Programiranje in algoritmi

skozi primere

Alenka Kavčič, Marko Privošnik, Ciril Bohak,
Matija Marolt in Saša Divjak

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
januar 2010

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

004.421(075.8)

PROGRAMIRANJE in algoritmi skozi primere / Alenka Kavčič ... [et al.] ; [izdajatelj] Fakulteta za računalništvo in informatiko. - 1. izd. - Ljubljana : Založba FE in FRI, 2010

ISBN 978-961-6209-76-2 (Fakulteta za računalništvo in informatiko)
1. Kavčič, Alenka, 1968-
249928704

Recenzenta: prof. dr. Damjan Zazula, izr. prof. dr. Viljan Mahnič
Založnik: Založba FE in FRI, Ljubljana
Izdajatelj: Fakulteta za računalništvo in informatiko, Ljubljana
Urednik: mag. Peter Šega

Natisnil: KOPIJA Mavrič, Ljubljana
Naklada: 200 izvodov
1. izdaja

VSEBINA

DRUGI DEL: Programski jezik Java

Predgovor

Gradivo je namenjeno študentom prvih letnikov bolonjskega univerzitetnega programa na Fakulteti za računalništvo in informatiko, ki pri predmetu Programiranje in algoritmi spoznavajo osnove programskih algoritmov in podatkovnih struktur, naprednejše koncepte programskega jezika Java in se tudi prvič srečajo s programiranjem v programskem jeziku C. Poleg bolj teoretične predstavitve algoritmov in podatkovnih struktur zajema tudi izbrane teme iz programiranja v jezikih Java in C ter skozi primere programov, ki so tudi podrobneje razloženi, prikazuje pristope k reševanju različnih problemov v obeh jezikih.

To gradivo je dopolnilo zapiskom s predavanj. Ni namenjeno popolnim začetnikom v programiranju, saj predpostavlja poznavanje in razumevanje osnovnih konceptov programskih jezikov (kot so spremenljivke in njihova uporaba, zanke, odločitveni stavki itd.) in objektno usmerjenega programiranja (kot so objekti, članske spremenljivke, metode, konstruktorji, dedovanje, vmesniki, abstraktni razredi itd.). Osredotoči se na posebnosti jezika C in na pristop k reševanju problemov v tem jeziku ter na naprednejše koncepte jezika Java.

Gradivo je razdeljeno na tri dele. V prvem delu obravnavamo osnovne koncepte algoritmov, zahtevnost algoritmov ter predstavimo temeljne abstraktne podatkovne tipe. Drugi del se ukvarja z naprednejšimi koncepti jezika Java in gradi na znanju, ki so ga študenti pridobili v prvem semestru pri predmetu Osnove programiranja. Tretji del pa je posvečen jeziku C in izpostavi predvsem posebnosti tega jezika napram Javi.

Zelo priporočljivo je, da v tem gradivu obdelane primere bralec tudi sam preizkusi. To pomeni, da program napiše v računalniku, ga izvede in preveri njegovo delovanje. Po potrebi kodo programa tudi ustrezno spremeni in preveri, ali se delovanje spremenjenega programa ujema z njegovimi predpostavkami. V gradivu se omejimo na delo v Linux okolju, katerega uporabljamo tudi na laboratorijskih vajah pri predmetu Programiranje in algoritmi. Tako okolje pa ni vezano le na operacijski sistem Linux, saj ga lahko vzpostavimo tudi na operacijskih sistemih Microsoft Windows s pomočjo brezplačnih orodij Cygwin (www.cygwin.com). Cygwin je okolje za Windows, ki je podobno Linuxu, in vključuje obsežno zbirko orodij.

Izvorne kode vseh primerov programov, kjer je označeno tudi ime programa, so priložene temu gradivu. Dosegljive so na spletni strani Programiranje in algoritmi skozi primere (http://lgm.fri.uni-lj.si/pa/primeri/).

PRVI DEL
Algoritmi in podatkovne strukture

1. Algoritmi

Kaj je algoritem

Algoritem je postopek za rešitev nekega problema ali izvedbo nekega opravila. Postopek, ki predstavlja algoritem, mora biti sestavljen iz končnega števila navodil in se mora za poljubne vhodne podatke izvesti v končnem številu korakov.

Posamezen algoritem predstavlja splošen postopek za rešitev cele množice sorodnih problemov. Tako lahko na primer nek algoritem za urejanje uporabimo tako pri urejanju naravnih števil, kakor tudi pri urejanju besed ali pri urejanju kuhinjskih loncev po prostornini.

Prav tako pa velja tudi obratno. Za rešitev nekega problema je lahko na voljo več algoritmov. Dve števili lahko na primer zmnožimo na klasičen način, kakor smo se učili v osnovni šoli, s pomočjo tabele zmnožkov, s pomočjo logaritmov ali še kako drugače.

