# Kako naredimo ...

... v programskem jeziku C

Pripravila: Alenka Kavčič

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
februar 2006

# KAZALO


| | | |
|---|---|---|
| **1.** | **Uvod** | **1** |
| | Priprava platforme za delo | 1 |
| | Primeri programov | 1 |
| **2.** | **Pisanje, prevajanje in izvajanje programov** | **3** |
| | Pisanje izvorne kode programa | 3 |
| | Prevajanje programa | 3 |
| | Izvajanje programa | 4 |
| | Uporaba matematične knjižnice | 4 |
| | Opozorila pri prevajanju | 5 |
| **3.** | **Prvi preprosti programi** | **7** |
| | Prepisovanje standardnega vhoda na izhod | 7 |
| | Spreminjanje velikih črk v male | 8 |
| | Štetje prebranih znakov ter velikih in malih črk | 9 |
| | Brisanje večkratnih presledkov | 11 |
| **4.** | **Kazalci, naslovi, polja - osnovni pojmi** | **13** |
| | Kazalci in naslovi | 13 |
| | Kazalci in argumenti funkcij | 15 |
| | Polja in kazalci | 16 |
| | Nizi znakov | 18 |
| | Večdimenzionalna polja in polja kazalcev | 22 |
| **5.** | **Formatiran izpis in branje** | **25** |
| | Formatiran izpis - funkcija | 25 |
| | Formatirano branje - funkcija | 26 |
| | Še en primer | 27 |
| **6.** | **Argumenti ukazne vrstice** | **29** |
| | Števila kot argumenti programa | 30 |
| | Opcije kot argumenti programa | 31 |
| **7.** | **Delo z datotekami** | **35** |
| | Standardne datoteke | 36 |
| | Nekaj primerov: branje in izpis po znakih | 37 |
| | Še en primer: branje in izpis po vrsticah | 39 |
| | In za konec še: formatirano branje | 40 |
| **8.** | **Rekurzija** | **43** |
| | Rekurzivno reševanje problema | 43 |
| | Podrobnosti o izvajanju rekurzivnega programa | 44 |
| | Reševanje problema: iterativno ali rekurzivno? | 45 |
| | *Eleganca napram učinkovitosti* | *47* |
| | *Po naravi rekurzivni problemi* | *48* |
| | Še nekaj primerov rekurzivnih rešitev problemov | 49 |
| | *Pretvorba med številskimi sistemi* | *49* |
| | *Palindromi* | *50* |
| | *Permutacije* | *52* |
| | *Izris vzorca iz pik* | *53* |
| | *Izpis obrnjene datoteke* | *54* |

**9. Kazalčni seznami in drevesa** **57**
Dinamično dodeljevanje pomnilnika 57
Strukture, kazalci na strukture in rekurzivne strukture 59
Kazalčni seznami 60
*Deklaracija elementov seznama 64*
*Izpis seznama 65*
*Iskanje elementa v seznamu 65*
*Dodajanje elementa v seznam 67*
*Brisanje elementa iz seznama 72*
*Brisanje vseh pojavitev elementa iz seznama 75*
*Brisanje celega seznama 76*
*Sestavimo program 77*
*Vračanje vrednosti funkcije preko argumenta 80*
Urejeni seznami 85
*Iskanje elementa v urejenem seznamu 86*
*Brisanje elementa iz urejenega seznama 86*
*Dodajanje elementa v urejen seznam 87*
*Sestavimo program 87*
Binarna drevesa 90
*Deklaracija vozlišča drevesa 91*
*Iskanje vrednosti v drevesu 92*
*Izpis drevesa 92*
*Dodajanje vozlišča v drevo 93*
*Brisanje vozlišča iz drevesa 94*
*Brisanje celega drevesa 95*
*Sestavimo program 95*
**10. Literatura** **101**

# 1. Uvod

Gradivo je namenjeno študentom prvih letnikov na Fakulteti za računalništvo in informatiko, ki se pri predmetih Programiranje 2 in Osnove programiranja 2 prvič srečajo s programiranjem v programskem jeziku C. Zajema izbrane teme iz osnov programiranja v jeziku C in skozi primere programov, ki so tudi podrobneje razloženi, prikazuje pristope k reševanju različnih problemov.

To gradivo ni učbenik za programski jezik C in ni namenjeno samostojnemu učenju jezika, temveč je le dopolnilo drugim učbenikom oziroma zapiskom s predavanj. Tako predpostavlja poznavanje in razumevanje osnovnih konceptov programskih jezikov, kot so spremenljivke in njihova uporaba, zanke, odločitveni stavki, itd. ter se osredotoči na posebnosti jezika C in na pristop k reševanju problemov v tem jeziku.

### Priprava platforme za delo

Zelo priporočljivo je, da v tem gradivu obdelane primere tudi sami preizkusite. To pomeni, da program napišete v računalniku, ga izvedete in preverite njegovo delovanje. V gradivu se omejimo na delo v Unix okolju, katerega uporabljamo tudi na laboratorijskih vajah iz predmetov Programiranje 2 in Osnove programiranja 2. Tako okolje pa ni vezano le na Unix/Linux operacijski sistem, saj ga lahko vzpostavimo tudi na Microsoft Windows operacijskih sistemih s pomočjo brezplačnih orodij Cygwin (www.cygwin.com). Cygwin je okolje za Windows, ki je podobno Linuxu, in vključuje obsežno zbirko orodij.

Za prevajanje in povezovanje programov priporočamo uporabo standardnega GNU C/C++ prevajalnika. Na Linux platformi je prevajalnik že del namestitvenega paketa. Za Windows platformo pa ga najdemo med orodji Cygwin, a moramo ponavadi njegovo namestitev posebej zahtevati (označiti pri namestitvi orodij).

### Primeri programov

Izvorne kode vseh primerov programov, kjer je označeno tudi ime programa, so priložene temu gradivu. Dobite jih na spletu pod dodatnimi gradivi na domači strani predmeta Osnove programiranja 2 (http://lgm.fri.uni-lj.si/op2/).

Ker nihče ni popoln, tudi to gradivo ni odporno na napake. Zato vas prosim, da najdene napake sporočite na naslov alenka.kavcic@fri.uni-lj.si, da jih bomo lahko popravili. Prav tako so dobrodošli tudi vsi predlogi za nove teme, nove primere, podrobnejše razlage opisanih primerov in podobno.

# 2. Pisanje, prevajanje in izvajanje programov

Najprej si bomo pogledali, kako napišemo enostaven program v programskem jeziku C in kaj vse moramo narediti, preden nas razveselijo rezultati (delovanje) našega programa.

Celoten postopek razdelimo na tri korake:
- pisanje izvorne kode programa,
- prevajanje programa ter
- izvajanje programa.

### Pisanje izvorne kode programa

Vsak program se začne pri izvorni kodi. Le-to lahko napišemo v poljubnem urejevalniku besedila, ki pa nam mora omogočati shranjevanje vsebine kot navadno besedilo (*plain text*). Ponavadi je preprost urejevalnik že sestavni del operacijskega sistema. Primera takih preprostih urejevalnikov sta *Beležnica* (*Notepad*) v Windows okolju ali *vi* (ali izboljšana različica *vim*) v Linux/Unix okolju. Seveda je uporaba določenega urejevalnika stvar okusa in navad vsakega posameznika.

Naj kot prvi primer napišemo enostaven program, ki na zaslon izpiše besedo `Pozdravljeni!` in skoči v novo vrstico. Odpremo torej urejevalnik besedil in vanj vpišemo naslednje vrstice:

```
main()
{
  printf("Pozdravljeni!\n");
}
```

Vsebino shranimo kot navadno besedilo v datoteko z imenom `prvi.c` (pri tem pazimo, da je ime datoteke res tako, kot smo ga podali). Imena datotek, ki vsebujejo izvorno kodo C, naj imajo vedno podaljšek `.c` (kot npr. naš program `prvi.c`). Tako lahko po imenu (mi in tudi prevajalnik, ki tako ime zahteva) vedno prepoznamo datoteko z izvorno C kodo.

### Prevajanje programa

Program prevedemo v ukazni lupini (odpremo terminalsko okno oziroma *Cygwin Bash Shell*). Za prevajanje uporabimo prevajalnik *GNU C/C++ Compiler*. Pokličemo ga z ukazom `gcc`, kateremu sledi ime datoteke z izvorno kodo in lahko tudi poljubne opcije.

Naš program, katerega izvorna koda se nahaja v datoteki `prvi.c`, prevedemo in povežemo z ukazom:

```
gcc prvi.c
```

Kot rezultat zgornjega ukaza (če med prevajanjem ni prišlo do napake zaradi napačno napisane kode) se ustvari izvršljiva datoteka z imenom `a.out` (oziroma `a.exe` v Cygwin okolju), ki vsebuje izvajalno kodo.

Pri klicu prevajalnika je priporočljivo dodati tudi opcijo `-o`, ki določi ime izhodne (izvršljive) datoteke (v našem primeru je to `program`; v tem primeru seveda ne ustvari datoteke `a.out`):

```
gcc -o program prvi.c
```

Prevajanje z ukazom `gcc` pravzaprav zajema štiri faze: predprocesiranje (*preprocessing*), prevajanje v ožjem smislu (*compilation*), zbiranje (*assembly*) in povezovanje (*linking*), vedno v tem vrstnem redu. Čeprav lahko s pomočjo stikal spreminjamo delovanje ukaza `gcc` in tako vklopimo le nekatere od omenjenih faz, bomo v naših primerih vedno uporabljali vse štiri.

## Izvajanje programa

Tudi izvajanje programa poteka v ukazni lupini. Ko smo s prevajanjem ustvarili izvršljivo datoteko (v našem primeru je to `program` oziroma `program.exe` v Cygwin okolju), lahko program zaženemo s klicem te datoteke v ukazni vrstici:

```
./program
```

Pri klicu programa moramo paziti, da podamo celotno pot do datoteke, če poti nimamo posebej nastavljene v sistemski spremenljivki (pred ime izvršljive kode moramo napisati tekoči direktorij, to je `./`).

Kakšen pa je rezultat? Program izpiše:

```
Pozdravljeni!
```

in nato skoči v novo vrstico ter konča.

## Uporaba matematične knjižnice

Kadar v programu uporabljamo matematične funkcije iz matematične knjižnice, moramo v kodo programa vključiti zaglavno datoteko (*header file*) `math.h`, v kateri so ustrezne prototipne deklaracije funkcij in spremenljivk iz knjižnice.

Primer enostavnega programa, ki uporablja funkcijo za računanje potence `pow`, je datoteka `potenca.c` z naslednjo vsebino:

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

main()
{
  double pot;

  pot = pow(5.2, 3.0);
  printf("5.2 na tretjo potenco je %f\n", pot);
}
```

Pri prevajanju programa moramo posebej paziti, da v izvršljivo datoteko povežemo tudi matematično knjižnico. Stavek `#include <math.h>` v program ne vključi matematične knjižnice same, ampak le pripadajočo zaglavno datoteko (lahko bi rekli, da na nek način le napove uporabo matematične knjižnice). V zaglavni datoteki najdemo le deklaracije funkcij (glava funkcije), medtem ko so ustrezne definicije funkcij (telo funkcije) v sami knjižnici. Vsako knjižnico, ki jo uporabljamo v programu, moramo posebej povezati s programom. Za matematično knjižnico to zahtevamo z opcijo `-lm` pri prevajanju:

```
gcc -lm -o potenca potenca.c
```

Standardna knjižnica C (na katero se nanaša tudi zaglavna datoteka `stdio.h`) vsebuje funkcije, ki jih določa ANSI/ISO C standard (kot je na primer funkcija `printf`). Ta knjižnica se privzeto vedno poveže pri prevajanju vsakega programa v C-ju in je ni potrebno posebej navajati.

## Opozorila pri prevajanju

Pri prevajanju izvorne kode lahko uporabimo tudi opcijo `-Wall` (*Warning all*), ki pri prevajanju vključi tudi izpis vseh opozoril. Tako bi ob prevajanju našega prvega programa z ukazom

```
gcc -Wall -o program prvi.c
```

dobili tri opozorila: privzet tip, ki ga vrača funkcija (`main`), je `int`; drugo opozorilo pravi, da je funkcija `printf` le implicitno deklarirana; tretje opozorilo pa pravi, da smo prišli do konca funkcije, ki ni `void`.

Čeprav naš program deluje kljub navedenim opozorilom, pa bi ga napisali pravilneje, če bi upoštevali tudi navedena opozorila. Torej moramo poskrbeti za deklaracijo funkcije `printf`. Ker je ta funkcija že deklarirana v zaglavni datoteki `stdio.h`, zadostuje, da v program vključimo to zaglavno datoteko. Obe preostali opozorili pa odpravimo, če

eksplicitno določimo tip, ki ga vrača funkcija `main`, in ob koncu funkcije s stavkom `return` tudi poskrbimo, da funkcija vrne ustrezno vrednost. Naš program dopolnimo, da dobimo naslednjo kodo (shranimo ga v datoteko `prvi1.c`):

```
#include <stdio.h>

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

Sedaj pri prevajanju programa z vključeno opcijo izpisov vseh opozoril ne dobimo nobenega opozorila več, torej smo napisali prevajalsko čist program.

# 3. Prvi preprosti programi

Za ogrevanje si poglejmo nekaj kratkih programov skupaj z razlago kode in delovanja programa.

### Prepisovanje standardnega vhoda na izhod

Začnimo s programom, ki bere znake s standardnega vhoda in jih sproti izpisuje na standardni izhod. Za branje znaka s standardnega vhoda (tipkovnice) bomo uporabili funkcijo `getchar` (pravzaprav je to makro, ki je definiran v zaglavni datoteki `stdio.h`), ki vrne naslednji znak z vhoda oziroma konstanto `EOF`, če je prišlo pri branju do napake ali pa do konca vhodnega toka. Za izpisovanje znakov uporabimo funkcijo `putchar` (tudi ta je definirana kot makro v zaglavni datoteki `stdio.h`), ki na izhod zapiše znak, ki je podan kot argument funkcije. Program `prepis.c` izgleda v prvi različici takole:

```
#include <stdio.h>

main()
{
  int c;

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

Funkcija `getchar` vrača celoštevilski tip, zato mora biti prebran znak (spremenljivka `c`) tudi celoštevilskega tipa. Poleg tega ima konstanta `EOF` ponavadi vrednost `-1`, ki je nikakor ne bi mogli spraviti v tip `char` (le pozitivne vrednosti!), saj se mora `EOF` ločiti od vseh drugih možnih znakov. Tudi argument funkcije `putchar` je po definiciji celo število.

Ko program poženemo, se navidezno ne zgodi nič. Program namreč čaka na vhod, torej na nas, da vpišemo znake preko tipkovnice. Vpis zaključimo s tipko *Enter* in s tem pošljemo vpisane znake naprej v obdelavo. V odgovor nam program na zaslon izpiše isto zaporedje znakov. In postopek se ponovi.

Napisan program deluje toliko časa, dokler ne prebere znaka za konec datoteke `EOF`. In kako lahko ta znak vpišemo preko tipkovnice? To dosežemo s kombinacijo tipk `Ctrl` in `d`. Znak `EOF` na tipkovnici je torej `Ctrl+d`.

Zgornji program lahko zapišemo tudi krajše in sicer na način, ki se pogosto uporablja v jeziku C. Ker prirejanje lahko uporabimo tudi v izrazih, bomo prireditveni stavek

`c=getchar();` vključili kar v sam pogoj `while` zanke. Tako program skrajšamo in zapišemo tudi bolj pregledno (`prepis1.c`):

```
#include <stdio.h>

main()
{
  int c;

  while( (c=getchar()) != EOF )
    putchar(c);
}
```

Pri tem moramo biti pozorni na postavitev oklepajev, saj z njimi določimo pravilen vrstni red izvajanja operacij (`!=` ima sicer večjo prioriteto kot `=`). V zanki se tako najprej prebere en znak z vhoda, prebrani znak se priredi spremenljivki `c`, nato pa se preveri, ali je ta znak enak znaku `EOF`. Če znaka nista enaka (prebrani znak ni `EOF`), se izvrši telo zanke, to je izpis znaka `c` na izhod. Nakar se zanka ponovi. Zanka se zaključi, ko pogoj zanke ni več izpolnjen, kar pomeni, da je prebrani znak enak `EOF`. Takrat se zaključi tudi program.

### Spreminjanje velikih črk v male

Dopolnimo zgornji program tako, da vse črke od prebranih znakov izpisujemo kot male črke. Program torej bere znake iz standardnega vhoda, jih izpisuje na standardni izhod, pri tem pa vse velike črke spremeni v male. Program je zelo podoben prejšnjemu, le da pred izpisom znaka preverimo, ali je le-ta velika črka.

Vsakemu znaku pripada neka koda po ASCII tabeli. Pri tem velja, da imajo zaporedne velike črke A do Z (po angleški abecedi, vseh črk je 26) tudi zaporedne kode (A ima kodo 65, Z pa 90). Podobno velja tudi za zaporedne male črke a do z, ki v tabeli ležijo za velikimi črkami (koda a je 97, z pa 122), in števke 0 do 9. Samih kod nam ni potrebno poznati, saj lahko z znaki tudi računamo (prištevamo, odštevamo) in jih medsebojno primerjamo. Tako nam na primer `'A'+1` da rezultat `'B'`, `'z'-25` pa je enako `'a'`.

Veliko črko spremenimo v malo tako, da ji prištejemo ustrezno število, ki je enako ravno razliki kod med veliko in malo črko. To razliko lahko izračunamo (koda A je 65, koda a je 97, razlika je 32) ali pa enostavno izrazimo s pomočjo znakov `'A'` in `'a'`, to je kot `'a'-'A'`.

Za pretvarjanje velikih črk v male moramo najprej ugotoviti, ali je prebrani znak velika črka. To pomeni, da mora biti prebrani znak večji ali enak znaku `'A'` in hkrati manjši ali enak znaku `'Z'` (`'A' <= znak <= 'Z'`).

Če vse povedano vgradimo v naš program, dobimo naslednjo kodo (`prepis2.c`):

```
#include <stdio.h>

main()
{
  int c;

  while( (c=getchar()) != EOF ) {
    if( (c>='A') && (c<='Z') )
      putchar(c + 'a'- 'A');
    else
      putchar(c);
  }
}
```

Dodali smo `if` stavek, v katerem preverimo, ali je prebrani znak velika črka. Če je pogoj izpolnjen, izpišemo znak, katerega koda je za 32 večja od prebranega znaka, sicer pa izpišemo kar prebrani znak.

Tu naj opozorimo še na razliko med operatorjema `&&` in `&`. Čeprav v našem primeru program deluje enako, če uporabimo enojni ali dvojni `&`, pa je njun pomen precej različen. Operator `&&` pomeni *logični in* in vrne 1 (resnično), kadar sta oba operanda resnična (različna od nič), sicer pa vrne 0 (neresnično). Tako `0&&1` vrne `0`, izraz `2&&1` pa vrne vrednost `1`. Operator `&` pa predstavlja *bitni in*, kar pomeni, da deluje nad posameznimi biti. Izraz `0&1` bo imel še vedno vrednost `0`, a izraz `2&1` da tudi vrednost `0`, saj se operacija izvede po bitih (2 je 10 dvojiško, 1 pa je 01 dvojiško). Podobno velja tudi za operatorja *logični ali* (||) in *bitni ali* (|).

## Štetje prebranih znakov ter velikih in malih črk

Naš zadnji program dopolnimo še tako, da na koncu izpiše, koliko je bilo vseh prebranih znakov in koliko od tega je bilo črk (velikih in malih posebej). Seveda vsota prebranih malih in velikih črk ne da števila vseh prebranih znakov, saj so med njimi lahko tudi števke, ločila in drugi znaki, ki ne spadajo med črke.

Nalogo rešimo v dveh korakih. Najprej bomo uvedli le en števec za štetje vseh prebranih znakov in na koncu dodali izpis tega števca. Za štetje znakov uvedemo celoštevilsko spremenljivko `stc`, kateri nastavimo začetno vrednost 0 (nismo prebrali še nobenega znaka). Ob vsakem prehodu zanke (ko preberemo en znak) pa vrednost spremenljivke povečamo za ena. Na koncu, ko se zanka izteče, dodamo še stavek `printf`, ki izpiše vrednost števca `stc`.

```
#include <stdio.h>

main()
{
  int c;
  int stc = 0;

  while( (c=getchar()) != EOF ) {
    stc++;
    if( (c>='A') && (c<='Z') )
      putchar(c + 'a'- 'A');
    else
      putchar(c);
  }
  printf("Skupaj je bilo prebranih %d znakov.\n", stc);
}
```

V drugem koraku uvedemo še dva števca: `stm` za preštevanje malih črk in `stv` za preštevanje velikih črk. Začetni vrednosti obeh števcev sta nič, v zanki pa ustrezno povečujemo oba števca.

Pri štetju velikih črk ni težav, saj prebrani znak vedno testiramo, ali je velika črka, ker v tem primeru izpišemo spremenjen znak. Tako lahko v `if` del stavka dodamo še povečevanje števca velikih števil in naloga je opravljena. Pri tem moramo paziti le, da oba stavka pod `if` obdamo z zavitimi oklepaji.

Več dela pa imamo pri štetju malih črk. Če prebran znak ni velika črka, ni nujno, da je ta znak mala črka (lahko je na primer tudi števka ali ločilo). Zato moramo podobno, kot smo preverili, ali je prebrani znak velika črka, preveriti tudi, ali je morda prebrani znak mala črka, in v tem primeru povečati števec malih črk. Tako smo tudi v `else` delu obstoječega `if` stavka napisali dva stavka (`if` stavek in izpis znaka), ki ju moramo obdati z zavitimi oklepaji.

Na koncu moramo seveda še dopolniti izpis in izpisati vrednosti vseh treh števcev. Celoten program izgleda takole (`prepis3.c`):

```
#include <stdio.h>

main()
{
  int c;
  int stc = 0;
  int stm, stv;

  stm = stv = 0;
  while( (c=getchar()) != EOF ) {
    stc++;  // stejemo vse prebrane znake
    if( (c>='A') && (c<='Z') ) {  // ali je velika crka
      putchar(c + 'a'- 'A');
```

```
      stv++;  // stejemo velike crke
    }
    else {
      if ( (c>='a') && (c<='z') ) // ali je mala crka
        stm++;  // stejemo male crke
      putchar(c);
    }
  }
  printf("Skupaj je bilo prebranih %d znakov.\n", stc);
  printf("Malih crk je %d, velikih pa %d.\n", stm, stv);
}
```

V programskem jeziku C se lokalne spremenljivke ne inicializirajo samodejno, torej imajo ob deklaraciji nedefinirano (naključno) vrednost. Zato je zelo dobra praksa, da spremenljivke vedno eksplicitno inicializiramo. Tako smo vsem števcem, ki jih uporabljamo v programu, nastavili vrednost na nič. V zgornjem programu smo to naredili na dva načina: z inicializacijo ob sami deklaraciji (spremenljivka `stc`) in s prireditvenim stavkom, ki sledi deklaraciji (`stm` in `stv`).

Bodite pozorni tudi na prireditve v zgornjem programu. Prireditev (=) je namreč v programskem jeziku C tudi operator, zato imamo lahko večkratne prireditve ali pa jih uporabimo v izrazih:

```
stm = stv = 0;  // večkratna prireditev
while( (c=getchar()) != EOF )  // prirejanje v izrazu
```

Pa še na eno stvar moramo opozoriti. V programu smo za povečanje vrednosti spremenljivke `stc` za ena uporabili operator inkrement (`stc++`). Ta operator, podobno kot operator dekrement (`stc--`), lahko uporabimo v prefiksni (`++stc`) ali postfiksni (`stc++`) obliki. V obeh primerih je končni rezultat enak, to je za ena večja vrednost spremenljivke `stc`. Razlika je opazna šele pri uporabi operatorja v izrazu, saj `++stc` poveča vrednost spremenljivke `stc` pred njeno uporabo v izrazu, medtem ko `stc++` poveča njeno vrednost po uporabi v izrazu. Če ima spremenljivka `stc` vrednost 10, stavek `st=stc++;` priredi spremenljivki `st` vrednost 10, stavek `st=++stc;` pa vrednost 11. V obeh primerih pa ima po izvršitvi stavka spremenljivka `stc` vrednost 11.

### Brisanje večkratnih presledkov

Poglejmo si še en primer. Tokrat napišimo program, ki bere vhod in prebrano izpisuje na izhod, le da pri tem zamenja več zaporednih presledkov z enim samim presledkom.

Program je zelo podoben našemu prvemu programu iz tega razdelka (`prepis.c`), le da znak izpišemo samo v primeru, da to ni ponovljeni presledek. Da lahko ugotovimo večkratno pojavitev presledka, si moramo zapomniti tudi znak, ki smo ga prebrali pred trenutnim znakom. Zato smo uvedli novo spremenljivko `zadnjic`, ki hrani vrednost znaka, prebranega pred trenutnim znakom `c` (to je predzadnji prebrani znak). Vrednost

tega znaka pred prvim prebranim znakom je nedoločena, zato spremenljivko `zadnjic` inicializiramo na nič (pravzaprav bi za inicializacijo v našem primeru lahko izbrali katerikoli znak razen znaka za presledek `' '`).

Tako imamo v programu shranjena zadnja dva prebrana znaka. Če sta oba enaka presledku, potem trenutnega znaka ne smemo izpisati, saj izpisujemo vedno le en zaporedni presledek. Zato smo pred izpis postavili `if` stavek, v katerem preverjamo ta pogoj, ki bi ga lahko opisali takole:

