`vseh` ter operatorja množenja in deljenja. Operatorja spadata med aritmetične operatorje, o katerih bomo takoj povedali še nekaj besed.

Malo prej smo povedali, da ima vsak izraz svojo vrednost. Vrednost priredilnega izraza je enaka vrednosti, ki se prenese z desne strani priredilnega operatorja na levo.

Aritmetični operatorji

C pozna naslednje operatorje, ki izvajajo aritmetične operacije in vračajo kot rezultat številsko vrednost:

operator	kaj naredi
`+`	seštevanje in pozitiven predznak
`-`	odštevanje in negativen predznak
`*`	množenje
`/`	deljenje
`%`	ostanek celoštevilskega deljenja

Povemo naj še to, da operator deljenja vrne celoštevilski rezultat, če sta deljenec in delitelj oba celoštevilska izraza. Pri tem ne zaokroža rezultata, temveč poreže del za decimalno piko. Če je vsaj en izraz realnega tipa, operator vrne realno vrednost. Izraz

```
9 / 4
```

bo imel tako vrednost 2, izraz

```
9 / 4.0
```

pa 2,25, kajti prevajalnik tolmači konstantno vrednost `4.0` kot realno število, ker vsebuje decimalno piko.

Pogosto se pripeti, da v izrazu deljenja nastopajo celoštevilske spremenljivke, mi pa vseeno želimo realen rezultat. Takrat lahko zahtevamo začasno pretvorbo tipa tako, da pred ime spremenljivke v oklepajih navedemo ime ustreznega tipa:

```
(float)x
```

Kadar je v izrazu več aritmetičnih operatorjev, je pomemben vrstni red, v katerem se izvajajo. Najprej se izvede negativen predznak, potem množenje, deljenje in ostanek ter na koncu seštevanje in odštevanje. Za operatorje, ki se izvedejo prej, pravimo, da imajo višjo prioriteto. Če je v izrazu zapovrstjo več aritmetičnih operatorjev iste prioritete (na primer sama seštevanja in odštevanja), potem se ti izvajajo večinoma z leve proti desni. Drugačen vrstni red določimo z okroglimi oklepaji.

V spodnji tabeli vidimo nekaj zgledov. Študent lahko poskusi za vajo sam napisati kratek program, s katerim bo preveril pravilnost tabele. Poskusite napisati še kakšne druge izraze, predvideti njihov rezultat in ga preveriti z računalnikom.

Pri izračunu izrazov v tabeli smo upoštevali naslednje deklaracije:

```
int x = 10, y = 3;
float a = 13.1, b = 3;
```

izraz	vrednost
`x * -5 + 6 * a`	28,6000
`x / y`	3
`x / b`	3,3333
`(9 + y) % x`	2
`x - - 3`	13
`y * + 5`	15

Zdaj lahko še razmislite in tudi preverite, kaj bi na koncu izpisal program `postevanka.c`, če bi se stavek za izračun odstotka glasil

```
odstotek = prav / vseh * 100;
```

Prioriteta in asociativnost operatorjev

Kadarkoli v enem izrazu nastopa več operatorjev, je pomembno, da vemo, v kakšnem vrstnem redu se ti izvajajo. Za vse operatorje veljata pravili prioritete in asociativnosti. Po *pravilu prioritete* (ang. precedence) se najprej izvedejo operatorji z višjo prioriteto. Če je več operatorjev iste prioritete, potem se ti izvajajo po vrstnem redu, ki ga določa *pravilo asociativnosti* (z leve proti desni oziroma z desne proti levi).

V naslednji tabeli so zbrani vsi cejevski operatorji. Operatorji v višjih vrsticah imajo višjo prioriteto. Operatorji, ki so v isti vrstici, imajo enake prioritete.

operatorji	vrstni red izvajanja
`()` `[]` `->` `.` `++`(za operandom) `--`(za operandom)	z leve proti desni
`!` `~` `++`(pred operandom) `--`(pred operandom) `+`(predznak) `-`(predznak) `*`(indirekcija) `&`	z desne proti levi
`*` `/` `%`	z leve proti desni
`+` `-`	z leve proti desni
`<<` `>>`	z leve proti desni
`<` `<=` `>` `>=`	z leve proti desni
`==` `!=`	z leve proti desni
`&`	z leve proti desni
`^`	z leve proti desni
`|`	z leve proti desni
`&&`	z leve proti desni
`||`	z leve proti desni

`?: = += -= *= /= %= &= ^= |= <<= >>=`	z desne proti levi
`,`	z leve proti desni

Ponavljanje je mati modrosti

Zato bomo v tem poglavju med drugim spoznali še en ponavljalni stavek.

Računalnik naj računa

Naslednji program izračuna fakulteto (faktorielo) vrednosti, ki jo vnesemo prek tipkovnice. Fakulteta števila *n* je definirana kot produkt celih števil med vključno 1 in *n*:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

IZPIS IZVORNE KODE fakulteta.c

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

unsigned long fak = 1;
int i, n;

int main()
{
   printf("Vnesi vrednost, jaz ti bom zračunal fakulteto: ");
   scanf("%d", &n);
   for (i = 2; i <= n; i++)
   {
      fak *= i;
   }
   printf("%d! = %lu\n", n, fak);
   return 0;
}
```

V programu `fakulteta.c` najdemo kar nekaj reči, o katerih še nismo govorili. Poglejmo jih lepo po vrsti.

Ponavljalni stavek for

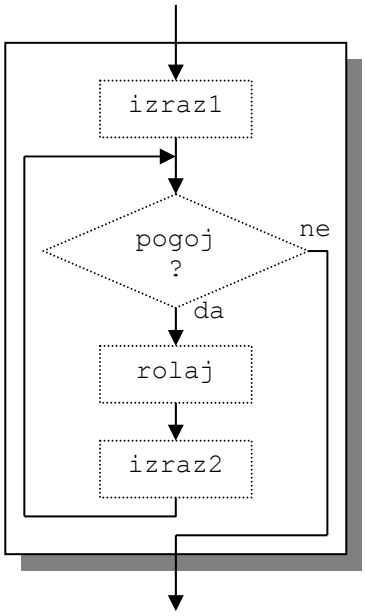
Stavek `for`, podobno kot `do..while` uporabljamo, kadar želimo, da se kakšen del programske kode izvrši večkrat. Ponavadi ga uporabljamo v primerih, ko je že vnaprej (preden se začne stavek `for` izvajati), znano, koliko ponovitev bo potrebnih.

Stavek zapišemo takole:

SKLADNJA ponavljalni stavek for

```
for (izraz1; pogoj; izraz2) rolaj
```

Stavek ima kar kompleksno zgradbo in najbolje bo, da si naslikamo vrstni red izvajanja izrazov in stavkov v njem z diagramom poteka.



Vidimo, da se najprej izvrši `izraz1`, ki se izvrši le enkrat. Potem se preveri `pogoj` in če je ta izpolnjen, se izvede stavek ali blok stavkov `rolaj` in na koncu še izraz `izraz2`. Izvajanje stavka se nato vrne na preverjanje pogoja in od tu se vse ponovi. Ko pogoj ni več izpoljnjen, se izvajanje stavka `for` konča. Opazimo tudi, da če `pogoj` že na začetku ni izpolnjen, se `rolaj` in `izraz2` sploh ne izvršita.

Kot smo že povedali, ta stavek največkrat uporabljamo v primerih, ko že vnaprej vemo, koliko ponovitev potrebujemo. Takrat v izrazu `izraz1` nastavimo nek števec na začetno vrednost, v izrazu `izraz2` ta števec ob vsakem obhodu povečamo (ali pomanjšamo) za ena, s pogojnim izrazom pa preverjamo, če je števec že dosegel končno vrednost.

V programu `fakulteta.c` smo za števec uporabili spremenljivko `i`, ki smo jo na začetku postavili na dve in jo povečevali vse do vrednosti spremenljivke `n`.

Da bomo povsem razumeli, kaj se v programu dogaja, se pogovorimo še malo o operatorjih.

Primerjalni operatorji

Uporaba primerjalnih, podobno kot aritmetičnih, operatorjev je v veliko primerih intuitivna. Če jih želimo polno izkoristiti in se zaščititi pred napačno uporabo, pa moramo o njih vedeti malo več.

C pozna naslednje primerjalne operatorje:

operator	kaj preverja
`==`	je enak?
`!=`	je različen?
`<`	je manjši?
`<=`	je manjši ali enak?
`>`	je večji?
`>=`	je večji ali enak?

S primerjalnimi operatorji gradimo primerjalne izraze, ki imajo lahko le dve različni vrednosti. V primeru, da je odgovor na vprašanje v tabeli poleg operatorja pritrdilno, je vrednost izraza ena, sicer je nič.

Zadnji štirje operatorji v tabeli (<, <=, > in >=) imajo vsi enako prioriteto, ki je večja od prioritete prvih dveh operatorjev. Kakšna je njihova prioriteta glede na ostale operatorje, si lahko ogledate na tabeli na strani 21.

Oglejmo si nekaj primerov primerjalnih izrazov in njihovih vrednosti ob predpostavki, da imamo v programu naslednjo deklaracijo:

```
int x = 10, y;
```

izraz	vrednost
`5 != x`	1
`10 > x`	0
`10 >= x`	1
`2 < y < 7`	1

Prvi trije izračuni so očitni, pri zadnjem pa bo verjetno večina bralcev menila, da gre za napako, saj nismo podali vrednosti spremenljivke `y` in potemtakem ne moremo ugotoviti, ali je trditev pravilna ali ne. V resnici ne gre za napako in vrednost izraza je pravilno izračunana.

Finta je v tem, da moramo poznati delovanje primerjalnih operatorjev. Če imamo v enem izrazu več operatorjev, se jih nikoli ne more izvršiti več hkrati. Operatorji se izvajajo eden za drugim po pravilih prioritete in asociativnosti. Tisti z višjo prioriteto se izvedejo prej, če pa je več operatorjev enake prioritete, se vrstni red določi po pravilu asociativnosti (z leve proti desni oziroma z desne proti levi). Primerjalni operatorji se izvajajo z leve proti desni.

Izraz

```
2 < y < 7
```

se tako izvede v dveh korakih. Najprej se izračuna izraz `2 < y`. Vrednost tega izraza je lahko le nič ali ena. V drugem koraku se dobljena vrednost uporabi kot levi operand drugega primerjalnega operatorja, in ker sta nič in ena obe manjši od sedem, je izračunana vrednost v vsakem primeru ena. Končna vrednost celotnega izraza je torej ena.

V praksi takšnih izrazov ne najdemo, je pa to tipičen primer napačne uporabe primerjalnih operatorjev. Ta napačna uporaba izvira iz dejstva, da je zapis matematično popolnoma pravilen. Kako po cejevsko ugotovimo, če leži neka spremenljivka znotraj določenega področja, bomo videli malo kasneje v poglavju o logičnih operatorjih.

Zdaj, ko vemo, kako delujejo primerjalni operatorji, si poglejmo še, kako odločitveni stavki obravnavajo izraze, ki jim jih podamo za pogoj. Velja, da lahko **za pogoj**

vstavimo poljubni izraz, ki vrne številsko vrednost. Če je vrednost izraza različna od nič, se smatra, da je pogoju zadoščeno, sicer ne.

Vzemimo za primer naslednji kos cejevskega programa:

```
int kriterij = 2;
if (kriterij)
{
      printf("Kriterij je izpolnjen\n");
}
else
{
      printf("Kriterij ni izpolnjen\n");
}
```

Ker je vrednost izraza v okroglih oklepajih različna od nič, se bo na zaslon izpisal stavek `Kriterij je izpolnjen.`

Preden se poslovimo od primerjalnih operatorjev, dajmo v razmislek še naslednjo kodo:

```
float pi = 3.14;
if (3.14 == pi)
{
      printf("Pi je enak 3.14\n");
}
else
{
      printf("Pi začuda ni enak 3.14\n");
}
```

Na zaslonu se bo izpisalo `Pi začuda ni enak 3.14`. Primerjalni operator (`==`) vrne vrednost ena namreč le v primeru, ko sta oba operanda, ki ju primerja, enaka do zadnjega bita. 3.14 se ne da brez napake pretvoriti v zapis s plavajočo vejico, tako se primerjata dva *približka* vrednosti 3.14. Prvi približek je tipa `float` (spremenljivka `pi`), drugi pa `double` (konstanta `3.14`). Cejevski prevajalnik namreč realne konstante pretvori v 64 bitni zapis s plavajočo vejico (`double`).

Nauk te zgodbe je ta, da realnih števil zaradi približkov, ki se jim ne moremo izogniti, ni dobro primerjati z operatorjem, ki primerja enakost (`==`) ali različnost (`!=`).

Operatorji spreminjanja vrednosti za ena

Zelo pogosta operacija je zmanjševanje ali povečevanje vrednosti spremenljivk za ena. Namesto dolgega zapisa, ki smo ga uporabljali doslej:

```
prav = prav + 1;
```

lahko uporabimo operator povečevanja vrednosti za ena (`++`):

```
prav++;
```

Oba stavka povečata vrednost spremenljivke `prav` za ena. Obstaja seveda tudi operator, ki zmanjša vrednost spremenljivke za ena (`--`):

```
i--;
```

Kadar spremenljivka, ki jo podvržemo kateremu od omenjenih dveh operandov, nastopa v kakšnem zapletenejšem izrazu, je pomembno, *kdaj* se njena vrednost poveča. Lahko se poveča, predno jo uporabimo v izračunu izraza, lahko pa se poveča šele po tem, ko smo jo že uporabili. Naslednja tabela kaže dve možni uporabi obeh operatorjev.

zapis	učinek
`i++`	poveča vrednost po uporabi
`++i`	poveča vrednost pred uporabo
`i--`	zmanjša vrednost po uporabi
`--i`	zmanjša vrednost pred uporabo

Poglejmo si primer:

```
int x = 1, y = 1, xx, yy;
xx = x++;
yy = ++y;
```

Ko se bo izvedla gornja koda, bodo imele vse spremenljivke vrednost 2, razen spremenljivke `xx`, ki bo imela vrednost 1. Tej spremenljivki smo namreč priredili vrednost spremenljivke `x`, ki smo jo sicer povečali za ena, ampak šele po uporabi (po izvršitvi operacije prirejanja). Drugače je v zadnjem stavku, kjer spremenljivko `y` tudi povečamo za ena, vendar to storimo, preden jo uporabimo v priredilnem stavku.

Kombinirani operatorji

Včasih želimo, da se vrednost enega od operandov, ki nastopajo v aritmetičnem ali kakem podobnem izrazu, spremeni. Takrat lahko v določenih okoliščinah uporabimo ustrezen *kombiniran operator*. Kombiniran operator dobimo tako, da običajnemu operatorju dodamo enačaj. S kombiniranim operatorjem množenja smo v programu `fakulteta.c` množili spremenljivko `fak` z vrednostjo spremenljivke `i`:

```
fak *= i;
```

Pri tem se je vrednost spremenljivke `fak` spremenila na enak način, kot če bi zapisali:

```
fak = fak * i;
```

Vrednost izraza, ki uporablja kombinirani operator, je enaka končni vrednosti spremenljivke na levi strani operatorja.

Ostale kombinirane operatorje, ki jih pozna C, si lahko ogledate v predzadnji vrstici tabele na strani 21.

Kako zdaj to deluje?

Zdaj so nam vsi zapisi, ki smo jih uporabili v programu `fakulteta.c` znani. Program je v resnici zelo preprost in če ga zaženemo, lahko vidimo takšen izpis:

```
DELOVANJE PROGRAMA          fakulteta.exe
Vnesi vrednost, jaz ti bom zračunal fakulteto: 11
11! = 39916800
_
```

Kdor zbira znanje, zbira tudi trud in muko (Michael E. de Montaigne)

Vsi podatki (spremenljivke), ki smo jih uporabljali doslej, so imeli obliko bodisi celih bodisi realnih števil. V matematiki takšnim podatkom pravimo skalarji, v računalništvu pa jih kličemo tudi *enostavni podatki*. Poleg enostavnih podatkov poznamo tudi *sestavljene podatke*, med katere spadajo *zbirke* (angl. array). V slovenski literaturi boste za zbirko zasledili tudi ime *polje*.

Zbirka

Matematično gledano je zbirka vektor. Zbirka združuje dva ali več podatkov oziroma *elementov* istega tipa, ki imajo vsi enako ime, med sabo pa jih ločimo s celoštevilskim *indeksom*.

Tako kot enostavne spremenljivke, moramo tudi zbirko deklarirati, preden jo lahko uporabimo v programu. Deklaracija zbirke se glasi takole:

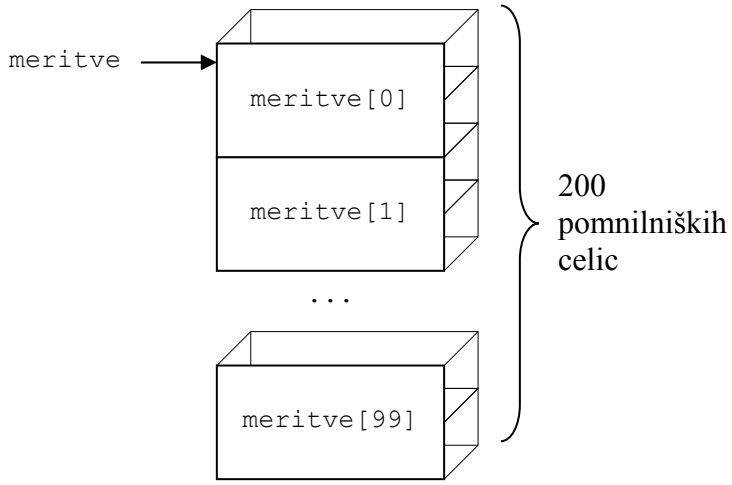
```
SKLADNJA            deklaracija zbirke
tip_elementa ime_zbirke[število_elementov];
```

S tem ustvarimo zbirko, ki obsega `število_elementov` elementov tipa `tip_elementa`.

Na primer:

```
short meritve[100];
```

ustvari zbirko 100 elementov tipa `short`. Elementi se v pomnilniku nanizajo eden za drugim in skupaj zasedejo toliko pomnilnika, kolikor bi ga enako število enostavnih spremenljivk istega tipa. `ime_zbirke` predstavlja pomnilniški naslov začetka prvega elementa v zbirki. O naslovih bomo podrobneje govorili v poglavju o kazalcih. V primeru zbirke `meritve` dobimo v pomnilniku takšno stanje:



Operacije nad zbirkami v Ceju izvajamo tako, da obdelujemo posamezne elemente. Do elementa zbirke pridemo tako, da imenu zbirke dodamo ustrezen indeks, ki ima lahko vrednost med nič in ena manj od števila elementov v zbirki. Na gornji sliki smo za 100 elementov zbirke `meritve` uporabili indekse med 0 in 99. Posamezne elemente obravnavamo na povsem identičen način kot spremenljivke istega tipa. Tako na primer 42. meritvi priredimo vrednost 93 z naslednjim priredilnim stavkom:

```
meritve[41] = 93;
```

42. element smo izbrali tako, da smo imenu zbirke dodali indeks 41 v oglatih oklepajih. Indeks smo tu zapisali kot konstantno vrednost, vendar lahko v splošnem za indeks podamo poljuben celoštevilski izraz.

Osvetlimo vse skupaj s primerom.

Meteorološki podatki za mesec februar

Naslednji program ima v zbirki `t_maks` shranjene najvišje dnevne temperature za vseh 28 februarskih dni. Program najprej zračuna srednjo vrednost temperature za ves mesec, potem pa izpiše zaporedne številke dni, v katerih je temperatura presegla povprečno mesečno temperaturo. Program za te dni izpiše tudi najvišjo dnevno temperaturo:

IZPIS IZVORNE KODE temperature.c

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

float t_maks[28] = {3.4, 2.7, 2.2, 3.1, 3.0, 7.9, 3.6,
                    3.0, 4.4, 1.7, 1.3, 2.0, 1.9, 2.7,
                    3.0, 1.8, 2.3, 0.1, 4.7, 8.4, 11.1,
                    10.0, 11.7, 3.9, 2.6, 5.0, 1.9, 1.2};
float t_sred = 0;
int i;

int main()
{
   for (i = 0; i < 28; i++)
   {
      t_sred += t_maks[i];
   }
   t_sred /= 28;
```

```
   printf("Dnevi z najvišjimi temperaturami,\n");
   printf("višjimi od mesečnega povprečja (%.1f):\n", t_sred);
   for (i = 0; i < 28; i++)
   {
      if (t_maks[i] > t_sred)
      {
         printf("%d. februar: %.1f stopinj\n", i + 1, t_maks[i]);
      }
   }
   return 0;
}
```

V programu smo zbirko `t_maks` ob njegovi deklaraciji tudi inicializirali. V splošnem izgleda deklaracija zbirke z inicializacijo takole:

SKLADNJA	deklaracija zbirke z inicializacijo
`tip ime_zbirke[stev_elementov] = {vredn_1, vredn_2, ..., vredn_n};`	

Pri tem mora veljati, da je število vrednosti, ki jih podamo v zavitih oklepajih, manjše ali enako velikosti (dimenziji) zbirke, ki smo jo podali v oglatih oklepajih:

```
n <= stev_elementov
```

Kadar je podanih manj vrednosti kot je elementov zbirke, se vsi ostali elementi nastavijo na nič. Upoštevaje to dejstvo, lahko na primer deklariramo zbirko 10 števcev in jih vse postavimo na nič takole:

```
int stevci[10] = {0};
```

Čim smo podali vrednost enega elementa, se ostalih devet avtomatično postavi na nič.