```
if(c==' ' && zadnjic==' ')
  ;            // znaka c ne izpišemo
else
  putchar(c);  // znak c izpišemo
```

Če znaka `c` ne izpišemo, pravzaprav ne naredimo nič. Tako je stavek v prvem delu `if` stavka prazen, zato je bolje, da pogoj `if` stavka obrnemo (negiramo) in dobimo:

```
if(c!=' ' || zadnjic!=' ')
  putchar(c);
```

Tokrat smo prazni stavek pod `else` enostavno izpustili.

Na koncu moramo pred ponovitvijo zanke poskrbeti še za to, da bo imela spremenljivka `zadnjic` pravilno vrednost ob naslednjem prehodu zanke. Zato ji priredimo vrednost spremenljivke `c`, kajti po naslednjem prebranem znaku, ki se zgodi v pogoju `while` zanke, bo to predzadnji prebrani znak. Če vse povedano zapišemo skupaj, dobimo naslednji program (`presledki.c`):

```
#include <stdio.h>

main()
{
  int c, zadnjic;

  zadnjic=0;
  while ((c=getchar()) != EOF) {
    if(c!=' ' || zadnjic!=' ')
      putchar(c);
    zadnjic=c;
  }
}
```

# 4. Kazalci, naslovi, polja - osnovni pojmi

Kazalci so zelo pomemben koncept v programskem jeziku C, saj se zelo pogosto uporabljajo (posredno ali neposredno). Marsikje se jim sploh ne moremo izogniti. Zato si bomo na začetku na kratko ogledali nekaj osnovnih pojmov v zvezi z njimi.

**Kazalci in naslovi**

Kazalec je spremenljivka, ki vsebuje naslov druge spremenljivke, ki je tako dosegljiva tudi posredno preko kazalca. Kot primer vzemimo celoštevilsko spremenljivko `x` in spremenljivko `px`, ki je kazalec na celoštevilsko spremenljivko. Oboje deklariramo kot:

```
int x;
int *px;
```

Kazalec na celoštevilsko spremenljivko `px` deklariramo s pomočjo operatorja `*`, ki svoj operand `px` razume kot naslov spremenljivke. Spremenljivka `(*px)` pa je deklarirana kot celo število.

Podobno operator `&` izračuna naslov svojega operanda. Torej lahko spremenljivki `px` priredimo naslov spremenljivke `x`:

```
px = &x;
```

Potem kazalec `px` kaže na `x` in inicializacijo spremenljivke x lahko enakovredno naredimo z enim od naslednjih dveh stavkov:

```
*px = 1;
x = 1;
```

Čeprav sta spremenljivki `x` in `*px` obe deklarirani kot celoštevilski, pa je med njima ena bistvena razlika. Medtem ko deklaracija `int x;` deklarira celoštevilsko spremenljivko `x` in zanjo tudi rezervira prostor v pomnilniku, se pri deklaraciji `int *px;` deklarira kazalec na celoštevilsko spremenljivko, v pomnilniku pa se rezervira le prostor za ta kazalec (torej naslov), ne pa tudi prostor za samo spremenljivko, na katero kaže `px`. Tako lahko brez težav uporabimo zaporedje stavkov:

```
int x;
x = 1;
```

medtem ko je spodnje zaporedje stavkov nepravilno:

```
int *px;
*px = 1;  // NAPAKA! prostor za *px ni rezerviran
```

Brez težav pa lahko uporabimo:

```
int x;
int *px;
px = &x;
*px = 1;  // px kaže na x, prostor je že rezerviran
```

Kazalcu lahko priredimo le ustrezen naslov, ali pa konstanto `NULL`, kar pomeni, da ta kazalec ne kaže nikamor. Omenjena konstanta je med drugim definirana tudi v zaglavni datoteki `stdio.h`, ki jo seveda vključimo v program.

Prostor za spremenljivko, na katero kaže `px`, pa lahko rezerviramo tudi neposredno s pomočjo funkcije `malloc(n)`. Funkcija rezervira `n` bajtov pomnilnika in vrne kazalec na začetek rezerviranega bloka pomnilnika. Za določitev velikosti bloka pomnilnika, ki ga želimo rezervirati, pogosto uporabljamo funkcijo `sizeof(tip)`, ki vrne velikost podanega tipa v bajtih. Oglejmo si uporabo obeh funkcij na primeru. Če želimo rezervirati prostor za celoštevilsko spremenljivko, uporabimo naslednje stavke:

```
int *px;
px = (int*) malloc(sizeof(int));
```

Funkcija `sizeof(int)` vrne število bajtov, ki jih zaseda tip `int`, funkcija `malloc` rezervira ustrezno velikost pomnilnika za eno `int` spremenljivko in vrne `void` kazalec na ta blok pomnilnika, le-tega pa s pretvorbo tipa `(int*)` pretvorimo v kazalec na celo število ter ga priredimo spremenljivki `px`, ki je tudi deklarirana kot kazalec na celo število. Sedaj lahko brez težav naredimo tudi prireditev `*px = 1;` in pri tem ne pride do napake.

Ker smo pomnilnik za celoštevilsko spremenljivko eksplicitno rezervirali, ga moramo po koncu uporabe spremenljivke tudi eksplicitno sprostiti. Za to uporabimo funkcijo `free`, ki ji kot argument podamo kazalec na rezerviran blok pomnilnika. V našem primeru je to `free(px)`. Vse skupaj lahko zapišemo:

```
int *px;
px = (int*) malloc(sizeof(int));  // rezerviramo pomnilnik
*px = 1;                          // uporabljamo spr.
...
free(px);                         // sprostimo pomnilnik
```

Pomnilnik moramo eksplicitno sprostiti le takrat, ko smo ga tudi eksplicitno rezervirali. V primeru deklaracije celoštevilske spremenljivke `int x;` pride do rezervacije pomnilnika že samodejno ob deklaraciji in zato se tudi pomnilnik sprosti samodejno takrat, ko spremenljivka pride iz območja uporabe (na primer, ko se konča funkcija, v kateri smo deklarirali spremenljivko).

Povzemimo uporabo kazalcev na spremenljivke z naslednjim primerom (program `kazalci.c`), ki na zaslon izpiše naslov in vrednost spremenljivke na tem naslovu (pri

izpisu naslova smo uporabili decimalni izpis, kar ni povsem primerno, a zadostuje za našo ponazoritev primera):

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

main()
{
  int *px;
  px = (int*) malloc(sizeof(int));
  *px = 105;
  printf("px = %d (naslov), *px = %d \n", px, *px);
  free(px);
}
```

### Kazalci in argumenti funkcij

V programskem jeziku C se argumenti funkcije vedno prenašajo po vrednosti. Zato funkcija teh argumentov ne more spremeniti tako, da bi bile spremembe vidne navzven. V primerih, ko želimo, da se spremembe vrednosti argumentov ohranijo tudi izven klicane funkcije, moramo uporabiti prenos argumentov po referenci. Kot pove že ime, so v tem primeru argumenti le reference na ustrezne spremenljivke, torej kazalci na te spremenljivke.

Poglejmo si naslednjo funkcijo z imenom `init`, ki prejme en argument, to je naslov celoštevilske spremenljivke:

```
void init(int *n)
{
  *n = 0;
}
```

Funkcija `init` postavi vrednost spremenljivke, katere naslov je argument funkcije, na nič. Funkcijo lahko pokličemo z naslednjim stavkom:

```
int stevilo;
init(&stevilo);
```

Če imamo celoštevilsko spremenljivko (ob deklaraciji se zanjo rezervira tudi prostor) `stevilo`, lahko podamo kot argument funkcije `init` naslov te spremenljivke, to je `&stevilo`. Ob klicu funkcije se ta naslov prenese po vrednosti in spremenljivka `n` v funkciji dobi vrednost tega naslova. Ali z drugimi besedami, `n` kaže na spremenljivko `stevilo`. Stavek `*n = 0;` v telesu funkcije priredi spremenljivki, na katero kaže `n`, vrednost 0. Z drugimi besedami, ta stavek postavi vrednost spremenljivke `stevilo` na nič. Ko se funkcija zaključi, spremenljivka `n` ne obstaja več, a učinek funkcije ostane, saj ima spremenljivka `stevilo` vrednost nič.

Seveda bi lahko klic funkcije `init` naredili tudi na malce daljši način, ki je enakovreden zgornjemu:

```
int stevilo;
int *p;

p = &stevilo;
init(p);
```

Delovanje opisane funkcije si lahko pogledamo na naslednjem primeru (program `kazfun.c`):

```
#include <stdio.h>

void init(int *n)
{
  *n = 0;
}

main()
{
  int stevilo = 5;

  printf("stevilo = %d \n", stevilo);
  init(&stevilo);
  printf("stevilo = %d \n", stevilo);
}
```

Prenos argumentov po referenci uporabljamo predvsem pri funkcijah, ki morajo vračati več kot eno vrednost. Tako lahko funkcija vrne vrednosti tudi preko svojih argumentov. Primer take funkcije je `scanf`, ki je opisana v naslednjem poglavju.

### Polja in kazalci

Polje definiramo tako, da za imenom spremenljivke v oglatih oklepajih navedemo število elementov polja. Polje desetih celih števil bi tako deklarirali kot:

```
int polje[7];
```

Zgornje polje je predstavljeno kot sedem zaporednih objektov tipa `int`, njihova imena pa so `polje[0]`, `polje[1]` in tako naprej do `polje[6]`. Velikost polja je 7, indeksi elementov polja pa tečejo od 0 do 6. `polje[i]` v splošnem pomeni `i`-ti element polja (element na `i`-tem mestu, računano od začetka polja).

Ime polja je pravzaprav kazalec na to polje (ime polja je sinonim za naslov elementa z indeksom nič). Torej, če je spremenljivka `p` kazalec na celo število (ki ga deklariramo s stavkom `int *p;`), bi lahko zapisali:

```
p = &polje[0];
```

ali na krajši način in bolj preprosto:

```
p = polje;
```

Edina razlika med imenom polja in kazalcem na to polje je, da je ime polja konstanta, kazalec na polje pa spremenljivka. Tako so (ob upoštevanju zgornjih deklaracij) legalni stavki:

```
p = polje;
p++;
```

ne pa tudi stavki:

```
// polje je konstanta, ne moremo ji spremeniti vrednosti
polje = p;  // NAPAKA!
polje++;    // NAPAKA!
```

Če `p` kaže na določen element polja, potem `p+1` kaže na naslednji element tega polja, `p+i` pa `i` elementov naprej. Indeks polja torej pove odmik od začetka polja. Tako lahko `i`-ti element polja dosežemo s `polje[i]` ali pa preko kazalca `*(polje+i)`, oboje je enakovredno.

Za prehod polja ponavadi uporabimo zanke, najenostavneje kar `for` zanko. Tako bi naše polje lahko inicializirali (vsem elementom polja nastavili začetne vrednosti, recimo enake 0) z naslednjima stavkoma:

```
int i;
for(i=0; i<7; i++)
  polje[i] = 0;
```

Pri tem moramo poznati velikost polja (ki smo jo podali pri deklaraciji polja; ponavadi jo zaradi enostavnosti in preglednosti zapišemo kot konstanto), saj v programskem jeziku C ne obstaja nobena privzeta spremenljivka ali funkcija, ki bi nam vrnila velikost podanega polja.

Isti rezultat bi dosegli tudi, če polje inicializiramo ob sami deklaraciji z eksplicitno navedbo njegovih elementov med zavitima oklepajema:

```
int polje[7] = {0,0,0,0,0,0,0};
```

Kot primer deklaracije in uporabe polja napišimo enostaven program (`polje.c`), ki v polje desetih celih števil vpiše po vrsti števila od 10 do 1 in jih nato po vrsti tudi izpiše na zaslon.

```
#include <stdio.h>

#define MAX 10

main()
{
  int i;
  int polje[MAX];

  for(i=0; i<MAX; i++)
    polje[i] = MAX-i;
  printf("Izpis vrednosti polja:\n");
  for(i=0; i<MAX; i++)
    printf("%d\n", polje[i]);
}
```

Z določilom `define` smo ustvarili simbolično konstanto `MAX`, ki določa velikost našega polja. Predprocesor potem zamenja vse pojavitve `MAX` v programu z vrednostjo `10`.

**Nizi znakov**

Programski jezik C pozna znakovni tip `char`, ne pa tudi tipa `string`, to je niza znakov, kot ga srečamo v nekaterih drugih jezikih. Prav tako tudi ne pozna operatorjev, ki bi omogočali operacije nad celotnimi nizi.

V jeziku C je niz znakov (ali krajše samo niz, po angleško *string*) definiran kot polje znakov, pri katerem je konec veljavnih znakov označen z 0 (to je znak `'\0'`, katerega koda je nič). Ker so nizi navadna polja znakov, jih tudi deklariramo tako kot polja:

```
char niz[10];  // polje 10 znakov
```

Ob deklaraciji se rezervira prostor za 10 znakov.

Nize lahko zapišemo kot konstante, znake niza pišemo med dvojnimi narekovaji (`"Pozdravljeni!"`, `"12345"` ali `"To je niz znakov."`). Začetek in konec niza torej določa znak `"`. Če niz zapišemo kot konstanto, je za pravilno zaključitev niza avtomatsko poskrbljeno (prevajalnik doda na koncu niza 0).

Niz lahko inicializiramo že ob sami deklaraciji, tako da uporabimo inicializacijo polja (kot to velja za vsa polja):

```
char niz[5]={'a','b','c','d','\0'};
```

Uporabimo lahko tudi:

```
char niz[5]="abcd";
char niz[]="abcd";
```

kjer se na konec niza samodejno postavi 0. V drugem primeru prevajalnik sam določi tudi potrebno velikost polja (to je 5).

Pri podajanju vrednosti niza kot konstante veljajo naslednja pravila. Če je deklarirano polje premajhno za podano konstanto, se le-ta skrajša (preostanek se odreže). Če je deklarirano polje večje od podane konstante, ostanejo preostali znaki polja nedefinirani. Če pa velikosti polja ne določimo, bo le-ta enako velikosti konstante, vključno z zaključnim znakom nič.

```
char niz[3] = "ABC01";
// niz vsebuje 3 znake: 'A', 'B' in 'C'
// to ni veljaven niz, ker ni zaključen z 0!

char niz[6] = "ABC01";
// niz vsebuje 6 znakov: 'A', 'B', 'C', '0', '1' in '\0'

char niz[] = "ABC01";
// niz vsebuje 6 znakov: 'A', 'B', 'C', '0', '1' in '\0'
```

Ker so nizi kazalci na znake, za kopiranje nizov ne moremo uporabiti prireditvenega operatorja. Poglejmo na primeru. Če zapišemo:

```
char *a = "abcde";
char *b;
b = a;
```

smo s prireditvijo `b=a;` spremenljivki `b` sicer prirediti vrednost `"abcde"`, vendar taka prireditev ne naredi kopije niza `"abcde"`, temveč le nastavi kazalec `b` tako, da ta kaže na isti niz znakov kot kazalec `a` (priredi naslov). Torej imamo na koncu en sam niz znakov `"abcde"` in dva kazalca nanj (`a` in `b`).

Kadar želimo narediti kopijo niza (da dobimo dva različna niza z isto vsebino), zato uporabimo funkcijo `strcopy(a,b)`, ki naredi kopijo vseh znakov niza `b` (do ničle, ki zaključuje niz) ter jih shrani v `a`.

Ker je ime polja konstanta, prirejanja vrednosti nizu tudi moremo narediti na naslednji način:

```
char a[10];
a = "abcd";  // NAPAKA!
```

Do napake pride, ker je `a` konstanta, zato ji ne moremo prirediti vrednosti. Lahko pa uporabimo funkcijo za kopiranje nizov, kajti v tem primeru konstanto podamo kot argument funkciji, kar se izvede na enak način kot pri spremenljivkah, to je s kazalcem:

```
strcopy(a,"abcd");
```

Tudi pri primerjanju enakosti dveh nizov ne moremo enostavno uporabiti operatorja enakosti ==. Primerjava a==b namreč samo testira, ali kazalca a in b kažeta na isto lokacijo v pomnilniku. Zato je neuporabna za primerjavo vsebine dveh nizov, saj sta niza lahko enaka, tudi če sta shranjena na različnih lokacijah.

Za primerjavo nizov tako uporabimo funkcijo strcmp(a,b). ki primerja podana niza in vrne vrednost 0, če sta niza enaka. V primeru, da je niz a manjši od niza b (po abecedi), funkcija vrne negativno vrednost. Če pa je niz a večji od niza b, funkcija vrne pozitivno vrednost.

Zelo uporabna je tudi funkcija strlen(niz), ki vrne število znakov v nizu niz. Funkcija prešteje vse znake od začetka do prve pojavitve znaka '\0'.

Naj omenimo še funkcijo strcat(a,b), ki stakne niza a in b tako, da doda znake niza b na konec niza a.

Vse omenjene funkcije za delo z nizi najdemo v standardni knjižnici, opisane pa so v zaglavni datoteki string.h, ki jo moramo vključiti v program.

Na koncu naj opozorimo še na razliko med znakom 'a' in nizom "a". Znakovna konstanta 'a' je tipa char in zato zavzame v pomnilniku prostor za en znak, to je en bajt. Niz "a" pa je polje znakov in v pomnilniku zavzame prostor za dva znaka ('a' in '\0'), to je skupaj dva bajta.

Uporabo nizov si poglejmo še na naslednjem primeru (nizi.c), ki ilustrira uporabo opisanih funkcij. Primer je sicer malo daljši, a ne potrebuje posebnega komentarja. Kaj posamezni stavki izpišejo, smo pripisali kot komentar v samem programu.

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

main()
{
  int i;
  char s[10] = "abcde";
  char a[10] = {'\0'};  // prazen niz
  char *b;

  b = s;
  printf("s:%s:  a:%s:  b:%s:\n", s, a, b);
  // izpis: s:abcde:  a::  b:abcde:

  s[0]='x';
  printf("s:%s:  a:%s:  b:%s:\n", s, a, b);
  // izpis: s:xbcde:  a::  b:xbcde:
```

```
  strcpy(a,"nizA");
  printf("s:%s:  a:%s:  b:%s:\n", s, a, b);
  // izpis: s:xbcde:  a:nizA:  b:xbcde:

  strcpy(b,"nizB");
  printf("s:%s:  a:%s:  b:%s:\n", s, a, b);
  // izpis: s:nizB:  a:nizA:  b:nizB:

  strcpy(b,a);
  printf("s:%s:  a:%s:  b:%s:\n", s, a, b);
  // izpis: s:nizA:  a:nizA:  b:nizA:

  // pogoj je resničen, ker s in b kažeta na isti niz
  if (s == b)
    printf("s==b\n");  // izpis: s==b
  else
    printf("s!=b\n");

  // pogoj NI resničen; a in b kažeta na različna niza,
  // čeprav sta vrednosti obeh nizov enaki
  if (a == b)
    printf("a==b\n");
  else
    printf("a!=b\n");  // izpis: a!=b

  // pogoj NI resničen; s in a kažeta na različna niza,
  // čeprav sta vrednosti obeh nizov enaki
  if (s == a)
    printf("s==a\n");
  else
    printf("s!=a\n");  // izpis: s!=a

  // pogoj je resničen; niza s in a sta enaka,
  // čeprav sta na različnih lokacijah
  if ( strcmp(s,a) == 0 )
    printf("strcmp(s,a)=0\n");  // izpis: strcmp(s,a)=0
  else
    printf("strcmp(s,a)<>0\n");

  for(i=0; i<strlen(s); i++)
    s[i]='A'+i;
  printf("s:%s:  a:%s:  b:%s:\n", s, a, b);
  // izpis: s:ABCD:  a:nizA:  b:ABCD:

  strcat(s,a);
  // paziti moramo, da je za polje s rezervirano dovolj
  // prostora, da vanj shranimo nov, daljši niz!
  printf("s:%s:  a:%s:  b:%s:\n", s, a, b);
  // izpis: s:ABCDnizA:  a:nizA:  b:ABCDnizA:
}
```

**Večdimenzionalna polja in polja kazalcev**

Čeprav se bomo tu osredotočili le na dvodimenzionalna polja, lahko vse, kar zanje velja, razširimo tudi na polja poljubnih dimenzij.

Dvodimenzionalna polja so v programskem jeziku C definirana kot enodimenzionalno polje, katerega vsak element je spet polje. Zato se indeksi pišejo v obliki `polje[i][j]`. Elementi so shranjeni po vrsticah: prvi indeks pomeni vrstico, drugi pa stolpec (če elemente indeksiramo po vrsti, se drugi indeks hitreje spreminja kot prvi).

Tudi dvodimenzionalno polje lahko inicializiramo ob sami deklaraciji s seznamom začetnih vrednosti:

```
int ocene[5][2] = {{10,9}, {10,10}, {8,9}, {6,6}, {9,10}};
```

Če dvodimenzionalno polje prenašamo kot argument funkciji, potem moramo pri deklaraciji argumenta v funkciji obvezno navesti število stolpcev (drugi indeks). Število vrstic (prvi indeks) ni pomembno, saj se polje prenaša po referenci, torej s kazalcem na polje. Če polje `ocene` prenesemo v funkcijo `povprecje`, uporabimo naslednjo deklaracijo:

```
double povprecje(ocene)
int ocene[5][2];
{
...
}
```

Ker število vrstic ni pomembno, lahko argument deklariramo kot:

```
int ocene[][2];
```

ali pa z uporabo kazalca na polje:

```
int (*ocene)[2];
```

Oklepaja pri deklaraciji argumenta moramo pisati zaradi prioritete operatorjev. Če ju izpustimo, smo namreč deklarirali polje dveh kazalcev na cela števila, saj operator `[]` bolj veže kot operator `*`:

```
int *ocene[2];
```

In kakšna je razlika med dvodimenzionalnim poljem in poljem kazalcev? Recimo, da želimo matriko 10 x 10 predstaviti s poljem:

```
int a[10][10];
int *b[10];
```

Uporaba obeh je podobna, saj sta tako `a[0][1]` kot `b[0][1]` pravilna zapisa za doseg celega števila. Toda `a` je pravo polje, kar pomeni, da se zanj rezervira vseh 10x10 = 100 celoštevilčnih podatkovnih celic, pri indeksiranju pa se izvrši običajen indeksni izračun. Pri deklaraciji polja `b` pa se rezervira le 10 podatkovnih celic za kazalce na cela števila. Vsak izmed njih sicer lahko kaže na zaporedje 10 celih števil, a moramo sami vsakemu kazalcu prirediti naslov tega zaporedja. Če vsako zaporedje zasede 10 celoštevilskih podatkovnih celic, to pri 10 zaporedjih znese skupaj 10x10 = 100 celic. Tako polje `b` zasede 100 celoštevilčnih podatkovnih celic in še 10 podatkovnih celic za kazalce. Polje `b` torej zasede več prostora kot polje `a`.

Zato pa je prednost polja `b`, da omogoča realizacijo matrike z različno dolgimi vrsticami. Ker so elementi polja `b` kazalci, lahko ti kažejo na poljubno zaporedje elementov, tudi na prazno zaporedje.

Najpogosteje v C-ju uporabljamo polja kazalcev na znake, ker omogočajo shranitev različno dolgih nizov v eno polje (tipičen primer je polje, ki hrani vrednosti argumentov ukazne vrstice, kot je to opisano v 6. poglavju).

# 5. Formatiran izpis in branje

Programi morajo pogosto komunicirati z uporabnikom, najenostavneje preko standardnega vhoda (tipkovnica) in standardnega izhoda (zaslon). Tako izpisi ponavadi potekajo na standardni izhod, branje podatkov pa s standardnega vhoda. Za delo z njima sta zelo uporabna formatiran izpis in formatirano branje, ki si ju bomo ogledali v nadaljevanju.

### Formatiran izpis - funkcija `printf`

Funkcija `printf` prejme enega ali več argumentov. Prvi argument funkcije je format, to je niz znakov, ki opisuje obliko izhoda. Znak `%` v formatu in znaki, ki mu sledijo, določajo, kako se pretvori naslednji argument funkcije. Uporabo funkcije `printf` si najlažje pogledamo na primerih.

Za izpis niza znakov uporabimo formatni znak `s`. Niz znakov, ki ga izpisujemo, mora biti zaključen z nič (znakom `'\0'`). V formatnem določilu so med znakoma `%` in `s` lahko tudi drugi znaki: znak minus (`-`) določa levo poravnavo, sledi širina (število znakov) izpisa, pika (`.`) in največje število izpisanih znakov niza. Za primer si poglejmo naslednje stavke, pri katerih je v komentarju napisan tudi izpis:

```
char niz[] = "Formatiran izpis";

printf(":%s:\n", niz);         //  :Formatiran izpis:
printf(":%10s:\n", niz);       //  :Formatiran izpis:
printf(":%-10s:\n", niz);      //  :Formatiran izpis:
printf(":%20s:\n", niz);       //  :    Formatiran izpis:
printf(":%-20s:\n", niz);      //  :Formatiran izpis    :
printf(":%20.10s:\n", niz);    //  :          Formatiran:
printf(":%-20.10s:\n", niz);   //  :Formatiran          :
printf(":%.10s:\n", niz);      //  :Formatiran:
```

Znak `\n` na koncu formata je znak za novo vrstico (*new line*) in pomeni, da po izpisu skoči kurzor v novo vrstico. Tako dosežemo, da se vsak stavek za izpis izpisuje podatke v svojo vrstico.

Za izpis enega samega znaka uporabimo formatni znak `c`. Izpis znaka `%` pa dosežemo z navedbo dvojnega znaka.

```
char znak = 'A';

printf("Znak je: %c\n", znak);   // Znak je: A
printf("Delez je 5%%.\n");       // Delez je 5%.
```

Za izpis števil uporabimo formatni znak `d` za cela števila v desetiški obliki ter znake `e`, `f` ali `g` za realna števila. Tudi tu določa znak minus (`-`) levo poravnavo, sledi širina

(število znakov) izpisa, pika (`.`) in število izpisanih decimalnih mest pri realnih številih (privzeto je 6). Nekaj primerov s pripadajočimi izpisi v komentarju:

```
int st = 12345;
double pi = 3.1415926;

printf("Stevilo <%d>\n", st);  // Stevilo <12345>
// f privzeto izpiše 6 decimalk, število ustrezno zaokroži
printf("PI = %f\n", pi);       // PI = 3.141593
printf("PI = %e\n", pi);       // PI = 3.141593e+00
// g privzeto izpiše 6 pomembnih števk
printf("PI = %g\n", pi);       // PI = 3.14159
printf("PI = %.10f\n", pi);    // PI = 3.1415926000
printf("PI = %.2f\n", pi);     // PI = 3.14
printf(":%-10f:\n", pi);       // :3.141593  :
printf(":%-10.2f:\n", pi);     // :3.14      :
printf(":%10f:\n", pi);        // :  3.141593:
printf(":%10.2f:\n", pi);      // :      3.14:
printf("2 + 2 = %d\n", 2+2);   // 2 + 2 = 4
```

Kadar želimo izpisati posebne znake, kot so na primer znak za novo vrstico ali tabulator, uporabimo ubežne sekvence. Te določa zaporedje dveh znakov (prvi je vedno povratna poševnica ali *backslash* `\`), ki skupaj predstavljata en sam znak. Tako na primer znak za novo vrstico, zaporedje `\n`, predstavlja en sam znak in lahko pišemo na primer:

```
char znak = '\n';
```

Med drugim C pozna sekvence za odmik ali tabulator (*tab*) `\t`, pomik nazaj (*backspace*) `\b`, dvojni narekovaj `\"` (samo znak `"` pomeni začetek oziroma konec niza) in podobno.

Vsi navedeni primeri se nahajajo v programu `formatizpis.c`.

### Formatirano branje - funkcija `scanf`

Funkcija `scanf` je analogna funkciji `printf`, le da pretvarja znake, ki jih prebere iz vhoda, in rezultate shranjuje v ustrezne naslednje argumente. Vsi argumenti za formatom morajo biti kazalci na objekte, v katere se shranjujejo rezultati.

Pri branju vhoda se presledki, odmiki in znaki za novo vrstico ignorirajo. Drugi znaki v formatu, ki niso znak `%` ali formatni znaki, se morajo popolnoma ujemati s prebranimi znaki. Formatni znaki imajo podoben pomen kot pri formatiranju izpisa: `d` pomeni celo število v desetiški obliki (argument je kazalec na celo število), `c` pomeni znak, `s` niz znakov (kateremu se na koncu avtomatično doda znak `'\0'`), `e`, `f` ali `g` pa realno število.

Delovanje funkcije si spet poglejmo na primeru. Napišimo naslednji program, ki zahteva branje treh podatkov (program `formatbranje.c`): niza znakov, celega števila in realnega števila. Argumenti funkcije, v katere se vpisujejo prebrani podatki, morajo biti vedno naslovi spremenljivk (po potrebi uporabimo operator `&`). Tako smo v spodnjem primeru uporabili naslov spremenljivke `n` (`&n`), naslov spremenljivke `stevilo` (`&stevilo`), spremenljivka `niz` pa je že po definiciji naslov prvega elementa polja (element z indeksom 0).

```
#include <stdio.h>

main()
{
  int n = 0;
  float stevilo = 0.0;
  char niz[20];

  scanf("%s in %d %f", niz, &n, &stevilo);
  printf("Prebrali smo ");
  printf("celo stevilo %d, ", n);
  printf("niz znakov \"%s\" in ", niz);
  printf("realno stevilo %g.\n", stevilo);
}
```

Če v vhodni vrstici vpišemo naslednje podatke:

```
Format in 5    3.14
```

se spremenljivki `niz` priredi vrednost "`Format`" (brez narekovajev, niz zaključen z 0), spremenljivka `n` dobi vrednost `5`, spremenljivki `stevilo` pa se priredi vrednost `3.14`. Presledki se ignorirajo, zaporedje znakov `in` pa se mora pojaviti v točno taki obliki. Prebrano lahko preverimo z izpisom vrednosti spremenljivk na koncu programa.

Funkcija `scanf` preneha z branjem, ko prebere potrebno število podatkov ali ko vhodni podatek ne ustreza podanemu formatu. Funkcija vrne število pravilno prebranih podatkov. Če pa je pri branju prišlo do konca datoteke, vrne `EOF` (konstanta, definirana v zaglavni datoteki `stdio.h`).

### Še en primer

Za konec pa si poglejmo še en primer uporabe branja podatkov in izpisa rezultata. Program naj prebere dve celi števili in izpiše njuno vsoto (`sestej.c`). Čeprav je program zelo enostaven, predstavlja uporabo obeh v tem poglavju opisanih funkcij v praksi.

```
#include <stdio.h>

main()
{
  int prvo, drugo;

  printf("Vpisite prvo stevilo:\n");
  scanf("%d",&prvo);
  printf("Vpisite drugo stevilo:\n");
  scanf("%d",&drugo);
  printf("%d+%d=%d\n", prvo, drugo, prvo+drugo);
}
```

Kot vidimo v zadnjem primeru, so lahko argumenti funkcije `printf` tudi izrazi.

# 6. Argumenti ukazne vrstice

Argumente iz ukazne vrstice operacijskega sistema lahko prenesemo tudi v program v C-ju. Pri klicu funkcije `main`, ki je prva klicana funkcija v programu, ta dobi dva parametra. Prvi vsebuje število vseh argumentov, ki jih je funkcija dobila iz okolja, in ga ponavadi označujemo z `argc`. Drugi pa je kazalec na polje znakovnih nizov, ki vsebujejo te argumente, po enega v vsakem nizu. Ponavadi ga označujemo z `argv`. Ustrezna deklaracija `main` funkcije izgleda takole:

```
main(argc,argv)
int argc;
char *argv[];
{
  ...
}
```

Podani argumenti programa (skupaj z imenom programa) se shranijo v polje nizov `argv`. Prvi element polja je vedno ime programa, vsebuje ga niz `argv[0]`. Iz tega sledi, da je vrednost `argc` vedno večja ali enaka 1. Nizi `argv[1]` do `argv[argc-1]` pa hranijo podane argumente programa.

Oglejmo si to na primeru. Spodnji program (`argumenti.c`) na zaslon izpiše svoje ime in vse argumente, ki jih prejme ob zagonu, vsakega v novo vrstico.

```
#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
  int i;

  printf("ime programa:\n  %s\n", argv[0]);
  printf("argumenti:\n");
  for(i=1; i<argc; i++)
    printf("  %s\n", argv[i]);
}
```

Če program prevedemo v izvršljivo datoteko `argumenti`:

```
gcc -o argumenti argumenti.c
```

in ga nato poženemo z ukazom:

```
./argumenti prvi drugi tretji
```

dobimo naslednji izpis:

```
ime programa:
  ./argumenti
argumenti:
  prvi
  drugi
  tretji
```

Program lahko zapišemo tudi krajše (`argumenti1.c`):

```
#include <stdio.h>

main(int argc, char *argv[])
{
  printf("ime programa:\n  %s\nargumenti:\n", argv[0]);
  while(--argc > 0)
    printf("  %s\n", *++argv);
}
```

Uporabili smo zanko `while`, ki se ponavlja, dokler je vrednost `argc` večja od nič. Pri vsakem prehodu zanke se vrednost `argc` zmanjša za ena (en argument smo že izpisali). Spremenljivka `argv` je kazalec na začetek polja argumentov. Če `argv` povečamo za ena (`++argv`), postavimo kazalec na prvi argument, to je na `argv[1]`. Vsako naslednje inkrementiranje (`++argv`) pomika ta kazalec po argumentih naprej. Zapis `*++argv` pomeni kazalec na niz. Bodite pozorni tudi na prefiksno uporabo operatorjev inkrement (`++`) in dekrement (`--`)!

## Števila kot argumenti programa

Kot smo videli v prejšnjem primeru, se argumenti programa interpretirajo kot nizi znakov. Včasih pa želimo kot argument podati tudi število, bodisi celo ali realno. V tem primeru moramo niz znakov pretvoriti v število. Na srečo nam tega ni treba narediti "na roke", ampak za to poskrbi že pripravljena funkcija. Niz znakov spremenimo v celo število s funkcijo `atoi`, za spreminjanje v realno število pa uporabimo funkcijo `atof`. Poglejmo si primer uporabe opisanih funkcij:

```
int celo;
double realno;

celo = atoi("123");
realno = atof("12.345");
```

Obe funkciji sta iz standardne knjižnice, opisani pa sta v zaglavni datoteki `stdlib.h`, ki jo moramo vključiti v program.

Napišimo še program `plus.c`, ki sešteje vse podane argumente in izpiše rezultat. Argumenti naj bodo bolj splošno podani kot realna števila (ki seveda zajemajo tudi cela

števila). Z `while` zanko pregledamo vse argumente, vsakega posebej pretvorimo iz niza v realno število in to število prištejemo vsoti.

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

main(int argc, char *argv[])
{
  double vsota = 0;

  while(--argc > 0)
    vsota += atof(*++argv);
  printf("%g\n", vsota);
}
```

### Opcije kot argumenti programa

Pri klicu programov pogosto uporabljamo opcije, ki spremenijo delovanje programa. Opcijske argumente ponavadi začenjamo z znakom minus (-) in jih pišemo v poljubnem vrstnem redu, lahko pa jih tudi združujemo. Tipičen primer uporabe opcij je ukaz `ls`, ki privzeto izpiše tekoči direktorij. Kadar želimo izpisati vse datoteke v direktoriju (tudi skrite), uporabimo opcijo `-a`. Z opcijo `-l` zahtevamo dolg izpis, ki poleg imena datoteke prikaže tudi ostale podatke. Seveda lahko obe opciji tudi združimo. Uporabimo lahko katerokoli od variant:

```
ls
ls -a
ls -l
ls -a -l
ls -l -a
ls -al
ls -la
```

Zelo uporabno je, da tudi programi, ki jih napišemo sami, po potrebi omogočajo podobno uporabo opcij. V nadaljevanju si bomo pogledali, kako lahko podane opcije razberemo iz argumentov programa.

Napišimo program, ki pregleda podane argumente ukazne vrstice in izpiše, katere od opcij smo uporabili, oziroma nas opozori, da smo podali neveljavno opcijo. Možne opcije naj bodo `a`, `b` in `c`. Opcije lahko tudi združujemo (pišemo skupaj).

Recimo, da se naš preveden program imenuje `opcije`. Navedimo nekaj primerov klica programa s samimi veljavnimi opcijami:

```
opcije -a -b -c
opcije -abc y
opcije -c x -ba
opcije -c -b
```

```
opcije -b
opcije a -b -c
```

Kot vidimo, ima lahko program tudi druge argumente, ki nas trenutno ne zanimajo, kajti gledamo le opcijske argumente (tiste, ki se začenjajo z `-`). Klici programa s podanimi neveljavnimi opcijami pa bi bili naslednji:

```
opcije -a -b -d
opcije -a -ce
opcije -e a
```

V prvem primeru je nepoznana opcija `d` (`-d`), v drugem in tretjem pa `e` (`-ce` in `-e`).

Pa se vrnimo h kodi programa. Kot smo rekli, niz `argv[0]` vsebuje ime programa, niz `argv[1]` pa prvi argument. Potem je prvi znak prvega argumenta `argv[1][0]`. Ta znak mora biti enak znaku minus `'-'`, če gre za opcijski argument. Temu znaku pa pri regularnih opcijah sledi `a` ali `b` ali `c` ali katerakoli kombinacija teh treh znakov. Če znaku minus sledi karkoli drugega, je opcija neveljavna.

Program mora pregledati vse podane argumente ter vsakega najprej testirati, ali se začenja z znakom minus. Za prehod preko vseh argumentov uporabimo kar `for` zanko. Zanka teče od 1 (ker je 1 indeks v polju `argv` prvega pravega argumenta programa; indeks 0 ima ime programa) pa do števila vseh argumentov:

```
for(i=1; i<argc; i++)
```

Testiranje na znak minus pa opravimo v `if` stavku, kjer prvi znak niza z `i`-tim argumentom `argv[i]` primerjamo z znakom `'-'`:

```
if(argv[i][0]=='-')
```

Če je pogoj izpolnjen, pregledamo še vse ostale znake v nizu `argv[i]`, za kar spet uporabimo `for` zanko. Tokrat zanka teče od 1 (prvi znak niza `argv[i][0]` smo že pogledali in je enak `'-'`) pa do konca niza `argv[i]`:

```
for(j=1; j<strlen(argv[i]); j++)
```

Za ugotavljanje konca niza smo uporabili funkcijo `strlen`, ki vrne dolžino podanega niza. In kaj naredimo znotraj notranje `for` zanke? Za vsak znak iz niza `argv[i][j]` moramo pogledati, ali ustreza regularnim opcijam, torej ali je enak znakom `'a'` ali `'b'` ali `'c'`. Če ni niti `a` niti `b` niti `c`, je opcija neveljavna. Ker testiramo znake in imamo tri oziroma štiri različne možnosti, je naša izbira `switch` stavek:

```
switch(argv[i][j]) {
  case 'a': ...  // opcija a
            break;
  case 'b': ...  // opcija b
            break;
  case 'c': ...  // opcija c
            break;
  default:  ...  // nepoznana opcija
}
```

V našem programu se zadovoljimo s tem, da izpišemo vsako od ugotovljenih opcij. Če vse povedano sestavimo skupaj, dobimo spodnji program (`opcije.c`):

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

main(int argc, char *argv[])
{
  int i,j;

  for(i=1; i<argc; i++)
    if(argv[i][0]=='-')
      for(j=1; j<strlen(argv[i]); j++)
        switch(argv[i][j]) {
          case 'a': printf("opcija a\n");
                    break;
          case 'b': printf("opcija b\n");
                    break;
          case 'c': printf("opcija c\n");
                    break;
          default:  printf("nepoznana opcija\n");
        }
}
```

Za konec pa poskusimo naš zadnji program zapisati še nekoliko drugače z uporabo kazalcev na argumente. Če je `argv` kazalec na začetek polja argumentov in drugi element polja, to prvi podani argument programu, dosežemo preko `(*++argv)`, potem je prvi znak prvega argumenta, ki mora biti pri opcijah enak znaku minus `'-'`, dosegljiv preko `(*++argv)[0]`. Temu znaku pa pri regularnih opcijah sledi `a` ali `b` ali `c`.

Preko vseh argumentov se tokrat sprehodimo z `while` zanko, v kateri zmanjšujemo vrednost `argc`:

```
while(--argc > 0)
```

Za preverjanje, ali je prvi znak posameznega argumenta enak minusu, tudi tu uporabimo `if` stavek. Hkrati ob testiranju pogoja tudi povečamo `argv` za ena, tako da kaže na ustrezen naslednji argument:

```
if( (*++argv)[0]=='-' )
```

v primeru, da je pogoj izpolnjen, moramo pregledati še vse preostale znake v argumentu. Zato nastavimo kazalec `s`, ki je deklariran kot znakovni kazalec, na drugi znak argumenta ( `argv[0]+1`). V zanki ta kazalec povečujemo, dokler ne pridemo do znaka `'\0'` (argumenti programa so nizi in so zato zaključeni z 0):

```
for(s=argv[0]+1; *s!='\0'; s++)
```

Stavek `switch`, ki sestavlja telo `for` zanke, ostaja enak, le da gledamo znak, na katerega kaže `s`.

Celoten program (`opcije1.c`) je naslednji:

```
#include <stdio.h>

main(int argc, char *argv[])
{
  char *s;

  while(--argc > 0)
    if((*++argv)[0]=='-')
      for(s=argv[0]+1; *s!='\0'; s++)
        switch(*s) {
          case 'a': printf("opcija a\n");
                    break;
          case 'b': printf("opcija b\n");
                    break;
          case 'c': printf("opcija c\n");
                    break;
          default:  printf("nepoznana opcija\n");
        }
}
```

# 7. Delo z datotekami

V tem poglavju si bomo pogledali, kako v programskem jeziku C uporabljamo datoteke, kako jih pripravimo za uporabo, iz njih beremo ali vanje pišemo. Vhodno/izhodne funkcije sicer niso del jezika, a jih pogosto uporabljamo, zato ne moremo mimo njih.

Datoteke v C-ju uporabljamo preko datotečnega kazalca, to je kazalca na strukturo `FILE`, v kateri so shranjeni podatki o datoteki. Spremenljivko `fp`, ki je kazalec na datoteko, deklariramo kot:

```
FILE *fp;
```

Struktura `FILE` in vse funkcije za delo z datotekami, ki si jih bomo ogledali v nadaljevanju, so definirane v standardni knjižnici oziroma v zaglavni datoteki `stdio.h`, katero moramo vključiti v program:

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

Preden datoteko lahko uporabimo (za branje ali pisanje), jo moramo odpreti. To naredimo s klicem sistemske funkcije `fopen`, ki vrne datotečni kazalec:

```
fp = fopen(ime, način);
```

Funkcija `fopen` prejme dva argumenta. Prvi (`ime`) je zunanje ime datoteke, podano kot znakovni niz. Drugi argument (`način`) pa je niz, ki določa operacijo, za katero datoteko odpiramo: `"r"` pomeni branje, `"w"` pisanje ter `"a"` dodajanje na konec datoteke. Če datoteko odpremo za pisanje ali dodajanje in datoteka še ne obstaja, funkcija ustvari novo datoteko. Če datoteka obstaja in jo odpremo za pisanje, prepišemo njeno vsebino (izgubimo njeno prejšnjo vsebino). Če pri odpiranju datoteke pride do napake, funkcija `fopen` vrne `NULL`. Do napake lahko pride, če datoteka, ki jo odpiramo za branje, ne obstaja, če nimamo ustreznih pravic za dostop do datoteke in podobno.

Vsako datoteko, ki smo jo odprli, moramo po uporabi tudi zapreti. Za zapiranje datoteke uporabimo sistemsko funkcijo `fclose`:

```
fclose(fp);
```

Branje datoteke oziroma pisanje v datoteko lahko izvedemo na več načinov. Najpreprostejši način je po znakih (znak za znakom), ki je tudi najpogosteje uporabljan. Lahko pa datoteke beremo/pišemo tudi po vrsticah (cela vrstica naenkrat) ali pa uporabimo formatirano branje ali izpis. Pa si oglejmo vse tri načine po vrsti.

Za branje datoteke po znakih uporabimo funkcijo `fgetc`, ki prebere naslednji znak iz datoteke `fp`. Ob koncu datoteke ali če pri branju pride do napake, vrne `EOF`. Funkcija vrača vrednost `int`. Za zapis znaka `c` v datoteko `fp` pa uporabimo funkcijo `fputc`:

```
c = fgetc(fp);
fputc(c,fp);
```

Namesto obeh funkcij lahko enakovredno uporabimo tudi makroja, ki sta definirana v zaglavni datoteki `stdio.h`:

```
c = getc(fp);
putc(c, fp);
```

Če želimo iz datoteke `fp` prebrati celo vrstico naenkrat, za branje uporabimo funkcijo `fgets`. Prebrana vrstica se shrani v polje `vrstica`, zadnji znak v polju pa dobi vrednost 0. Funkcija prebere največ `max-1` znakov in vrne kazalec na polje `vrstica` oziroma `NULL` ob napaki ali koncu datoteke. Za zapis niza `vrstica` v datoteko `fp` pa pokličemo funkcijo `fputs`:

```
fgets(vrstica, max, fp);
fputs(vrstica, fp);
```

Za formatirano branje oziroma izpis uporabimo funkciji `fscanf` in `fprintf`, ki sta sorodni funkcijama `scanf` in `printf` iz 5. poglavja. Edina razlika je dodatni argument `fp`, ki določa datoteko, iz katere beremo oziroma vanjo zapisujemo:

```
fscanf(fp, format);
fprintf(fp, format);
```

Funkcija `fscanf`, enako kakor tudi funkcija `scanf`, vrne število uspešno prebranih in prirejenih vhodnih postavk, oziroma `EOF` ob neuspelem branju.

Pri rokovanju z datotekami nam pogosto pride prav tudi funkcija `feof(fp)`, ki preveri, ali je datoteke `fp` že konec. Če smo v datoteki že prišli do konca, funkcija vrne neničelno vrednost (resnično), sicer pa 0 (neresnično).

## Standardne datoteke

Pri zagonu vsakega programa se vedno samodejno odprejo tri datoteke in določijo kazalci nanje. To se standardni vhod, standardni izhod in standardni izhod za napake. Pripadajoči datotečni kazalci so `stdin`, `stdout` in `stderr`. Ponavadi so povezane s tipkovnico (standardni vhod) in zaslonom (standardni izhod in standardni izhod za napake).

V 3. poglavju, ko smo prvič uporabili funkciji `getchar` in `putchar`, smo omenili, da sta obe definirani kot makro v zaglavni datoteki `stdio.h`. Obe definiciji lahko sedaj razumemo brez težav:

```
#define getchar()   getc(stdin)
#define putchar(c)  putc((c), stdout)
```

**Nekaj primerov: branje in izpis po znakih**

Delo z datotekami si poglejmo še v praksi. Najprej na enostavnem primeru, kjer program na standardni izhod (zaslon) izpiše vsebino datoteke. Ker že poznamo tudi uporabo argumentov programa, bomo datoteko podali kot argument ukazne vrstice.

Spremenljivka `argv` je kazalec na polje nizov, kamor se shranijo argumenti programa. Naš program pričakuje najmanj en argument, to je ime datoteke, ki jo želimo izpisati. Zato mora biti vrednost `argc` večja ali enaka 2. Potem je `argv[1]` drugi element polja, ki hrani prvi programu podani argument, to je ime datoteke za izpis (v `argv[0]` je shranjeno ime programa). Datoteko odpremo samo za branje s stavkom:

```
fp=fopen(argv[1],"r");
```

V primeru, da je pri odpiranju datoteke prišlo do napake, dobi spremenljivka `fp` vrednost `NULL`. Seveda v primeru napake ne želimo (ne moremo) nadaljevati z izpisom datoteke, zato program predčasno zaključimo s stavkom `exit(1)`:

```
if(fp == NULL)
  exit(1);
```

Argument funkcije `exit` je vrednost, ki jo program vrne operacijskemu sistemu. Po dogovoru je uspeh označen z 0, neuspeh pa z vrednostjo, različno od nič. Upoštevajoč to navado, vrnemo ob napaki vrednost 1.

Seveda lahko prireditveni stavek uporabimo kot del izraza in oboje skupaj zapišemo nekoliko krajše:

```
if( (fp=fopen(argv[1],"r")) == NULL )
  exit(1);
```

Preostanek programa je preprost. V zanki beremo znake in jih izpisujemo na standardni izhod, dokler ne pridemo do konca datoteke (do `EOF`). Ko zaključimo z izpisom datoteke, jo moramo zapreti. Če se program uspešno zaključi, to sporočimo z vrnjeno vrednostjo 0. Celoten program izgleda takole (`izpisi.c`)

```
#include <stdio.h>

main(int argc, char *argv[])
{
  FILE *fp;
  int c;
```

```
  if(argc < 2)
    exit(2);
  if( (fp=fopen(argv[1],"r")) == NULL )
    exit(1);
  while( (c=fgetc(fp)) != EOF )
    fputc(c, stdout);
  fclose(fp);
  exit(0);
}
```

Seveda bi v programu namesto stavka `fputc(c,stdout)` lahko pisali tudi `putchar(c)` ali pa uporabili makro `putc(c,stdout)` (oziroma makro `getc(fp)` za branje).

Program ni najbolj prijazen do uporabnika, saj se ob napaki zaključi, ne da bi uporabnika obvestil, da je do napake prišlo. Tako bi bilo bolje, da bi ob napaki pred izhodom iz programa izpisali tudi vrsto napake.

Sedaj pa napisan program dopolnimo tako, da program znakov ne bo izpisoval na standardni izhod, ampak v drugo datoteko. Tudi ime slednje naj bo podano kot argument ukazne vrstice.

V tem primeru moramo programu podati najmanj dva argumenta, zato mora biti `argc` večje ali enako tri. Prvi podani argument je še vedno ime datoteke, ki jo želimo prepisati. Drugi argument pa določa datoteko, v katero želimo prepisati prvo datoteko. Drugo datoteko odpremo za pisanje (`"w"`, vsebina obstoječe datoteke se prepiše), kazalec nanjo pa shranimo v spremenljivko `fp2`:

```
fp2=fopen(argv[2],"w");
```

Če odpiranje druge datoteke ne uspe, zaključimo s programom, še pred tem pa moramo zapreti prvo datoteko, ki smo jo že uspešno odprli. Zanka v programu ostaja enaka, le da tokrat namesto na standardni izhod znake izpisujemo v datoteko `fp2`:

```
fputc(c, fp2);
```

Seveda moramo po uporabi tudi drugo datoteko zapreti s stavkom:

```
fclose(fp2);
```

Rezultat je program (`kopiraj.c`), ki prekopira prvo datoteko v drugo.