Če zbirko ob deklaraciji tudi inicializiramo, v oglatih oklepajih ni potrebno navesti števila elementov. V tem primeru bo prevajalnik sam ugotovil, koliko elementov ima zbirka in bo rezerviral zadostno količino pomnilnika:

SKLADNJA	deklaracija zbirke z inicializacijo
`tip ime_zbirke[] = { vredn_1, vredn_2, ..., vredn_n };`	

Sicer pa ni program nič posebnega. Najprej v stavku `for` seštejemo vseh 28 temperatur in nato dobljeno vsoto delimo z 28. Potem, spet v zanki `for`, za vsakega od elementov preverimo, če je njegova vrednost večja od povprečne. Če je, izpišemo zaporedno številko elementa, povečano za ena (elemente začnemo šteti od 0, dneve pa od 1), in vrednost elementa. Ker so vrednosti realna števila tipa `float`, smo za izpis uporabili formatno določilo `%f`. Kot smo že izvedeli, lahko pri izpisu realnih števil določimo, koliko mest za decimalno piko se bo izpisalo. To storimo tako, da v formatno določilo vrinemo piko in desetiško številko, ki določa število decimalnih mest, ki se jih bo izpisalo. V našem primeru smo želeli izpis temperatur z enim decimalnim mestom (`%.1f`).

Ko program zaženemo, dobimo na zaslonu takšen izpis:

DELOVANJE PROGRAMA	temperature.exe

`Dnevi z najvišjimi temperaturami,`
`višjimi od mesečnega povprečja (3.9):`
`6. februar: 7.9 stopinj`
`9. februar: 4.4 stopinj`	

```
19. februar: 4.7 stopinj
20. februar: 8.4 stopinj
21. februar: 11.1 stopinj
22. februar: 10.0 stopinj
23. februar: 11.7 stopinj
26. februar: 5.0 stopinj
_
```

Besedila

Z večanjem zmogljivosti računalnikov se spreminja tudi narava komunikacije med človekom in računalnikom, ki postaja vse bolj slikovno in zvočno usmerjena. Kljub temu pa tudi v računalništvu še vedno ostaja precejšnja potreba po pisani besedi, ki verjetno še dolgo ne bo izpodrinjena.

Pisana besedila so sestavljena iz znakov, zato si najprej oglejmo, kaj je znak in kako ga uporabljamo.

Znaki

V Ceju nimamo posebnega tipa, ki bi bil namenjen samo znakom. Znake shranjujemo v osembitni celoštevilski tip `char`, ki smo ga že spoznali. Namesto znaka samega se v pomnilnik shrani njegova ASCII koda. Pretvorba med osembitno ASCII kodo in dejanskim znakom se vrši na zahtevo programerja in izključno na nivoju komunikacije človek-računalnik. Računalnik interno obdeluje znake vedno kot osembitna cela števila.

Za branje in izpis znakov uporabljamo formatno določilo `%c`.

Kot primer vzemimo deklaracijo

```
char znak;
```

Naslednji stavek izpiše vrednost spremenljivke `znak` v obliki znaka:

```
printf("%c", znak);
```

Naslednji stavek s tipkovnice prebere poljuben znak in njegovo ASCII kodo shrani v spremenljivko `znak`:

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

Za primer si poglejmo kratek program, ki izpisuje ASCII kode znakov, ki jih vnašamo. Izvajanje programa se konča, ko vnesemo zvezdico (`*`).

```
IZPIS IZVORNE KODE          ascii.c
#include "stdafx.h"
#include <stdio.h>

char znak;

int main()
{
   printf("Vnasaj znake, jaz ti bom kazal njihove\n");
```

```
   printf("ASCII kode. Za konec vnesi zvezdico (*):\n");
   do
   {
      scanf("%c", &znak);
      fseek(stdin, 0, SEEK_END);
      printf("Znak %c ima kodo %d\n", znak, znak);
   } while (znak != '*');
   return 0;
}
```

Ko vtipkamo znak, moramo njegov vnos še potrditi s pritiskom na tipko `return`, da funkcija `scanf()` začne sprejemati vhodne podatke. Nastopi manjša težava, ker znak `return` ostane neprebran v medpomnilniku in naslednji klic funkcije `scanf()` bi prebral ta znak. Zato potrebujemo klic funkcije `fseek()`, ki postavi kurzor, ki kaže na vhodni tok podatkov, na konec tega toka. Tako je ob naslednjem klicu funkcije `scanf()` vhodni medpomnilnik prazen in funkcija čaka na nov vnos.

Pri klicu funkcije `printf()`, ki sledi klicu `fseek()`, vidimo, da je za obliko izpisa spremenljivke znakovnega tipa odgovorno izključno formatno določilo. Ista spremenljivka se namreč enkrat izpiše kot znak in drugič kot celoštevilska vrednost.

V programu `ascii.c` vidimo tudi, kako v cejevskem programu zapisujemo konstantne znake. V stavku `do..while` na koncu primerjamo spremenljivko `znak` s konstantnim znakom `*`. Znak moramo dati v enojne navednice. Namesto znaka bi lahko v program vpisali tudi njegovo ASCII kodo. Tako bi dobili isto, če bi se pogoj v stavku `do..while` glasil

```
znak != 42
```

42 je namreč ASCII koda zvezdice.

Če po zagonu programa vtipkamo zaporedje znakov `$6Z@*`, dobimo takšen izpis:

```
DELOVANJE PROGRAMA          ascii.exe
Vnasaj znake, jaz ti bom kazal njihove
ASCII kode. Za konec vnesi zvezdico (*):
$
Znak $ ima kodo 36
6
Znak 6 ima kodo 54
Z
Znak Z ima kodo 90
@
Znak @ ima kodo 64
*
Znak * ima kodo 42
_
```

Znakovni nizi

Če želimo dobiti besedilo, moramo več znakov združiti v zbirko. Zbirki znakov pravimo tudi *znakovni niz* (angl. string). Spomnimo se, da vse operacije nad zbirkami v cejevskih programih izvajamo tako, da izvajamo operacije nad posameznimi elementi. Pri obravnavi besedila bi bilo takšno gledanje v mnogih pogledih neudobno, zato obstaja droben trik, ki nam omogoča, da v določenih primerih na besedilo gledamo kot na celoto.

Deklarirajmo si za primer znakovni niz `geslo` in ga nastavimo na vrednost `banana`:

```
char geslo[] = {'b', 'a', 'n', 'a', 'n', 'a', 0};
```

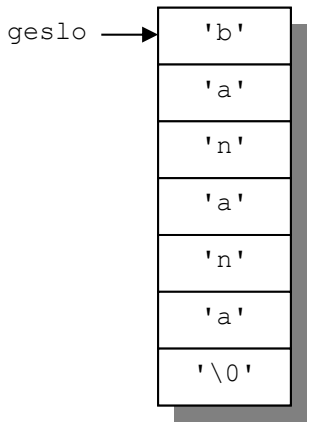
poleg znakov, ki smo jih vnesli v celice, rezervirane za znakovni niz, smo na koncu dodali ničlo. Ker so besedila sestavljena iz znakov, tudi to ničlo pogosto pišemo kot znak `'\0'`, ki mu navadno rečemo *zaključni ničelni znak* (angl. terminating NULL character). To je znak, katerega ASCII vrednost je 0. Narobe bi bilo, če bi pisali `'0'`, kajti to predstavlja desetiško cifro nič, katere ASCII vrednost je 48.

Zaključni ničelni znak igra v znakovnem nizu vlogo markerja, ki označuje konec besedila. Kasneje bomo videli, da nam ravno zaključni znak omogoča, da nize obravnavamo kot navidezno celoto.

Inicializacija znakovnega niza, kakršno smo pravkar videli, je zelo nerodna in v resnici kaj takega nikjer ne vidimo. C nam omogoča, da konstantne znakovne nize pišemo kot zaporedje znakov med dvojnimi narekovaji:

```
char geslo[] =  "banana";
```

Cejevski prevajalnik pri tem na koncu tudi avtomatično doda zaključni ničelni znak. Naš znakovni niz v pomnilniku izgleda takole:



Ker je znakovni niz zbirka, tudi tu velja, da predstavlja njegovo ime (v našem primeru `geslo`) naslov prvega elementa niza (`'b'`).

Skrivno geslo

Zdaj, ko vemo, kako je zgrajen znakovni niz, si poglejmo, kako z nizi upravljamo. Začnimo s praktičnim zgledom, ki s tipkovnice prebere geslo in preveri, če geslo ustreza varnostnim zahtevam. Geslo mora vsebovati vsaj 8 znakov. Program izgleda takole:

IZPIS IZVORNE KODE	geslo.c

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

char geslo[20];
int i;
```

```
int main()
{
   printf("Vpiši novo geslo: ");
   gets(geslo);
   for (i = 0; geslo[i] != 0; i++);
   if (i < 8)
   {
      puts("Geslo je prekratko");
   }
   return 0;
}
```

V programu smo za `geslo` rezervirali 20 pomnilniških celic, kar pomeni, da je lahko geslo dolgo največ 19 znakov. Ne smemo namreč pozabiti na zaključni ničelni znak, ki mora biti vedno na koncu niza. Branje niza s tipkovnice nam omogoča funkcija `gets()`, lahko pa bi uporabili tudi funkcijo `scanf()` s formatnim določilom `%s`. Naslednja dva klica sta skoraj enakovredna:

```
gets(geslo);
scanf("%s", geslo);
```

Razlika je v tem, da funkcija `gets()` prebere vse znake, dokler ne pritisnemo `enter`, medtem ko `scanf()` prebere le znake do prvega presledka ali tabulatorja. Obe funkciji delujeta tako, da pobirata znake s tipkovnice in jih zlagata v pomnilnik od naslova `geslo` naprej. Na koncu avtomatično dodata zaključni ničelni znak. Naj opozorimo na to, da v funkciji `scanf()` pred ime niza ne pišemo naslovnega operatorja (`&`), kajti ime niza že samo po sebi predstavlja naslov.

Znakovni niz lahko izpišemo z enim od naslednjih dveh klicev, ki data enakovreden rezultat:

```
puts(geslo);
printf("%s\n", geslo);
```

Če po izpisu niza ne želimo pomakniti kurzorja v začetek nove vrstice, lahko uporabimo funkcijo `printf()` na tak način:

```
printf(geslo);
```

Ta klic že poznamo, spomniti se moramo le dejstva, da je zaporedje znakov med dvojnimi narekovaji pravzaprav znakovni niz:

```
printf("Juhuhu!");
```

Ko smo geslo prebrali, moramo prešteti, koliko znakov vsebuje. To smo storili s pomočjo ponavljalnega stavka `for`:

```
for (i = 0; geslo[i] != 0; i++);
```

prazen stavek

Števec `i` smo uporabili za štetje znakov. S štetjem zaključimo, ko ni več izpolnjen pogoj `geslo[i] != 0`, se pravi, ko prispemo do zaključnega ničelnega znaka. Naj opozorimo na prazen stavek na koncu stavka `for`. Čeprav razen štetja znakov ne počnemo ničesar drugega, mora stavek tam vseeno biti, sicer bi se znotraj stavka `for` znašel stavek, ki sledi (v našem primeru stavek `if`).

Za štetje znakov pa bi lahko uporabili tudi knjižnično funkcijo `strlen()`, ki vrne število znakov v nizu brez zaključnega znaka. Tako bi v programu `geslo.c` vrstico s stavkom `for` lahko nadomestili z

```
i = strlen(geslo);
```

Oglejmo si še primer delovanja programa:

```
DELOVANJE PROGRAMA          geslo.exe
Vpiši novo geslo: B52
Geslo je prekratko
_
```

Išči, pa boš našel

Za primer navajamo še program, ki s tipkovnice prebere najprej neko besedilo, nato pa še en znak. Program prešteje, kolikokrat se znak pojavi v besedilu in število ponovitev izpiše na zaslon.

```
IZPIS IZVORNE KODE          iskanje.c
#include "stdafx.h"
#include <stdio.h>

char besedilo[256];
char znak;
int i, stevec = 0;

int main()
{
   printf("Vpiši besedilo: ");
   gets(besedilo);
   printf("Vpiši znak: ");
   scanf("%c", &znak);
   for (i = 0; besedilo[i] != 0; i++)
   {
      if (besedilo[i] == znak)
      {
         stevec++;
      }
   }
   printf("V besedilu se znak %c pojavi %d-krat\n", znak, stevec);
   return 0;
}
```

V programu ni ničesar posebno novega. Opozorimo naj le na to, da se v kodi pojavi stavek `for`, s katerim se sprehodimo od prvega do zadnjega znaka niza. Stavek se začne točno tako, kot smo videli v programu `geslo.c`, le da na koncu nimamo praznega stavka, saj moramo za vsak znak niza še nekaj početi. Za vsak znak besedila preverimo, če se ujema z vtipkanim znakom (`besedilo[i] == znak`), in če se, potem povečamo števec znakov za ena.

Program deluje takole:

```
DELOVANJE PROGRAMA          iskanje.exe
Vpiši besedilo: Smisel življenja je ležanje na plaži
Vpiši znak: a
V besedilu se znak a pojavi 4-krat
_
```

Še vsakega po malo

V tem kratkem poglavju bomo spoznali še en ponavljalni stavek in še nekaj operatorjev.

Taylorjeva vrsta

Veliko digitalnih računalnikov izračunava vrednosti raznih matematičnih funkcij s pomočjo Taylorjeve vrste. Funkcija sin x, razvita v vrsto, je

$$\sin x = x - x^3/3! + x^5/5! - x^7/7! + \ldots$$

Napišimo program, ki bo izračunal sinus kota, ki ga vnesemo prek tipkovnice. Ker je vrsta neskončna, moramo izračun nekje ustaviti. Odločimo se, da bomo pri izračunu upoštevali vse člene, ki so po absolutni vrednosti večji od $\varepsilon = 10^{-8}$. Ker so členi vrste padajoči, bomo prenehali z računanjem, čim najdemo člen, ki po absolutni vrednosti ni več večji od ε. Program izgleda takole:

IZPIS IZVORNE KODE sinus.c

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

double sinus = 0;
double kot, clen;
int i = 2;

const double epsilon = 1e-8;

int main()
{
   printf("Vnesi kot v radianih: ");
   scanf("%lf", &kot);
   clen = kot;
   while (clen < -epsilon || clen > epsilon)
   {
      sinus += clen;
      clen = -clen * kot * kot / (i * (i + 1));
      i += 2;
   }
   printf("sin(%.2lf) = %.2lf\n", kot, sinus);
   return 0;
}
```

V programu opazimo reči, ki jih doslej še nismo srečali. Oglejmo si jih pobliže.

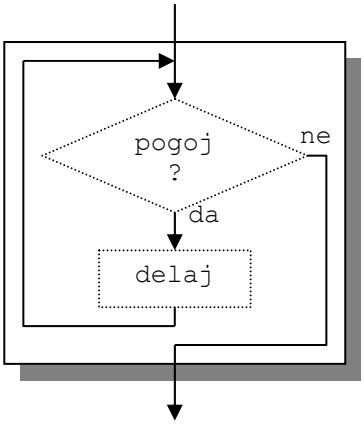
Ponavljalni stavek while

V programu `sinus.c` nastopa, ponavljalni stavek `while`, ki se le malo razlikuje od stavka `do..while`, ki smo ga že spoznali. Njegov zapis je takšen:

SKLADNJA ponavljalni stavek while

```
while (pogoj) delaj;
```

Stavek se od stavka `do..while` razlikuje v tem, da se pri njem pogoj preverja na začetku, tako da ni nujno, da se stavek `delaj` sploh izvrši. Diagram poteka za ta stavek izgleda takole:



Logični operatorji

Dve pokončni črti (`||`), ki ju opazimo v pogoju stavka `while`, predstavljata *logični operator* ALI. C pozna naslednje logične operatorje:

operator	logična funkcija
`!`	negacija
`&&`	logični IN
`||`	logični ALI

Prioriteto operatorjev si lahko ogledate v tabeli na strani 21.

Podobno kot primerjalni operatorji, tudi logični operatorji vrnejo vrednost nič ali ena. Delovanje logičnih operatorjev prikazujejo naslednje tabele:

`izraz`	`!izraz`
0	1
različen od 0	0

`izraz1`	`izraz2`	`izraz1 && izraz2`
0	0	0
0	različen od 0	0
različen od 0	0	0
različen od 0	različen od 0	1

`izraz1`	`izraz2`	`izraz1 || izraz2`
0	0	0
0	različen od 0	1
različen od 0	0	1
različen od 0	različen od 0	1

Zdaj lahko pogledamo, kako se izračuna vrednost izraza

```
clen < -epsilon || clen > epsilon
```

ki smo ga uporabili za pogoj v stavku `while` v programu `sinus.c`. Ker ima logični ALI (`||`) nižjo prioriteto od primerjalnih operatorjev, se najprej izračunata vrednosti obeh primerjalnih izrazov. Če je vrednost vsaj enega od izrazov ena, bo takšna tudi vrednost celotnega izraza.

Takšno razmišljanje pomaga, da razumemo, kako logični operatorji dejansko delujejo. V praksi pa gornji izraz večinoma tolmačimo bolj po človeško: Pogoj bo izpolnjen, če je `clen` manjši od `-epsilon` ali večji od `epsilon`.

Za vajo izračunajmo še nekaj vrednosti logičnih izrazov, pri čemer bomo upoštevali naslednjo deklaracijo:

```
int q = 5, w = 0;
```

izraz	vrednost
`!w`	1
`q && w`	0
`q || w`	1
`q > 3 && q < 10`	1

Izraz v zadnji vrstici tabele ugotavlja, če leži vrednost spremenljivke `q` na intervalu med 3 in 10. Kot smo že povedali v poglavju o primerjalnih operatorjih, je pogosta napaka začetnikov, da tak pogoj zapišejo kot `3 < q < 10`.

Bitni logični operatorji

Ko že govorimo o logičnih operatorjih, si na hitro oglejmo še bitne logične operatorje. Ti, za razliko od operatorjev, ki smo jih spoznali v prejšnjem razdelku, izvajajo logične operacije nad posameznimi biti.

C pozna naslednje bitne operatorje:

operator	bitna operacija
`&`	IN
`|`	ALI
`^`	izključni ALI
`>> n`	pomik bitov v desno za *n* mest
`<< n`	pomik bitov v levo za *n* mest
`~`	eniški komplement (negacija bitov)

Prvi trije operatorji izvajajo logične operacije nad istoležnimi biti. Pomiki bitov se izvajajo tako, da se z leve oziroma desne (odvisno od smeri pomikanja) dodajajo ničle. Na primer, pri pomiku v levo za 5 mest se z desne doda 5 ničel.

Naslednja tabela prikazuje nekaj primerov izrazov, ki vsebujejo bitne logične operatorje. Pri tem predpostavimo, da imamo deklarirano spremenljivko `x` kot nepredznačeno osembitno celo število:

```
unsigned char x = 5, y = 255;
```

izraz	vrednost
`~x`	250
`6 & 21`	4
`x | 9`	13
`y ^ 15`	240
`3 << 2`	12
`17 >> 3`	2

Vrednosti v gornji tabeli dobimo najlaže z vmesno pretvorbo v dvojiški zapis. Izraz `6 & 21` tako dobi obliko:

```
  00000110
& 00010101
----------
  00000100
```

Oblike zapisov realnih števil

Realna števila v cejevskih programih običajno zapisujemo v pozicijskem zapisu z decimalno piko (na primer `3.1416`). Kadar imamo opravka z zelo velikimi ali zelo majhnimi vrednostmi, raje uporabimo *eksponentno* obliko. V programu `sinus.c` smo tak zapis uporabili za konstanto `epsilon`. V splošnem tako zapišemo vrednost $a \cdot 10^p$ kot `aep`. Kadar želimo realne vrednosti izpisovati na zaslon, uporabimo formatno določilo `%e`.