```
#include <stdio.h>

main(int argc, char *argv[])
{
  FILE *fp1, *fp2;
  int c;

  if(argc < 3)
    exit(2);
  if( (fp1=fopen(argv[1],"r")) == NULL )
    exit(1);
  if( (fp2=fopen(argv[2],"w")) == NULL ) {
    fclose(fp1);  // fp1 je bila uspešno odprta
    exit(1);
  }
  while( (c=fgetc(fp1)) != EOF )
    fputc(c, fp2);
  fclose(fp1);
  fclose(fp2);
  exit(0);
}
```

### Še en primer: branje in izpis po vrsticah

Poglejmo si še enkrat primer programa, ki na standardni izhod (zaslon) izpiše vsebino tekstovne datoteke. Vendar tokrat datoteko beremo in izpisujemo po vrsticah, torej celo vrstico naenkrat. Za hranjenje ene vrstice bomo uporabili polje znakov, zato moramo največjo dolžino vrstice predvideti vnaprej. Tako predpostavimo, da dolžina vrstice v datoteki ne presega 100 znakov (konstanta `MAX`). Prebrano vrstico shranimo v polje `vrstica`, ki je deklarirano kot:

```
char vrstica[MAX];
```

Zgornja deklaracija poskrbi tudi za to, da se za to spremenljivko v pomnilniku rezervira prostor za `MAX` znakov.

Za branje ene vrstice tako uporabimo funkcijo `fgets`, ki prebrano vrstico shrani v polje `vrstica` in zadnji znak v polju nastavi na 0:

```
fgets(vrstica, MAX, fp);
```

Funkcija prebere največ `MAX-1` znakov, tako da je ob upoštevanju zaključnega znaka velikost niza `vrstica` največ `MAX` znakov, kar ustreza rezerviranemu prostoru zanjo ob deklaraciji.

Če je pri branju prišlo do napake ali do konca datoteke, funkcija `fgets` vrne `NULL`. To je tudi naš pogoj za izstop iz zanke. Pogoj `while` zanke lahko torej zastavimo takole:

```
while( fgets(vrstica, MAX, fp) != NULL )
```

Če je ta pogoj izpolnjen (kar pomeni, da smo vrstico uspešno prebrali), se izvrši telo zanke, kjer prebrano vrstico preprosto izpišemo na standardni izhod:

```
fputs(vrstica, stdout);
```

Če naredimo program malo prijaznejši do uporabnika in dodamo izpise obvestil o napakah ob predčasnih izhodih iz programa (pred klicem funkcije `exit`), dobimo naslednji program (`izpisi1.c`):

```
#include <stdio.h>

#define MAX 100

main(int argc, char *argv[])
{
  FILE *fp;
  char vrstica[MAX];

  if(argc < 2) {
    printf("Uporaba: %s ime_datoteke\n", argv[0]);
    exit(2);
  }
  if( (fp=fopen(argv[1],"r")) == NULL ) {
    printf("Napaka: odpiranje %s za branje\n", argv[1]);
    exit(1);
  }
  while( fgets(vrstica, MAX, fp) != NULL )
    fputs(vrstica, stdout);
  fclose(fp);
  exit(0);
}
```

**In za konec še: formatirano branje**

Za konec si poglejmo še program, ki iz datoteke bere realna števila, jih sproti sešteva in na koncu izpiše njihovo vsoto.

Realna števila so v datoteki zapisana na poljuben način: z decimalno piko ali brez, na eksponentni način, s predznakom ali brez, torej v vseh možnih oblikah. Števila so med seboj ločena s presledki ali pa z novo vrstico. Primer je datoteka `vsota.txt`.

Če bi to datoteko brali znak po znak (ali pa po vrsticah), bi imeli veliko dela, da bi znake združili v pravilno realno število. Na srečo se nam problema ni potrebno lotiti na ta način, ampak lahko uporabimo formatirano branje iz datoteke, ki samodejno poskrbi za to, da preberemo vse znake, ki sestavljajo eno realno število. Tako bomo za branje

enega števila uporabili funkcijo `fscanf`, kjer bomo v formatu podali, da želimo prebrati eno realno število (`%g`):

```
fscanf(fp, "%g", &stevilo);
```

Prebrano število se shrani v spremenljivko `stevilo`, ki je deklarirana kot realno število:

```
float stevilo;
```

Števila beremo v zanki `while` toliko časa, dokler lahko uspešno preberemo naslednje realno število iz datoteke (funkcija `fscanf` vrne 1). V telesu zanke potem vsako prebrano število prištejemo vsoti. V telo zanke smo dodali tudi izpis vsakega prebranega števila; tako imate vpogled tudi v branje posameznih števili.

Ko ne moremo več prebrati nobenega realnega števila, torej ko smo prišli do konca datoteke (funkcija vrne `EOF`) ali pa prebrano ne ustreza realnemu številu (funkcija vrne 0), se zanka zaključi in izpišemo vsoto vseh prebranih števil.

Program (`vsota.c`) je naslednji:

```
#include <stdio.h>

main(int argc, char *argv[])
{
  FILE *fp;
  float stevilo;
  double vsota = 0;

  if(argc < 2) {
    printf("Uporaba: %s ime_datoteke\n", argv[0]);
    exit(2);
  }
  if( (fp=fopen(argv[1],"r")) == NULL ) {
    printf("Napaka: odpiranje %s za branje\n", argv[1]);
    exit(1);
  }
  while( fscanf(fp, "%g", &stevilo) > 0 ) {
    printf("prebrano stevilo: %g\n", stevilo);
    vsota += stevilo;
  }
  fclose(fp);
  printf("Vsota stevil iz %s je %g\n", argv[1], vsota);
  exit(0);
}
```

Pri zagonu programa ne pozabite podati datoteke s števili:

```
./vsota vsota.txt
```

# 8. Rekurzija

Programski jezik C dopušča tudi rekurzivno klicanje funkcij, kar pomeni, da lahko funkcija neposredno ali posredno kliče sama sebe.

Rekurzivno reševanje problema poteka tako, da se kompleksen računski problem razdeli na več enostavnejših problemov. Rekurzija pomeni, da se pri reševanju problema ponovijo isti izračuni.

Rekurzivno reševanje problema tipično vključuje:

- opis rešitve problema za enostaven primer (trivialen primer) in
- opis rešitve problema z uporabo dela podatkov in rešitev istega problema nad preostalimi podatki.

V nadaljevanju si bomo pogledali nekaj primerov rekurzivnih rešitev problemov, primerjali rekurzivno in iterativno rešitev istega problema ter spregovorili o primernosti (oziroma neprimernosti) uporabe rekurzivnih funkcij.

### Rekurzivno reševanje problema

Začnimo z enim najbolj razširjenih primerov, to je z izračunom fakultete naravnega števila `n` (kar zapišemo kot `n!`), ki je definirana na naslednji način:

```
n! = n * (n-1) * (n-2) * ... * 2 * 1
```

Pri tem velja, da je

```
0! = 1
```

Iterativno rešitev sestavimo brez težav: v zanki zmnožimo števila od 1 do `n` (vključno). Primer za `n=0` bi sicer lahko obravnavali ločeno, a ni potrebe: v tem primeru se zanka `for` ne izvrši niti enkrat in funkcija vrne pravilno vrednost 1. Funkcija za izračun fakultete je naslednja (program `fak-iter.c`):

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

Rešitev tega problema pa si lahko zamislimo tudi rekurzivno in ga rešimo z uporabo rekurzije. Z nekaj matematike zapišemo formulo za izračun `n!` malo drugače kot:

```
n! = n * (n-1)!
```

Do rezultata smo prišli z upoštevanjem naslednjih dveh enačb:

```
n! = n * (n-1) * (n-2) * ... * 2 * 1

(n-1)! = (n-1) * (n-2) * ... * 2 * 1
```

Sedaj lahko problem opišemo na naslednji način: `n!` izračunamo tako, da izračunamo `(n-1)!` in rezultat pomnožimo z `n`. Pri tem še vedno velja, da je `0!` enako 1, torej je rezultat v primeru `n=0` enak 1.

Postopek izračuna `n!` lahko z drugimi besedami opišemo tudi takole: če je `n` enak 0, je rezultat kar 1, sicer (za `n>0`) pa je rezultat enak produktu števila `n` in rezultata izračuna `(n-1)!`. Ta postopek lahko zapišemo tudi v jeziku C:

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

Celoten program je v datoteki `fak-rek.c`.

Primer, ko je `n` enako 0 (izračun `0!`), je najenostavnejši primer izračuna fakultete (včasih ga imenujemo tudi trivialni primer) in je hkrati tudi izstopni pogoj iz rekurzije (robni pogoj ali pogoj zaustavitve). V primeru `n=0` se namreč ne kliče več rekurzivno funkcije `fakulteta`, temveč funkcija vrne rezultat 1 in tako zaključi z rekurzivnimi klici same sebe. Rešitev trivialnega primera je obvezna in ne sme manjkati, saj je to izstopni pogoj iz rekurzije, kar pomeni, da se tu rekurzija konča. Če bi ga izpustili, bi funkcija klicala samo sebe v nedogled oziroma dokler ji ne bi zmanjkalo pomnilnika.

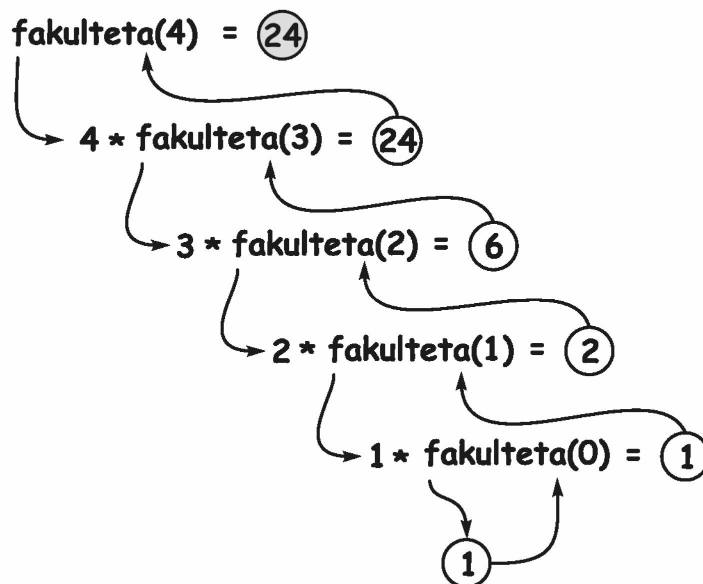
Kot lahko vidimo iz primera, ima vsaka rekurzivna funkcija najmanj dva dela:
- izstopni pogoj iz rekurzije (to je ponavadi rešitev trivialnega problema) in
- klic same sebe (nad manjšim/enostavnejšim problemom).

**Podrobnosti o izvajanju rekurzivnega programa**

Poglejmo si, kako se izvaja rekurzivna funkcija `fakulteta`. Najlažje to prikažemo kar na primeru, tako da sledimo izvajanju kode.

Recimo, da želimo izračunati vrednost `4!` in zato pokličemo funkcijo `fakulteta(4)`. Koda v telesu funkcije najprej preveri, ali je funkciji podan argument (v našem primeru

število 4) enak 0. Ker pogoj ni izpolnjeni (4 ni enako 0), se izvrši stavek pod `else`, to je `return(n*fakulteta(n-1));`. Funkcija torej vrne rezultat, ki je enak zmnožku števila 4 (naš `n`) in rezultata, ki ga vrne funkcija `fakulteta(3)`. Preden lahko funkcija izračuna in vrne rezultat, mora dobiti vrednost, ki jo vrne klic `fakulteta(3)`. To pomeni, da ponovno pokličemo isto funkcijo `fakulteta`, le da tokrat z manjšim številom kot argumentom funkcije. Postopek ponavljamo, dokler ne pridemo do klica `fakulteta(0)`. V tem primeru preverjanje `(n==0)` vrne resnično in se zato izvrši koda pod `if` delom stavka (in ne pod `else`, kot v vseh prejšnjih klicih). Tako funkcija vrne vrednost 1 in s tem se končajo tudi rekurzivni klici te funkcije (vrednost argumenta 0 je izstopni pogoj iz rekurzije). Sedaj se le še do konca izvršijo vsi predhodni klici funkcij, tako da se izračuna in vrne ustrezen rezultat. Celoten postopek (vrstni red klicev ter računanje in vračanje rezultatov) je prikazan na spodnji sliki.




Pri vsakem ponovnem klicu funkcije se vsi podatki trenutnega klica funkcije shranijo na sklad (torej na vrh podatkov o prejšnjem klicu funkcije). Na sklad se shranijo funkcijski parametri, njene lokalne spremenljivke ter naslov vrnitve (*return address*), ki pove, kje je naslednji ukaz, ki naj se izvede, ko se funkcija konča. Ti podatki se nalagajo na sklad ob vsakem novem klicu funkcije. Ko se posamezna funkcija zaključi, se s sklada poberejo podatki prejšnjega klica funkcije (vedno vzamemo podatke na vrhu sklada). To se ponavlja do prvega klica funkcije.

**Reševanje problema: iterativno ali rekurzivno?**

Poglejmo si še en znan primer, to je izračun `n`-tega člena Fibonaccijevega zaporedja. Prva dva člena zaporedja imata vrednost 0 in 1 po vrsti, vsak naslednji člen pa je vsota

dveh predhodnih členov zaporedja. Prvih deset členov Fibonaccijevega zaporedja je tako:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
```

Zaporedje lahko zapišemo tudi z naslednjimi tremi formulami, kjer je `n` naravno število, večje od dva:

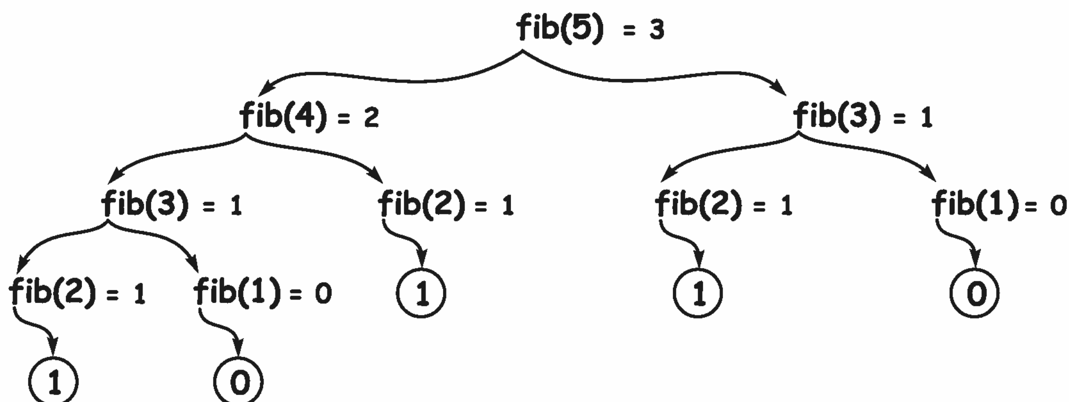
```
fib(1) = 0
fib(2) = 1
fib(n) = fib(n-1) + fib(n-2)
```

Ker je problem izračuna `n`-tega Fibonaccijevega števila podan rekurzivno, lahko enostavno sestavimo rekurzivno funkcijo, ki izračuna to število za podani `n` (`n>0`):

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

Koda je kratka in enostavno razumljiva, celoten program najdete med primeri tega poglavja v datoteki `fib-rek.c`.

Ker funkcija `fib` dvakrat kliče samo sebe, lahko njeno delovanje (izvajanje) predstavimo z binarnim drevesom. Poglejmo si primer klica funkcije `fib(5)` za računanje petega Fibonaccijevega števila (glej spodnjo sliko).




Rezultat `fib(5)` je enak vsoti rezultatov `fib(4)` in `fib(3)`. Rezultat `fib(4)` spet izračunamo tako, da seštejemo rezultata `fib(3)` in `fib(2)`. Slednji je enak 1, saj je to eden od izstopnih pogojev pri rekurziji (`n=2`). Rezultat `fib(3)` pa moramo ponovno

izračunati in je enak vsoti rezultatov `fib(2)` in `fib(1)`. Tu se ta veja rekurzije zaključi, saj `n` pri obeh klicih ustreza enemu od izstopnih pogojev. Funkciji vrneta rezultata 1 in 0 zaporedoma. Sedaj lahko izračunamo tudi rezultat funkcije `fib(3)`, to je vsota 1 in 0, torej 1. Ta rezultat nam omogoči izračun rezultata `fib(4)`, to je 1 plus 1, torej 2. Podobno se izračuna tudi desna stran drevesa (rezultat `fib(3)`, ki je enak 1) in tako dobimo rezultat `fib(5)`, ki je enak 3 (vsota 2 in 1).

Če si pogledamo drevo klicev funkcije `fib(5)`, lahko hitro ugotovimo, da se ena in ista vrednost računa večkrat (`fib(3)` in `fib(1)` računamo dvakrat, `fib(2)` pa trikrat), kar sigurno ne pripomore k učinkovitosti programa. To še posebej velja za večje vrednosti `n`, kjer je ponovljenih izračunov še več.

Pa poiščimo še iterativno rešitev tega problema. Rešitev je malo daljša, a vseeno dovolj enostavna: v zanki sproti računamo vsak naslednji člen zaporedja tako, da seštejemo dva predhodna člena (spremenljivki `n1` in `n2`), dokler ne izračunamo `n`-tega člena.

```
int fib(int n)
{
  int i, n1, n2, f;
  if( n==1 )
    return(0);
  if( n==2 )
    return(1);
  n1 = 1;
  n2 = 0;
  for(i=3; i<=n; i++) {
    f = n2 + n1;
    n2 = n1;
    n1 = f;
  }
  return(f);
}
```

Celoten program je v datoteki `fib-iter.c`.

### *Eleganca napram učinkovitosti*

Rekurzivne rešitve so pogosto zelo elegantne in krajše od iterativnih, a so zato manj učinkovite, saj zasedajo veliko prostora na skladu. Pogosto je problem tudi v časovni kompleksnosti, posebej tam, kjer se ena in ista rešitev računa večkrat (primer Fibonaccijevo zaporedje).

Rekurzija ne varčuje s pomnilnikom (ponavadi ga porabi več od iterativne rešitve, saj se veliko podatkov shrani na sklad), niti ne deluje hitreje. Njena glavna prednost je, da je program pogosto napisan krajše in bolj razumljivo. Slednje velja še posebej pri problemih, ki so rekurzivni po naravi (po definiciji), kot so recimo drevesa.

***Po naravi rekurzivni problemi***

Veliko problemov lahko rešimo na oba načina, iterativno ali rekurzivno, obe rešitvi pa sta podobno enostavni in razumljivi. Pri takih problemih je bolje uporabiti iterativno rešitev. Obstajajo pa tudi problemi, ki so po naravi rekurzivni in zato nimajo enostavne iterativne rešitve. Tu s pridom uporabimo rekurzijo. Primeri takih problemov so problem vračanja drobiža (ta primer si bomo ogledali v nadaljevanju), drevesa (binarna so opisana v zadnjem poglavju), problem Hanoiskih stolpov in podobni.

Da bi si lažje predstavljali koristnost in uporabnost rekurzije, si oglejmo tako imenovani problem vračanja drobiža: na koliko različnih načinov lahko vrnemo drobiž za določen znesek?

Problem lahko definiramo tudi na naslednji način. Imamo nek znesek `znesek` v evrih (oziroma centih) in nas zanima, na koliko različnih načinov lahko ta znesek izplačamo v kovancih po 1 cent, 2 centa, 5 centov, 10 centov, 20 centov ali 50 centov. Na koliko načinov lahko torej iz teh kovancev sestavimo poljuben znesek?

Primer: znesek 5 centov lahko sestavimo na 4 različne načine (1 x 5 centov, 2 x 2 centa in 1 x 1 cent, 1 x 2 centa in 3 x 1 cent ali pa 5 x 1 cent). Število kombinacij zelo hitro naraste, pri znesku en evro (100 centov) imamo že 4562 načinov.

Rešitev problema je precej enostavna, če jo zapišemo rekurzivno. Recimo, da vse razpoložljive kovance zapišemo v določenem vrstnem redu, recimo kar od najmanjšega do največjega. Število vseh načinov zamenjave nekega zneska `znesek` z uporabo `n` različnih kovancev je enako vsoti števila načinov za zamenjavo zneska `znesek` z uporabo vseh kovancev razen prvega in števila načinov zamenjave zneska, zmanjšanega za vrednost prvega kovanca (`znesek-vrednost`), z uporabo vseh `n` kovancev. Tu `vrednost` pomeni vrednost prvega kovanca. To lahko zapišemo tudi s formulo:

```
drobiz(znesek, n) =
  drobiz(znesek, n-1) + drobiz(znesek-vrednost(n), n)
```

Kot je razvidno iz opisanega, smo načine zamenjave določenega zneska razdelili na dve skupini:

- vse tiste načine, ki ne uporabljajo prvega kovanca in
- vse tiste načine, ki uporabljajo tudi prvi kovanec.

Skupno število načinov zamenjave je tako enako vsoti vseh načinov v obeh skupinah. Število načinov zamenjave zneska iz druge skupine (kjer uporabimo prvi kovanec) pa je nadalje enako številu načinov zamenjave zneska, ki ostane, če prvi kovanec uporabimo. Tako naš problem prevedemo na isti problem z manjšim zneskom in/ali z manjšim številom kovancev.

Preden pa se lotimo zapisa rekurzivne funkcije, se moramo dogovoriti še nekaj pravil. Če je znesek enak 0, bomo rekli, da imamo le en način zamenjave drobiža. Če je znesek manjši od 0, ne moremo narediti nobene zamenjave (število načinov zamenjave je nič). Podobno ne moremo narediti zamenjave tudi v primeru, ko je število razpoložljivih kovancev enako nič. Opisano lahko zapišemo tudi s formulami:

```
drobiz(0, n) = 1
drobiz(znesek, n) = 0, če znesek < 0
drobiz(znesek, 0) = 0
```

Predpostavimo, da imamo že podano funkcijo `vrednost(n)`, ki vrne vrednost `n`-tega kovanca po vrsti. Če smo kovance sortirali naraščajoče po velikosti, dobimo:

```
vrednost(1) = 1
vrednost(2) = 2
vrednost(3) = 5
vrednost(4) = 10
vrednost(5) = 20
vrednost(6) = 50
```

Sestavljene in opisane formule sedaj zlahka prevedemo v programsko kodo v rekurzivni proceduri `drobiz`:

```
int drobiz(znesek, n)
{
  if(znesek == 0)
    return(1);
  if(znesek < 0)
    return(0);
  if(n == 0)
    return(0);
  return(drobiz(znesek, n-1) + drobiz(znesek-vrednost(n), n));
}
```

Datoteka `menjava.c` vsebuje celoten program, skupaj z vsemi potrebnimi funkcijami.

Kot vidimo, je zapisana rekurzivna rešitev precej kratka in enostavna. Poskusite razmisliti o iterativni rešitvi. Bi jo znali zapisati?

## Še nekaj primerov rekurzivnih rešitev problemov

### *Pretvorba med številskimi sistemi*

Začnimo s programom, ki pretvarja med številskimi sistemi. Funkciji podamo naravno število `n` (v desetiški obliki, `n>0`) in poljubno bazo `b` (kjer je `b` večji od 1 in manjši od 10), ta pa nam izpiše število `n` v številskem sistemu z bazo `b`.

Problem je enostavno rešljiv. Izstopni pogoj iz rekurzije je pri trivialni rešitvi, ko je `n` enak 0 (v tem primeru ne izpišemo nič, saj mora biti `n>0`). Sicer pa število `n` delimo z bazo `b` in postopek ponovimo nad dobljenim rezultatom celoštevilskega deljenja. Na koncu, ko se rekurzivni klici zaključijo, pa izpišemo še vse števke, ki so enake ostankom števila `n` pri deljenju z `b` (tako dobimo pravilno obrnjen izpis od leve proti desni).

```
void pretvori(int n, int b)
{
  if( n>0 ) {
    pretvori(n/b, b);
    printf("%d", n%b);
  }
}
```

Program najdete v datoteki `pretvorba.c`.

Kako pa bi se lotili pretvorbe števila `n`, ki je zapisano v številskem sistemu z bazo `b`, v desetiški sistem? Tudi ta problem je z rekurzijo enostavno rešljiv. Robni pogoj dobimo, kadar je število, ki ga pretvarjamo, enako nič (`n=0`). V tem primeru je tudi rešitev (pretvorba) enaka nič. Sicer pa problem rešimo tako, da posebej obravnavamo zadnjo števko podanega števila (`n%10`), preostale števke (`n/10`) pa predstavljajo enak, a manjši podproblem in nad njimi rekurzivno pokličemo isto funkcijo (funkcija `dec`). Rezultat je vsota z bazo pomnožene rešitve podproblema (`dec(n/10,b)*b`) in zadnje števke podanega števila (`n%10`).

Funkcija `dec` je naslednja (cel program je v datoteki `pretvorba10.c`):

```
int dec(int n, int b)
{
  if( n==0 )
    return(0);
  return(dec(n/10, b)*b+n%10);
}
```

### *Palindromi*

Eden od problemov, ki ga elegantno rešimo s pomočjo rekurzije, je tudi preverjanje, ali je podana beseda palindrom. Palindrom je taka beseda, ki se enako bere naprej in nazaj, kot so na primer besede *a*, *AA*, *ata*, *abba*, *cepec* ali *radar*.