Delovanje programa

Program `sinus.c` smo prevedli in zagnali. Takole nam je izračunal sinus kota pi:

```
DELOVANJE PROGRAMA          sinus.exe
Vnesi kot v radianih: 3.14159
sin(3.14) = 0.00
_
```

Izbire toliko, da glava peče

Kadar se moramo odločati med dvema možnostima, lahko uporabimo pogojni stavek `if..else` z ustreznim pogojem. Kadar je možnosti več, uporabimo gnezdene stavke `if..else` in `if`. Pogosto postane takšna rešitev nepregledna, takrat uporabimo *izbirni stavek* `switch`. Stavek `switch`, tako kot vsi ponavljalni in oba pogojna stavka, spada med krmilne stavke.

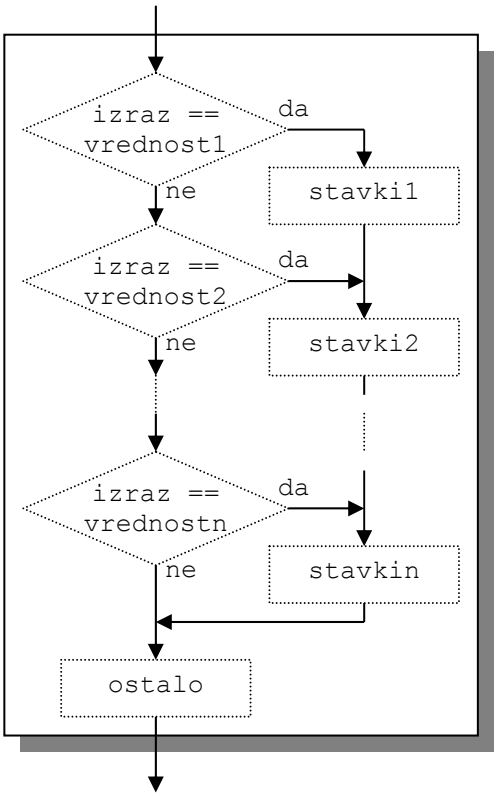
Izbirni stavek switch

Splošna oblika stavka je takšna:

```
SKLADNJA             izbirni stavek switch
switch (izraz)
{
   case vrednost1: stavki1;
   case vrednost2: stavki2;
      ...
   case vrednostn: stavkin;
   default: ostalo;
}
```

V stavku `switch` se najprej izračuna vrednost izraza `izraz`, ki mora biti **celoštevilskega tipa**. Nato se med vsemi besedami `case` poišče tista, za katero stoji vrednost, ki je enaka izračunani vrednosti. Na koncu se izvršijo vsi stavki od dotične besede `case` pa do konca stavka `switch`. Če izračunana vrednost ne ustreza nobeni od podanih vrednosti, se izvedejo samo stavki za besedo `default` (angl. default = privzeto).

Poglejmo si še grafično podobo tega stavka:



Tri, dve, ena... šibamo!

Naslednji program z besedami odšteva čas do štarta. Odštevanje začne pri vrednosti, ki mu jo poda uporabnik.

IZPIS IZVORNE KODE odstevanje.c

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

int n;

int main()
{
   printf("Začenja se odštevanje. Koliko še do štarta? ");
   scanf("%d", &n);
   switch (n)
   {
      case 10: printf("deset, ");
      case 9: printf("devet, ");
      case 8: printf("osem, ");
      case 7: printf("sedem, ");
```

```
        case 6: printf("šest, ");
        case 5: printf("pet, ");
        case 4: printf("štiri, ");
        case 3: printf("tri, ");
        case 2: printf("dve, ");
        case 1: printf("ena... gremo!\n");
    }
    return 0;
}
```

V stavku `switch` smo izpustili del z besedo `default`. Tako se v primeru, da vnesemo vrednost, ki je večja od 10 ali manjša od 1, ne zgodi nič.

Program deluje takole:

```
DELOVANJE PROGRAMA          odstevanje.exe
Začenja se odštevanje. Koliko še do štarta? 3
tri, dve, ena... gremo!
_
```

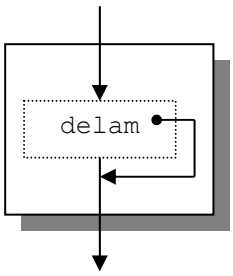
Pozoren študent bo sedaj protestiral, kajti to ni ravno to, kar smo obljubljali. Obljubljali smo stavek, ki nam bo omogočal **izbirati** med večimi možnostmi, podobno, kot lahko s stavkom `if..else` izbiramo med dvema možnostima. Čisto malo nam manjka, da bo stavek `switch` znal tudi to. To malo je stavek `break`.

Dovolj mi je vsega, jaz grem ven

Včasih si želimo zapustiti ponavljalni stavek predčasno, ko je pogoj za ponavljanje še izpolnjen. In skoraj v večini primerov želimo, da izbirni stavek res deluje kot izbirni stavek. Oboje lahko dosežemo s stavkom `break`.

Stavek break

S stavkom `break` izstopimo iz ponavljalnega ali izbirnega stavka. Izvajanje programa se nadaljuje s stavkom, ki sledi stavku, iz katerega smo izstopili. Grafično lahko delovanje stavka `break` ponazorimo na naslednji način. Stavek `delam` je poljuben ponavljalni ali izbirni stavek:



Da bo študent sit in učiteljica Slovenščine zadovoljna

Slovenščina je precej neprijazen jezik za računalnike, ker ima za razliko od angleščine ogromno oblikoslovnih vzorcev. Tako, na primer, ne le, da imamo dvojino, ampak obstaja v primeru, da uporabimo števnik, posebna oblika tudi za tri in štiri.

Poglejmo si primer programa, ki bo izpisal naročilo za sendviče v pravilni slovenščini:

```
IZPIS IZVORNE KODE          sendvici.c
#include "stdafx.h"
#include <stdio.h>

int n;

int main()
{
   printf("Koliko sendvičev želiš? ");
   scanf("%d", &n);
   printf("Prosim, čimhitreje dostavite %d ", n);
   switch (n)
   {
      case 1: printf("sendvič.\n");
              break;
      case 2: printf("sendviča.\n");
              break;
      case 3:
      case 4: printf("sendviče.\n");
              break;
      default: printf("sendvičev.\n");
   }
   return 0;
}
```

V stavku `switch` smo uporabili stavke `break`, da se bo vsakič izpisala samo ena oblika besede sendvič. Za 3 in 4 je oblika enaka, zato pod `case 3` nismo zapisali nobenega stavka, izvedel se bo stavek pod `case 4`.

Ko program zaženemo, se zgodi tole:

```
DELOVANJE PROGRAMA          sendvici.exe
Koliko sendvičev želiš? 5
Prosim, čimhitreje dostavite 5 sendvičev.
_
```

Praštevila

Naravnim številom, ki so deljiva le sama s seboj in z ena, pravimo praštevila. Če želimo preveriti, ali je število *n* praštevilo, jo enostavno delimo z vsemi vrednostmi med 2 in *n*-1. Čim se katero od deljenj izide brez ostanka, že vemo, da *n* ni praštevilo. Če se nobeno deljenje ne izide, potem je *n* praštevilo.

V programu `prastevilo.c` smo uporabili stavek `break`, s katerim končamo preverjanje (ponavljalni stavek `for`), čim se deljenje izide brez ostanka. Na koncu imamo dve možnosti:

1. Ponavljalni stavek se je končal, ker pogoj `i < n` ni bil več izpolnjen. V tem primeru je `i` enak `n`. Nobeno deljenje se ni izšlo, torej je `n` praštevilo.
2. Ponavljalni stavek se je končal, ker je bil izpolnjen pogoj `n % i == 0`, in se je izvedel stavek `break`. V tem primeru `i` ni enak `n`. Vsaj eno deljenje se je izšlo, `n` torej ni praštevilo.

Program izgleda takole:

```
IZPIS IZVORNE KODE          prastevilo.c
#include "stdafx.h"
#include <stdio.h>

int i, n;

int main()
{
   printf("Vpiši naravno število: ");
   scanf("%d", &n);
   for (i = 2; i < n; i++)
   {
      if (n % i == 0)
      {
         break;
      }
   }
   if (n == i)
   {
      printf("%d je praštevilo.\n", n);
   }
   else
   {
      printf("%d ni praštevilo.\n", n);
   }
   return 0;
}
```

Ko smo vnesli število 293, nam je program o njem povedal tole:

```
DELOVANJE PROGRAMA          prastevilo.exe
Vpiši naravno število: 293
293 ni praštevilo.
_
```

Stavek goto in strukturirano programiranje

Običajno se stavki v programu izvajajo eden za drugim, v vrstnem redu, kakor so zapisani. Temu pravimo *zaporedno izvajanje* (angl. sequential execution). Videli smo, da obstaja vrsta odločitvenih stavkov, ki nam omogočajo, da se izvrši kateri od stavkov, ki ni naslednji po vrsti, čemur pravimo *prenos nadzora* (angl. transfer of control). Prenos nadzora nam omogoča s programskimi jeziki zapisovati veliko splošnejše rešitve problemov, kot bi jih lahko brez odločitvenih stavkov.

V grafičnih prikazih odločitvenih stavkov smo opazili, da imajo vsi en sam vhod in en sam izhod. Takšnim stavkom pravimo *strukturirani stavki*, in programiranju, v katerem uporabljamo strukturirane stavke, pravimo *strukturirano programiranje* (angl. structured programming). Vendar programi niso bili vedno takšni.

Do šestdesetih let prejšnjega stoletja so programerji za prenos nadzora (po analogiji z zbirnikom včasih temu pramimo skok) uporabljali izključno stavek `goto`. V šestdesetih letih je začelo vedno bolj postajati jasno, da je brezbrižna uporaba skokov vir večine težav s katerimi so se soočali razvijaci programov. Okrivili so stavek `goto`, ki je omogočal programerju, da skoči na ogromno različnih mest v programu.

Sredi šestdesetih sta Bohm in Jacopini pokazala, da je mogoče pisati programe brez enega samega stavka `goto`. Pokazala sta, da je programiranje možno z uporabo treh osnovnih stavčnih struktur: zaporedno, izbirno in ponavljalno. Vse tri smo že spoznali:

- zaporedna struktura (blok stavkov oziroma sestavljen stavek)
- izbirna struktura (`if`, `if..else`, in `switch`)
- ponavljalna struktura (`for`, `while` in `do..while`)

Programerji so strukturirano programiranje začeli jemati resno šele v začetku sedemdesetih. Rezultati so bili impresivni. Razvijalci programske opreme so začeli poročati o vedno več softverskih projektih, ki so jih končali v roku in v okvirih predvidenega proračuna. Ključ do tega izjemnega uspeha je dejstvo, da so strukturirani programi bolj pregledni, laže je v njih najti napake in jih vzdrževati, in konec koncev je verjetnost, da so v njih sploh napake, manjša.

Stavek `goto` je vseeno preživel. Vendar ne zato, da bi bili cejevski programi podobni zbirniškim programom ali programom iz šestdesetih, ampak zato, ker je včasih še vedno uporaben, na primer kot nadomestek stavka `break` pri izhodu iz gnezdenih zank.

Oglejmo le, kako stavek `goto` uporabimo:

SKLADNJA	stavek goto

```
      goto pojdi
       ...
pojdi:
```

Čim se izvajanje programa znajde v vrstici z `goto` stavkom, se izvajanje nadaljuje v vrstici z oznako `pojdi`.

Funkcije

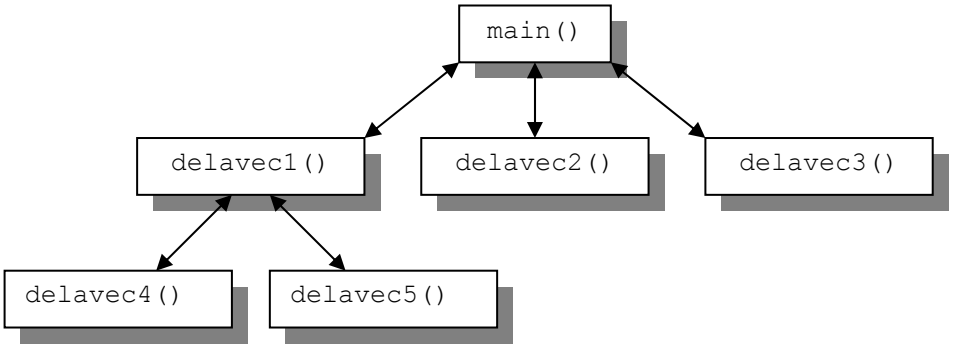
Večina računalniških programov, ki rešujejo realne probleme, je mnogo daljših od primerov, ki smo jih spoznali doslej. Izkušnje so pokazale, da je najboljši način za razvoj in vzdrževanje obsežnih programov, da jih zgradimo iz manjših kosov, ki jim v Ceju pravimo *funkcije*. Tako grajeni programi so veliko bolj pregledni in obvladljivi.

Pri načrtovanju z vrha navzdol, ki smo ga spoznali v začetku semestra, smo ugotovili, da lahko vsak kompleksnejši problem opišemo s hierarhično urejenim skupkom enostavnejših problemov in rešimo vsakega posebej. Funkcije pri tem pogosto predstavljajo rešitve posameznih enostavnih problemov.

Kadarkoli v programu želimo, da se izvede kakšna funkcija, moramo funkcijo *klicati* (angl. call). Funkcijo kličemo z imenom, v oklepajih pa podamo podatke (parametre), ki jih funkcija potrebuje, da bi opravila svoje delo. Ko funkcija vrne (angl. return) kontrolo klicočemu delu kode, je klic končan in izvajanje programa se nadaljuje v naslednji vrstici.

Delovanje funkcij v programih lahko ponazorimo s primerom delovanja hierarhično organiziranega podjetja. Šef naroči (kliče) podrejenemu delavcu naj opravi določeno nalogo in mu sporoči rezultate, ko bo opravil. Šef ne ve, kako bo delavec opravil nalogo. Delavec lahko celo kliče druge delavce in šef tega ne bo vedel. Takšno "skrivanje" podrobnosti precej prispeva k učinkovitosti programiranja.

Naslednja slika prikazuje kako funkcija `main()` na hierarhičen način komunicira z večimi delavskimi funkcijami. Pri tem se funkcija `delavec1()` obnaša kot šef funkcijama `delavec4()` in `delavec5()`.



Funkcije, ki rešujejo probleme, na katere pogosto naletimo, so večinoma že napisane in jih najdemo v knjižnicah, priloženih prevajalnikom. Takšnim funkcijam pravimo *knjižnične funkcije* (angl. library functions) in nekatere izmed njih smo že spoznali. V tem poglavju nas bo bolj zanimalo, kako napišemo in uporabimo svoje lastne funkcije.

Pri pisanju funkcije moramo najprej napisati njeno *deklaracijo* oz. *prototip*. Prototip funkcije definira vmesnik do funkcije. Z drugimi besedami, definira, kako bomo funkcijo uporabljali, ne pove pa ničesar o tem, kaj bo funkcija dejansko počela. Slednje določimo v *definiciji* funkcije.

V prototipu funkcije določimo *ime*, *tip* in *parametre* funkcije:

SKLADNJA prototip funkcije
`tip_funkcije ime_funkcije(tip1 parameter1, tip2 parameter2, ...);`

Za ime funkcije veljajo ista pravila, kot smo jih povedali za imena spremenljivk. Tip funkcije je lahko kakršenkoli veljaven cejevski tip. Funkciji lahko določimo poljubno število parametrov, ki jih navedemo v oklepajih in ločimo z vejicami. Za vsakega moramo podati tip in ime.

Kot primer si oglejmo prototipa knjižničnih funkcij `rand()` in `srand()`, ki smo ju spoznali v programu `postevanka.c`:

```
void srand(unsigned int seed);
int rand();
```

Funkcija `srand()` sprejme en nepredznačen celoštevilski parameter, funkcija `rand()` pa *vrne* naključno celoštevilsko vrednost. Tip funkcije predstavlja tip vrednosti, ki jo funkcija vrne, zato je funkcija `rand()` tipa `int`, funkcija `srand()` pa tipa `void`, ker ne vrne nobene vrednosti. Spomnimo se, ko smo govorili o izrazih, smo povedali, da ima vsak izraz svoj tip in vrednost. V tem pogledu se klic funkcije torej nič ne razlikuje od konstante ali spremenljivke oziroma od kakršnegakoli veljavnega cejevskega izraza. Če smo razumeli vlogo konstant in spremenljivk, bomo znali uporabljati tudi funkcije.

Pri definiciji funkcije ponovimo to, kar smo že določili v prototipu, le da zraven dodamo še *telo*:

SKLADNJA	definicija funkcije

```
tip_funkcije ime_funkcije(tip1 parameter1, tip2 parameter2, ...)
{
   ...
   return vrednost;
}
```

V telo funkcije vpišemo kodo, ki določa, kaj bo funkcija počela. Funkcija mora imeti enega ali več stavkov `return`, ki skrbijo za to, da funkcija vrne ustrezno vrednost. Pri tem lahko za `vrednost` vstavimo poljuben izraz tipa `tip_funkcije`. Kot bomo kmalu spoznali, delujejo parametri funkcije, ki jih podamo v definiciji, kot lokalne spremenljivke, katerim se vrednosti priredijo ob klicu funkcije. Tem parametrom zato pravimo tudi *formalni parametri*. Parametrom, ki jih podajamo ob vsakokratnem klicu funkcije, pravimo *dejanski parametri*.

Za primer si poglejmo, kako bi definirali funkcijo `fakulteta()`, ki sprejme celoštevilski parameter in vrne njegovo fakulteto:

```
int fakulteta(int n) //n -> formalni parameter
{
      int fak = 1;
      int i;
      for (i = 2; i <= n; i++)
      {
            fak *= i;
      }
      return fak;
}
```

To funkcijo lahko sedaj uporabimo kjerkoli, kjer bi uporabili poljuben celoštevilski izraz, na primer v priredilnem stavku:

```
x = fakulteta(5); //5 -> dejanski parameter
```

ali pa

```
printf("%d! = %d\n", 5, fakulteta(5));
```

V prvem primeru bo spremenljivka `x` dobila vrednost, ki jo vrne funkcija `fakulteta()`. Za podani parameter 5 bo to 120. V drugem primeru se bo na mestu drugega formatnega določila `%d` izpisala vrednost, ki jo vrne funkcija `fakulteta()`. Funkcija `printf()` bo tako izpisala "5! = 120".

Funkcijo `fakulteta()` lahko uporabimo tudi za izračun binomskega simbola

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

ki izračuna, na koliko načinov lahko izberemo *m* elementov izmed *n* elementov, ne glede na vrstni red. Funkcijo `binom()` bomo definirali takole:

```
int binom(int n, int m)
{
      return fakulteta(n) / (fakulteta(m) * fakulteta(n - m));
}
```

Iz funkcije `binom()` smo trikrat klicali funkcijo `fakulteta()`. Vidimo, da je klic funkcije sestavljen iz imena funkcije, ki mu v oklepaju sledijo dejanski parametri. Dejanskih parametrov mora biti natanko toliko, kolikor je v prototipu formalnih parametrov. `fakulteta()` sprejme le en parameter. Dejanski parameter lahko podamo v obliki poljubnega izraza, tako smo pri tretjem klicu funkcije za parameter podali izraz `n-m`. Če je parametrov več, so med seboj ločeni z vejico. Tako bomo funkcijo `binom()` klicali na primer takole:

```
binom(10, 3)
```

Takšen klic je sicer sintaktično pravilen, vendar sam po sebi nima nobenega smisla, ker za sabo ne pušča nobenih posledic. Ker je funkcija `binom()` tipa `int`, lahko (in tudi moramo, če želimo učinek) gornji izraz uporabimo kot del kompleksnejšega izraza. Na primer, vrednost, ki jo vrne, lahko shranimo v spremenljivko x:

```
x = binom(10, 3);
```

Ali pa vrnjeno vrednost izpišemo na zaslon:

```
printf("%d\n", binom(10, 3));
```

Vrednosti izrazov, ki jih podamo funkciji kot dejanske parametre, se ob klicu prenesejo v istoležne formalne parametre, ki se znotraj funkcije obnašajo kot običajne spremenljivke. Tako se v primeru gornjega klica funkcije `binom()` vrednost 10 prenese v parameter `n`, vrednost 3 pa v parameter `m`.

Funkcijo `binom()` bomo uporabili, da izračunamo, kolikšna je verjetnost, da iz legla *n* mačjih mladičev, v katerem je *m* samcev, izberemo same samce, če na slepo izberemo *m* mladičev:

IZPIS IZVORNE KODE mladici.c

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

int fakulteta(int n);
int binom(int n, int m);

int main()
{
   int mladici, samci;
   printf("Vnesi skupno število mladičev: ");
   scanf("%d", &mladici);
   printf("Vnesi število samcev: ");
   scanf("%d", &samci);

   printf("\nVerjetnost, da so od %d izbranih mladičev\n", samci);
   printf("vsi samci, je %.2f.\n", 1.0 / binom(mladici, samci));
   return 0;
}
```

```
int fakulteta(int n)
{
   int fak = 1;
   int i;
   for (i = 2; i <= n; i++)
   {
      fak *= i;
   }
   return fak;
}

int binom(int n, int m)
{
   return fakulteta(n) / (fakulteta(m) * fakulteta(n - m));
}
```

Z uporabo funkcije `binom()` je program enostaven. Vse, kar moramo narediti za izračun verjetnosti je, da izračunamo število vseh možnih izbir (s klicem funkcije `binom()`) in izračunamo razmerje vseh izbir proti ena. Ena sama izbira je namreč tista, ki nam da same samce. Program `mladici.c` je lep primer hierarhične organizacije klicev funkcij. Brez uporabe funkcij bi bil ta sicer enostaven program precej nepregleden.