Rekurzivna rešitev je zelo enostavna, če sam problem definiramo rekurzivno: beseda je palindrom, če je prvi znak enak zadnjemu znaku, vsi znaki vmes pa tudi sestavljajo palindrom. Pri tem velja, da je prazna beseda kot tudi beseda dolžine ena (en sam znak) vedno palindrom. Slednje uporabimo kot izstopni pogoj iz rekurzije.

Poglejmo si tako preverjanje na primeru besede *cepec*. Prvi znak (znak *c*) je enak zadnjemu znaku, zato je beseda *cepec* palindrom natanko takrat, ko je tudi beseda *epe* palindrom. Pri preverjanju slednje spet ugotovimo, da se njen prvi znak ujema z zadnjim znakom (znaka *e*), torej je tudi ta beseda palindrom natanko takrat, ko je palindrom tudi beseda *p*. Vsaka beseda dolžine 1 je po definiciji palindrom, torej je beseda *p* palindrom. Potem sledi, da je palindrom tudi beseda *epe*, prav tako pa tudi beseda *cepec*.

Za rešitev problema moramo torej poznati dolžino besede, ki jo poleg kazalca na začetek besede podamo funkciji. Glavo funkcije lahko potemtakem zapišemo kot:

```
int palindrom(char *niz, int n)
```

Funkcija `palindrom` prejme dva argumenta: `niz` je kazalec na začetek besede, ki jo preverjamo, `n` pa določa dolžino te besede. Funkcija vrača sicer vrednost tipa `int`, a sta v praksi to le dve možni vrednosti: 1 v primeru, da je podana beseda palindrom, sicer pa je vrnjena vrednost 0. Če imamo besedo definirano kot:

```
char *beseda = "cepec";
```

potem lahko funkcijo `palindrom` pokličemo takole:

```
int pal = palindrom(beseda, strlen(beseda));
```

V telesu funkcije `palindrom` najprej določimo izstopni pogoj iz rekurzije. V našem primeru je to beseda, ki jo sestavlja en sam znak, ali pa je celo brez znakov (prazna beseda). Če je dolžina besede torej manjša od 2, je beseda sigurno palindrom, zato funkcija v tem primeru vrne vrednost 1 (resnično).

```
if( n < 2 )
  return(1);
```

V primeru, da izstopni pogoj iz rekurzije ni izpolnjen, najprej preverimo, ali se prvi znak besede, to je `niz[0]`, ujema z zadnjim znakom besede, to je `niz[n-1]` (če je `n` dolžina besede, je prvi znak na mestu z indeksom `0`, zadnji znak pa na mestu z indeksom `n-1`). V primeru neujemanja podana beseda sigurno ni palindrom, zato funkcija vrne vrednost 0 (neresnično) in ni potrebno nobeno nadaljnje preverjanje.

```
if( niz[0] != niz[n-1] )
  return(0);
```

V primeru ujemanja prvega in zadnjega znaka v besedi pa je podana beseda palindrom natanko takrat, ko je palindrom tudi beseda od drugega do predzadnjega znaka, torej za dva znaka krajša beseda. Slednje preverimo s klicem funkcije `palindrom` na ustrezno krajši besedi (nova beseda se začne pri drugem znaku, torej `niz+1` kaže na začetek te krajše besede, njena dolžina pa je za 2 manjša). Tako smo problem prevedli na nekoliko

manjši podproblem. Funkcija v tem primeru vrne natanko tisto, kar vrne klic funkcije `palindrom(niz+1, n-2)`, kar zapišemo kot:

```
return(palindrom(niz+1, n-2));
```

Funkcija `palindrom`, zapisana v celoti, je naslednja:

```
int palindrom(char *niz, int n)
{
  if( n < 2 )
    return(1);
  if( niz[0] != niz[n-1] )
    return(0);
  return(palindrom(niz+1, n-2));
}
```

Celoten program najdete v datoteki `palindrom.c` med primeri.

***Permutacije***

Napišimo še program, ki izpiše vse možne permutacije `n` elementov. Pri rešitvi si ponovno pomagamo z rekurzijo.

Elemente, ki jih želimo permutirati, imamo shranjene v polju `a`. Elementi polja `a` so kar cela števila od 1 do `n` po vrsti (tako bomo pri izpisu polja tudi najlažje opazili narejene permutacije). Uporabili smo tudi funkcijo `izpis()`, ki izpiše elemente polja `a`. Funkcija `perm` je sicer malo daljša, a v primerjavi z iterativno rešitvijo bolj razumljiva. Pri zapisani rešitvi smo zaradi enostavnosti predpostavili, da je polje `a` deklarirano kot globalna spremenljivka in je zato dostopno tudi v sami funkciji.

```
void perm(int k)
{
  int i, x;
  if( k==0 ) {
    izpis();
  }
  else {
    perm(k-1);
    for(i=0; i<k; i++) {
      x = a[i]
      a[i] = a[k];
      a[k] = x;
      perm(k-1);
      x = a[i];
      a[i] = a[k];
      a[k] = x;
    }
  }
}
```

Edini argument funkcije je `k`, ki pove, nad koliko elementi delamo permutacije (kako veliko je polje). Pri prvem klicu funkcije `perm` je `k` seveda enak `n`. Če je `k` enak 0, to pomeni, da ne delamo permutacij nad nobenim elementom polja, zato je to naš izstopni pogoj iz rekurzije. V tem primeru le izpišemo polje, kar predstavlja eno rešitev problema (eno od permutacij).

Sicer pa imamo dve možnosti. V prvi pustimo `k`-ti element na svojem mestu in permutiramo vse ostale elemente (za eno stopnjo manjši problem, klic `perm(k-1)`). Druga možnost pa je, da `k`-ti element prestavimo na neko drugo mesto tako, da ga zamenjamo z `i`-tim elementom (`i` izbiramo v `for` zanki po vrsti od prvega do zadnjega elementa), nato permutiramo vse ostale elemente (za eno stopnjo manjši problem, ponovno klic `perm(k-1)`) ter na koncu `k`-ti element ponovno postavimo nazaj na svoje prvotno mesto (ponovno zamenjamo `k`-ti in `i`-ti element). Postopek ponovimo za vse možne zamenjave (`for` zanka).

Cel program je malo daljši, saj zajema tudi funkcijo za izpis polja ter glavni program, kjer polje napolnimo z ustreznimi elementi (števili od 1 do `n`). Ker vrednost `n` podamo šele ob klicu programa, moramo prostor za elemente polja rezervirati dinamično. Funkciji `perm` kot argumenta podamo tudi polje `a` in velikost tega polja `n` (polje ni več deklarirano kot globalno) ter dodamo števec, ki prešteje vse narejene (in izpisane) permutacije elementov. Program je v datoteki `permutacije.c`.

### *Izris vzorca iz pik*

Program naj v odvisnosti od podanega naravnega števila `m` izriše (rekurzivno) naslednji vzorec iz pik:

```
m=1          m=2          m=3          m=4          m=5

.            .            .            .            .
             . .          . .          . .          . .
             .            . . .        . . .        . . .
                          . .          . . . .      . . . .
                          .            . . .        . . . . .
                                       . .          . . . .
                                       .            . . .
                                                    . .
                                                    .
```

Napišimo funkcijo `pike`, katero bomo rekurzivno klicali, da dobimo ustrezen izris vzorca. Funkcija naj prejme dva parametra, prvi parameter `n` pove trenutno vrstico, ki jo izpisujemo (in hkrati število pik, ki jih zaporedoma napišemo), drugi pa je `m`, ki določa število izpisanih stolpcev (največjo dolžino zaporedja izpisanih pik v eni vrstici). Ker na

začetku najprej izpišemo prvo vrstico (eno samo piko), mora biti ob prvem klicu funkcije `n` enak 1.

Izstopni pogoj iz rekurzije je pri `n>m`, saj je največje število pik, ki jih izpišemo v eni vrstici, enako `m` (zato `n` ne sme biti večji od `m`). V tem primeru se rekurzivno klicanje funkcije zaključi. Če pa je `n` (to je število pik, ki jih izpisujemo v trenutni vrstici) manjše ali enako `m`, potem v `for` zanki izpišemo najprej `n` pik in nato še znak za novo vrstico. Postopek ponovimo za `n+1`, saj moramo v naslednji vrstici izpisati eno piko več. Na koncu pa ponovimo izpis pik simetrično še na spodnji strani vzorca (druga `for` zanka), a le v primeru, kadar ne izpisujemo srednje vrstice (natanko `m` pik), saj se le-ta izpiše le enkrat.

Celoten postopek zapišemo z naslednjo funkcijo:

```
void pike(int n, int m)
{
  int i;
  if( n <= m ) {
    for(i=0; i<n; i++)
      printf(".");
    printf("\n");
    pike(n+1, m);
    if( n < m ) {
      for(i=0; i<n; i++)
        printf(".");
      printf("\n");
    }
  }
}
```

Ker je ob prvem klicu funkcije `n` enak 1, je njen klic naslednji (za poljuben `m`):

```
pike(1, m);
```

Celoten program najdete v datoteki `pike.c`.

***Izpis obrnjene datoteke***

Za konec pa napišimo še rekurzivno funkcijo, ki izpiše vsebino poljubno dolge datoteke v obratnem vrstnem redu. Pri iterativni rešitvi bi morali prebrati in si zapomniti vsebino cele datoteke, preden bi jo lahko začeli izpisovati. Pri rekurzivni rešitvi pa za pomnjenje prebranih znakov poskrbi kar sam ponovni klic funkcije, ko se vse lokalne spremenljivke shranijo na sklad. Rešitev je sicer zelo požrešna do virov (pomnilnik), a je zato bolj enostavna in jo lažje zapišemo.

V funkciji najprej preverimo, ali smo že prišli do konca datoteke (to je izstopni pogoj iz rekurzije). V primeru, da datoteke še ni konec, preberemo en znak iz datoteke, ponovno pokličemo funkcijo (ki poskrbi za nadaljnje branje datoteke) in na koncu še izpišemo prebrani znak:

```
void obrat(FILE *fp)
{
  int c;
  if( !feof(fp) ) {
    c = fgetc(fp);
    obrat(fp);
    fputc(c, stdout);
  }
}
```

Celoten program je v datoteki `obrni.c`.

Za konec pa še vprašanje: kaj se zgodi, če zamenjamo stavka `obrat(fp);` in `fputc(c, stdout);` ter najprej izpišemo znak in šele nato kličemo funkcijo `obrat`? Vaš odgovor preverite na primeru!

# 9. Kazalčni seznami in drevesa

V tem poglavju si bomo ogledali neurejene in urejene linearne seznamе ter urejena binarna drevesa.

Kazalčni seznami in drevesa so dinamične podatkovne strukture. Predstavljajo enega od načinov za dinamično shranjevanje podatkov v pomnilniku. Ne glede na število podatkov zavzame seznam vedno le toliko pomnilnika, kot ga potrebuje za hranjenje teh podatkov. Ko v seznam dodajamo nove elemente, se seznam daljša (več podatkov) in zavzema več prostora v pomnilniku (sproti, v času izvajanja, rezerviramo prostor za nove podatke). Ko iz seznama brišemo elemente, se le-ta krajša (manj podatkov) in zavzema manj prostora v pomnilniku (sprostimo prostor, ki so ga zasedali zbrisani elementi).

**Dinamično dodeljevanje pomnilnika**

Oglejmo si naslednji primer. Recimo, da želimo shraniti poljubno število celih števil. Za hranjenje števil bi lahko uporabili polje, vendar moramo pri deklaraciji polja navesti tudi njegovo velikost. Zato se moramo že med pisanjem programa odločiti, kako veliko bo polje. Recimo, da je ta velikost 100:

```
int stevila[100];
```

To pomeni, da bomo v polje `stevila` lahko shranili do največ 100 celih števil. Dokler jih je manj, bomo imeli nezaseden prazen prostor, če pa je števil več kot 100, naše polje ne zadošča več.

Problem lahko rešimo tako, da najprej ugotovimo število elementov polja (spremenljivka `max`) in nato rezerviramo ravno prav prostora za vse elemente polja (`max` krat velikost enega elementa polja). Tokrat polje deklariramo kar s kazalcem na prvi element polja:

```
int *stevila;
int max;
max = 10;  // določimo število elementov polja
stevila = (int*) malloc(max*sizeof(int));
```

Tako smo rezervirali blok pomnilnika, ki je ravno dovolj velik za vsa naša števila. Pomnilniški blok se dodeli šele v času izvajanja programa. Vsi elementi polja so v enem bloku pomnilnika in si zaporedoma sledijo. Če želimo v polje k obstoječim dodati še več elementov, moramo na novo dodeliti ustrezno velik blok pomnilnika za vse elemente in prepisati obstoječe elemente, kar ni najbolj enostavna rešitev. Na srečo si tu lahko pomagamo s funkcijo `realloc`, ki delo opravi namesto nas.

Funkcija `realloc(p,n)` spremeni velikost pomnilniškega prostora, ki smo ga pred tem rezervirali s pomočjo funkcije `malloc`. Argument `p` je kazalec na prvotno rezerviran prostor, `n` pa določa skupno število bajtov želenega pomnilniškega prostora.

Funkcija vrne kazalec na začetek novo rezerviranega pomnilniškega bloka oziroma `NULL` v primeru, ko rezervacija pomnilnika ne uspe. Hkrati funkcija tudi poskrbi za prepis vsebine prvotnega pomnilniškega bloka v nov blok, če je to potrebno (če ni bilo mogoče preprosto spremeniti velikosti že obstoječega bloka). Tako bi delovanje funkcije `realloc(stari, vel)` lahko simbolično opisali kot:

```
novi = malloc(vel);
if (novi != NULL) {
  memcpy(novi, stari, min(vel, stara_vel));
  free(stari);
}
return(novi);
```

Naše polje števil lahko torej enostavno povečamo, če se izkaže potreba:

```
int *tmp;
max = 15;  // določimo novo število elementov polja
tmp = realloc(stevila, max*sizeof(int));
if( tmp != NULL)
  stevila = tmp;
else
  printf("Napaka: polja ne moremo povecati\n");
```

Primer programa, ki demonstrira dinamično dodeljevanje pomnilnika, je v datoteki `alokacija.c`. Bodite pozorni na sproščanje pomnilnika – vsak dinamično dodeljen kos pomnilnika (z uporabo funkcije `malloc` ali `realloc`) moramo tudi eksplicitno sprostiti (z uporabo funkcije `free`).

Z dinamičnim dodeljevanjem pomnilnika smo problem določanja ustrezne velikosti bloka pomnilnika za elemente polja prenesli v čas izvajanja programa. Še vedno pa za polje velja, da si elementi sledijo zaporedoma, kar lahko privede do težav pri dodajanju novih elementov ali brisanju posameznih elementov. Kadar elementov ne dodajamo le na konec polja, ampak jih vrivamo na poljubno mesto (kot na primer pri urejenem polju), moramo najprej na ustreznem mestu zagotoviti prazen prostor za nov element, kar dosežemo s prepisovanjem (premikanjem) vseh ostalih elementov. Podobno velja tudi za brisanje poljubnih elementov polja: če ne želimo imeti "lukenj" v našem polju, moramo po vsakem brisanju prepisati elemente in zapolniti "luknjo", ki ostane za izbrisanim elementom.

Idealno bi seveda bilo, da bi elemente lahko poljubno dodajali in brisali, ob tem pa nam ne bi bilo potrebno skrbeti, kje v pomnilniku se ti elementi nahajajo, saj bi za vsak element lahko rezervirali prostor v pomnilniku neodvisno od preostalih elementov. Rešitev predstavlja kazalčni seznam. Posamezni elementi seznama se nahajajo v poljubnem delu pomnilnika, med seboj pa so povezani s kazalci (tako, da jih lahko najdemo). Tako lahko elemente seznama po potrebi sproti dodajamo ali brišemo, ne da bi zato morali posegati v vse preostale elemente seznama.

Izgradnja seznama temelji na strukturah, zato si najprej oglejmo nekaj osnovnih pojmov o strukturah.

**Strukture, kazalci na strukture in rekurzivne strukture**

Kadar želimo združevati več spremenljivk, ki so lahko tudi različnih tipov, uporabimo strukture. Na ta način lahko združimo podatke, ki logično sodijo skupaj, kot so na primer podatki o študentu: ime, priimek, letnik itd.

Za opis enega študenta bi lahko uporabili naslednje tri spremenljivke:

```
char ime[10];
char priimek[10];
int letnik;
```

Ker se vse tri spremenljivke `ime`, `priimek` in `letnik` nanašajo na istega študenta, je smiselno, da jih združimo. Tako definiramo strukturo `student`, ki zajema vse tri spremenljivke:

```
struct student {
  char ime[10];
  char priimek[10];
  int letnik;
};
```

Sedaj lahko deklariramo spremenljivko `stud`, ki opisuje enega študenta, kot spremenljivko tipa `struct student`:

```
struct student stud;
```

Posamezne komponente strukture dosegamo preko operatorja `.` (pika); v našem primeru lahko na primer določimo vrednosti posameznih komponent spremenljivke `stud` na naslednji način:

```
stud.letnik = 1;
strcpy(stud.ime, "Janez");
strcpy(stud.priimek, "Novak");
```

Bodite pozorni na nastavitve vrednosti imena in priimka. Ker sta oba deklarirana kot niza, torej kot polja znakov, smo morali uporabili funkcijo `strcpy`.

Če imamo spremenljivko deklarirano kot kazalec na strukturo:

```
struct student *pstud;
```

potem do komponente strukture dostopamo preko operatorja `->` (napišemo ga kot zaporedje znakov minus in večje). Tako bi lahko kot primer podatke o našem študentu izpisali na naslednji način:

```
pstud = &stud;  // pstud kaže na stud
printf("%s %s, ", pstud->ime, pstud->priimek);
printf("%d\n", pstud->letnik);
```

Seveda bi lahko namesto `pstud->letnik` pisali tudi `(*pstud).letnik`, kar je enakovreden zapis, le malo daljši, saj zaradi višje prioritete operatorja `.` od operatorja `*` potrebujemo tudi oklepaje.

Če ima struktura (vsaj eno) komponento, ki je tipa kazalec na to isto strukturo, rečemo, da je rekurzivna, saj je rekurzivno definirana. Rekurzivna struktura je torej tista, katere komponenta se nanaša na strukturo samo.

V deklaraciji strukture sicer ne moremo deklarirati komponente, ki bi bila spet tipa te iste strukture, lahko pa deklariramo komponento, katere tip je kazalec na to strukturo. Poglejmo si primer:

```
struct a {
  struct a b;  // NAPAKA!
};
```

Zgornja deklaracija strukture je napačna, saj je tip komponente `b` strukture `a` kar sama struktura `a`. Pravilno bi komponento `b` zapisali kot kazalec na strukturo `a`:

```
struct a {
  struct a *b;
};
```

Rekurzivne strukture bomo uporabili pri deklaraciji elementov seznama, kar si bomo ogledali v nadaljevanju.

## Kazalčni seznami

Kazalčni seznam predstavlja preprost način povezave množice elementov, ki jih razvrstimo v seznam ali vrsto. Najenostavnejši seznami so neurejeni, kar pomeni, da je vrstni red elementov v seznamu poljuben. Zato lahko nove elemente v seznam dodajamo vedno na isto mesto, najlažje na konec ali na začetek seznama.

Najpreprostejše kazalčne seznami imenujemo tudi enosmerno povezani seznami, ker imajo ti seznami dva elementa povezana le z eno zvezo, to je s kazalcem na element naslednika.

Vsak element seznama vedno sestavljata dve komponenti: podatkovni del in kazalec na naslednji element seznama. V nadaljevanju bomo zaradi enostavnosti podatkovni del omejili le na eno celoštevilsko vrednost, kar zadostuje za ponazoritev dela s seznami. Element seznama tako opišemo s strukturo `struct el` z dvema komponentama. Prva je `x`, ki predstavlja celoštevilsko vrednost elementa. Druga komponenta, to je spremenljivka `n`, pa je povezava na naslednji element seznama. Ker je naslednji element seznama prav tako struktura `el`, je ta povezava kazalec na to strukturo:

```
struct el
{
  int x;
  struct el *n;
};
```

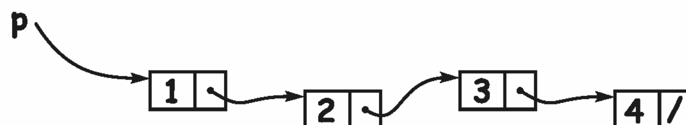
Spremenljivko `p`, ki je kazalec na začetek seznama, lahko deklariramo kot:

```
struct el *p;
```

Element seznama ponavadi simbolično prikažemo z dvodelno škatlo, kjer je v enem delu vpisana vrednost elementa (v našem primeru vrednost spremenljivke `x`), iz drugega dela pa vodi puščica, ki kaže na naslednji element seznama. Naslednje slike prikazujejo nekaj načinov simboličnega prikaza elementa seznama:

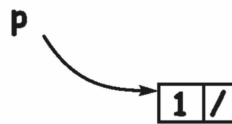
Na levi je prikazan element seznama z vrednostjo `1` in kazalcem na naslednji element. Srednja slika prikazuje zadnji element seznama, ki ima prav tako vrednost `1`, nima pa naslednjega elementa, kar ponazorimo s poševnico (ker je to zadnji element seznama, kazalec ne kaže nikamor). Na desni pa je simboličen prikaz prvega elementa seznama, na katerega kaže kazalec `p` (to je kazalec na začetek seznama).

In kako bi simbolično prikazali cel seznam? Škatle z elementi enostavno povežemo skupaj. Vzemimo za primer seznam, ki hrani štiri elemente z vrednostmi 1, 2, 3 in 4 po vrsti. Simbolično ga narišemo, kot prikazuje slika:



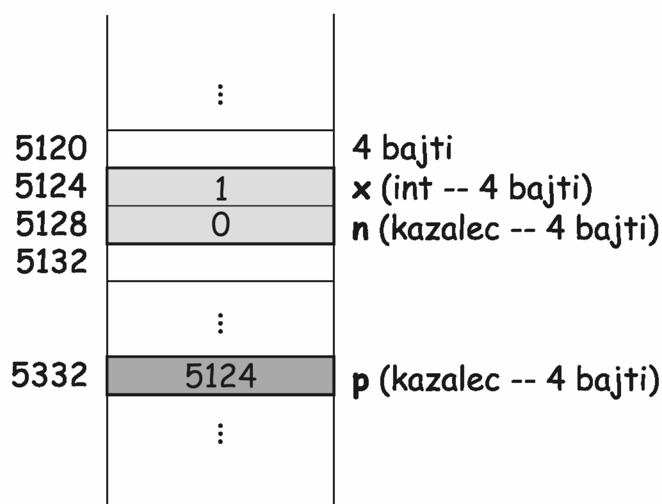

Tu je spremenljivka `p` kazalec na začetek seznama, element z vrednostjo 4 pa je zadnji element seznama.

Za boljše razumevanje kazalčnih seznamov si poglejmo še, kako je seznam predstavljen v pomnilniku. Za začetek vzemimo seznam, ki ga sestavlja en sam element. Spremenljivka `p` kaže na začetek seznama, vrednost elementa seznama pa je 1. To simbolično narišemo, kot prikazuje slika:

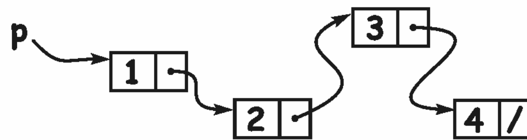



In kako je ta kratek seznam predstavljen v pomnilniku? Spremenljivka `p`, ki je kazalec na strukturo `el`, zavzame v pomnilniku štiri bajte, torej eno lokacijo na spodnji sliki. Na levi strani so napisani naslovi lokacij, ki so seveda za naš primer izmišljeni. Tako se naša spremenljivka `p` nahaja na naslovih 5332 do 5335 (na sliki smo jo označili temnejše sivo). Strukturo `el` sestavljata celoštevilska spremenljivka `x`, ki zasede štiri bajte, in kazalec na isto strukturo `n` (zasede naslednje štiri bajte). Tako cela struktura zasede skupaj osem bajtov, ki se na naši sliki nahajajo na lokacijah 5124 do 5131. Del pomnilnika, ki ga zaseda prvi element seznama, označuje svetlejša sivina.

Vrednost spremenljivke `x` je 1, kazalec `n` pa ne kaže nikamor, ker je to zadnji element v seznamu, zato ima vrednost 0. Po dogovoru imajo kazalci, ki nikamor ne kažejo, vrednost 0 oziroma `NULL` (slednja je konstanta, definirana v zaglavni datoteki `stdio.h`). Kam pa kaže kazalec `p`? Na prvi element seznama, seveda. Ko rečemo, da `p` kaže na element seznama, to pomeni, da je njegova vrednost enaka naslovu, na katerem se nahaja ta element. V našem primeru je to naslov 5124, kar je tudi vrednost spremenljivke `p`.