Ko smo program zagnali, nam je povedal tole:

```
DELOVANJE PROGRAMA           mladici.exe
Vnesi skupno število mladičev: 5
Vnesi število samcev: 3

Verjetnost, da so od 3 izbranih mladičev
vsi samci, je 0.10.
_
```

Področje spremenljivke

V programu `mladici.c` se je celoštevilska spremenljivka `n` pojavila dvakrat. Prvič kot parameter funkcije `fakulteta()` in še kot parameter funkcije `binom()`. Zastavi se vprašanje, ali je to dovoljeno in če je, kako obe spremenljivki med sabo ločimo. V Ceju velja, da niso vse spremenljivke dostopne vsem delom programa, zaradi česar lahko izberemo eno in isto ime spremenljivke večkrat.

Vsaka spremenljivka je dostopna le znotraj svojega *področja* (angl. scope). Omenili bomo le dve področji:

Področje datoteke - spremenljivke, ki jih deklariramo zunaj katerekoli funkcije, so dostopne vsem delom programa in jim pravimo *globalne* spremenljivke.

Področje funkcije - spremenljivke, deklarirane znotraj funkcije, so dostopne le funkciji, znotraj katere so deklarirane. Takšnim spremenljivkam pravimo, da so *lokalne*.

Parametri funkcije se obnašajo kot lokalne spremenljivke, ki se jim, kakor smo že videli, nastavi vrednost ob klicu funkcije.

Kadar ima lokalna spremenljivka enako ime kot kakšna globalna spremenljivka, potem globalna spremenljivka znotraj funkcije ni dostopna.

Poglejmo si primer, ki, razen tega, da ilustrira področje spremenljivk, ne počne nič pametnega:

```
IZPIS IZVORNE KODE          podrocje.c
#include "stdafx.h"
#include <stdio.h>

int f1(int x);
int f2(int y);

int a = 17;


int main()
{
   int x = 3, y = 4;
   f1(x);
   f2(x);
   printf("V funkciji main():\n");
   printf("  a = %d\n", a);
   printf("  x = %d\n", x);
   printf("  y = %d\n", y);
   return 0;
}

int f1(int x)
{
   x++;
   printf("V funkciji f1():\n");
   printf("  a = %d\n", a);
   printf("  x = %d\n", x);
   return 0;
}

int f2(int y)
{
   int a = 10;
   printf("V funkciji f2():\n");
   printf("  a = %d\n", a);
   printf("  y = %d\n", y);
   return 0;
}
```

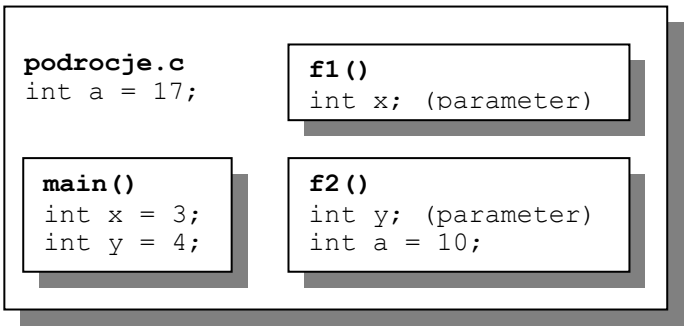
V programu imamo eno samo globalno spremenljivko `a`, ki je dostopna povsod, razen znotraj funkcije `f2()`, ki ima lastno, lokalno spremenljivko z istim imenom.

Ob klicu funkcije `f1()` se prepiše vrednost lokalne spremenljivke `x` funkcije `main()` v parameter `x` funkcije `f1()`. Da bi videli, da gre res za drugo spremenljivko (z istim imenom), smo v funkciji `f1()` povečali vrednost lokalne spremenljivke `x` za ena. Ko program zaženemo, dobimo takšen izpis:

```
DELOVANJE PROGRAMA          podrocje.exe
V funkciji f1():
  a = 17
  x = 4
V funkciji f2():
  a = 10
  y = 3
V funkciji main():
  a = 17
  x = 3
  y = 4
_
```

V funkciji `f1()` spremenljivka `y` ni dostopna, prav tako tudi ni dostopna spremenljivka `x` v funkciji `f2()`.

Področje spremenljivk v programu `podrocje.c` si ponazorimo še z naslednjim diagramom:



Kadar želimo ugotoviti, če je kakšna spremenljivka dostopna posameznemu delu programa (funkciji), uporabimo preprosto pravilo. Najprej pogledamo, če je spremenljivka parameter funkcije, ali če je deklarirana lokalno. Če ni ne eno ne drugo, potem pogledamo, če obstaja globalna spremenljivka z istim imenom. Če tudi ta ne obstaja, potem spremenljivka ni dostopna. Tako iz zgornje slike na primer vidimo, da sta funkciji `f1()` dostopna le lokalna spremenljivka `x` (parameter) in globalna spremenljivka `a`.

Za vajo razmislite, kaj je narobe z naslednjim programom, ki naj bi izpisal fakultete števil od 1 do 8:

IZPIS IZVORNE KODE problem.c

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

int i;

int fakulteta(int n);

int main()
{
   for (i = 1; i <= 8; i++)
   {
      printf("%d! = %d\n", i, fakulteta(i));
   }
   return 0;
}

int fakulteta(int n)
{
   int fak = 1;
   for (i = 2; i <= n; i++)
   {
      fak *= i;
   }
   return fak;
}
```

Program v resnici izpiše tole:

```
DELOVANJE PROGRAMA          problem.exe
2! = 1
4! = 6
6! = 120
8! = 5040
_
```

Obstoj spremenljivke

Spremenljivka obstaja, dokler ima rezerviran prostor v pomnilniku. Med obstojem in področjem spremenljivke obstaja določena povezava. Očitno je, da spremenljivka, ki je dostopna, tudi obstaja, vendar obratno ne velja vedno. C loči dva tipa obstoja spremenljivk:

Avtomatičen obstoj - spremenljivka, ki ima avtomatičen obstoj, obstaja samo v času izvajanja funkcije v kateri je deklarirana. Če drugače ne določimo, imajo vse lokalne spremenljivke avtomatičen obstoj.

Statičen obstoj - statičen obstoj pomeni, da spremenljivka obstaja ves čas izvajanja programa. Takšen obstoj imajo globalne spremenljivke. Tudi lokalnim spremenljivkam lahko določimo statičen obstoj in sicer tako, da pred njihovo deklaracijo dopišemo besedo `static`.

V praksi lokalni spremenljivki dodelimo statičen obstoj, kadar želimo, da se njena vrednost ohrani med posameznimi klici funkcije. Kot primer si poglejmo uporabo funkcije, s pomočjo katere neko pozitivno celoštevilsko vrednost razstavimo na prafaktorje. Funkcija deluje tako, da jo prvič kličemo s pozitivnim celoštevilskim parametrom, ki ga želimo razstaviti. Vrednost se shrani v statično lokalno spremenljivko `komad`. Potem funkcijo kličemo s parametrom 0 (nič) toliko časa, dokler nam ne vrne ničle. Funkcija vsakič sproti vrne enega od prafaktorjev podane vrednosti, pri čemer ustrezno zmanjša vrednost spremenljivke `komad`.

```
IZPIS IZVORNE KODE          prafaktor.c
#include "stdafx.h"
#include <stdio.h>

unsigned int prafaktor(unsigned int n);

int main()
{
   unsigned int stevilo, f;
   printf("Vnesi število, ki bi ga rad\n");
   printf("razstavil na prafaktorje: ");
   scanf("%u", &stevilo);
   printf("\nPrafaktorji:\n%d", prafaktor(stevilo));
   while (f = prafaktor(0))
   {
      printf(", %d", f);
   }
   printf("\n");
   return 0;
}

unsigned int prafaktor(unsigned int n)
{
   static unsigned int komad;
   unsigned int i;
   if (n > 0)
   {
      komad = n;
   }
```

```
   else if (komad == 1)
   {
      return 0;
   }
   for (i = 2; i <= komad; i++)
   {
      if (komad % i == 0)
      {
         komad /= i;
         return i;
      }
   }
   return i;
}
```

Program deluje takole:

```
DELOVANJE PROGRAMA          prafaktor.exe
Vnesi število, ki bi ga rad
razstavil na prafaktorje: 64311

Prafaktorji:
3, 13, 17, 97
_
```

Problem bi seveda lahko rešili tudi z uporabo globalne spremenljivke, vendar se je iz varnostnih razlogov bolje izogibati pretirani uporabi globalnih spremenljivk (glej program `problem.c`).

Prazen tip (void)

Včasih se pripeti, da je narava kakšne funkcije takšna, da ne sprejema parametrov (npr. knjižnična funkcija `rand()`) oziroma ne vrača vrednosti (npr. knjižnična funkcija `srand()`, ki inicializira generator naključnih števil). Za takšne primere uporabimo *prazen podatkovni tip* `void`. Prazen podatkovni tip je tip, ki nima nobene vrednosti. Tako bi funkcijo, ki niti ne sprejema parametrov niti ne vrača vrednosti, deklarirali in definirali takole:

```
SKLADNJA             void funkcija brez parametrov
void ime_funkcije(void);     //prototip

void ime_funkcije(void)      //definicija
{
   ...
}
```

Ključna beseda `void` v oklepajih pomeni, da funkcija ne pričakuje nobenega parametra, vendar uporaba besede `void` ni obvezna: lahko pišemo enostavno samo prazen par oklepajev:

```
SKLADNJA             void funkcija brez parametrov
void ime_funkcije(); //prototip

void ime_funkcije()  //definicija
{
   ...
}
```

Ker funkcija tipa `void` ne vrača nobene vrednosti, v telesu ne potrebuje stavka `return`.

Seveda lahko napišemo tudi funkcijo, ki ne vrača vrednosti, vendar sprejema parametre in tudi funkcijo, ki ne sprejema parametrov, vendar vrača vrednost.

Funkcijo, ki ne sprejema nobenega parametra, kličemo s praznim parom oklepajev. Prevajalnik namreč po oklepajih loči funkcijo od spremenljivke. Primer takšne funkcije je knjižnična funkcija `rand()`, ki vrne naključno celo število med 0 in `RAND_MAX`:

```
zreb = rand();
```

Poglejmo si enostaven primer definicije in uporabe funkcije, ki ne sprejema parametrov niti ne vrača vrednosti. Naslednji program na zaslon izpiše trenuten čas. Za branje ure lahko uporabimo knjižnično funkcijo `time()`, ki vrne število sekund od polnoči, 1. januarja 1970. Ker nas datum ne zanima, od vrednosti, ki jo funkcija `time()` vrne, vzamemo le celoštevilski ostanek deljenja z 86400. Toliko sekund namreč vsebuje povprečen dan. Iz tega bomo potem izračunali ure in minute, ki predstavljajo trenutni čas:

IZPIS IZVORNE KODE ura.c

```
#include "stdafx.h"
#include <time.h>
#include <dos.h>
#include <stdio.h>

void izpisiUro(void);

int main()
{
   izpisiUro();
   return 0;
}

void izpisiUro()
{
   unsigned long t;
   int ure, minute;

   t = time(NULL) % 86400;
   ure = t / 3600;
   minute = t / 60 % 60;

   printf("Ura je %d in %d minut.\n", ure, minute);
}
```

Ure smo izračunali tako, da smo vzeli celi del količnika skupnega števila sekund s 3600, kolikor je sekund v uri:

```
ure = t / 3600;
```

Minute smo izračunali tako, da smo skupno število sekund najprej delili s šestdeset in od dobljenega vzeli ostanek pri deljenju s šestdeset (ta ostanek predstavlja število minut čez polno uro):

```
minute = t / 60 % 60;
```

Če program zaženemo dvanajst minut pred enajsto dopoldne, dobimo takšen izpis:

DELOVANJE PROGRAMA ura.exe
Ura je 10 in 48 minut. _

Kdaj in kako pisati lastne funkcije

Kadar se lotimo pisanja kakšne lastne funkcije, je dobro vedeti, kaj je že napisanega, da ne bi izumljali tople vode. Kadar je le mogoče, uporabite katero od bogate izbire standardnih knjižničnih funkcij. Sem in tja je priporočljivo malo pobrskati po priloženi pomoči (angl. online help) prevajalnika, s časom boste imeli vedno več občutka za to, kaj vse je že napisanega.

Ko pišete funkcijo, se držite zlatega pravila, da naj funkcija opravlja eno samo, dobro definirano nalogo. S tem dosežete, da postanejo napisane funkcije splošno uporabne tudi v drugih projektih, ki jih boste pisali vi ali vaši kolegi v podjetju. Za funkcijo izberite kratko ime, ki čimbolje opisuje nalogo, ki jo funkcija opravlja. Če tega ne morete storiti, potem obstaja verjetnost, da skuša vaša funkcija izvajati več različnih opravil. V takšnem primeru velja razmisliti, če ne bi morda funkcijo razbili na več krajših funkcij.

Vzemimo za primer gornjo funkcijo `izpisiUro()`. Ta funkcija počne 3 reči:

-prebere sistemski čas,
-pretvori sekunde v ure, minute in sekunde ter
-izpiše uro

Te funkcije ne moremo uporabiti, če bi želeli izpis ure v drugačni obliki, na primer 10:48. Poskusite za vajo napisati tri ločene funkcije, ki vsaka opravi le eno od treh opravil.

Večdimenzionalne zbirke

Vemo že, da zbirke uporabljamo v primeru, ko imamo opravka z množico podatkov istega tipa, ki tematsko spadajo skupaj. Če so ti podatki enodimenzionalnega tipa (npr. meritve temperature ob določenih časovnih intervalih), potem jih shranjujemo v enodimenzionalne zbirke, ki, matematično gledano, ustrezajo vektorjem. Včasih se pripeti, da je narava podatkov večdimenzionalna, takrat lahko uporabimo večdimenzionalno zbirko. V matematiki so to matrike. Večdimenzionalno zbirko deklariramo tako, da za imenom zbirke podamo število elementov za vsako dimenzijo posebej:

SKLADNJA deklaracija n-dimenzionalne zbirke
tip_elementa ime_zbirke[št_el1][št_el2] ... [št_eln];

Tako na primer ustvarimo dvodimenzionalno realno matriko z dvema vrsticama in tremi stolpci z naslednjo deklaracijo:

```
float m[2][3];
```

Do posameznih elementov takšne matrike pridemo na enak način kot do elementov enodimenzionalne zbirke, le da je potrebno za imenom navesti dva indeksa, prvi predstavlja zaporedno številko vrstice, drugi pa stolpca. Matrika `m` bo izgledala takole:

	stolpec 0	stolpec 1	stolpec 2
vrstica 0	`m[0][0]`	`m[0][1]`	`m[0][2]`
vrstica 1	`m[1][0]`	`m[1][1]`	`m[1][2]`

Enako kot enodimenzionalno zbirko lahko matriko ob deklaraciji tudi inicializiramo. Če zapišemo

```
float m[2][3] = {{1.2}, {-1.0, 0.8, -3.1}};
```

bo imel element matrike `m[0][0]` vrednost 1,2, elementa `m[0][1]` in `m[0][2]` vrednost 0, elementi `m[1][0]`, `m[1][1]` in `m[1][2]`, pa po vrsti vrednosti -1, 0,8 in -3,1.

Vrtenje točke v tridimenzionalnem prostoru

Poglejmo si praktičen primer vrtenja točke v tridimenzionalnem prostoru za kot φ okrog osi z. Takšne in podobne transformacije se uporabljajo v risarskih programih, kakršen je AutoCAD.

Tridimenzionalno točko zavrtimo okrog osi z za kot φ tako, da jo množimo z rotacijsko matriko

$$\Delta = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Naslednji program s tipkovnice prebere tridimenzionalno točko in kot rotacije v radianih ter na zaslon izpiše koordinate zavrtene točke:

IZPIS IZVORNE KODE rotacija.c

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

int main()
{
   float rot[3][3];
   float t1[3], t2[3];
   float fi;
   int i, j;

   printf("Vnesi koordinate točke (x, y, z): ");
   for (i = 0; i < 3; i++)
   {
      scanf("%f", &t1[i]);
   }
   printf("Vnesi kot vrtenja okrog osi z (v radianih): ");
```

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


   rot[0][0] = cos(fi);
   rot[0][1] = -sin(fi);
   rot[0][2] = 0;
   rot[1][0] = sin(fi);
   rot[1][1] = cos(fi);
   rot[1][2] = 0;
   rot[2][0] = 0;
   rot[2][1] = 0;
   rot[2][2] = 1;

   for (i = 0; i < 3; i++)
   {
      t2[i] = 0;
      for (j = 0; j < 3; j++)
      {
         t2[i] += rot[i][j] * t1[j];
      }
   }

   printf("Zavrtena točka: (%.1f, %.1f, %.1f)\n", t2[0], t2[1],
      t2[2]);
   return 0;
}
```

Program prikazuje uporabo dvodimenzionalne zbirke, sicer pa ni nič posebnega. V devetih vrsticah, ki sledijo branju kota vrtenja, nastavimo elemente matrike, potem pa v dveh gnezdenih stavkih `for` izvedemo množenje točke z rotacijsko matriko. Zunanja zanka `for` skrbi za to, da obdelamo vse vrstice (indeks `i`), notranja pa stolpce (indeks `j`) matrike.

Program deluje takole:

```
DELOVANJE PROGRAMA          rotacija.exe
Vnesi koordinate točke (x, y, z): 1 1 1
Vnesi kot rotacije okrog osi z (v radianih): 3.14
Zarotirana točka: (-1.0, -1.0, 1.0)
_
```

Joj, kam bi del!

Ko količine podatkov, ki jih obdelujemo, naraščajo, se začnejo pojavljati potrebe po dodatni organizaciji podatkov. V tem poglavju se bomo pogovarjali o združevanju podatkov različnih tipov v takoimenovane strukture.

Strukture

Kadar imamo opravka z večimi pomensko vezanimi podatki istega tipa, lahko takšne podatke združimo v zbirko. Če pa podatki niso istega tipa, vendar so pomensko vseeno tesno povezani, jih združujemo v *strukture*. Strukturo definiramo takole:

```
SKLADNJA             definicija strukture
struct ime_strukture
{
   tip1 komponenta1;
   tip2 komponenta2;
      ...
   tipn komponentan;
};
```

S tem smo le povedali, katere podatke bi radi združili skupaj, vendar v pomnilniku še nismo ustvarili prostora zanje. Definicijo strukture si lahko predstavljamo kot definicijo novega podatkovnega tipa. Tako lahko deklariramo spremenljivko strukturnega tipa na podoben način kot običajno skalarno spremenljivko:

SKLADNJA	deklaracija strukturne spremenljivke

```
struct ime_strukture str_spremenljivka;
```

Deklaracija se razlikuje od deklaracije skalarne spremenljivke le v tem, da na začetku deklaracije namesto tipa napišemo rezervirano besedo `struct` in pa ime strukture.

Definicijo strukturnega tipa in deklaracijo strukturne spremenljivke lahko tudi združimo:

SKLADNJA	definicija tipa in deklaracija sprem.