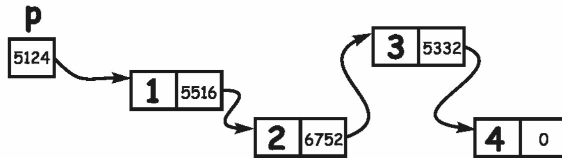

Pa se vrnimo na naš prejšnji seznam s štirimi elementi. Simbolično ga narišemo takole:




Kako pa je ta seznam predstavljen v pomnilniku? Posamezni elementi seznama zasedajo po osem bajtov in ležijo v različnih delih pomnilnika, ne nujno po vrsti. Edina povezava med elementi je naslov naslednjega elementa, spremenljivka `p` pa je kazalec na začetek seznama. Če torej pregledujemo elemente seznama po vrsti od začetka, bomo s pomočjo povezav med njimi prišli preko vseh elementov seznama do konca. Tako nas začetek seznama (spremenljivka `p`) usmeri na prvi element na naslovu 5124. Ta element ima vrednost 1 in povezavo na naslednji element na naslovu 5516. Na tem naslovu najdemo vrednost 2 in naslov naslednjega elementa 6752. To je naslov tretjega elementa, katerega vrednost je 3, povezava na naslednji element pa je naslov 5332. Na omenjenem naslovu je zadnji element našega seznama, ki ima vrednost 4, naslov naslednjega elementa pa je 0, kar pomeni, da naslednjega elementa seznama ni. Posamezni elementi seznama so na spodnji sliki poudarjeni z močnejšo obrobo, podobno kot tudi začetek seznama.

| | | |
|---|---|---|
| | ⋮ | |
| 3484 | 5124 | p |
| | ⋮ | |
| 5120 | | |
| 5124 | 1 | |
| 5128 | 5516 | |
| 5132 | | |
| | ⋮ | |
| 5332 | 4 | |
| 5336 | 0 | |
| | ⋮ | |
| 5516 | 2 | |
| 5520 | 6752 | |
| | ⋮ | |
| 6752 | 3 | |
| 6756 | 5332 | |
| | ⋮ | |

Simbolično risbo seznama bi lahko predstavili tudi takole:




kjer vsak element seznama hrani dva podatka: vrednost elementa in naslov naslednjega elementa. Kazalec na začetek seznama `p` pa hrani naslov prvega elementa seznama.

V nadaljevanju bomo za ponazoritev seznamov uporabljali simbolične risbe s škatlicami in puščicami. Gradili bomo sezname, katerih elementi bodo hranili celoštevilske vrednosti, pogledali pa si bomo vse pomembnejše funkcije za delo s seznami, kot so dodajanje elementa, brisanje elementa ali izpis seznama.

### *Deklaracija elementov seznama*

Za nadaljnje delo s seznami moramo najprej definirati, kakšni so elementi seznama. Kot smo že rekli, posamezen element seznama predstavlja struktura, ki ima v našem primeru dve komponenti: celoštevilsko vrednost elementa in kazalec na naslednji element seznama. Strukturo definiramo takole (izbrana imena spremenljivk so malo daljša, a več povejo o sami vlogi spremenljivke):

```
struct element
{
  int vrednost;
  struct element *naslednji;
};
```

Potem lahko deklariramo tudi spremenljivko `zacetek`, ki se nanaša na začetek našega seznama:

```
struct element *zacetek;
```

Na začetku je naš seznam prazen, saj nismo dodali še nobenega elementa, zato spremenljivko `zacetek` inicializiramo na `NULL` (ker seznama ni, `zacetek` ne kaže nikamor):

```
zacetek = NULL;
```

Kot smo že rekli, je spremenljivka `zacetek` kazalec na začetek našega seznama. Ker so elementi seznama dostopni le preko tega kazalca na začetek seznama, je zelo pomembno, da te reference pomotoma ne izgubimo. V primeru izgube reference na

začetek seznama namreč ne bi le izgubili dostopa do elementov seznama (torej celega našega seznama), temveč bi ti elementi tudi po nepotrebnem zasedali prostor v pomnilniku, katerega ne bi mogli več sprostiti, saj ne bi vedeli, kje v pomnilniku je ta prostor.

### *Izpis seznama*

Začeli bomo z eno najenostavnejših operacij nad seznamom, to je izpis elementov seznama. Pri izpisu se moramo le sprehoditi preko vseh elementov in za vsakega izpisati njegovo vrednost. Funkcija prejme en argument, to je kazalec na začetek seznama, ki ga želimo izpisati. Ker funkcija le izpisuje elemente seznama in nič ne vrača, je tipa `void`. Telo funkcije sestavlja `while` zanka, v kateri se sprehodimo preko celega seznama. Na začetku zanke kaže `p` na prvi element seznama (saj smo rekli, da je argument `p` kazalec na začetek seznama). V zanki nato najprej izpišemo vrednost tega elementa seznama (`p->vrednost`), nato pa kazalec `p` prestavimo na naslednji element seznama (vrednost `p` se torej spremeni tako, da kazalec `p` kaže na isti element kot kazalec `p->naslednji`). Zanka se zaključi, ko s kazalcem `p` pridemo do konca seznama, torej ko je `p` enako `NULL`.

Cela funkcija je zelo enostavna, zato je tudi koda kratka:

```
void izpisi(struct element *p)
{
  while( p!=NULL ) {
    printf("%d  ", p->vrednost);
    p = p->naslednji;
  }
  printf("\n");
}
```

Tu bi opozorili še na dejstvo, da se kazalec na začetek seznama v funkcijo prenese po vrednosti, torej je `p` nova spremenljivka, v katero se prepiše vrednost funkciji podanega argumenta, ki je kazalec na začetek seznama. Zato s spreminjanjem vrednosti spremenljivki `p` (pri prehodu preko seznama) ne izgubimo dejanskega kazalca na začetek seznama, saj ob koncu funkcije spremenljivka `p` tako ali tako ne obstaja več.

### *Iskanje elementa v seznamu*

Iskanje elementa v seznamu je v osnovi zelo podobno izpisu seznama, le da tu ne gre več le za enostaven sprehod skozi seznam, temveč gledamo tudi vrednosti posameznih elementov seznama.

Kot je v navadi pri pisanju funkcij v programskem jeziku C, naj funkcija `poisci` vrne vrednost 1 (resnično), če iskani element najdemo v seznamu, sicer pa 0 (neresnično).

Glava funkcije je malo drugačna, saj funkcija prejme dva argumenta (kazalec na začetek seznama in vrednost elementa, ki ga iščemo v seznamu) in vrne celoštevilsko vrednost:

```
int poisci(struct element *p, int vred)
```

Glavnino telesa funkcije zaseda spet `while` zanka, saj se moramo sprehoditi preko seznama, dokler ne najdemo iskanega elementa oziroma pridemo do konca seznama, kar se pač zgodi prej. Tako bi zanko lahko zapisali:

```
while( p!=NULL ) {
  if( p->vrednost == vred)
    break;
  p = p->naslednji;
}
```

Vrednost vsakega element seznama torej primerjamo s podano vrednostjo `vred` in če sta vrednosti enaki, smo iskani element našli (nanj kaže kazalec `p`) in lahko prekinemo iskanje. Sicer pa nadaljujemo iskanje, dokler ne pridemo do konca seznama.

Zanko bi lahko krajše in elegantneje zapisali, če bi primerjavo vrednosti elementa z iskano vrednostjo vključili kar v sam pogoj zanke:

```
while( (p!=NULL) && (p->vrednost != vred)) {
  p = p->naslednji;
}
```

Obe primerjavi (ali smo že prišli do konca seznama in ali je vrednost elementa različna od iskane vrednosti) združimo z operatorjem *in* (`&&`). Če katerikoli od obeh pogojev ni izpolnjen (to pomeni, da smo prišli do konca seznama ali pa da smo našli iskano vrednost), je celoten pogoj zanke neresničen in zanka se zaključi.

Tu moramo opozoriti, da je vrstni red zapisa obeh pogojev zelo pomemben. Čeprav je vrednost izraza `((p!=NULL)&&(p->vrednost!=vred))` enaka vrednosti izraza `((p->vrednost!=vred)&&(p!=NULL))`, pa pride do velike razlike, kadar izraza uporabimo v pogoju `while` zanke: medtem ko je prvi izraz pravilen, drugi v določenih primerih povzroči napako.

Zakaj? Odgovor je enostaven, če poznamo način, kako se v programskem jeziku C računajo izrazi. Izrazi, združeni z operatorjema *in* (`&&`) oziroma *ali* (`||`), se računajo z leve proti desni in izračun se ustavi takoj, ko je znana resničnost oziroma neresničnost izraza. V našem primeru moramo tako najprej preveriti, ali `p` kaže na katerega od elementov seznama, preden pogledamo vrednost tega elementa. V primeru, ko `p` ne kaže nikamor (`p!=NULL` je neresnično), je celoten izraz neresničen in do primerjave vrednosti elementa z iskano vrednostjo (`p->vrednost!=vred`) sploh ne pride (kar je pravilno, saj bi dostop do vrednosti `p->vrednost` povzročil napako).

Na koncu torej le preverimo, ali smo element našli v seznamu, in vrnemo ustrezne vrednosti. Če je kazalec `p` prazen (`p==NULL`), to pomeni, da smo se sprehodili preko celega seznama in elementa nismo našli. Če pa kazalec `p` kaže na enega od elementov seznama (in torej `p!=NULL`), hrani ta element iskano vrednost.

Funkcijo `poisci` zapišimo še enkrat v celoti:

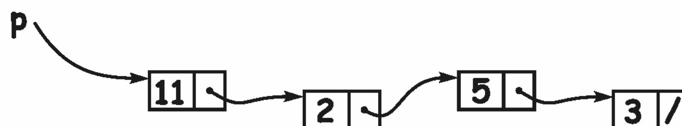
```
int poisci(struct element *p, int vred)
{
  while( (p!=NULL) && (p->vrednost!=vred) )
    p = p->naslednji;
  if(p!=NULL)
    return(1);
  else
    return(0);
}
```

***Dodajanje elementa v seznam***

Dodajanje elementa v seznam je ena od najosnovnejših operacij nad seznamom, saj s postopnim dodajanjem elementov lahko zgradimo celoten seznam.

Najpreprostejše je vstavljanje elementov na začetku seznama. To pomeni, da vsak novo dodan element postane vedno prvi element seznama.

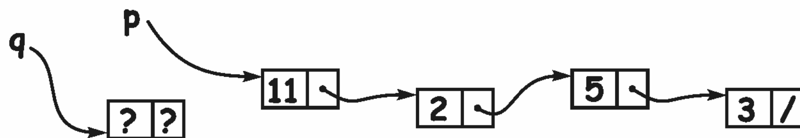
Poglejmo si operacijo dodajanja na začetek seznama podrobneje s pomočjo simbolične predstavitve seznama. Recimo, da imamo seznam, v katerem so štiri števila: 11, 2, 5 in 3 po vrsti. Kazalec `p` kaže na začetek seznama, to je na prvi element (element z vrednostjo 11).




Temu seznamu želimo na začetek dodati nov element z vrednostjo 8. Preden ta element dodamo, ga moramo seveda ustvariti, torej zanj rezervirati prostor v pomnilniku in mu določiti vrednosti obeh komponent. Koliko prostora v pomnilniku pa sploh potrebujemo za en element? Če element seznama določa struktura `struct element`, potem potrebujemo za en element toliko prostora, kot ga zaseda ta struktura, torej `sizeof(struct element)`. Naj bo `q` kazalec na novo ustvarjen element. Prostor za ta element rezerviramo z naslednjim stavkom:

```
q = (struct element*) malloc(sizeof(struct element));
```

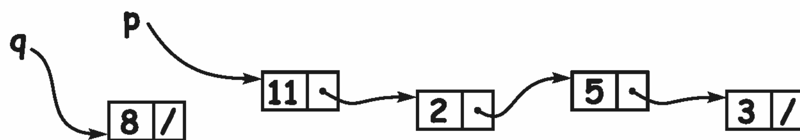
Funkcija `malloc` vrne kazalec `void`, zato moramo vrnjen kazalec pretvoriti v kazalec na strukturo `struct element`, preden ga priredimo spremenljivki `q`. Kar smo naredili v zgornjem stavku, simbolično predstavlja spodnja slika:




Zaenkrat sta obe komponenti strukture še nedefinirani, ker jima še nismo priredili nobene vrednosti. Slednje naredimo z dvema prireditvenima stavkoma:

```
q->vrednost = 8;
q->naslednji = NULL;
```

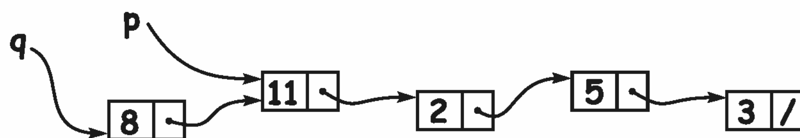
Zaenkrat smo kazalec na naslednji element nastavili kar na `NULL`, ker še ne vemo točno, kam naj bi kazal. Seveda ta prireditev ni nujno potrebna, je pa priporočljiva, saj so tako stvari bolj urejene (sicer ostaja kazalec nedefiniran). Trenutno stanje prikazuje slika:




Nadaljujmo z najpomembnejšim delom, to je z načinom, kako nov element povežemo v obstoječi seznam. Ker mora novi element postati prvi element v seznamu (dodajamo na začetek seznama!), mora trenutni prvi element seznama (na ta element kaže kazalec `p`) postati drugi element, to je element, ki sledi prvemu elementu. Torej moramo kazalec `q->naslednji` nastaviti tako, da kaže na isti element, na katerega kaže `p`:

```
q->naslednji = p;
```

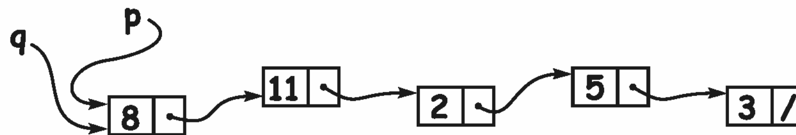
Sedaj tudi vidimo, zakaj začetna prireditev vrednosti kazalca `q->naslednji=NULL;` ni bila potrebna, saj smo vrednost `q->naslednji` takoj v naslednjem koraku že prepisali. Opisan korak prikazuje naslednja slika:



Do konca manjka le še en korak. Ker je `p` po definiciji kazalec na začetek seznama, moramo `p` prestaviti tako, da bo kazal na začetek spremenjenega seznama, torej na isti element, kot kaže kazalec `q`:

```
p = q;
```

Slednje lahko spet prikažemo simbolično:




Sedaj ko poznamo algoritem za dodajanje elementa, zlahka zapišemo funkcijo za dodajanje elementa na začetek seznama. Funkciji kot argumenta podamo kazalec na začetek seznama, v katerega želimo dodati element, in vrednost elementa, ki ga dodajamo. Ker se kazalec na začetek seznama ob operaciji dodajanja elementa spremeni (kaže na drug element, zato se spremeni naslov, ki je zapisan v spremenljivki `p`), mora funkcija vrniti nov kazalec na začetek seznama. Glavo funkcije lahko zapišemo takole:

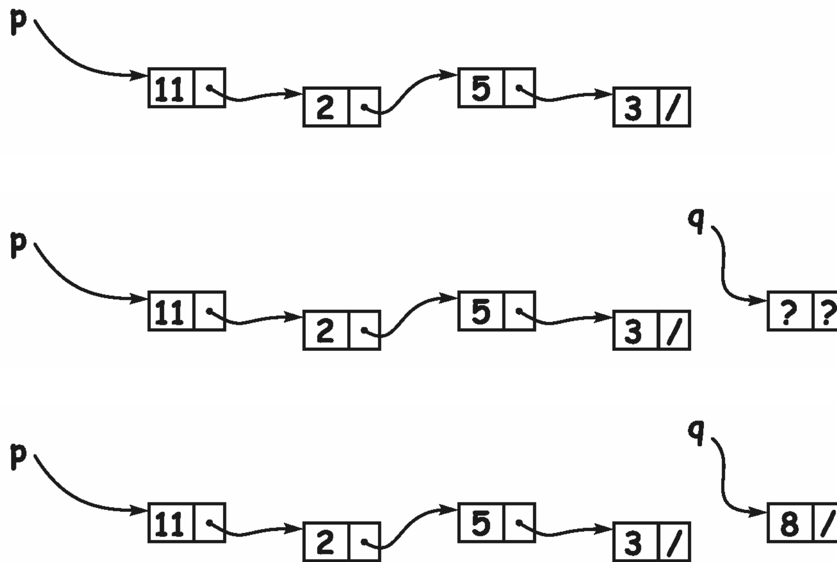
```
struct element *dodajZacetek(struct element *p, int vred)
```

Telo funkcije zajema celoten postopek, ki smo ga že opisali. Pri tem lahko kodo tudi malo skrajšamo tako, da opustimo nepotrebne prireditve (`q->naslednji=NULL;` in `p=q;`). Funkcija vrne kar kazalec `q`, ki na koncu kaže na začetek celega seznama. Celotna funkcija je naslednja:

```
struct element *dodajZacetek(struct element *p, int vred)
{
  struct element *q;

  q = (struct element*) malloc(sizeof(struct element));
  q->vrednost = vred;
  q->naslednji = p;
  return(q);
}
```

Pri preverjanju pravilnosti napisane kode, moramo preveriti njeno delovanje tudi v mejnih primerih. Razmislimo, kako se naša funkcija obnaša v primeru, ko je seznam prazen in torej dodajamo prvi element v seznam. To pomeni, da je `p` enak `NULL`, vrednost `p` pa se priredi spremenljivki `naslednji` prvega elementa (`q->naslednji`), ki tako tudi postane `NULL`. Funkcija na koncu vrne kazalec na novo ustvarjeni element `q`. Cel postopek torej poteka pravilno in v skladu s pričakovanji, kar potrjuje pravilnost naše kode.

Kot vidimo, je dodajanje elementa na začetek seznama precej enostavno. Malo več dela pa imamo pri dodajanju elementa na konec seznama. Prvi del (ustvarjanje novega elementa) je enak kot pri dodajanju na začetek, razlika je le v sami povezavi elementa v obstoječi seznam. Prve korake že poznamo in jih ne bomo ponovno opisovali. Predstavimo jih le simbolično na naslednjih slikah:




Tu je stavek `q->naslednji=NULL;` nujno potreben, saj bo element `q` po dodajanju zadnji element seznama, torej kazalec na naslednji element ne kaže nikamor.

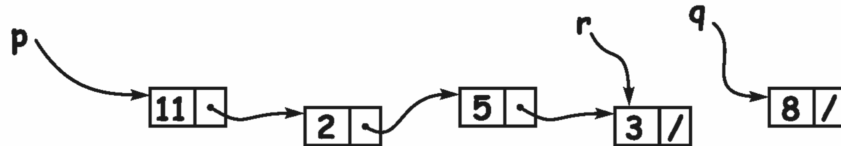
Za povezavo novo ustvarjenega elementa v obstoječi seznam potrebujemo še kazalec na zadnji element v seznamu (imenujmo ga `r`), saj novi element pride za zadnji element v seznam. In kako poiščemo zadnji element seznama? Ker je naša edina referenca le kazalec na začetek seznama `p`, moramo začeti s tem kazalcem. Od `p` se sprehodimo preko vseh elementov do zadnjega elementa v seznamu, torej do tistega, katerega kazalec na naslednji element ne kaže nikamor. Z drugimi besedami bi to lahko zapisali:

```
r = p;
while( r->naslednji!=NULL )
  r = r->naslednji;
```

Ko se zanka konča, kaže kazalec `r` na zadnji element seznama, saj takrat velja, da je kazalec `r->naslednji` enak `NULL` (to je pogoj za izstop iz `while` zanke). Zgornja zanka ne deluje pravilno (povzroči napako) v enem primeru: ko je seznam prazen. Takrat je `p` enako `NULL`, torej tudi `r` postane `NULL` in do komponente `r->naslednji` ne smemo dostopati. Zato se moramo pred vstopom v zanko prepričati, da seznam `p` ni prazen. Primer praznega seznama pa obravnavamo posebej:

```
if (p == NULL)
  // posebna obravnava dodajanja prvega elementa v seznam
else
  // sprehod preko seznama do zadnjega elementa
```

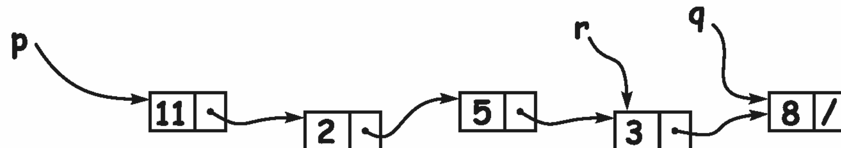
Naše tri kazalce `p`, `q` in `r` po izteku zanke prikazuje naslednja slika:




Kaj še manjka? Odločilni zadnji korak: povezava seznama z novim elementom. Zadnji element seznama, na katerega kaže `r`, ne sme biti več zadnji element, ampak mora imeti enega naslednika, to je element, na katerega kaže `q`. Narediti moramo naslednjo prevezavo:

```
r->naslednji = q;
```

Simbolično ta zadnji korak prikazuje spodnja slika:




Kot vidimo, kazalec `p` še vedno ostaja kazalec na začetek seznama.

Kot smo že omenili, moramo v primeru praznega seznama (`p` je enak `NULL`) dodajanje elementa posebej obravnavati. V tem primeru je novo ustvarjen element edini element seznama, torej lahko kot kazalec na začetek seznama vrnemo kar kazalec `q`.

Cela funkcija dodajanja elementa na konec seznama je naslednja:

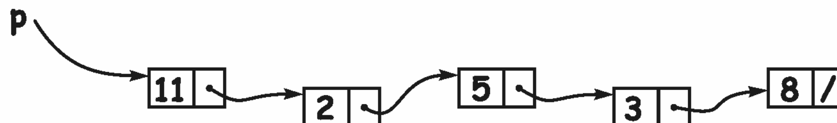
```
struct element *dodajKonec(struct element *p, int vred)
{
  struct element *q, *r;

  q = (struct element*) malloc(sizeof(struct element));
  q->vrednost = vred;
  q->naslednji = NULL;
  if (p == NULL)
    return(q);
  r = p;  // p ni prazen seznam
  while( r->naslednji!=NULL )
    r = r->naslednji;
  r->naslednji = q;
```

```
  return(p);
}
```

### *Brisanje elementa iz seznama*

Za začetek napišimo funkcijo, ki iz seznama zbriše en element s podano vrednostjo. Ker je v seznamu lahko več elementov z isto vrednostjo, naj funkcija zbriše prvi tak element, ki ga najde v seznamu. Če elementa v seznamu ni, funkcija vrne nespremenjen seznam.

Poglejmo si brisanje elementa tudi na primeru. Recimo, da imamo seznam z elementi 11, 2, 5, 3 in 8 po vrsti:



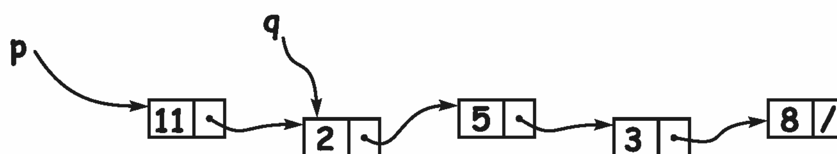

Iz tega seznama bi želeli zbrisati element, ki ima vrednost 5 (v našem primeru tretji element seznama).

Postopek lahko razdelimo na dva dela: najprej moramo element v seznamu poiskati (če sploh obstaja), potem pa ga moramo še zbrisati (če smo ga seveda pred tem našli).

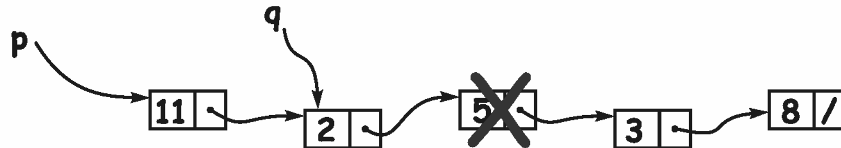
Postopek iskanja elementa s podano vrednostjo že poznamo. Zaradi praktičnosti pa bomo raje poiskali element, ki je v seznamu pred elementom, ki ga želimo zbrisati. Zakaj, bomo videli malo kasneje. S pomožnim kazalcem `q` se sprehodimo preko seznama, dokler vrednost naslednjega elementa ni enaka iskani vrednosti. Seveda moramo pri tem paziti, da prej ne pridemo do konca seznama, kar se nam lahko zgodi, kadar iskanega elementa ni v seznamu.

```
q = p;
while( (q->naslednji!=NULL) && (q->naslednji->vrednost!=vred) )
  q = q->naslednji;
```

To lahko ponazorimo tudi s sliko:

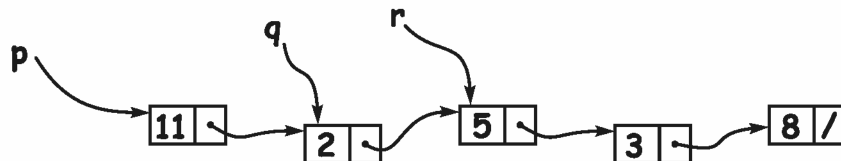


Ko se zanka zaključi, kazalec `q->naslednji` kaže na element, ki ga želimo brisati (če je ta element v seznamu, sicer pa ima vrednost `NULL`):




Priredimo to vrednost pomožnemu kazalcu `r`:

```
r = q->naslednji;
```




Če elementa nismo našli v seznamu, je `r` enako `NULL` in seznam ostane nespremenjen.