```
struct ime_strukture
{
   tip1 komponenta1;
   tip2 komponenta2;
      ...
   tipn komponentan;
} str_spremenljivka;
```

Strukturno spremenljivko uporabljamo tako, da uporabljamo njene posamezne komponente. Do konkretnega elementa pridemo tako, da navedemo ime strukturne spremenljivke in s selektorjem izberemo željeno komponento:

SKLADNJA	izbira komponente

```
str_spremenljivka.komponentan
```

Pika med obema imenoma predstavlja selektor. Tako dobljeni izraz uporabljamo enako, kot bi uporabljali običajno spremenljivko tipa `tipn` (to je tip komponente `komponentan`).

Kadar imamo dve strukturni spremenljivki istega tipa, in želimo prepisati vrednosti vseh komponent ene spremenljivke v drugo, lahko uporabimo priredilni opearator:

SKLADNJA	kopiranje strukturnih spremenljivk

```
str_spremenljivka1 = str_spremenljivka2
```

Poglejmo si kratek primer, ki s tipkovnice prebere ime in višino dveh sošolcev in na zaslon izpiše, kdo je večji:

IZPIS IZVORNE KODE	sosolci.c

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

struct dijak
{
   char ime[20];
   int visina;
} d1, d2;

int main()
{
   printf("Vnesi ime in višino 1. dijaka: ");
   scanf("%s%d", d1.ime, &d1.visina);
   printf("Vnesi ime in višino 2. dijaka: ");
```

```
   scanf("%s%d", d2.ime, &d2.visina);
   if (d1.visina > d2.visina) printf("Večji je %s.\n", d1.ime);
   else if (d1.visina < d2.visina) printf("Večji je %s.\n", d2.ime);
   else printf("Oba sta enako visoka");
   return 0;
}
```

V programu smo definirali strukturo `dijak`, ki je sestavljena in znakovnega niza `ime` in celoštevilske komponente `visina`. Deklarirali smo tudi strukturni spremenljivki `d1` in `d2`. Na primeru vidimo, da vsako strukturno spremenljivko obravnavamo kot več ločenih spremenljivk različnih tipov, ki jih druži le ime pred selektorjem. Tako na primer zapis `d2.visina` uporabljamo kot običajno spremenljivko tipa `int`, zapis `d1.ime` pa kot običajen znakovni niz.

Program deluje takole:

```
DELOVANJE PROGRAMA          sosolci.exe
Vnesi ime in višino 1. dijaka: Miha 186
Vnesi ime in višino 2. dijaka: Jure 183
Večji je Miha.
_
```

Seveda bi lahko, podobno kot s skalarnimi spremenljivkami, tudi s strukturnimi spremenljivkami ustvarili zbirko. Denimo, da je v 3. A razredu 35 dijakov:

```
struct dijak tretjiA[35];
```

Njihova imena bi izpisali takole:

```
for (i = 0; i < 35; i++)
{
      puts(tretjiA[i].ime);
}
```

Vidimo, da moramo najprej izmed zbirke strukturnih spremenljivk z indeksom izbrati določen element, potem šele s selektorjem izberemo ustrezno komponento (v našem primeru `ime`).

Največjega dijaka bi poiskali takole:

```
struct dijak najvecji;

najvecji = tretjiA[0];
for (i = 1; i < 35; i++)
{
      if (najvecji.visina < tretjiA[i].visina)
            najvecji = tretjiA[i];
}
printf("Največji je %s\n", najvecji.ime);
```

V primeru, da je več dijakov enako velikih, bi se izpisalo ime tistega, ki ima v seznamu nižji indeks.

Inicializacija strukture ob deklaraciji

Podobno, kot smo to videli pri ostalih podatkovnih tipih, lahko tudi komponente strukturnih spremenljivk ob deklaraciji nastavimo na začetne vrednosti. Princip je enak kakor pri zbirkah, kjer vrednosti elementov preprosto naštejemo med parom zavitih oklepajev:

```
struct dijak nekdo = {"Luka", 204};
```

Tudi, če deklariramo zbirko strukturnih spremenljivk, lahko posamezne komponente nastavimo na začetne vrednosti. Upoštevati moramo zgolj dejstvo, da je vsak element zbirke struktura:

```
struct dijak tretjiA[] = {{"Lun", 178},
                          {"Gal", 185},
                          {"Pia", 171}};
```

Kazalci

Kazalci (angl. pointers) so precej zahtevna in obsežna tematika vsakega programskega jezika, ki dopušča neposredno delo z njimi. Po drugi strani nam ponujajo kazalci ogromno moč manipulacije s podatki in kodo, vendar le, če jih dobro razumemo in obvladamo. Čas nam ne dopušča, da bi se v okviru našega predmeta preveč ukvarjali s kazalci, tematike se bomo le dotaknili.

Če hočemo razumeti kazalce, se moramo najprej prepričati, da razumemo vsa zakulisna dogajanja v naslednjem trivialnem programu:

IZPIS IZVORNE KODE	vrednost.c

```
#include "stdafx.h"

int main()
{
   int x;
   x = 2014;
   printf("%d\n", x);
   return 0;
}
```

Vse dogajanje v programu je strnjeno v treh vrsticah kode v funkciji `main()`. V prvi vrstici deklariramo celoštevilsko spremenljivko `x`, s čimer v pomnilniku rezerviramo dve pomnilniški celici, ki bosta služili izključno za hranjenje vrednosti spremenljivke `x`. V drugi vrstici v ti dve rezervirani celici vpišemo vrednost 2014. V zadnji vrstici s pomočjo funkcije `printf()` na zaslon izpišemo vrednost, ki se nahaja v pomnilniku, rezerviranem za spremenljivko `x`.

Ko program zaženemo, dobimo izpis:

DELOVANJE PROGRAMA	vrednost.exe

```
2014
_
```

S tem enostavnim primerom smo želeli opomniti na dejstvo, da, kadarkoli v programski kodi zasledimo ime kakšne spremenljivke, v resnici beremo ali pišemo iz dela pomnilnika, rezerviranega izključno za to spremenljivko.

Kje natančno je v pomnilniku rezerviran prostor za določeno spremenljivko, nas programerje večinoma ne zanima. Na srečo lahko zaupamo prevajalniku, da bo za spremenljivko rezerviral kos še neuporabljenega pomnilnika, in s tem smo zadovoljni. Pogosto pa naletimo na operacije, ki na tak ali drugačen način potrebujejo informacijo o tem, kje v pomnilniku se določena spremenljivka nahaja oziroma, kakšen je njen *naslov*. Do naslova spremenljivke pridemo s pomočjo naslovnega operatorja (`&`), ki ga postavimo pred ime spremenljivke, kot kaže naslednji primer.

```
IZPIS IZVORNE KODE          naslov.c
#include "stdafx.h"

int main()
{
   int x;
   x = 2014;
   printf("%u\n", &x);
   return 0;
}
```

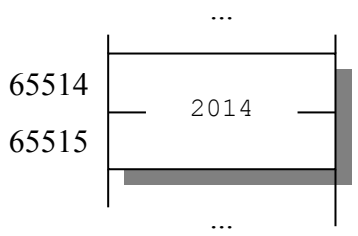
Primer se od programa `vrednost.c` razlikuje le v zadnji vrstici, ki namesto vrednosti spremenljivke `x` izpiše njen naslov. Kot že vemo, je naslov nepredznačeno celo število, zato moramo za izpis uporabiti formatno določilo `%u`.

Ko smo gornji program zagnali, je izpisal

```
DELOVANJE PROGRAMA          naslov.exe
65514
_
```

To pomeni, da je vrednost spremenljivke `x` shranjena v celicah 65514 in 65515. Dejanska vrednost, ki jo vrne izraz `&x`, je v resnici odvisna od konfiguracije sistema, na katerem zaganjamo program, vedno pa predstavlja pomnilniški naslov spremenljivke `x`.

Naslednja slika prikazuje stanje v pomnilniku med zagonom programa `naslov.exe`.



Pa ne že spet znakovni nizi

Spomnimo se, da so znakovni nizi poseben primer zbirk. So zbirke znakov, ki imajo na koncu zaključni ničelni znak. Večkrat smo že omenili, da predstavlja ime zbirke brez indeksa naslov začetka zbirke oziroma naslov prvega elementa v zbirki. Zdaj, ko

smo uradno spoznali naslovni operator, lahko to dejstvo zapišemo tudi po cejevsko. Če imamo deklaracijo

```
char sms[160];
```

Potem je izraz

```
sms
```

enakovreden izrazu

```
&sms[0]
```

Oba predstavljata naslov prvega elementa zbirke oziroma začetka zbirke. Videli smo, da vsaka funkcija, ki izvaja operacije nad znakovnimi nizi, za svoje delo vedno potrebuje ta naslov. Vedeti mora namreč, kje v pomnilniku je začetek znakovnega niza, ki ga mora obdelati. Funkcija potem od tega naslova naprej po vrsti obdeluje znake enega za drugim, dokler ne pride do zaključnega ničelnega znaka.

Poigrajmo se nekoliko s to idejo:

IZPIS IZVORNE KODE spetdoma.c

```
#include "stdafx.h"
#include <string.h>

int main()
{
   char msg[10] = "SPET DOMA";
   printf("%s %s\n", msg, &msg[5]);
   return 0;
}
```

V tem programu smo funkciji `printf()` podali dva naslova. Najprej smo ji podali naslov začetka znakovnega niza `msg`, potem pa še naslov šestega znaka istega niza. Funkcija seveda ne ve, od kod je dobila naslov in se ne sprašuje, ali je naslov res začetek kakšnega znakovnega niza. Vse, kar ta funkcija naredi, je to, da začne z delom na naslovu, ki ga je dobila in konča pri zaključnem ničelnem znaku.

Ko program zaženemo, bo izpisal tole:

DELOVANJE PROGRAMA spetdoma.exe

```
SPET DOMA DOMA
_
```

Na takšen način se seveda da preslepiti katerokoli funkcijo, ki dela z znakovnimi nizi. Če bi klicali

```
strlen(&msg[2])
```

bi dobili vrnjeno vrednost 7. To je namreč dolžina niza `msg` od vključno tretjega znaka naprej.

Deklarirani kazalci

Videli smo, da z imenom znakovnega niza (oz. katerekoli zbirke), indeksnim operatorjem in naslovnim operatorjem že lahko nekoliko manipuliramo s funkcijami, ki kot parameter pričakujejo naslov. Še več možnosti manipulacije nam nudijo pravi kazalci.

Kazalec definiramo kot **spremenljivko, katere vrednost predstavlja naslov pomnilniške lokacije.** V skladu s to definicijo bi lahko rekli, da je ime zbirke kazalec. V resnici je ime zbirke *konstanten* kazalec, njegova vrednost je naslov začetka zbirke, vendar te vrednosti ne moremo spreminjati.

Naslednji primer prikazuje uporabo pravega kazalca.

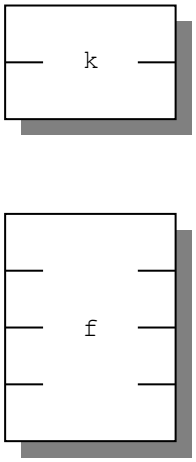
IZPIS IZVORNE KODE kazalci1.c

```
#include "stdafx.h"

int main()
{
   float f, *k;
   f = 500;
   k = &f;
   printf("%.1f %.1f\n", f, *k);
   return 0;
}
```

V programu `kazalci1.c` smo poleg običajne realne spremenljivke `f` deklarirali še kazalec `k`. Da gre za kazalec, spoznamo po zvezdici (`*`), ki je v deklaraciji postavljena pred ime spremenljivke:

```
float *k;
```

Naslednja slika prikazuje pomnilnik, rezerviran za spremenljivki `f` in `k`.

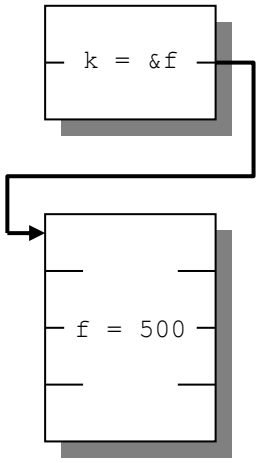


Spremenljivka `f` zasede 4 pomnilniške celice, ker je tipa `float`, kazalci pa so vedno nepredznačena cela števila, saj hranijo le pomnilniški naslov. Za potrebe našega predmeta bomo predpostavili, da so kazalci 16 bitni, čeprav to ne drži.

Prva dva stavka programa `kazalci1.c` vpišeta v `f` vrednost 500, v `k` pa naslov spremenljivke `f`:

```
f = 500;
k = &f;
```

V pomnilniku imamo sedaj takšno stanje:



Odebeljena puščica, ki kaže od kazalca `k` proti spremenljivki `f`, simbolno nakazuje, da smo ustvarili povezavo med obema spremenljivkama. S tem, ko smo v kazalec vpisali naslov spremenljivke `f`, smo kazalec usmerili na to spremenljivko. Pravimo, da kazalec `k` *kaže* na `f`.

Verjetno se je že prej komu zastavilo vprašanje, zakaj moramo pred deklaracijo kazalca navesti tip, če pa smo rekli, da je kazalec vedno nepredznačeno celo število. Tip kazalca ne pomeni tipa njegove vrednosti, pač pa tip podatka, na katerega kaže. Ta informacija je zelo pomembna, kajti le tako lahko preko naslova, ki je shranjen v kazalcu, pravilno dostopamo do spremenljivke, na katero kazalec kaže.

Do podatka, na katerega kazalec kaže, pridemo tako, da uporabimo *operator indirekcije* (`*`), ki ga postavimo pred ime kazalca. Tako smo ta operator uporabili v zadnji vrstici programa `kazalci1.c`, da bi izpisali vrednost, na katero kaže `k`:

```
printf("%.1f %.1f\n", f, *k);
```

Ker `k` kaže na `f`, se je izpisala dvakrat ista vrednost:

```
DELOVANJE PROGRAMA          kazalci1.exe
500.0 500.0
_
```

Temu, da dostopamo do podatkov posredno prek kazalca, pravimo *posredno naslavljanje* (angl. indirect addressing).

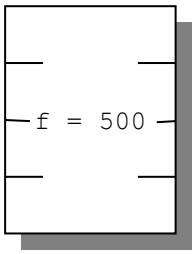
Naslednji program je zelo podoben programu `kazalci1.c`:

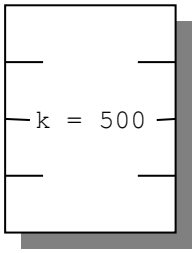
```
IZPIS IZVORNE KODE          kazalci2.c
#include "stdafx.h"

int main()
{
   float f, k;
   f = 500;
   k = f;
   printf("%.1f %.1f\n", f, k);
   return 0;
}
```

In tudi njegov izpis je enak:

```
DELOVANJE PROGRAMA          kazalci2.exe
500.0 500.0
_
```

Vendar obstaja med obema programoma bistvena razlika. V programu `kazalci1.c` prihajata obe izpisani vrednosti 500 iz ene in iste lokacije v pomnilniku. V programu `kazalci2.c` pa izvira vsaka od izpisanih vrednosti iz ločene lokacije. Naslednja slika prikazuje stanje v pomnilniku med izpisovanjem obeh vrednosti v programu `kazalci1.c`:





Dejstvo, da gre v programu `kazalci1.c` za en in isti podatek, medtem ko imamo v programu `kazalci2.c` opravka z dvema (sicer enakima) podatkoma lahko preverimo tako, da tik pred stavek `printf` v vsakega od programov dodamo naslednjo kodo:

```
f = 42;
```

V prvem primeru bo program izpisal dvakrat vrednost 42, v drugem primeru pa 42 in 500.

V programu `kazalci1.c` smo uporabili operator indirekcije za branje podatka z lokacije, na katero je kazal kazalec. Na podoben način lahko prek kazalca pišemo v pomnilnik:

```
IZPIS IZVORNE KODE          kazalci3.c
#include "stdafx.h"

int main()
{
   float f, *k;
   f = 500;
   k = &f;
   *k = 42;
   printf("%.1f\n", f);
   return 0;
}
```

V tem primeru smo najprej postavili `f` na vrednost 500, potem pa smo nanj usmerili kazalec `k`. Prek tega kazalca smo potem v `f` vpisali vrednost 42:

```
*k = 42;
```

Dokaz za to je naslednji izpis:

```
DELOVANJE PROGRAMA          kazalci3.exe
42.0
_
```

Na tem mestu se še enkrat spomnimo, da mora biti na levi strani priredilnega operatorja izraz, ki določa pomnilniško lokacijo. Izraz `*k` predstavlja lokacijo z naslovom, ki je shranjen v `k`. V našem konkretnem zgledu je izraz `*k` enakovreden izrazu `f`.

Neinicializiran kazalec

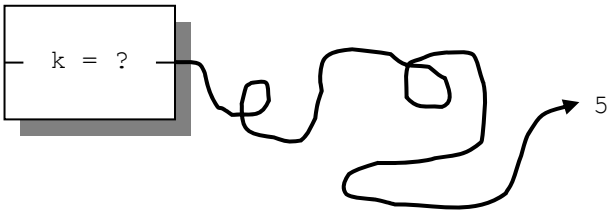
V programu `kazalci3.c` smo pisali v pomnilnik prek kazalca `k`, ki smo ga usmerili na deklarirano spremenljivko `f`. Zelo pomembno je, da vedno, kadar pišemo ali beremo podatke prek kazalca, kazalec *inicializiramo*. To pomeni, da kazalec usmerimo na veljaven (rezerviran) pomnilniški naslov.

V naslednjem primeru kazalca nismo inicializirali:

```
IZPIS IZVORNE KODE          pozor.c
#include "stdafx.h"

int main()
{
   int *k;
   *k = 5;  //UPS!!
   printf("%d\n", *k);
   return 0;
}
```

Poglejmo, kaj se tu dogaja. V programu smo rezervirali prostor za kazalec. Nismo pa rezervirali prostora, na katerega bo kazalec kazal, niti nismo kazalca nikamor usmerili (nismo mu priredili nobene vrednosti). Kazalcu, ki ga nismo nikamor usmerili, pravimo *neinicializiran* kazalec.

Stanje v pomnilniku med izvajanjem programa `pozor.c` prikazuje naslednja slika:



Ker kazalcu `k` nismo določili vrednosti, je njegova vrednost naključna. Tako z izrazom `*k = 5` pišemo petico v nakjučen del pomnilnika. Posledice takega ravnanja so nepredvidljive in so lahko zelo različne. Lahko se na primer pripeti, da program, v katerem imamo neinicializiran kazalec, ne daje vedno pravilnih rezultatov, lahko se sesuje operacijski sistem, ali pa se zgodi kakšna tretja nevšečnost. Če je program kratek, se navadno napaka ne pokaže, tako bo naš primer z veliko verjetnostjo brez vidnih zapletov izpisal na zaslon petico.

Ničelni kazalec

Da bi se izognili težavam, ki smo jih pravkar opisali, navadno neinicializiran kazalec postavimo na vrednost nič oziroma `NULL`:

```
int *k = NULL;
```

Takemu kazalcu pravimo ničelni kazalec (angl. NULL pointer). Uporabo neinicializiranega kazalca nato preprečimo tako, da najprej preverimo, če je njegova vrednost različna od `NULL`:

```
if (k != NULL)
{
      *k = 18;
}
```

Ali krajše

```
if (k)
{
      *k = 18;
}
```

Praktična uporaba kazalcev

Videli smo, da nam kazalci nudijo posreden dostop do pomnilnika, kar je povezano z določenim tveganjem. Na ta način se namreč določena mera nadzora nad uporabo pomnilnika prenese s prevajalnika na programerja, kar zahteva dodatno pazljivost pri programiranju. Nastopi upravičeno vprašanje, čemu se sploh ubadati s kazalci, če imamo s tem same probleme. V okviru tega predmeta se žal ne bomo naučili toliko, da bi lahko do konca odgovorili na to vprašanje. Bomo pa v tem razdelku pokazali eno od možnih rab kazalcev.

Predpostavimo, da želimo napisati funkcijo, ki med seboj zamenja vrednosti dveh celoštevilskih spremenljivk, podanih kot parametra. Prototip funkcije, ki sprejme dva celoštevilska parametra, izgleda takole:

```
void menjaj(int a, int b);
```

Vendar, če malo bolje razmislimo, ugotovimo, da to ne bo delovalo. Spomnimo se, da se ob klicu funkcije vrednosti izrazov, podanih v oklepaju, prekopirajo v parametre funkcije, ki delujejo kot lokalne spremenljivke. Ko se klic funkcije konča, lokalnih spremenljivk ni več in takšna funkcija ne bo imela nobenega učinka. Tako, na primer, klic

```
menjaj(x, y);
```

ne more imeti nobenega učinka na vrednosti spremenljivk `x` in `y`. Kadar podajamo funkciji parametre na takšen način, pravimo, da jih *podajamo po vrednosti*. Funkcija dobi zgolj kopijo podanih parametrov.

Če želimo, da bo imela funkcija dostop do spremenljivk, ki jih podamo kot parametre, ji ne smemo podati njihovih vrednosti, raje ji podamo njihove naslove. Prototip funkcije, ki med sabo zamenja vrednosti dveh podanih spremenljivk, bo izgledal takole:

```
void menjaj(int *a, int *b);
```

Parametra funkcije sta zdaj kazalca, v katera se ob njenem klicu prepišeta naslova podanih spremenljivk. Potem lahko funkcija prek kazalcev spreminja njune vrednosti.

Definicija funkcije bo izgledala takole:

```
void menjaj(int *a, int *b)
{
      int tmp = *a;
      *a = *b;
      *b = tmp;
}
```

Če želimo menjati vrednosti spremenljivk `x` in `y`, podamo funkciji njuna naslova:

```
menjanj(&x, &y);
```

Takemu podajanju parametrov pravimo podajanje po *referenci*. Podajanje parametrov po referenci srečujemo že ves čas pri funkciji `scanf()`:

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

Funkcijo `menjaj()` bomo uporabili v programu, ki uredi 5 celih števil po velikosti od najmanjšega proti največjemu.

```
IZPIS IZVORNE KODE          mehurcki.c
#include "stdafx.h"

void menjaj(int *a, int *b);

int main()
{
   int i, menjava;
   int p[5];

   puts("Vpiši 5 celih števil, jaz jih bom uredil po velikosti:");
   for (i = 0; i < 5; i++)
   {
      scanf("%d", &p[i]);
   }
   do
   {
      menjava = 0;
      for (i = 1; i < 5; i++)
      {
         if (p[i-1] > p[i])
         {
            menjaj(&p[i-1], &p[i]);
            menjava = 1;
         }
      }
   } while (menjava == 1);

   puts("Števila, urejena po velikosti:");
   for (i = 0; i < 4; i++)
   {
      printf("%d, ", p[i]);
   }
   printf("%d\n", p[i]);
   return 0;
}

void menjaj(int *a, int *b)
{
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
```

V programu smo uporabili algoritem urejanja z mehurčki (angl. bubblesort). Program deluje takole:

```
DELOVANJE PROGRAMA          mehurcki.exe
Vpiši 5 celih števil, jaz jih bom uredil po velikosti:
9 3 15 0 2
Števila, urejena po velikosti:
0, 2, 3, 9, 15
_
```

Kazalčna aritmetika

Nad kazalci lahko izvajamo aritmetični operaciji seštevanja in odštevanja:

Prištevanje celoštevilske vrednosti - kazalcu lahko prištejemo poljubno celoštevilsko vrednost. Kazalcu se vrednost poveča za prišteto vrednost, pomnoženo s številom celic, ki jih zaseda podatkovni tip, ki mu kazalec pripada. Vzemimo za primer deklaracijo

```
long *kazalec;
```

Stavek

```
kazalec++;
```

poveča vrednost kazalca za 4, toliko celic namreč zasede podatek tipa `long`.

Odštevanje celoštevilske vrednosti - kazalcu lahko odštejemo poljubno celoštevilsko vrednost. Kazalcu se vrednost zmanjša za odšteto vrednost, pomnoženo s številom celic, ki jih zaseda podatkovni tip, ki mu kazalec pripada. Vzemimo, da imamo še vedno deklaracijo iz prejšnjega primera. Potem stavek

```
kazalec -= 2;
```

zmanjša vrednost kazalca za 8.

Poleg tega lahko kazalce primerjamo po velikosti, lahko pa tudi dva kazalca med sabo odštejemo. Vse te operacije so seveda v glavnem smiselne le, kadar imamo kazalce, ki kažejo na zbirke podatkov.

Kazalci na zbirke

Na začetku poglavja o kazalcih smo ugotovili, da predstavlja ime zbirke konstanten kazalec. Konstantno vrednost lahko seveda s priredilnim operatorjem vedno prepišemo v spremenljivko. Tako dobimo kazalec na zbirko:

```
char razna_sara[100];
char *kaz;

kaz = razna_sara;
```

Kazalec `kaz` zdaj kaže na prvi znak zbirke `razna_sara`. Z operatorjem indirekcije lahko tako prek tega kazalca pridemo do prvega znaka niza `razna_sara`. Ta znak lahko izpišemo na zaslon takole

```
printf("%c", *kaz);
```

Prek tega kazalca lahko pridemo seveda do kateregakoli znaka. Spomnimo se, da lahko kazalcu prištejemo poljubno celoštevilsko vrednost. Na ta način lahko izpišemo na primer 2. znak takole:

```
printf("%c", *(kaz + 1));
```

Kadar imamo opravka s kazalci na zbirke, včasih namesto operatorja indirekcije uporabimo indeksni operator, ki prime tudi na kazalce. Tako velja, da je izraz

```
*kaz
```

enakovreden izrazu

```
kaz[0]
```

Prav tako je izraz

```
*(kaz + n)
```

Enak izrazu

```
kaz[n]
```

Zdaj je tudi jasno, zakaj se v Ceju začne štetje elementov zbirke od 0. Indeks v resnci predstavlja odmik od naslova 1. elementa zbirke.

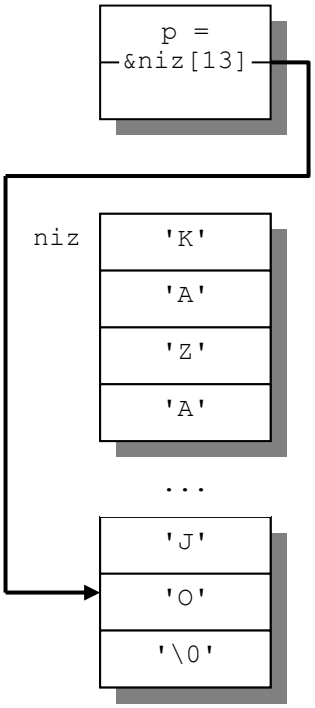
Jasno postane tudi, zakaj prištevanje konstantne vrednosti 1 kazalcu njegove vrednosti ne poveča za 1, ampak za število bajtov, ki jih v pomnilniku zaseda podatek, na katerega kazalec kaže. Povečanje kazalca za ena namreč smiselno prestavi kazalec na začetek naslednjega podatka v pomnilniku.

Za konec poglejmo primer, ki s tipkovnice prebere znakovni niz in ga obrnjenega izpiše na zaslon s pomočjo kazalca.

IZPIS IZVORNE KODE obrni.c

```
#include "stdafx.h"
#include <string.h>

char niz[100], *p;

int main()
{
   printf("Vpiši besedilo, da ti ga obrnem:");
   gets(niz);
   printf("Obrnjeno besedilo:");
   p = &niz[strlen(niz)-1];
   for (; p >= niz; p--) printf("%c", *p);
   printf("");
   return 0;
}
```

V četrti vrstici funkcije `main()` v programu `obrni.c` v kazalec `p` vpišemo naslov zadnjega znaka v prebranem nizu. S tem smo kazalec usmerili na konec niza. Naslednja slika kaže stanje v pomnilniku za primer, ko smo vtipkali niz `"KAZALCI RULAJO"`:



V zanki `for` potem postopoma zmanjšujemo `p` in vsakič izpišemo znak, na katerega `p` kaže. To počnemo toliko časa, dokler je `p` večji ali enak naslovu začetka niza.

Program se odziva takole:

```
DELOVANJE PROGRAMA          obrni.exe
Vpiši besedilo, da ti ga obrnem:
KAZALCI_RULAJO
Obrnjeno besedilo:
OJALUR_ICLAZAK
_
```

In še nekaj malega v razmislek:

```
IZPIS IZVORNE KODE          pozor1.c
#include "stdafx.h"
#include <string.h>

char niz[100], *p;

int main()
{
   strcpy(niz, "test");
   strcpy(p, "test"); //UPS!!
   //...
   return 0;
}
```

Opomba: funkcija `strcpy()` prekopira cel znakovni niz (z zaključnim ničelnim znakom vred), ki ga podamo kot drugi parameter, na naslov, ki ga podamo kot prvi parameter. Prvi klic funkcije `strcpy()` bo tako kopiral znake `'t'`, `'e'`, `'s'`, `'t'` in `'\0'` na naslove od `niz` do vključno `niz + 4`. Kaj se zgodi pri drugem klicu funkcije `strcpy()`.

Primerjava znakovnih nizev

Spomnimo se, da primerjalni operator == v jeziku C vrne 1, če imata leva in desna stran operatorja enaki vrednsoti in 0, če sta njuni vrednosti različni. Če boste preverili vrednost naslednjega izraza:

```
"enak" == "enak"
```

boste morda dobili vrednost 1, vendar ne nujno. Prvo vprašanje, ki se nam zastavi, je, kakšno vrednost sploh ima izraz `"enak"`. Navajeni smo že, da ima sleherni izraz v ceju določeno vrednost, ki je vedno elementaren podatek, se pravi številka ali znak (ki je konec koncev tudi številka). Izraz `"enak"` pa je sestavljen iz petih različnih osembitnih števil, ki predstavljajo kode ASCII štirih znakov ter zaključnega ničelnega znaka. Kakšna je torej njegova vrednost? Izkaže se, da konstanten znakovni niz kot vrednost vrne naslov, na katerem je shranjen v pomnilniku in ta naslov lahko izpišemo:

```
printf("%u", "enak");
```

V gornjem primerjalnem izrazu torej primerjamo naslova dveh sicer enakih nizev, ki pa nista nujno shranjena na isti lokaciji v pomnilniku. Odločitev, ali bo uporabljen en in isti niz ali pa bosta v pomnilniku dva enaka niza na različnih lokacijah, je prepuščena prevajalniku, zato ne moremo predvideti, kakšno vrednost bo imel izraz `"enak" == "enak"`. Takšen izraz torej nima pravega smisla.

Če želimo dva znakovna niza med seboj primerjati, bomo uporabili funkcijo `strcmp()`, ki ji kot parametra podamo naslova dveh znakovnih nizev. Funkcija bo podana znakovna niza primerjala znak za znakom in vrnila 0, če sta niza popolnoma enaka, ter -1 ali 1, če sta niza različna, odvisno od tega ali je prvi niz po abecedi (glede na kode ASCII) pred ali za drugim nizem:

```
strcmp("enak", "enak") //vrne 0
strcmp("razlicen", "enak") //vrne 1
strcmp("ena", "enak") //vrne -1
```

Kazalec na strukturno spremenljivko

Tako kot na vsako spremenljivko, lahko deklariramo tudi kazalec na spremenljivko strukturnega tipa. Vzemimo za primer strukturo `zgoscenka`, ki jo bomo definirali takole:

```
struct zgoscenka
{
      char naslov[21];
      int koda;
      int cena;
      int zaloga;
};
```

Deklarirajmo še strukturno spremenljivko `cd1` in kazalec `k_cd1`, ter usmerimo kazalec na spremenljivko:

```
struct zgoscenka cd1, *k_cd1;
```

```
k_cd1 = &cd1;
```

Spomnimo se, da potrebujemo za dostop do posamezne komponente strukturne spremenljivke selektor (`.`). Tako lahko nastavimo zalogo na 10 takole:

```
cd1.zaloga = 10;
```

Če bi hoteli nastaviti zalogo prek kazalca, bi morali z operatorjem indirekcije (`*`) najprej priti do strukturne spremenljivke, potem pa s selektorjem izbrati komponento `zaloga`:

```
(*k_cd1).zaloga = 10;
```

Tak zapis je že nekoliko nepregleden, zato obstaja še en selektor, ki ga uporabljamo, kadar dostopamo do strukturne spremenljivke preko kazalca. Selektor ima obliko puščice (`->`). Naslednji zapis je skrajšan zapis prejšnjega stavka:

```
k_cd1->zaloga = 10;
```

Osvetlimo si stvar s primerom. V naslednjem programu smo deklarirali zbirko zgoščenk in jo ob deklaraciji hkrati inicializirali. Spomnimo se, da strukturno spremenljivko ob deklaraciji inicializiramo na podoben način kot zbirko - med zavitimi oklepaji po vrsti navedemo vrednosti posameznih komponent:

SKLADNJA	deklaracija strukturne sprem. z inicializacijo
struct ime_strukture str_spremenljivka = {vk_1, vk_2, ..., vk_n};	

Kot zadnji element zbirke struktur smo vnesli prazen znakovni niz za naslov in ničle za kodo, ceno in zalogo. Ta element bomo uporabili na podoben način, kot smo uporabili zaključni ničelni znak pri detekciji zaključka znakovnega niza.

Program `trgovina.c` s tipkovnice prebere kodo zgoščenke, ki jo želimo prodati, in najprej poišče zgoščenko v seznamu. Za iskanje smo napisali funkcijo `poisci()`, ki sprejme pomnilniški naslov prve iz zbirke zgoščenk in vtipkano kodo ter vrne indeks zgoščenke, ki ji koda pripada. Če zgoščenke s podano kodo ni v seznamu, ali pa je njena zaloga enaka nič, funkcija vrne negativno vrednost.

Če smo zgoščenko našli, potem s funkcijo `izpisi()` izpišemo njen naslov in ceno ter s funkcijo `prodaj()` zmanjšamo zalogo za ena.

IZPIS IZVORNE KODE trgovina.c

```
#include "stdafx.h"

struct zgoscenka
{
   char naslov[21];
   int koda;
   float cena;
   int zaloga;
};
```

```
void prodaj(struct zgoscenka *cd);
void izpisi(struct zgoscenka *cd);
int poisci(struct zgoscenka *cd, int koda);

struct zgoscenka cd[] =  {{"ZMELKOOW - DEJ, NOSO", 566, 8.99, 20},
                          {"SIDDHARTA - NORD", 123, 9.99, 20},
                          {"SCOOTER - WE BRING T", 852, 12.99, 20},
                          {"JACKSON, MICHAEL - G", 342, 12.99, 20},
                          {"SMOLAR, ADI - NE SE ", 739, 5.99, 20},
                          {"", 0, 0, 0}};

int main()
{
   int i, koda;
   puts("Vpiši kodo zgoščenke:");
   scanf("%d", &koda);
   i = poisci(cd, koda);
   if (i < 0)
   {
      puts("Artikla ni na zalogi.");
   }
   else
   {
      izpisi(&cd[i]);
      prodaj(&cd[i]);
   }
   return 0;
}

void prodaj(struct zgoscenka *cd)
{
   cd->zaloga--;
}

int poisci(struct zgoscenka *cd, int koda)
{
   int i;
   for (i = 0; cd[i].koda != 0; i++)
   {
      if (cd[i].koda == koda)
      {
         if (cd[i].zaloga == 0) return -1;
         else return i;
      }
   }
   return -1;
}

void izpisi(struct zgoscenka *cd)
{
   printf("%s  %.2f EUR\n", cd->naslov, cd->cena);
}
```

Program deluje takole:

```
DELOVANJE PROGRAMA          trgovina.exe
Vpiši kodo zgoščenke:
566
ZMELKOOW - DEJ, NOSO  8.99 EUR
_
```

Sklad

V programiranju je sklad abstraktna podatkovna struktura, ki deluje po načelu "zadnji noter, prvi ven" (angl. last in first out, oz. LIFO). Sklad s pridom uporabljajo tako operacijski sistemi (npr. za hranjenje lokalnih spremenljivk) kot različni uporabniški programi. Vsi rekurzivni algoritmi (spoznamo jih po tem, da funkcije kličejo same sebe), na primer, uporabljajo sklad implicitno. Rekurzivne rešitve je mogoče izvesti tudi tako, da sklad uporabljamo eksplicitno. Kot primer si bomo na koncu ogledali pretvarjanje desetiške vrednosti v dvojiško.

Sklad si predstavljamo kot skladovnico vrednosti, položene eno na drugo:

67
2
852

Običajno sklad izvedemo kot zbirko vrednosti, ki ima lahko nespremenljivo velikost ali pa se po potrebi prilagaja količini podatkov. Ker slednjega z našim znanjem še ne znamo izvesti, se bomo zadovoljili s skladom konstantne velikosti. Ker vsa mesta v skladu ne bodo vedno zasedena, potrebujemo tudi nek mehanizem, ki nam bo sporočal, do kje je sklad napolnjen, oziroma, kje je *vrh* sklada. Vrh bo preprosto zaporedna številka celice, ki je prosta za odlaganje naslednjega podatka. Tako bo, na primer, prazen sklad izgledal takole:

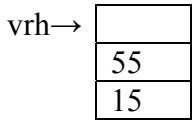
vrh→	

Vrh ima v tem primeru vrednost 0.

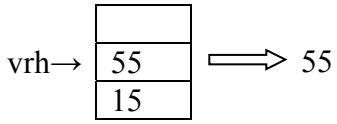
Nad skladom bomo izvajali naslednje operacije, ki so tudi zelo običajne za sklad:
init - vzpostavi začetno stanje (prazen sklad)
push - odloži vrednost na sklad
pop - vzemi vrednost z vrha sklada
peek - poglej, kakšna vrednost je na vrhu sklada
empty - sporoči, ali je sklad prazen ali ne

Zadnji dve operaciji ne vplivata na vsebino sklada, prve tri pa. Zavedati se moramo, da lahko operacija push naleti na sklad, ki je že do vrha poln. Če izvedemo operacijo push nad polnim skladom, potem pravimo, da pride do preliva sklada (angl. overflow). Prav tako se lahko zgodi, da izvedemo operacijo pop nad praznim skladom. Takrat pravimo, da se je zgodila podkoračitev (angl. underflow).

Poglejmo si primer, ko na prazen sklad potisnemo dve vrednosti (15 in 55). Vrh dobi vrednost 2, na skladu pa sta vrednosti položeni ena nad drugo:



Če sedaj nad skladom izvedemo operacijo pop (vzemi vrednost z vrha sklada), dobimo vrnjeno vrednost 55, vrh pa se pomakne za eno mesto navzdol:



Vrednost 55 je nalašč ostala prikazana na skladu, kajti jemanje vrednosti s sklada te vrednosti dejansko ne pobriše iz pomnilnika. Kako le, saj pomnilnika fizično ni mogoče brisati, kakor hitro imamo namreč na njem napetost, se posamezni biti postavijo bodisi na ena ali na nič, tako da je v njem vedno vpisana takšna ali drugačna vrednost. To, da je pozicija na skladu dejansko prosta, izvemo iz vrednsoti, ki jo ima vrh sklada. Če je vrh enak nič, potem je sklad prazen, kakor smo že videli malo prej.

Tu je zdaj primer sklada:

```
#define DIM 20

struct sklad {
	float buff[DIM];
	int vrh;
	int napaka;
};

void init(struct sklad *s)
{
	s->vrh = 0;
	s->napaka = 0;
}

void push(struct sklad *s, float v)
{
	if (s->vrh >= DIM)
	{
		s->napaka = 1; //overflow
		return;
	}
	s->buff[s->vrh++] = v;
}

float pop(struct sklad *s)
{
	if (s->vrh == 0)
	{
		s->napaka = 1; //underflow
		return 0;
	}
	s->vrh--;
	return s->buff[s->vrh];
}
```

```
int peek(struct sklad *s)
{
	return s->buff[s->vrh - 1];
}

int empty(struct sklad *s)
{
	return !s->vrh;
}
```

Pretvarjanje desetiške vrednosti v dvojiško

Kot primer uporabe sklada bomo uporabili pretvarjanje desetiške vrednsoti v dvojiško. Postopek pretvorbe je tak, da desetiško vrednost delimo z dve in si zapisujemo ostanke:

44 / 2 = 22 (ostanek: 0)
22 / 2 = 11 (ostanek: 0)
11 / 2 = 5 (ostanek: 1)
5 / 2 = 2 (ostanek: 1)
2 / 2 = 1 (ostanek: 0)
1 / 2 = 0 (ostanek: 1)

Ostanke zdaj zapišemo od zadnjega proti prvemu in dobimo dvojiško vrednost 101100. Problem, ki ga imamo pred seboj, je kot nalašč za sklad, saj zadnjo vrednost, ki smo jo dobili, rabimo najprej, prvo dobljeno vrednost pa rabimo čisto na koncu. Takole bo izgledal program:

```
int main()
{
	int desetisko = 44;
	struct sklad mojSklad;
	init(&mojSklad);

	printf("%d v dvojiskem zapisu je: ", desetisko);
	do {
		push(&mojSklad, desetisko % 2);
		if (mojSklad.napaka == 1)
		{
			printf("Napaka: preliv.\n");
		}
		desetisko /= 2;
	} while (desetisko);

	while (!empty(&mojSklad))
	{
		printf("%d", (int)pop(&mojSklad));
	}


	return 0;
}
```

Kalkulator RPN

Obratni poljski zapis (Reverse Polish Notation - RPN), je dobil ime po matematiku poljskega rodu Janu Łukasiewiczu. V tem zapisu zapisujemo izraze tako, da najprej navedemo vse operande in šele potem operacijo, ki jo želimo izvesti. Tako na primer zapišemo seštevanje vrednosti 3 in 8 kot `3 8 +`. Ena velikih prednosti takšnega zapisa je, da ne potrebuje oklepajev za spreminjanje vrstnega reda izvajanja računskih operacij, zaradi česar je enostaven za programiranje. Je pa zapis za človeškega uporabnika dokaj okoren, zato se v praksi uporablja v glavnem tam, kjer izraze računalnik tako generira kot tudi tolmači njihove vrednosti.