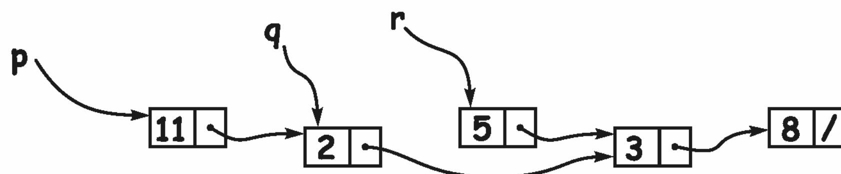
Sedaj, ko smo element našli (nanj kaže `r`), sledi drugi del, to je brisanje samega elementa iz seznama. To enostavno naredimo tako, da seznam prevežemo mimo tega elementa:

```
q->naslednji = r->naslednji;
```

kar lahko enakovredno zapišemo tudi (glej zgornjo sliko):

```
q->naslednji = q->naslednji->naslednji;
```

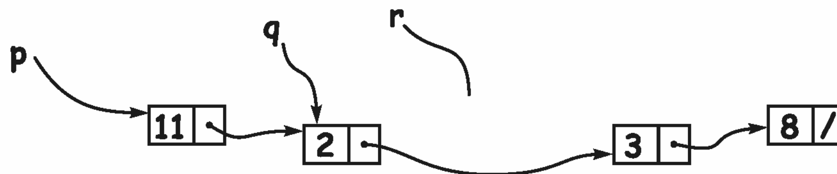
Sedaj tudi vidimo, zakaj smo iskali element pred brisanim elementom (nanj kaže `q`); brez te reference namreč ne bi mogli narediti prevezave seznama:




Na koncu moramo le še sprostiti pomnilnik, ki ga zaseda brisani element. Ne pozabimo, da smo pomnilnik eksplicitno rezervirali z ukazom `malloc`, zato ga moramo tudi eksplicitno sprostiti z ukazom `free` (zato smo tudi potrebovali kazalec na element `r`):

```
free(r);
```

Po klicu funkcije `free` element z vrednostjo 5 ni obstaja več, vrednost kazalca `r` pa je enaka `NULL`, kar smo na sliki prikazali, kot da ne kaže nikamor:




V opisanem postopku smo predpostavili, da seznam `p` ni prazen, torej moramo prazen seznam obravnavati posebej:

```
if( p==NULL )
  return(p);
```

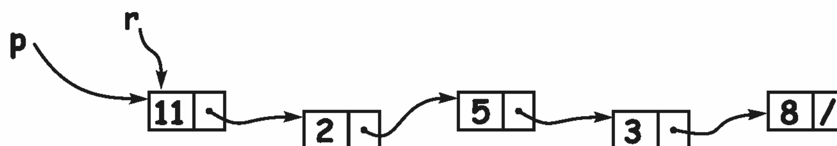
Tudi primer, ko je element, ki ga želimo zbrisati, prvi element seznama, zahteva posebno obravnavo. Recimo, da želimo v našem prvotnem seznamu zbrisati element z vrednostjo 11. Najprej torej preverimo, ali je vrednost prvega elementa enaka vrednosti, ki jo brišemo, in v primeru izpolnjenega pogoja ustrezno ukrepamo:

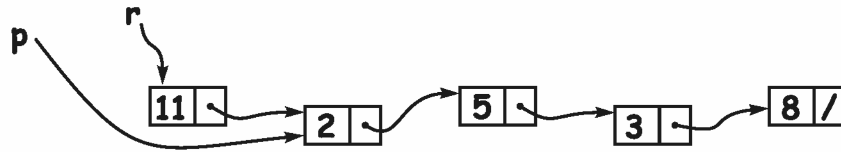
```
if( p->vrednost==vred ) {
  // brišemo prvi element seznama
}
```

Kaj pa moramo narediti v primeru brisanja prvega elementa seznama? Podobno kot prej, si moramo zapomniti ta element, da lahko kasneje sprostimo pomnilnik, ki ga zaseda, torej postavimo kazalec `r`, da kaže nanj. Sledi prevezava, kjer nastavimo `p`, da kaže na drugi element seznama. Po sprostitvi prostora s pomočjo `free` smo končali. Kazalec `p` kaže na nov seznam brez prvega elementa.

```
r = p;
p = p->naslednji;
free(r);
return(p);
```

Prva dva stavka postopka prikazujeta spodnji sliki:






Za konec le še združimo vso kodo v naslednjo funkcijo:

```
struct element *brisi(struct element *p, int vred)
{
  struct element *q, *r;

  if( p==NULL )
    return(p);
  if( p->vrednost==vred ) {
    r = p;
    p = p->naslednji;
    free(r);
    return(p);
  }
  q = p;
  while( (q->naslednji!=NULL)&&(q->naslednji->vrednost!=vred) )
    q = q->naslednji;
  r = q->naslednji;
  if( r!=NULL ) {
    q->naslednji = r->naslednji;
    free(r);
  }
  return(p);
}
```

***Brisanje vseh pojavitev elementa iz seznama***

V prejšnjem razdelku smo si ogledali brisanje enega elementa iz seznama. Seveda se lahko v seznamu ista vrednost tudi večkrat ponovi (imamo več elementov z isto vrednostjo). Včasih bi želeli zbrisati vse pojavitve neke vrednosti v seznamu. Zato napišimo še funkcijo `brisiVse`, ki zbriše iz seznama vse elemente, katerih vrednost je enaka podani vrednosti.

Tudi ta funkcija ima dva dela: najprej preverimo, ali moramo brisati elemente na začetku seznama. Dokler je vrednost prvega elementa seznama enaka podani vrednosti, toliko časa brišemo prvi element in kazalec na začetek seznama prestavljamo na naslednji element. Pri tem moramo seveda paziti, da prej ne pridemo do konca seznama.

```
while( (p!=NULL) && (p->vrednost== vred) ) {
  r = p;
  p = p->naslednji;
  free(r);
}
```

Drugi del pa pregleduje vse elemente seznama do zadnjega elementa v seznamu (s kazalcem `q` se sprehodimo preko seznama) in sproti preverja, ali je vrednost elementa enaka podani vrednosti. Kadar najde element, za katerega je ta pogoj izpolnjen, ga enostavno zbriše iz seznama:

```
q = p;
while( q->naslednji!=NULL ) {
  if( q->naslednji->vrednost == vred) {
    r = q->naslednji;
    q->naslednji = r->naslednji;
    free(r);
  }
  else
    q = q->naslednji;
}
```

Ko smo pregledali celoten seznam in zbrisali vse elemente, katerih vrednost se ujema s podano vrednostjo, vrnemo kazalec na začetek (novega) seznama `p`. Funkcija ima naslednjo kodo:

```
struct element *brisiVse(struct element *p, int vred)
{
  struct element *q, *r;

  while( (p!=NULL) && (p->vrednost== vred) ) {
    r = p;
    p = p->naslednji;
    free(r);
  }
  if (p == NULL )
    return(p);
  q = p;
  while( q->naslednji!=NULL ) {
    if( q->naslednji->vrednost == vred) {
      r = q->naslednji;
      q->naslednji = r->naslednji;
      free(r);
    }
    else
      q = q->naslednji;
  }
  return(p);
}
```

### *Brisanje celega seznama*

Pogosto želimo zbrisati tudi vse elemente seznama, ne glede na njihovo vrednost, torej izprazniti seznam. Če je `p` kazalec na začetek seznama, bi to najenostavneje naredili tako, da kazalec `p` postavimo na `NULL`:

```
p = NULL;
```

Na prvi pogled je rešitev dobra, toda pozor! Elementi seznama še vedno obstajajo in zasedajo prostor v pomnilniku. Še slabša novica pa je, da smo s postavitvijo `p` na `NULL` izgubili vsakršno referenco na te elemente seznama, tako da ne moremo niti do njih dostopati (in jih uporabljati) niti sprostiti prostora, ki ga zasedajo.

Vidimo, da izbris celega seznama le ni tako trivialna operacija, kot bi se nam lahko zdelo na prvi pogled. Vendar funkcija `izprazni` vseeno ni preveč zapletena. Vse, kar moramo narediti, je, da se sprehodimo preko seznama in sproti sproščamo prostor za vsakega od elementov seznama. Na koncu je kazalec `p` enak `NULL`, kar funkcija tudi vrne kot rezultat:

```
struct element *izprazni(struct element *p)
{
  struct element *r;

  while( p!=NULL ) {
    r = p;
    p = p->naslednji;
    free(r);
  }
  return(p);
}
```

Ta funkcija je še posebej uporabna ob koncu dela s seznamom, ko spremenljivke `zacetek` ne potrebujemo več. Preden se program zaključi, moramo namreč ves eksplicitno rezerviran prostor v pomnilniku tudi eksplicitno sprostiti, kar lahko sedaj naredimo enostavno s klicem funkcije `izprazni(zacetek)`.

### *Sestavimo program*

Če vse povedano o kazalčnih seznamih združimo in dodamo še `main` funkcijo, ki prikazuje uporabo seznama, dobimo naslednji program, ki demonstrira delo z enosmernim neurejenim linearnim kazalčnim seznamom (`seznam.c`).

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


struct element
{
  int vrednost;
  struct element *naslednji;
};
```

```
void izpisi(struct element *p)
{
  while( p!=NULL ) {
    printf("%d  ", p->vrednost);
    p = p->naslednji;
  }
  printf("\n");
}


int poisci(struct element *p, int vred)
{
  while( (p!=NULL) && (p->vrednost!=vred) )
    p = p->naslednji;
  if(p!=NULL)
    return(1);
  else
    return(0);
}


struct element *dodajZacetek(struct element *p, int vred)
{
  struct element *q;

  q = (struct element*) malloc(sizeof(struct element));
  q->vrednost = vred;
  q->naslednji = p;
  return(q);
}


struct element *dodajKonec(struct element *p, int vred)
{
  struct element *q, *r;

  q = (struct element*) malloc(sizeof(struct element));
  q->vrednost = vred;
  q->naslednji = NULL;
  if (p == NULL)
    return(q);
  r = p;  // p ni prazen seznam
  while( r->naslednji!=NULL )
    r = r->naslednji;
  r->naslednji = q;
  return(p);
}


struct element *brisi(struct element *p, int vred)
{
  struct element *q, *r;

  if( p==NULL )
    return(p);
```

```
  if( p->vrednost==vred ) {
    r = p;
    p = p->naslednji;
    free(r);
    return(p);
  }
  q = p;
  while( (q->naslednji!=NULL)&&(q->naslednji->vrednost!=vred) )
    q = q->naslednji;
  r = q->naslednji;
  if( r!=NULL ) {
    q->naslednji = r->naslednji;
    free(r);
  }
  return(p);
}


struct element *brisiVse(struct element *p, int vred)
{
  struct element *q, *r;

  while( (p!=NULL) && (p->vrednost== vred) ) {
    r = p;
    p = p->naslednji;
    free(r);
  }
  if (p == NULL )
    return(p);
  q = p;
  while( q->naslednji!=NULL ) {
    if( q->naslednji->vrednost == vred) {
      r = q->naslednji;
      q->naslednji = r->naslednji;
      free(r);
    }
    else
      q = q->naslednji;
  }
  return(p);
}


struct element *izprazni(struct element *p)
{
  struct element *r;

  while( p!=NULL ) {
    r = p;
    p = p->naslednji;
    free(r);
  }
  return(p);
}
```

```
main() {
  struct element *zacetek;
  int i;

  zacetek = NULL;
  for(i=1; i<6; i++) {
    zacetek = dodajZacetek(zacetek,i);
    izpisi(zacetek);
  }
  for(i=1; i<6; i++) {
    zacetek = dodajKonec(zacetek,i);
    izpisi(zacetek);
  }
  zacetek = dodajKonec(zacetek,0);
  izpisi(zacetek);
  zacetek = brisi(zacetek,5);
  izpisi(zacetek);
  zacetek = brisiVse(zacetek,3);
  izpisi(zacetek);
  if( poisci(zacetek,0) == 1)
    printf("Element 0 je v seznamu.\n");
  else
    printf("Elementa 0 ni v seznamu.\n");
  zacetek = brisiVse(zacetek,0);
  izpisi(zacetek);
  if( poisci(zacetek,0) == 1)
    printf("Element 0 je v seznamu.\n");
  else
    printf("Elementa 0 ni v seznamu.\n");
  zacetek = izprazni(zacetek);
  printf("Na koncu je seznaam prazen:\n");
  izpisi(zacetek);
}
```

***Vračanje vrednosti funkcije preko argumenta***

Pri programiranju imamo vedno več načinov, na katere lahko rešimo en in isti problem. Nekatere rešitve so boljše (na primer enostavnejše, bolj pregledne, hitrejše ali pa porabijo manj prostora), druge slabše, nekatere pa so bolj ali manj enakovredne. Med slednje spada tudi način vračanja vrednosti kazalca na začetek seznama, ki ga bomo opisali v nadaljevanju.

V predhodnih razdelkih smo napisali kar nekaj funkcij, ki so kot enega izmed argumentov prejele kazalec na začetek seznama in tudi vrnile kazalec na začetek seznama (funkcije za brisanje in dodajanje elementa ter izpraznitev seznama). V vseh teh funkcijah se je (lahko) spremenila vrednost kazalca na začetek seznama, zato so morale funkcije novo vrednost vrniti, torej posredovati v okolje, iz katerega je bila funkcija poklicana.

Novo vrednost kazalca pa bi lahko vrnili tudi preko samega argumenta funkcije. Kot smo zapisali že v razdelku *Kazalci in argumenti funkcij* v četrtem poglavju, moramo v funkciji uporabiti prenos argumentov po referenci, kadar želimo, da se spremembe vrednosti argumentov ohranijo tudi izven klicane funkcije. V tem primeru so argumenti le reference na ustrezne spremenljivke, torej kazalci na te spremenljivke.

Vzemimo za primer funkcijo `dodajZacetek`. Funkcija prejme kazalec na začetek seznama kot prvi argument (drugi argument je vrednost elementa, ki ga dodajamo), doda na začetek seznama en element (torej se spremeni kazalec na začetek seznama, ki sedaj kaže na novo dodani element!) in vrne kazalec na začetek tega spremenjenega seznama kot rezultat. Glava funkcije je naslednja:

```
struct element *dodajZacetek(struct element *p, int vred)
```

Lahko bi funkcijo napisali tudi tako, da bi spremenjeno vrednost kazalca na začetek seznama funkcija vrnila preko samega argumenta. V tem primeru moramo argument podati kot referenco na spremenljivko, torej kot kazalec na kazalec na začetek seznama, kar zapišemo:

```
void dodajZacetek1(struct element **p, int vred)
```

Sama funkcija je `void`, saj rezultat vrne preko argumenta, drugi argument (vrednost elementa, ki ga dodajamo v seznam) pa ostaja nespremenjen.

Ker je podan `p` v tem primeru naslov kazalca na začetek seznama, je `(*p)` kazalec na začetek seznama. Telo funkcije moramo zato ustrezno popraviti, na koncu mora `*p` tudi kazati na začetek novega seznama:

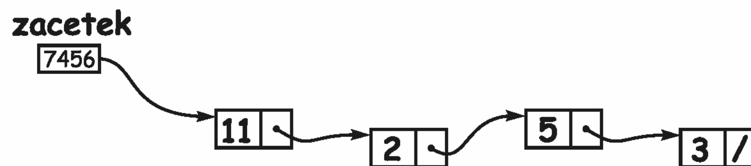
```
void dodajZacetek1(struct element **p, int vred)
{
  struct element *q;

  q = (struct element*) malloc(sizeof(struct element));
  q->vrednost = vred;
  q->naslednji = *p;
  *p = q;
}
```

Zaradi privzete prioritete operatorjev smo lahko `*p` zapisali brez oklepajev. Vendar pozor! Oklepaji so nujno potrebni pri dostopu do posameznih komponent elementa `(*p)->vrednost` in `(*p)->naslednji`.

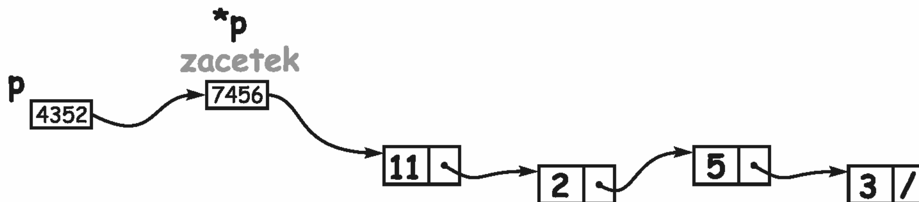
Ker smo spremenili glavo funkcije, to funkcijo tudi pokličemo drugače. Če je spremenljivka `zacetek` kazalec na začetek seznama, potem moramo funkciji `dodajZacetek1` kot argument podati naslov tega kazalca, to je `&zacetek`. Drugi argument pa je vrednost, ki jo dodajamo v seznam, naj bo le-ta v našem primeru 8.

```
struct element *zacetek;
...
dodajZacetek1(&zacetek, 8);
```

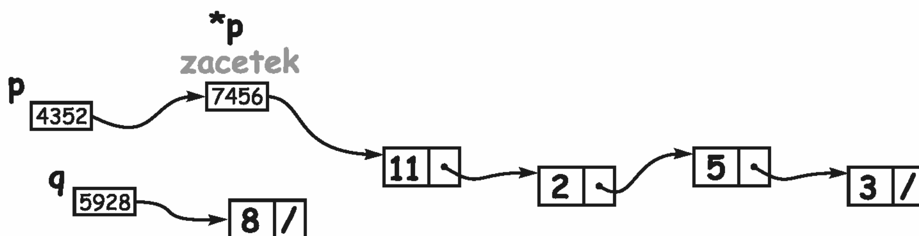
Poglejmo si to še na sliki, na primeru našega prejšnjega seznama, kateremu dodamo na začetek element z vrednostjo 8. Na sliki smo začetek seznama označili kot spremenljivko, v kateri je zapisan naslov prvega elementa seznama, to je (izmišljena vrednost) 7456.



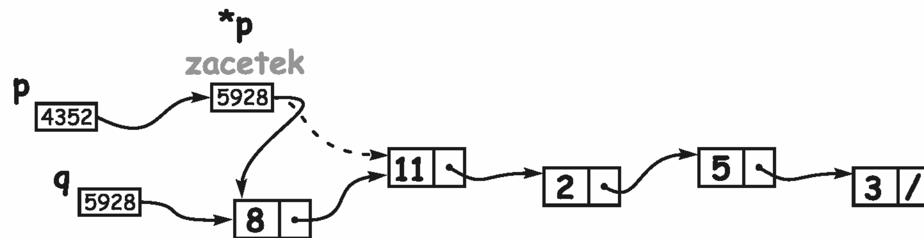

Ob klicu funkcije `dodajZacetek1` se naslov spremenljivke `zacetek` (recimo, da je to 4352) prenese kot argument v spremenljivko `p`. Tako `p` kaže na `zacetek` oziroma, povedano z drugimi besedami, kaže na kazalec na začetek seznama. Vrednost spremenljivke `p` je torej 4352. V sami funkciji spremenljivke `zacetek` ni, zato je na sliki prikazana sivo. Na isto lokacijo kot `zacetek` pa se nanaša tudi `*p`.



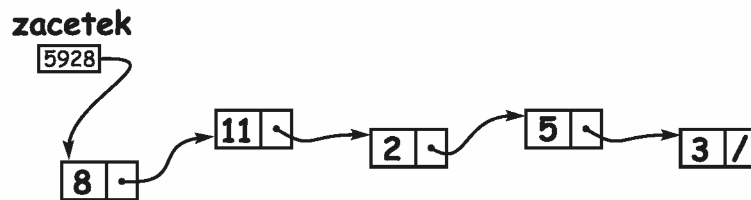

V funkciji `dodajZacetek1` najprej ustvarimo nov element, na katerega kaže kazalec `q`. Recimo, da se ta element nahaja na naslovu 5928, torej je to tudi vrednost `q`.




V naslednjem koraku novi element povežemo v seznam. S tem se spremeni tudi vrednost kazalca na začetek seznama `*p`, ki sedaj kaže na novo ustvarjeni element, torej se `*p` priredi vrednost `q`, to je 5928.




Spremenljivke `p` v funkciji nismo nič spreminjali, služila nam je le kot referenca na kazalec na začetek seznama (`*p`). Ko se funkcija konča, spremenljivki `p` in `q` ne obstajata več (zato smo ju odstranili s spodnje slike). Kot vidimo, pa se je spremenljivki `zacetek` vrednost spremenila: prej je hranila naslov 4352, sedaj pa 5928. Tako spremenljivka `zacetek` sedaj pravilno kaže na začetek spremenjenega seznama.



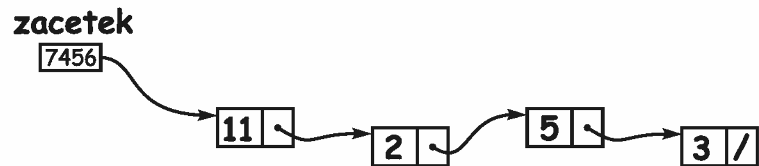

Da bi bolje ponazorili razliko med obema načinoma prenosa rezultata funkcije, si poglejmo še, kaj se zgodi ob klicu funkcije za dodajanje elementa `dodajZacetek`, ki smo jo prvo napisali. Za osvežitev spomina je tukaj ponovno njena koda:

```
struct element *dodajZacetek(struct element *p, int vred)
{
  struct element *q;

  q = (struct element*) malloc(sizeof(struct element));
  q->vrednost = vred;
  q->naslednji = p;
  return(q);
}
```

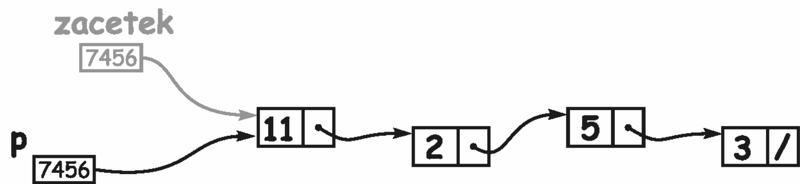
in primer klica funkcije:

```
struct element *zacetek;
...
zacetek = dodajZacetek(zacetek, 8);
```

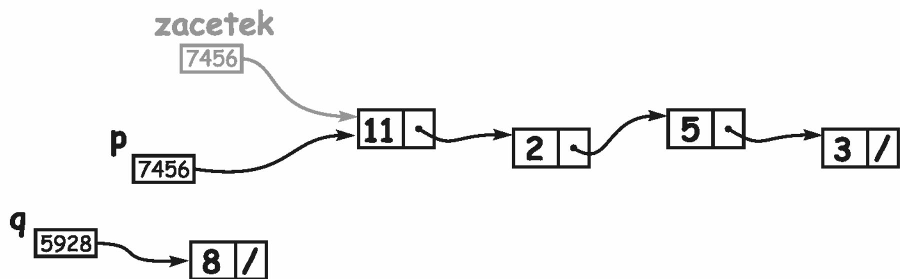
Dogajanje spet ponazorimo slikovno na primeru istega seznama, na katerega kaže spremenljivka `zacetek`, ki hrani naslov prvega elementa seznama, to je 7456:



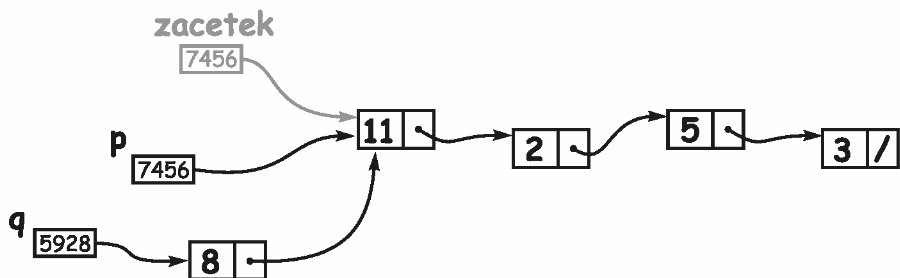

Ob klicu funkcije smo podali dva argumenta (oba po vrednosti): prvi je kazalec na začetek seznama, drugi pa vrednost, ki jo vstavljamo. V funkciji `dodajZacetek` tako spremenljivka `p` dobi vrednost spremenljivke `zacetek`, spremenljivka `vred` pa vrednost 8 (slednja nas v tem primeru pravzaprav ne zanima). Spremenljivka `p` je torej nova spremenljivka, ki pa ima isto vrednost kot spremenljivka `zacetek`, torej kaže na začetek seznama. Na sliki smo spremenljivko `zacetek` posivili, saj v funkciji ni vidna.



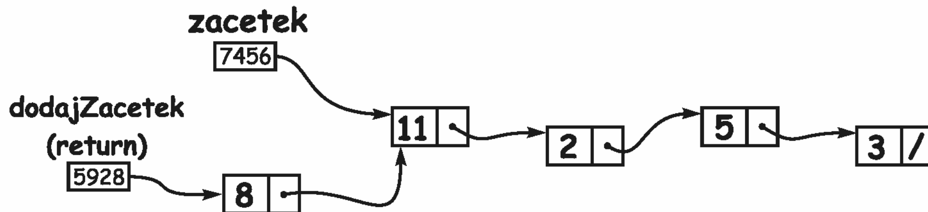

Sledi ustvarjanje novega elementa. V našem primeru je ta element na naslovu 5928 in nanj kaže kazalec `q`.



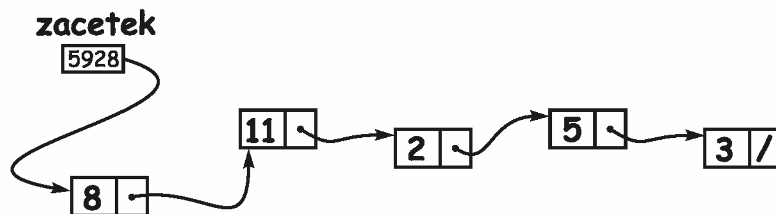

Ko nov element pravilno povežemo v seznam, je `q` kazalec na spremenjen seznam. Začetek seznama je sedaj na naslovu 5928, to pa je tudi vrednost, ki jo funkcija vrne s stavkom `return`.



Po koncu funkcije spremenljivki `p` in `q` ne obstajata več, spremenljivka `zacetek` pa je ohranila svojo vrednost in sedaj kaže na drugi element seznama.



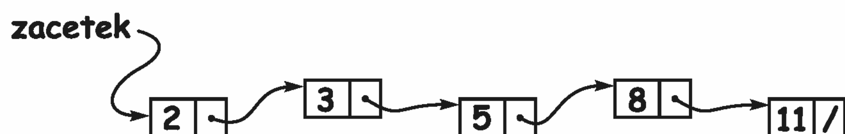

Vendar pa začetka seznama kljub temu nismo izgubili, saj je funkcija `dodajZacetek` vrnila naslov prvega elementa seznama, to vrednost pa potem priredimo spremenljivki `zacetek`, ki tako spet pravilno kaže na začetek seznama.




**Urejeni seznami**

Seznam je urejen, kadar si elementi seznama sledijo v določenem zaporedju. Vsak neurejen seznam lahko uredimo glede na podano relacijo med elementi in dobimo urejen seznam.

Tako lahko naš primer neurejenega seznama uredimo po vrednosti elementa od najmanjšega do največjega in dobimo naslednji seznam, kjer so elementi urejeni po velikosti:




Ker so si neurejeni in urejeni seznami zelo podobni (razlika je le v urejenosti seznama), sta funkciji za izpis seznama in brisanje celega seznama pri urejenem seznamu enaki kot smo ju zapisali že pri neurejenem seznamu. Funkciji iskanje elementa v seznamu in brisanje elementa iz seznama sta sicer podobni, a nam urejenost seznama omogoča, da kodo optimiziramo, zato ju bomo napisali na novo. Velika razlika pa je pri funkciji za dodajanje elementa v seznam, saj novega elementa ne dodamo več po naši izbiri na začetek ali na konec seznama, ampak ga moramo vstaviti na pravo mesto v seznamu.

### *Iskanje elementa v urejenem seznamu*

Ker je podan seznam urejen, nam pri iskanju določene vrednosti v seznamu ni potrebno pregledati celega seznama, temveč le do prvega elementa, ki ima vrednost večjo od iskane. Če iskane vrednosti nismo našli v pregledanem delu seznama, je sigurno ne bomo našli v preostanku seznama, saj imajo vsi elementi (zaradi urejenosti seznama) vrednosti večje od iskane. Pogoj zanke, s katero pregledujemo seznam, lahko zato spremenimo:

```
while( (p!=NULL) && (p->vrednost<vred) )
```

Zanka se torej lahko konča iz dveh razlogov: ali smo prišli do konca seznama ali pa je vrednost elementa večja ali enaka iskani vrednosti. Če smo element našli, `p` kaže na ta element. Če pa elementa nismo našli, je `p` enak `NULL` ali pa kaže na element z večjo vrednostjo od iskane. Zato moramo popraviti pogoj `if` stavka, v katerem vračamo vrednost funkcije:

```
if( (p!=NULL) && (p->vrednost==vred) )
```

Funkcija v celoti izgleda takole:

```
int poisci(struct element *p, int vred)
{
  while( (p!=NULL) && (p->vrednost<vred) )
    p = p->naslednji;
  if( (p!=NULL) && (p->vrednost==vred) )
    return(1);
  else
    return(0);
}
```

### *Brisanje elementa iz urejenega seznama*

Pri brisanju elementa iz seznama je prvi korak iskanje tega elementa v seznamu, zato lahko tudi funkcijo brisanja pri urejenem seznamu poenostavimo. Ko element najdemo v seznamu, ga izbrišemo na enak način, kot pri neurejenem seznamu. Celotna koda funkcije je naslednja:

```
struct element *brisi(struct element *p, int vred)
{
  struct element *q, *r;

  if( p==NULL )
    return(p);
  if( p->vrednost==vred ) {
    // briši prvi element seznama
    r = p;
```

```
    p = p->naslednji;
    free(r);
    return(p);
  }
  // poišči element v seznamu in ga zbriši
  q = p;
  while( (q->naslednji!=NULL) && (q->naslednji->vrednost<vred) )
    q = q->naslednji;
  r = q->naslednji;
  if( (r!=NULL) && (r->vrednost==vred) ) {
    q->naslednji = r->naslednji;
    free(r);
  }
  return(p);
}
```

***Dodajanje elementa v urejen seznam***

Ko dodajamo nov element v urejen seznam, moramo paziti, da ga postavimo na pravo mesto v seznamu. To je lahko na začetku, na koncu ali pa nekje v sredini, odvisno pač od vrednosti elementov v seznamu in od vrednosti elementa, ki ga dodajamo. Funkcija je naslednja:

```
struct element *dodaj(struct element *p, int vred)
{
  struct element *q, *r;

  q = (struct element*) malloc(sizeof(struct element));
  q->vrednost = vred;
  q->naslednji = NULL;
  if( p==NULL )
    return(q);
  if( p->vrednost>=vred ) {  // dodaj element na začetek seznama
    q->naslednji = p;
    return(q);
  }
  r = p;  // poišči pravo mesto za nov element
  while( (r->naslednji!=NULL) && (r->naslednji->vrednost<vred) )
    r = r->naslednji;
  q->naslednji = r->naslednji;
  r->naslednji = q;
  return(p);
}
```

***Sestavimo program***

Podobno kot smo napisali program, ki je prikazoval delo z neurejenim seznamom, lahko sedaj sestavimo tudi program, ki demonstrira delo z enosmernim urejenim linearnim

kazalčnim seznamom (`urejen.c`). Tudi tu smo že opisanim funkcijam dodali še `main` funkcijo, ki prikazuje uporabo teh funkcij.

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


struct element
{
  int vrednost;
  struct element *naslednji;
};


void izpisi(struct element *p)
{
  while( p!=NULL ) {
    printf("%d  ", p->vrednost);
    p = p->naslednji;
  }
  printf("\n");
}


int poisci(struct element *p, int vred)
{
  while( (p!=NULL) && (p->vrednost<vred) )
    p = p->naslednji;
  if( (p!=NULL) && (p->vrednost==vred) )
    return(1);
  else
    return(0);
}


struct element *dodaj(struct element *p, int vred)
{
  struct element *q, *r;

  q = (struct element*) malloc(sizeof(struct element));
  q->vrednost = vred;
  q->naslednji = NULL;
  if( p==NULL )
    return(q);
  if( p->vrednost>=vred ) {
    q->naslednji = p;
    return(q);
  }
  r = p;
  while( (r->naslednji!=NULL) && (r->naslednji->vrednost<vred) )
    r = r->naslednji;
  q->naslednji = r->naslednji;
  r->naslednji = q;
  return(p);
```

```
}


struct element *brisi(struct element *p, int vred)
{
  struct element *q, *r;

  if( p==NULL )
    return(p);
  if( p->vrednost==vred ) {
    r = p;
    p = p->naslednji;
    free(r);
    return(p);
  }
  q = p;
  while( (q->naslednji!=NULL) && (q->naslednji->vrednost<vred) )
    q = q->naslednji;
  r = q->naslednji;
  if( (r!=NULL) && (r->vrednost==vred) ) {
    q->naslednji = r->naslednji;
    free(r);
  }
  return(p);
}


struct element *izprazni(struct element *p)
{
  struct element *r;

  while( p!=NULL ) {
    r = p;
    p = p->naslednji;
    free(r);
  }
  return(p);
}


main() {
  struct element *zacetek;
  int i;

  zacetek = NULL;
  for(i=1; i<6; i++) {
    zacetek = dodaj(zacetek,2*i);
    izpisi(zacetek);
  }
  zacetek = dodaj(zacetek,1);
  izpisi(zacetek);
  zacetek = dodaj(zacetek,10);
  izpisi(zacetek);
  zacetek = dodaj(zacetek,20);
  izpisi(zacetek);
```

```
    zacetek = dodaj(zacetek,5);
    izpisi(zacetek);
    zacetek = brisi(zacetek,1);
    izpisi(zacetek);
    zacetek = brisi(zacetek,20);
    izpisi(zacetek);
    zacetek = brisi(zacetek,6);
    izpisi(zacetek);
    if( poisci(zacetek,2) == 1)
      printf("Element 2 je v seznamu.\n");
    else
      printf("Elementa 2 ni v seznamu.\n");
    zacetek = brisi(zacetek,2);
    izpisi(zacetek);
    if( poisci(zacetek,2) == 1)
      printf("Element 2 je v seznamu.\n");
    else
      printf("Elementa 2 ni v seznamu.\n");
    zacetek = izprazni(zacetek);
    printf("Na koncu je seznaam prazen:\n");
    izpisi(zacetek);
  }
```

### Binarna drevesa

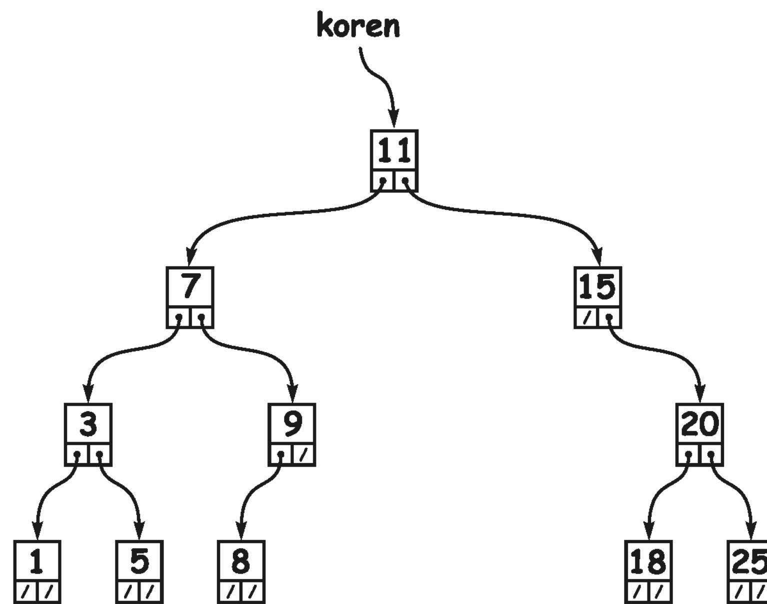
Drevo je hierarhična struktura na zbirki elementov. Posameznim elementom v drevesu pravimo vozlišča, eno od njih je korensko vozlišče (koren drevesa). Vozlišča so med seboj v relaciji oče/sin, torej ima vozlišče lahko naslednike in enega predhodnika. Vozlišča, ki nimajo naslednikov, imenujemo listi drevesa. Vsa vozlišča, razen korenskega vozlišča, imajo predhodnike.

V binarnem drevesu ima vsako vozlišče največ dva naslednika. Tako strukturo najlažje definiramo rekurzivno: drevo sestavljajo koren ter levo in desno poddrevo, ki sta tudi drevesi. Seveda je drevo lahko tudi prazno (v tem primeru je koren enak `NULL`). Tako rekurzivno definicijo drevesa prikazuje slika:

Če so vozlišča binarnega drevesa urejena, strukturo imenujemo urejeno binarno drevo. S takimi drevesi se bomo ukvarjali v nadaljevanju.

Urejeno binarno drevo je torej podatkovna struktura, kjer so elementi (vozlišča) urejeni tako, da so vrednosti elementov v levem poddrevesu manjše od vrednosti korena, vrednosti elementov v desnem poddrevesu pa so večje ali enake vrednosti korena.

Primer urejenega binarnega drevesa prikazuje spodnja slika. Drevo smo sestavili z zaporednim dodajanjem elementov 11, 7, 3, 15, 9, 8, 20, 18, 1, 25 in 5.




V nadaljevanju si bomo pogledali najosnovnejše operacije nad urejenim binarnim drevesom. Ker je drevo definirano rekurzivno, tudi operacije nad drevesom najlažje opišemo s pomočjo rekurzije.

### *Deklaracija vozlišča drevesa*

Vozlišče drevesa (en element) definiramo s pomočjo strukture, ki ima najmanj tri komponente. Hraniti mora vrednost elementa ter kazalca na levo in desno poddrevo, ki sta prav tako drevesi:

```
struct vozlisce
{
  int vrednost;
  struct vozlisce *levo;
  struct vozlisce *desno;
};
```

Kazalec na začetek drevesa ponavadi imenujemo `koren` in ga deklariramo z naslednjim stavkom:

```
struct element *koren;
```

***Iskanje vrednosti v drevesu***

Funkcijo za iskanje določene vrednosti v drevesu zlahka zapišemo, če upoštevamo rekurzivno definicijo drevesa in pravilo urejenosti. Če je drevo prazno, elementa ni v drevesu. To je tudi prvi izstopni pogoj iz rekurzije. Če je vrednost korena enaka iskani vrednosti, smo element našli (drugi izstopni pogoj rekurzije). Sicer pa nadaljujemo iskanje na enak način v poddrevesih: če je iskana vrednost manjša od vrednosti korena, preiščemo levo poddrevo, sicer pa desno poddrevo.

```
int poisci(struct vozlisce *koren, int vred)
{
  if(koren == NULL)
    return(0);  // iskane vrednosti ni v drevesu
  if(vred == koren->vrednost)
    return(1);  // iskano vrednost smo našli
  if(vred < koren->vrednost)
    return(poisci(koren->levo, vred));  // išči v levem poddrev.
  if(vred > koren->vrednost)
    return(poisci(koren->desno, vred));  // išči v desnem
}
```

***Izpis drevesa***

Binarno drevo lahko izpišemo na tri načine, odvisno od vrstnega reda izpisa korena in obeh poddreves. Če najprej izpišemo koren, zatem celo levo poddrevo in na koncu celo desno poddrevo, je to *prefiksna oblika* izpisa. *Infiksna oblika* izpisa najprej izpiše levo poddrevo, sledi koren in na koncu desno poddrevo. Zaporedje levo poddrevo, desno poddrevo in koren pa imenujemo *postfiksna oblika*.

Izpis drevesa na zgornji sliki je v posameznih oblikah naslednji:

Prefiksna oblika: `11 7 3 1 5 9 8 15 20 18 25`

Infiksna oblika: `1 3 5 7 8 9 11 15 18 20 25`

Postfiksna oblika: `1 5 3 8 9 7 18 25 20 15 11`

Funkcije za vse tri načine so podobne, razlikujejo se le v vrstnem redu stavkov za izpis korena in poddreves. Vidimo, da nam infiksna oblika izpiše elemente drevesa po velikosti od najmanjšega do največjega (tako kot je urejeno drevo).

Infiksna oblika izpisa:

```
void izpisiIn(struct vozlisce *koren)
{
  if( koren==NULL )
    return;
  izpisiIn(koren->levo);
  printf("%d  ", koren->vrednost);
  izpisiIn(koren->desno);
}
```

Prefiksna oblika izpisa:

```
void izpisiPre(struct vozlisce *koren)
{
  if( koren==NULL )
    return;
  printf("%d  ", koren->vrednost);
  izpisiPre(koren->levo);
  izpisiPre(koren->desno);
}
```

Postfiksna oblika izpisa:

```
void izpisiPost(struct vozlisce *koren)
{
  if( koren==NULL )
    return;
  izpisiPost(koren->levo);
  izpisiPost(koren->desno);
  printf("%d  ", koren->vrednost);
}
```

### *Dodajanje vozlišča v drevo*

Novo vozlišče v urejeno binarno drevo dodajamo vedno med liste drevesa. Mesto za novo vozlišče je zaradi urejenosti drevesa točno določeno z vrednostmi elementov. Funkcijo bi lahko zapisali takole: če je koren drevesa enak `NULL`, postavi nov element namesto korena, sicer pa vstavi nov element v levo poddrevo, če je njegova vrednost manjša od vrednosti korena, v nasprotnem primeru pa v desno poddrevo.

```
struct vozlisce *dodaj(struct vozlisce *koren, int vred)
{
  if(koren == NULL) {
    koren = (struct vozlisce*) malloc(sizeof(struct vozlisce));
    koren->vrednost = vred;
    koren->levo = NULL;
    koren->desno = NULL;
  }
  if(vred < koren->vrednost)
```

```
    koren->levo = dodaj(koren->levo, vred);
  if(vred > koren->vrednost)
    koren->desno = dodaj(koren->desno, vred);
  return(koren);
}
```

***Brisanje vozlišča iz drevesa***

Brisanje elementa iz drevesa je ena najtežjih operacij med tukaj opisanimi. Brisanje lista v drevesu nam sicer ne povzroča težav, problem pa se pojavi, kadar želimo izbrisati vozlišče, ki ima (enega ali dva) naslednika.

Najprej pa moramo seveda poiskati element, ki ga želimo izbrisati, v drevesu. Postopek je naslednji. Če je koren enak `NULL`, elementa ni v drevesu in ne naredimo nič (to je tudi izstopni pogoj rekurzije). Če je vrednost elementa, ki ga želimo brisati, manjša od vrednosti korena drevesa, moramo element poiskati in zbrisati v levem poddrevesu. Če pa je vrednost tega elementa večja od vrednosti korena, element brišemo iz desnega poddrevesa. V primeru, da moramo zbrisati koren (vrednost korena je enaka vrednosti, ki jo želimo zbrisati), pa imamo spet tri možnosti. Če koren nima naslednikov ali pa ima le enega naslednika (levo ali/in desno poddrevo je `NULL`), je naloga enostavna - koren enostavno prevežemo na naslednika in sprostimo prostor, ki ga zaseda vozlišče korena. Če pa ima koren oba naslednika, se brisanje nekoliko zaplete. Ker vozlišča korena ne moremo kar ukiniti (saj sta nanj pripeti dve poddrevesi), bomo vrednost korena zamenjali z najmanjšo vrednostjo iz desnega poddrevesa (ali pa z največjo vrednostjo levega poddrevesa, oboje ohrani urejenost drevesa) in potem zbrisali to vozlišče z najmanjšo vrednostjo desnega poddrevesa. Slednje lahko naredimo kar s klicem te iste funkcije `brisi`. Brisanje tega elementa je enostavno, saj zanj (po definiciji urejenosti) velja, da nima levega poddrevesa. Opisan postopek lahko zapišemo z naslednjo kodo:

```
struct vozlisce *brisi(struct vozlisce *koren, int vred)
{
  struct vozlisce *q;

  if(koren == NULL )
    return(koren);
  if(vred < koren->vrednost ) {
    koren->levo = brisi(koren->levo, vred);
    return(koren);
  }
  if(vred > koren->vrednost ) {
    koren->desno = brisi(koren->desno, vred);
    return(koren);
  }
  if(vred == koren->vrednost ) {
    if(koren->levo == NULL) {
      q = koren;
      koren = koren->desno;
      free(q);
```

```
      }
      else if(koren->desno == NULL) {
        q = koren;
        koren = koren->levo;
        free(q);
      }
      else {
      // poiščemo najmanjši element v desnem poddrevesu,
      // vrednost korena postane enaka vrednosti tega elementa
        q = koren->desno;
        while(q->levo != NULL)
          q = q->levo;
        koren->vrednost = q->vrednost;
        koren->desno = brisi(koren->desno, q->vrednost);
      }
      return(koren);
    }
  }
```

### *Brisanje celega drevesa*

Pri brisanju celega drevesa tudi uporabimo rekurzijo. Izstopni pogoj je, da je koren enak `NULL`. Če pa koren obstaja (to pomeni, da je različen od `NULL`), moramo najprej zbrisati celo levo in celo desno poddrevo (če obstajata), potem pa še koren. Brisanje korena pomeni sprostiti pomnilnik, ki ga zaseda koren. V obeh primerih funkcija vrne `NULL`.

```
struct vozlisce *izprazni(struct vozlisce *koren)
{
  if( koren==NULL )
    return(koren);
  if( koren->levo!=NULL ) {
    koren->levo = izprazni(koren->levo);
  }
  if( koren->desno!=NULL ) {
    koren->desno = izprazni(koren->desno);
  }
  if( (koren->levo==NULL) && (koren->desno==NULL) ) {
    free(koren);
  }
  return(NULL);
}
```

### *Sestavimo program*

Na koncu vse funkcije združimo s funkcijo `main` in dobimo naslednji program, ki demonstrira delo z urejenim binarnim drevesom (`drevo.c`).

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


struct vozlisce
{
  int vrednost;
  struct vozlisce *levo;
  struct vozlisce *desno;
};



void izpisiIn(struct vozlisce *koren)
{
  if( koren==NULL )
    return;
  izpisiIn(koren->levo);
  printf("%d  ", koren->vrednost);
  izpisiIn(koren->desno);
}


void izpisiPre(struct vozlisce *koren)
{
  if( koren==NULL )
    return;
  printf("%d  ", koren->vrednost);
  izpisiPre(koren->levo);
  izpisiPre(koren->desno);
}


void izpisiPost(struct vozlisce *koren)
{
  if( koren==NULL )
    return;
  izpisiPost(koren->levo);
  izpisiPost(koren->desno);
  printf("%d  ", koren->vrednost);
}


void izpisi(struct vozlisce *koren)
{
  printf("Infiksna oblika:\n");
  izpisiIn(koren);
  printf("\nPrefiksna oblika:\n");
  izpisiPre(koren);
  printf("\nPostfiksna oblika:\n");
  izpisiPost(koren);
  printf("\n\n");
}
```

```
int poisci(struct vozlisce *koren, int vred)
{
  if(koren == NULL)
    return(0);  // iskane vrednosti ni v drevesu
  if(vred == koren->vrednost)
    return(1);  // iskano vrednost smo našli
  if(vred < koren->vrednost)
    return(poisci(koren->levo, vred));  // iš?i levo
  if(vred > koren->vrednost)
    return(poisci(koren->desno, vred));  // iš?i desno
}


struct vozlisce *dodaj(struct vozlisce *koren, int vred)
{
  if(koren == NULL) {
    koren = (struct vozlisce*) malloc(sizeof(struct vozlisce));
    koren->vrednost = vred;
    koren->levo = NULL;
    koren->desno = NULL;
  }
  if(vred < koren->vrednost)
    koren->levo = dodaj(koren->levo, vred);
  if(vred > koren->vrednost)
    koren->desno = dodaj(koren->desno, vred);
  return(koren);
}


struct vozlisce *brisi(struct vozlisce *koren, int vred)
{
  struct vozlisce *q;

  if(koren == NULL )
    return(koren);
  if(vred < koren->vrednost ) {
    koren->levo = brisi(koren->levo, vred);
    return(koren);
  }
  if(vred > koren->vrednost ) {
    koren->desno = brisi(koren->desno, vred);
    return(koren);
  }
  if(vred == koren->vrednost ) {
    if(koren->levo == NULL) {
      q = koren;
      koren = koren->desno;
      free(q);
    }
    else if(koren->desno == NULL) {
      q = koren;
      koren = koren->levo;
      free(q);
    }
    else {
```

```
    // poiščemo najmanjši element v desnem poddrevesu,
    // vrednost korena postane enaka vrednosti tega elementa
      q = koren->desno;
      while(q->levo != NULL)
        q = q->levo;
      koren->vrednost = q->vrednost;
      koren->desno = brisi(koren->desno, q->vrednost);
    }
    return(koren);
  }
}


struct vozlisce *izprazni(struct vozlisce *koren)
{
  if( koren==NULL )
    return(koren);
  if( koren->levo!=NULL ) {
    koren->levo = izprazni(koren->levo);
  }
  if( koren->desno!=NULL ) {
    koren->desno = izprazni(koren->desno);
  }
  if( (koren->levo==NULL) && (koren->desno==NULL) ) {
    free(koren);
  }
  return(NULL);
}


main() {
  struct vozlisce *koren;

  koren = NULL;
  koren = dodaj(koren,11);
  izpisi(koren);
  koren = dodaj(koren,7);
  izpisi(koren);
  koren = dodaj(koren,3);
  izpisi(koren);
  koren = dodaj(koren,15);
  izpisi(koren);
  koren = dodaj(koren,9);
  izpisi(koren);
  koren = dodaj(koren,8);
  izpisi(koren);
  koren = dodaj(koren,20);
  izpisi(koren);
  koren = dodaj(koren,18);
  izpisi(koren);
  koren = dodaj(koren,1);
  izpisi(koren);
  koren = dodaj(koren,25);
  izpisi(koren);
  koren = dodaj(koren,5);
```

```
  izpisi(koren);
  koren = brisi(koren,1);
  izpisi(koren);
  koren = brisi(koren,7);
  izpisi(koren);
  koren = brisi(koren,11);
  izpisi(koren);
  if( poisci(koren,5) == 1)
    printf("Element 5 je v drevesu.\n\n");
  else
    printf("Elementa 5 ni v drevesu.\n\n");
  koren = brisi(koren,5);
  izpisi(koren);
  if( poisci(koren,5) == 1)
    printf("Element 5 je v drevesu.\n\n");
  else
    printf("Elementa 5 ni v drevesu.\n\n");
  koren = izprazni(koren);
  printf("Na koncu je drevo prazno:\n");
  izpisi(koren);
}
```

# 10. Literatura

Brian W. Kernighan in Dennis M. Ritchie: *Programski jezik C*, 4. izdaja, slovenski prevod, prevajalec Leon Mlakar, Fakulteta za elektrotehniko in računalništvo, Ljubljana, 1994.

Andrew Koenig: *C Traps and Pitfalls*, Addison-Wesley, 1989.

Alfred V Aho, John E. Hopcroft in Jeffrey D. Ullman: *Data Structures and Algorithms*, Addison-Wesley, 1982.

Spletni viri: www.cs.waikato.ac.nz/Teaching/COMP134B/lectures/recursion.pdf