Računanje vrednosti izrazov poteka po zelo enostavnem postopku:
-kadar naletimo na operand, ga enostavno odložimo na vrh sklada
-kadar naletimo na operacijo, vzamemo z vrha sklada potrebno število operandov, izvedemo operacijo in rezultat odložimo nazaj na sklad.

Kot primer si oglejmo računanje vrednosti izraza `932*-4+`. Zaradi enostavnosti vzemimo, da je vsaka števka (cifra) operand zase.

Simbol	Operacija	Vsebina sklada
'9'	push 9	9
'3'	push 3	9,3
'2'	push 2	9,3,2
'*'	pop, pop, množenje, push produkt	9, 6
'-'	pop, pop, odštevanje, push razlika	3
'4'	push 4	3, 4
'+'	pop, pop, seštevanje, push vsota	7

Vrednost, ki na koncu ostane na skladu, je končna vrednsot izraza, kar je v našem primeru 7.

Tu je primer funkcije, ki vrne vrednost izraza v obliki RPN. Izraz podamo v obliki znakovnega niza, kjer predstavlja vsaka cifra en operand, možne operacije pa so seštevanje (+), odštevanje (-), množenje (*) in deljenje (/). Funkcija sprejme še en parameter (napaka), ki sporoča stanje po operaciji: 0 pomeni, da je računanje uspelo, 1 pa pomeni, da je prišlo do napake (preliv, podkoračitev, deljenje z 0, itd):

```
float RPN(char *izraz, int *napaka)
{
	struct sklad d;
	int i;
	float x, y;
	*napaka = 0;
	init(&d);
```

```
	for (i = 0; izraz[i]; i++)
	{
		if (izraz[i] >= '0' && izraz[i] <= '9')
		//Če je znak cifra (operand)
		{
			push(&d, izraz[i] - '0');
			if (d.napaka) //preliv
			{
				*napaka = 1;
				return 0;
			}
		}
		else
		//Če ni cifra (operator)
		{
			y = pop(&d);
			x = pop(&d);
			if (d.napaka) //podkoračitev (preveč operatorjev)
			{
				*napaka = 1;
				return 0;
			}
			switch (izraz[i])
			{
			case '+': push(&d, x + y); break;
			case '-': push(&d, x - y); break;
			case '*': push(&d, x * y); break;
			case '/':
				if (y == 0) //deljenje z 0
				{
					*napaka = 1;
					return 0;
				}
				push(&d, x / y); break;
			default: *napaka = 1; return 0; //neznan simbol
			}
		}

	}
	if (d.vrh != 1) //premalo operatorjev
	{
		*napaka = 1;
		return 0;
	}
	return pop(&d);
}
```

In še primer uporabe gornje funkcije:

```
int napaka;
printf("%f\n", RPN("31+", &napaka));
printf("%d\n", napaka);
```

Dinamična alokacija pomnilnika

Oglejmo si program, ki s tipkovnice prebere 4 pregovore in enega od njih izpiše na zaslon:

```
IZPIS IZVORNE KODE          pregovori.c
#include <stdafx.h>
#include <stdlib.h>
#include <time.h>

int main()
{
   char pregovor[4][256];
   int i;
   puts("Vpiši štiri pregovore:");
   for (i = 0; i < 4; i++)
   {
      gets(pregovor[i]);
   }
   srand((unsigned)time(NULL));
   printf("\nDanašnji pregovor:\n");
   puts(pregovor[rand() % 4]);
   return 0;
}
```

V programu smo za pregovore deklarirali dvodimenzionalno zbirko znakov, velikosti 4 krat 256. S štirimi klici funkcije `gets()` s tipkovnice preberemo štiri pregovore in vsakega shranimo v svojo vrstico zbirke. Pri tem upoštevamo dejstvo, da dobimo naslov prvega znaka v i-ti vrstici z izrazom

```
&pregovor[i][0]
```

kar lahko, podobno kot smo to že videli pri enodimenzionalni zbirki, zapišemo krajše

```
pregovor[i]
```

Program smo zagnali in vtipkali štiri pregovore Alberta Einsteina:

```
DELOVANJE PROGRAMA          pregovori.exe
Vpiši štiri pregovore:
Domišljija je pomembnejša od znanja.
Zdrava pamet je zbirka predsodkov, ki si jo ustvarimo do 18. leta.
Pomembno je, da se nikoli ne nehaš spraševati.
Čudež je, da radovednost preživi skozi izobraževalni sistem.

Današnji pregovor:
Pomembno je, da se nikoli ne nehaš spraševati.
_
```

Težava primera, ki smo ga pravkar videli, je ta, da smo rezervirali štirikrat po 256 bajtov pomnilnika, čeprav smo ga kasneje potrebovali veliko manj. Toliko pomnilnika moramo rezervirati, ker moramo predvideti najslabši možni primer, ki pa ga v večini primerov ne dosežemo. V našem primeru to sicer ni zaskrbljujoč problem, pri večjih količinah podatkov pa lahko postane.

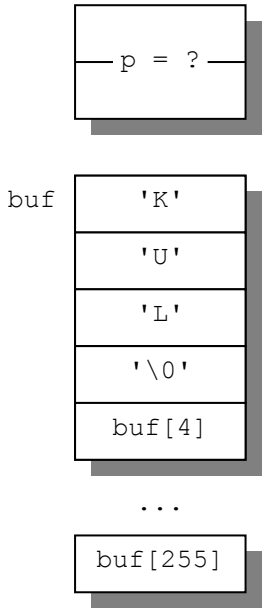
Obstaja mehanizem, ki nam omogoča bolj gospodarno rabo pomnilnika. Privoščimo si lahko, da pomnilnik rezerviramo šele takrat, ko ga dejansko potrebujemo, in ne že v času prevajanja programa. Temu pravimo *dinamična rezervacija oz. alokacija pomnilnika* (angl. dynamic memory allocation). Pomnilnik rezerviramo s pomočjo knjižnične funkcije `malloc()`, ki ji kot parameter podamo število bajtov pomnilnika,

ki ga potrebujemo. Funkcija rezervira želeno količino pomnilnika in vrne naslov začetka rezerviranega pomnilniškega bloka. Če v sistemu ni na voljo dovolj pomnilnika, funkcija vrne vrednost `NULL`.

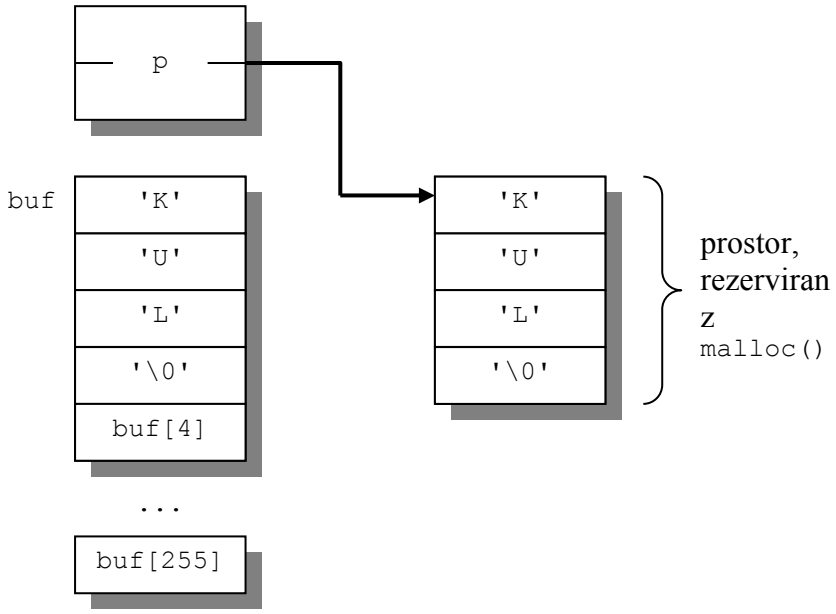
Naslednji primer kaže, kako uporabimo to funkcijo. Najprej s tipkovnice preberemo besedilo in ga shranimo v medpomnilnik (angl. buffer), za katerega smo že ob deklaraciji predvideli 256 bajtov prostora (`char buf[256];`). Potem rezerviramo točno toliko pomnilnika, kolikor ga potrebujemo za hranjenje vnešenega besedila (dolžine `strlen()` znakov plus zaključni ničelni znak), in na rezervirani blok usmerimo kazalec `p`. Če je kazalec različen od `NULL`, pomeni, da je rezervacija uspela, in besedilo prepišemo v rezervirani del pomnilnika s funkcijo `strcpy()`.

```
char *p, buf[256];

gets(buf);
p = malloc(strlen(buf) + 1);
if (p != NULL)
{
      strcpy(p, buf);
}
```

Naslednja slika prikazuje stanje v pomnilniku tik po tem, ko se je izvedla funkcija `gets()` in smo vtipkali besedo `"KUL"`. Kazalec `p` v tem hipu še ni inicializiran.



S funkcijo `malloc()` potem rezerviramo blok 4 pomnilniških celic, nanj usmerimo `p`, in s funkcijo `strcpy()` vanj (preko kazalca `p`) prepišemo besedilo, shranjeno v znakovnem nizu `buf`. V pomnilniku imamo sedaj takšno stanje.



Ko prenehamo uporabljati pomnilnik, ki smo ga rezervirali z `malloc()`, moramo pomnilnik sprostiti. To storimo s funkcijo `free()`, ki ji kot parameter podamo kazalec na blok pomnilnika, ki ga želimo sprostiti. V našem primeru bi pomnilnik sprostili s stavkom:

```
free(p);
```

Naslednji program je varčnejša različica programa `pregovori.c`. Ob deklaraciji smo tokrat rezervirali le 256 bajtov za mendpomnilnik

```
char buffer[256];
```

in zbirko štirih kazalcev:

```
char *pregovor[4];
```

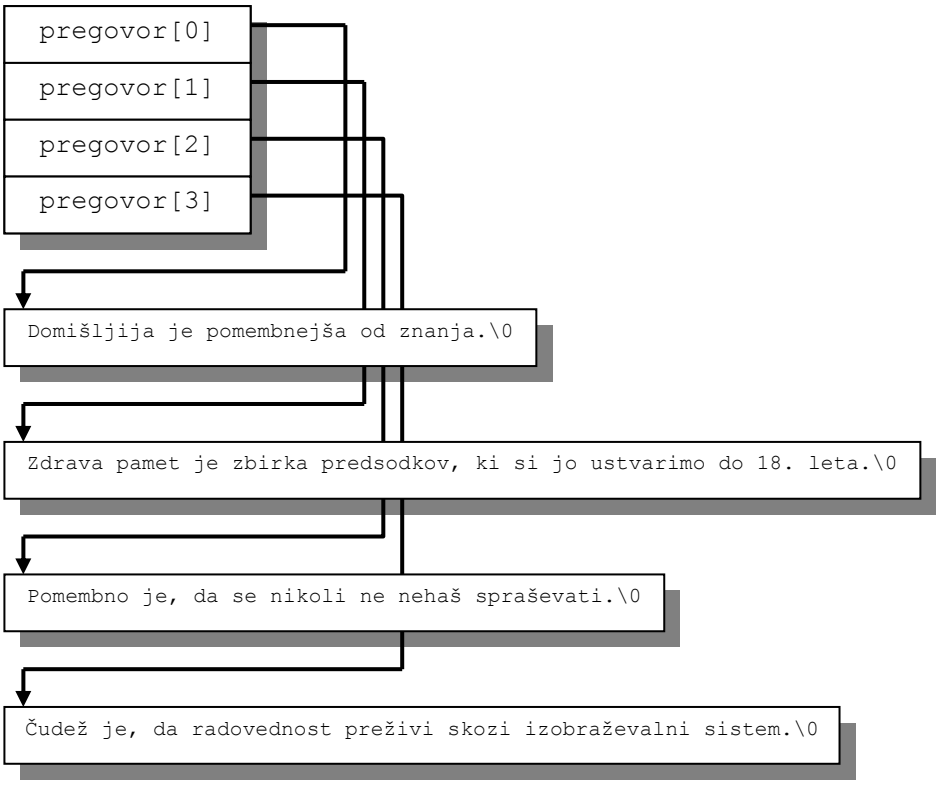
Zdaj za vsak prebrani pregovor najprej dinamično rezerviramo ravno dovolj pomnilnika in nato vanj prepišemo pregovor. Če bi pomnilnika slučajno zmanjkalo, to sporočimo na zaslon in predčasno zaključimo program s funkcijo `exit()` (izhod).

IZPIS IZVORNE KODE pregovori1.c

```
#include <stdafx.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
   char *pregovor[4];
   char buffer[256];
   int i;
   puts("Vpiši štiri pregovore:");
   for (i = 0; i < 4; i++)
   {
      gets(buffer);
      pregovor[i] = malloc(strlen(buffer) + 1);
```

```
      if (pregovor[i] == NULL)
      {
         printf("Ni dovolj pomnilnika.\n");
         exit(0);
      }
      strcpy(pregovor[i], buffer);
   }
   srand((unsigned)time(NULL));
   printf("\nDanašnji pregovor:\n");
   puts(pregovor[rand() % 4]);

   for (i = 0; i < 4; i++)
   {
      free(pregovor[i]);
   }
   return 0;
}
```

Naslednja slika prikazuje stanje v pomnilniku, ko preberemo in v pomnilnik shranimo vse štiri pregovore:



Zbirko kazalcev včasih zaključimo z ničelnim kazalcem. Če bi imeli na koncu zbirke pregovorov ničelni kazalec, bi lahko izpisali vse pregovore takole:

```
for (i = 0; pregovor[i] != NULL; i++)
{
      puts(pregovor[i]);
}
```

Načrtovanje programov

Večino problemov, ki smo jih obravnavali doslej, je bilo enostavnih in smo jih rešili s kratkimi programi. Problemi, ki jih rešujemo v realnem življenju, pogosto zahtevajo

nekoliko več znanja in truda. Pri načrtovanju večjih sistemov pogosto uporabimo metodo načrtovanja z vrha navzdol, kjer najprej določimo osnovna opravila, ki jih mora program početi, potem pa ta opravila postopoma drobimo na bolj in bolj osnovna, dokler posameznih opravil ni mogoče neposredno zapisati s preprostimi cejevskimi stavki. Poleg tega so nam pri načrtovanju v veliko pomoč orodja za razhroščevanje. Najprej bomo rekli besedo ali dve o razhroščevanju, potem pa se bomo lotili nekoliko obsežnejšega projekta. Napisali bomo preprosto računalniško igrico.

Razhroščevanje

V programerskem svetu se napake v programih že dolgo časa imenujejo hrošči (angl. bug), iskanju napak pa pravimo razhroščevanje (angl. debugging). Beseda daje lažen občutek, da so napake v programih nadležne prej kot škodljive. Vendar imajo lahko programske napake v delujočih napravah ravno tako katastrofalne posledice kot katerekoli druge napake.

Naslednje točke lahko pripomorejo h kvalitetnejši programski opremi:

- **preden začnete program kodirati, ga načrtujte** Nihče ne pomisli, da bi gradil hišo, ne da bi jo prej načrtoval. Zakaj bi bilo s programsko opremo kaj drugače?
- **izogibajte se kompleksnih rešitev** Programi naj bodo grajeni modularno, uporabljajte čimveč lokalnih spremenljivk, dobro definirane vmesnike itd.
- **izdelan program naj bo čimbolj v skladu s tem, kar smo načrtovali**

Obstaja več načinov, kako najti napake v programih, čeprav je mnogo enostavneje, da napak v programe sploh ne vnašamo (to lahko v določeni meri dosežemo z načrtovanjem). Študije so pokazale, da dlje ko pustimo napako v programu, višji so stroški, povezani z njeno odpravo. To še zlasti drži za strukturne in načrtovalske napake. Obstaja množica metod in orodij, ki nam pomagajo pri odkrivanju napak v programih:

Metoda 1 V program vključite čimveč `printf` stavkov. Tako lahko primerjate dejanski potek programa s pričakovanim.

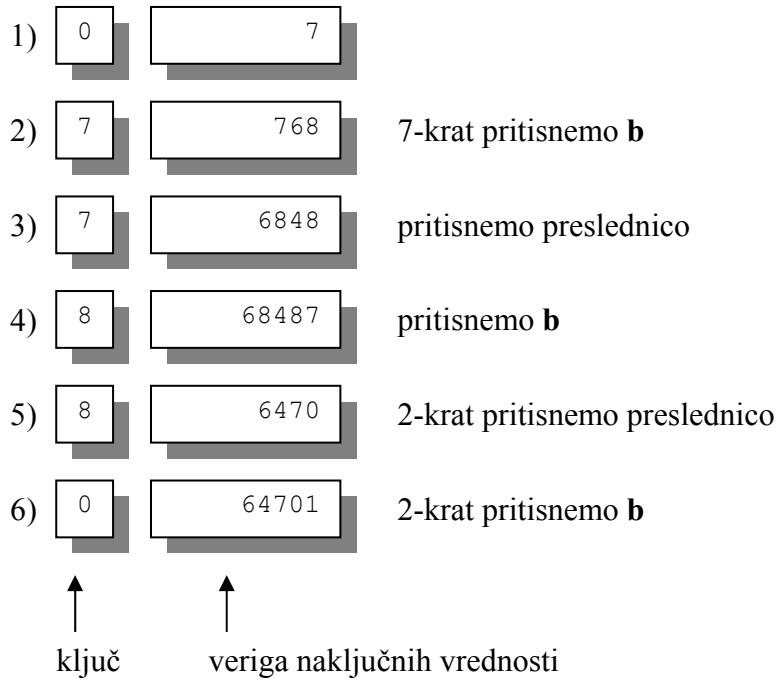
Metoda 2 Uporabite razhroščevalno orodje, s katerim lahko:

- koračno (angl. single step) izvajate programe
- opazujete vrednosti spremenljivk
- spreminjate vrednosti spremenljivk

Metoda 3 Uporabite tehten premislek

Enodimenzionalni tetris

Zdaj znamo dovolj, da se lahko lotimo resnejših stvari. Napisali bomo preprosto računalniško igrico, katere delovanje prikazuje naslednja slika:



Igra se odvija v eni sami vrstici besednega zaslona. Ta vrstica je, kot vidimo na sliki, sestavljena iz dveh polj: **ključa** in **verige naključnih vrednosti**. Veriga naključnih vrednosti je niz desetiških cifer, ki se korakoma pomika z desne proti levi, ob vsakem pomiku pa se na desni strani verige pripne nova desetiška cifra, katere vrednost je naključna. Naloga igralca je, da s pomočjo ključa čim dlje uničuje člene bližajoče se verige. Ko postane veriga predolga, je igre konec. Primer, ki ga prikazuje gornja slika, kaže, kako igralec nastavlja vrednost ključa in potem z njim uničuje člene verige. Z vsakokratnim pritiskom na tipko **b** se vrednost ključa poveča za ena. Ko pride ključ do 9 se ob pritisku na **b** vrne nazaj na 0. Če se ključ ujema s katerim od členov verige, pritisk na preslednico ta člen odstrani. Če je več členov enakih, uničimo s pritiskom na preslednico le enega. Členi, levo od odstranjenega, se pomaknejo za eno mesto v desno.

Lotimo se načrtovanja programa po metodi z vrha navzdol. Problem bomo načrtovali v treh korakih. V prvem koraku bomo načrtovali vsebino funkcije `main()`:

```
ALGORITEM                    tetris (funkcija main())
do
   inicializiraj igro (nastavi začetno vrednost verige in ključa ter
      nastavi število doseženih točk na nič)
   igraj igro
   izpiši število doseženih točk
   vprašaj igralca, če želi novo igro
while igralec želi novo igro
```

V gornjem algoritmu je potrebno razčleniti samo še vrstico `igraj igro`. Algoritem igranja igre, ki ga sestavimo v drugem koraku načrtovanja, bomo kodirali v funkciji `akcija()`.

```
ALGORITEM                    igraj igro (funkcija akcija())
do
   izberi naslednji člen verige
   premakni verigo in dodaj izbrani člen
   prikaži novo stanje na zaslon
   čakaj določen čas, medtem glej, če igralec pritisne kakšno
      tipko. Če jo, potem ustrezno ukrepaj
   if veriga je preveč zrasla
      nastavi konec igre
while konec igre
```

Od tega algoritma bomo razčlenili le še del, ko program čaka na tipko. Ta del bo kasneje kodiran v stavku `for` v funkciji `akcija()`.

```
ALGORITEM                    čakaj na tipko
začni meriti čas
do
   if tipka pritisnjena
      pritisnjen 'b': povečaj ključ
      pritisnjen ' ': odstrani člen
      pritisnjen 'q': nastavi konec igre
while čas še ni potekel
```

Tu se moramo zavedati problema hipnega reagiranja na pritisk tipke. Hipno v našem primeru pomeni dovolj hitro, da igralec ne zazna zakasnitve med pritiskom tipke in odzivom sistema. V primeru naše preproste igrice se bomo zadovoljili s pristopom poizvedovanja (angl polling), ki je opisan tudi v dodatku, kjer opisujemo sistem Arduino. Naš problem je, kako doseči, da se bo veriga v enakomernih časovnih presledkih gibala po zaslonu, igrica pa bo med presledki takoj reagirala na pritisnjene tipke. Problem je rešen v zadnjem koraku gornjega načrtovalskega postopka: časovni presledek med dvema pomikoma verige je zakodiran s ponavljalnim stavkom `do..while`, ki se ponovi tolikokrat, da mine določen čas. Ob vsaki njegovi ponovitvi hkrati preverjamo, če je na voljo kakšna tipka, in če je, takoj ukrepamo. Potem se vrnemo nazaj v ponavljalni stavek in ves postopek se nadajuje.

Dokončan program izgleda takole:

```
IZPIS IZVORNE KODE           tetris.c
#include <stdafx.h>
#include <time.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>

//konstante:
const int premor = 1000; //1 sekunda
const int dolzina_verige = 8;

//globalne spremenljivke:
char *veriga = NULL;
char kljuc;
long tocke;

//prototipi funkcij:
void inicializiraj_verigo(char *veriga, int dolzina);
void povecaj_kljuc(char *k);
int odstrani_clen(char k, char *veriga);
void premakni_verigo(char nov_clen, char *veriga);
int ponovi(void);
void izpisi_tocke(void);
void izpisi_igralno_polje(void);
char naslednji_clen(void);
void akcija(void);
void pobrisi_zaslon(void);
```

```
int main()
{
   srand((unsigned)time(NULL));
   veriga = (char *)malloc(dolzina_verige + 1);
   if (veriga == NULL)
   {
      puts("Napaka: premalo pomnilnika");
      exit(0);
   }
   do
   {
      kljuc = '0';
      tocke = 0;
      inicializiraj_verigo(veriga, dolzina_verige);
      pobrisi_zaslon();
      akcija();
      izpisi_tocke();
      pobrisi_zaslon();
   } while (ponovi());

   if (veriga != NULL)
   {
      free(veriga);
   }
   return 0;
}

void inicializiraj_verigo(char *veriga, int dolzina)
{
   int i;

   for (i = 0; i < dolzina; i++)
   {
      veriga[i] = ' ';
   }
   veriga[i] = 0;
}

char naslednji_clen()
{
   return rand() % 10 + '0';
}

void povecaj_kljuc(char *k)
{
   (*k)++;
   if (*k > '9')
   {
      *k = '0';
   }
}

int odstrani_clen(char k, char *veriga)
{
   char *tmp_veriga;

   //strchr() vrne naslov najdenega znaka v nizu
   tmp_veriga = strchr(veriga, k);

   //če znaka ni v nizu, vrne NULL
   if (tmp_veriga != NULL)
   {
      while (tmp_veriga != veriga)
      {
         tmp_veriga--;
         *( tmp_veriga + 1) = * tmp_veriga;
      }
      return 100;    //100 točk za zadetek
   }
   return 0;         //ni zadetka, brez točk
}
```

```
int ponovi()
{
   printf("\rželiš ponoviti? (d/n) ");

   while (1)  //neskončna zanka
   {
      switch(getch())
      {
         case 'd': return 1;
         case 'n': return 0;
      }
   }
}

void premakni_verigo(char nov_clen, char *veriga)
{
   int i;

   for (i = 1; veriga[i] != 0; i++)
      veriga[i - 1] = veriga[i];
   veriga[i - 1] = nov_clen;
}

void izpisi_tocke()
{
   printf("\rTOČKE: %4ld - Pritisni tipko za nadaljevanje.", tocke);
   getch();
}

void izpisi_igralno_polje()
{
   printf("\r%c %s", kljuc, veriga);
}

void pobrisi_zaslon()
{
   printf("\r                                                    \r");
}

void akcija()
{
   unsigned long cas;
   char clen;
   int nadaljuj = 1;

   do
   {
      clen = naslednji_clen();
      premakni_verigo(clen, veriga);
      izpisi_igralno_polje();

      cas = clock(); //začni meriti čas
      do
      {
         if (kbhit())
         {
            switch (getch())
            {
              case 'b':
                      povecaj_kljuc(&kljuc);
                      izpisi_igralno_polje();
                      break;
              case ' ':
                      tocke += odstrani_clen(kljuc, veriga);
                      izpisi_igralno_polje();
                              break;
              case 'q':
                      nadaljuj = 0;
            }
            if (!nadaljuj) break;
         }
      } while (clock() < cas + premor);     //je čas že potekel?
      if(veriga[1] != ' ') nadaljuj = 0;
   } while (nadaljuj);
}
```

Bolj zagnani študentje boste za vajo program dopolnili. Dodali boste težavnostne stopnje (krajšanje verige, večanje hitrosti), lestvico najboljših, itd.

Dodatek: Arduino

V drugem delu poletnega semestra se bomo spoznavali s programiranjem sistema Arduino. Opisi, ki sledijo, so bolj strnjeni in skopi kot so bili doslej, precej pa je sklicevanja na spletne strani v angleškem jeziku. Kot bodoči inženirji elektrotehnike se boste morali naučiti poiskati, brati in razumeti besedila iz najrazličnejših virov tudi (in v večini) v angleškem jeziku. Z nekaj izkušnjami boste ugotovili, da je splet nepogrešljiv in neizčrpen vir informacij (knjig, primerov, tečajev, forumov) in danes si že težko predstavljamo življenje inženirja brez dostopa do spletnih vsebin.

Izhodišče za spoznavanje sistema Arduino je (poleg predavanj) uradna spletna stran sistema Arduino `www.arduino.cc`.

Splošno namenske vhodno-izhodne sponke

Integrirana vezja imajo pogosto na voljo določeno število sponk, katerih obnašanje ni vnaprej določeno in jih lahko uporabnik konfigurira bodisi kot vhodne bodisi kot izhodne sponke. Uporabimo jih lahko za priključevanje in komunikacijo z zunanjimi napravami, na primer stikali, senzorji, motorji, prikazovalniki LCD, piskači, in tako dalje. V literaturi se splošno namenske sponke pogosto označujejo s kratico GPIO (General-Purpose input/output).

Digitalni vhodi in izhodi

Najprej moramo določiti ali bomo sponko uporabljali kot vhod (`INPUT`) ali izhod (`OUTPUT`). To storimo s klicem funkcije `pinMode()`, na primer:

```
pinMode(42, OUTPUT); //sponka 42 naj bo izhod
pinMode(41, INPUT); //sponka 41 naj bo vhod
```

Prek digitalnih sponk lahko prenašamo le logično enico (`HIGH` - visok napetostni potencial, navadno enak napajalni napetosti) ali logično ničlo (`LOW` - nizek napetostni potencial, navadno enak napetosti 0V). Na sponko, ki je določena kot izhodna lahko pišemo s funkcijo `digitalWrite()`, s sponke, ki je določena kot vhod, pa beremo s funkcijo `digitalRead()`. Na primer:

```
digitalWrite(42, HIGH); //na sponki 42 povzročimo visok
                        //napetostni potencial
int pin = digitalRead(41); //odčitamo stanje na sponki 41
                           //in ga shranimo v spremenljivko pin
```

Primer vezave diode LED in programa, ki povzroči njeno utripanje:
`arduino.cc/en/Tutorial/Blink`

Analogno digitalni pretvornik

Na vhod *n*-bitnega analogno digitalnega (A/D) pretvornika pripeljemo analogno napetost med 0V in referenčno napetostjo V_{ref}, ki je pogosto enaka napajalni napetosti sistema (3,3V ali 5V). Na izhodu dobimo *n*-bitno nepredznačeno celoštevilsko vrednost *vredn* med 0 in 2^n-1. Ob odčitani digitalni vrednosti *vredn* lahko izračunamo napetost na vhodu U_{in} po naslednji enačbi:

$U_{in} = V_{ref} * vredn / (2^n-1)$

Na primer, če imamo 10 bitni pretvornik A/D, ki je priključen na referenčno napetost 3,3V, in na izhodu preberemo vrednost 361, potem je napetost na vhodu U_{in}:

$U_{in} = 3.3V* 361 / 1023 = 1.16V$

Ker pretvornik A/D vhodno analogno napetost preslika v eno od 2^n diskretnih vrednosti, se mora vhodna napetost spremeniti vsaj za $V_{ref} / (2^n-1)$, da lahko spremembo zaznamo na izhodu. -1 v števcu je zato, ker je intervalov med vrednostmi vedno za ena manj kot pa je vrednosti. Tej minimalni spremembi, ki jo lahko zaznamo na izhodu pretvornika A/D, pravimo *ločljivost* pretvornika A/D.

Vrednost z izhoda pretvornika A/D odčitamo s pomočjo funkcije `analogRead()`. Na primer:

```
int val = analogRead(A0); //Odčitaj izhodno vrednost
                          //pretvornika A/D, priključenega
                          //na vhodno sponko A0
```

Primer priključitve potenciometra na analogni vhod A0, ki vključuje tudi uporabo serijskega monitorja: `arduino.cc/en/Tutorial/AnalogReadSerial`

Analogni izhod (PWM)

Če želimo na izhodu računalnika proizvesti analogen napetostni potencial, lahko to storimo bodisi z uporabo digitalno analognega pretvornika bodisi s pomočjo pulzno širinske modulacije (PWM - `arduino.cc/en/Tutorial/PWM`). Osnovna ideja modulacije PWM je ta, da generira pravokotne pulze spremenljive širine in konstantne frekvence. S spreminjanjem širine pulza dosežemo, da se spreminja srednja vrednost napetosti na sicer digitalni sponki. Funkcija `analogWrite()` sprejme kot argument osembitno vrednost, ki določa relativno širino pravokotnih pulzov. Vrednost 0 pomeni, da pulzov ni in na izhodu imamo konstanten signal napetosti 0V. Vrednost 255 pomeni, da so pulzi 100% časa na vrednosti HIGH, kar na izhodu povzroči konstanten signal enak napajalni napetosti.

Primer krmiljenja svetilnosti diode LED:
`arduino.cc/en/Tutorial/AnalogInOutSerial`

Serijski monitor

Zlasti v fazi načrtovanja in pisanja programov je zelo koristen serijski monitor, ki je del razvojnega okolja Arduino. Komunikacija poteka prek istega priključka USB, ki služi tudi programiranju in napajanju sistema. Za komunikacijo s serijskim monitorjem skrbi objekt `Serial` (`arduino.cc/en/Reference/Serial`), ki ima med drugim metodo `begin()`, ki odpre serijsko komunikacijo. Argument funkcije

`begin()` je hitrost komunikacije v bitih na sekundo. Privzeta hitrost vgrajenega monitorja je 9600 bitov na sekundo, zato najpogosteje uporabimo to hitrost. Za pisanje na serijski monitor lahko uporabimo funkcijo `print()`, oz `println()`, če želimo, da se po izpisanem podatku kurzor na monitorju pomakne v novo vrstico.

Primer: `arduino.cc/en/Tutorial/AnalogReadSerial`

Digitalni vhod in upori pull-up

Če digitalno sponko uporabljamo kot vhodno, potem naletimo na težavo, da stanje na tej sponki ni določeno, če je vhodna naprava odklopljena. Preprost primer je tipka, ki jo vežemo med vhodno sponko in maso: ko je tipka spuščena, potencial na vhodni sponki ni določen. Med vhodno sponko in napajalno napetosjo v ta namen vežemo t.i. pull-up upor, ki "povleče" napetost vhodne sponke proti napajalni, kadar je tipka spuščena. Sodobni mikrokrmilniki imajo pull-up upore že vgrajene in jih je potrebno samo vklopiti. V sistemu arduino to storimo tako, da funkciji `pinMode()` kot drugi parameter podamo vrednost `INPUT_PULLUP`.

Več o uporih pull-up si lahko preberete na strani `en.wikipedia.org/wiki/Pull-up_resistor`, primer uporabe vgrajenega upora pull-up pa najdete na strani `arduino.cc/en/Tutorial/InputPullupSerial`

Na enak način kot pull-up deluje tudi upor pull-down, le da je vezan na maso. Primer sistema z zunanjim uporom pull-down najdete na strani `arduino.cc/en/Tutorial/Button`

Odskakovanje kontaktov (bouncing)

Mehanski kontakti ob vklopu in izklopu pogosto odskakujejo, zaradi česar dobimo namesto čistega prehoda iz enice v ničlo in obratno nekaj kratkih pulzov, ki jih lahko digitalni sistem interpretira kot dodatne vklope in izklope. Težavo se da reševati na različne načine, možne pa so tako programske kot tudi strojne rešitve. Več o tem si lahko preberete na strani `en.wikipedia.org/wiki/Debounce#Contact_bounce`

Matrična tipkovnica

Na strani pcbheaven.com/wikipages/How_Key_Matrices_Works/ si lahko ogledate nazoren prikaz delovanja matrične tipkovnice.

V sistemu Arduino je na voljo knjižnica za delo z matrično tipkovnico. Primer njene uporabe najdete na strani `playground.arduino.cc/Main/KeypadTutorial`
Če vas zanima, kako je knjižnica napisana, si lahko ogledate vsebino datoteke `Keypad.cpp` (npr. tukaj: `github.com/kernd/Arduino-Libraries/blob/master/Keypad/Keypad.cpp`). Knjižnica je napisana v jeziku C++, vendar jo boste z nekoliko truda verjetno vseeno lahko razumeli.

Prikazovalnik LCD

Tudi za delo s prikazovalnikom obstaja knjižnica, ki je zelo enostavna za uporabo. Knjižnica uporablja protokol 44780 (`www.8052.com/tutlcd`), ki je dobil ime po Hitachijevem krmilniku HD44780 (`en.wikipedia.org/wiki/44780`), ki je najpogosteje uporabljen krmilnik za krmiljenje prikazovalnikov LCD. Protokol uporablja 3 krmilne in 8 podatkovnih vodov.

Krmilni vodi

-RS (Register Select): kadar je vrednost na sponki LOW, je podatek na podatkovnem vodu ukaz (npr. briši zaslon ali prestavi kurzor), kadar pa je vrednost HIGH, je podatek koda ASCII znaka, ki se bo prikazal na zaslonu.

-RW (Read Write): kadar je vrednost na sponki LOW, prikazovalnik sprejema podatke, kadar pa je vrednost HIGH, prikazovalnik prek podatkovne sponke D7 javlja ali je zaposlen (D7 je HIGH) ali pa je pripravljen na naslednji ukaz (D7 je LOW).

-E (Enable): pred pošiljanjem podatkov na LCD mora biti ta sponka postavljena na vrednost LOW. Ko so vsi podatki in krmilni signali nastavljeni, je potrebno sponko E za nekaj časa postaviti na HIGH in šele takrat LCD obdela podatke in ukaze, ki smo mu jih poslali. Ko preteče čas, ki ga potrebuje LCD za branje ukaza (ta čas lahko razberemo iz uporabniškega priročnika), ali pa se D7 postavi na vrednost LOW (če je RW vod postavljen na HIGH), lahko sponko E zopet nastavimo na vrednost LOW in cikel se lahko ponovi.

Podatkovni vodi

Prikazovalnik LCD ima osem priključkov za podatke, označene z D7 do D0. LCD lahko priključimo na sistem s štirimi ali osmimi podatkovnimi vodi. Če priključimo le štiri vode, potem moramo priključiti vode od D7 do D4 in podatek na LCD se prenese v dveh korakih: najprej gornji in potem še spodnji štirje biti.

Primer uporabe knjižnice `LiquidCrystal.h` s priključenimi štirimi podatkovnimi vodi

(Sponka RW je vezana na maso, zato lahko v tem primeru samo pišemo podatke na LCD, ne moremo pa preverjati njegovega statusa. Kompleten primer si lahko ogledate na `arduino.cc/en/Tutorial/LiquidCrystal`)

```
#include <LiquidCrystal.h> //vključi knjižnico
LiquidCrystal lcd(12, 11, 5, 4, 3, 2); //ustvari objekt lcd in
      //upoštevaj naslednje povezave: RS je vezan na digitalno sponko
      //12, E je vezan na sponko 11, podatkovne sponke D4 do
      //D7 pa so vezane na sponke 5 do 2. Ker je RW vezan na
      //maso, ga tu ni potrebno navajati
lcd.begin(16, 2); //lcd ima 2 vrstici in 16 stolpcev
```

Vgrajeni sistemi

Ko smo na sistem Arduino priklopili tipkovnico in prikazovalnik LCD, si lahko predstavljamo, da je tak sistem lahko že povsem neodvisen od računalnika, prek

katerega pišemo in preizkušamo programe. Če odklopimo vodilo USB in priklopimo avtonomno napajanje (`playground.arduino.cc/Learning/WhatAdapter`). Dobljen sistem bi že lahko poimenovali *vgrajen sistem* (angl. embedded system). Najosnovnejša značilnost vgrajenih sistemov je, da so ti sistemi posvečeni točno določeni nalogi, za razliko od splošno namenskih računalnikov, ki jih lahko uporabniki navadno poljubno programirajo. Primer vgrajenega sistema je predvajalnik MP3, ki je lahko samostojna enota, lahko pa je vgrajen na primer v avtomobil. Vgrajen sistem je lahko tudi sistem za nadzor tlaka v pnevmatikah ali pa modul GPS. Vgrajeni sistemi pogosto delujejo v realnem času.

Sistemi v realnem času

Za sisteme, ki delujejo v *realnem času* (angl. real time) je značilno, da se morajo odzivati v točno določenih časovnih okvirih. Prav tako kot ne smejo biti prepočasni, ne smejo biti niti prehitri. Predstavljajte si digitalno uro, ki je tudi primer vgrajenega sistema. Ura, ki prehiteva, ni nič boljša kot ura, ki zaostaja. Tudi semafor se ne sme takoj odzvati na zahtevo pešca za prečkanje ceste, saj mora upoštevati zavorno pot avtomobilov in zeleni val, če so v bližini še kakšni drugi semaforji. Predolgo pa tudi ne sme pustiti čakati pešča.

Časovnik

Glede na to, da vgrajeni sistemi delujejo večimona v realnem času, potrebujejo za svoje delo časovnike (angl. timer). Časovniki so poleg GPIO med najpomembnejšimi stvarmi, ki jih mora razvijalec vgrajenih sistemov dobro poznati. Sicer redko uporabljen način uporabe časovnika v resnih sistemih, vendar dovolj enostaven za razlago osnovnega principa delovanja, je način s *poizvedovanjem* (angl. polling). Poizvedovanje je princip, kjer krmilnik znova in znova preverja stanje časovnika in ukrepa, ko je čas za ukrepanje zrel. Princip je podoben človeku, ki mora na primer vsake deset minut pomešati golaž, vmes pa ima seveda še ogromno časa za druge reči. Vsakič, ko pomeša golaž, izračuna, kdaj bo napočil trenutek za naslednje mešanje. Med drugimi opravili neprestano pogleduje na uro in primerja prebrano uro s časom, ki si ga je zapomnil. Ko ugotovi, da je ura že dosegla čas, ko je potrebno spet pomešati, to naredi in spet izračuna naslednji čas mešanja.

Sistem arduino pozna funkcijo `millis()`, ki ob slehernem klicu vrne število milisekund, ki so minile od vklopa sistema. Tako je klic funkcije millis() podoben gledaju na uro. Potebujemo seveda še neko spremenljivko, npr z imenom `marker`, ki hrani naslednji čas, ob katerem je treba kaj postoriti.

```
if ((long)(millis() - marker) > 0)
{
      mesaj(); //postori, kar je za postoriti

      marker += interval; //nastavi naslednji čas, ko bo potrebno to
                          //spet postoriti
}
```