Kratka zgodovina algoritmov

Čeprav pojem algoritma v sodobnem času povezujemo z računalniki, ta za svoj obstoj in uporabo ne potrebuje računalnika. Računalnik je zgolj naprava, ki omogoča hitro izvajanje algoritmov, implementiranih v obliki programov.

Algoritmi so bili predmet študija že v pradavnini in mnogo izmed danes uporabljanih algoritmov so odkrili že davno pred pojavom sodobnih računalnikov. Tak primer je Evklidov algoritem za izračun največjega skupnega delitelja. Ta algoritem, ki predstavlja učinkovit postopek za rešitev problema izračuna največjega skupnega delitelja, je grški matematik Evklid opisal že 300 let pr. n. št., verjetno pa je bil poznan že prej.

Izraz algoritmi izhaja iz imena perzijskega astronoma in matematika Al-Khwārizmī-ja, ki je živel v osmem in devetem stoletju. Al-Khwārizmī velja za utemeljitelja algebre, njegovo delo pa je omogočilo razširitev hindujsko-arabskega številskega sistema v Evropo.

Pomemben vpliv na področje algoritmov je imel tudi razvoj mehanskih avtomatov, ki se je začel z izumom mehanskih ur v srednjem veku in se nadaljeval vse do mehanskih računskih avtomatov Charlesa Babbagea in grofice Ade Lovelace v devetnajstem stoletju.

Sodoben pogled na algoritme so izoblikovali matematiki devetnajstega in dvajsetega stoletja. K razvoju tega področja so pomembno prispevali George Boole, David Hilbert, Alonzo Church, Emil Post, Alan Turing in mnogi drugi, ki so matematično opredelili pojem algoritma, simboličnega jezika in izračunljivosti.

Primer preprostega algoritma

Nek problem lahko rešimo na več načinov z uporabo različnih algoritmov, pri čemer je lahko nek algoritem v danem okviru boljši od ostalih. Kot primer vzemimo iskanje besede v slovarju. Besedo lahko v slovarju iščemo zaporedoma od prve besede proti zadnji besedi, vse dokler ne naletimo nanjo. Tovrstno iskanje, ki predstavlja izvedbo iskalnega algoritma z imenom linearno iskanje, je počasno, razen v primeru, ko se iskana beseda nahaja na začetku slovarja.

Pri iskanju besede v slovarju lahko izkoristimo lastnost slovarja, in sicer da so besede v slovarju urejene. Na ta način hitrejše iskanje besede v slovarju dosežemo tako, da preverimo besedo na sredini seznama besed v slovarju in se v primeru, če s tem že nismo naleteli na iskano besedo, odločimo, ali bomo z iskanjem besede nadaljevali v prvi ali v drugi polovici seznama. Nato v izbrani polovici seznama besedo iščemo enako, kot smo jo iskali v celotnem seznamu besed v slovarju, vse dokler ne naletimo na iskano besedo. Tovrstno iskanje besede predstavlja izvedbo iskalnega algoritma, ki se imenuje binarno iskanje, in omogoča hitrejše iskanje kot linearno iskanje.

Predstavitev in zapis algoritmov

Algoritme lahko predstavimo na mnogo različnih načinov. Najosnovnejšo (a pogosto dvoumno in ne dovolj natančno) visokonivojsko predstavitev algoritma lahko podamo z uporabo naravnega jezika.

Bolj natančen zapis lahko dosežemo z uporabo psevdokode, ki omogoča neformalen, visokonivojski ter strukturiran zapis algoritmov. Psevdokoda predstavlja zapis algoritma, ki je namenjen človeku in ne računalniku. Kljub temu, da uporablja podobne strukturne elemente kot nekateri programski jeziki, je neodvisna od nekega določenega programskega jezika.

Algoritme lahko predstavimo na slikovni način z diagrami poteka, ki prav tako omogočajo visokonivojski ter strukturiran zapis, ki je neodvisen od nekega določenega programskega jezika.

Najbolj natančen in nedvoumen zapis algoritmov omogočajo programski jeziki, ki so namenjeni računalniški obdelavi. Algoritem, zapisan v programskem jeziku, lahko obravnavamo kot njegovo formalno specifikacijo.

Opis algoritma v naravnem jeziku, njegov zapis v psevdokodi ter zapis v programskem jeziku Java si oglejmo na primeru algoritma za iskanje največjega števila v nepraznem seznamu števil.

Opis algoritma v naravnem jeziku:

Naj prvo število v seznamu prevzame vlogo trenutno največjega števila. Za vsako od preostalih števil v seznamu preverimo, če je večje od trenutno največjega števila, in če je temu tako, naj to število prevzame vlogo trenutno največjega števila. Ko preverimo vsa števila v seznamu, je trenutno največje število tudi največje število v seznamu.

Zapis algoritma v psevdokodi:

```
1. največje ← prvo število v seznamu
2. za števila v seznamu od drugega do zadnjega naredimo:
    2.1. če število > največje, potem največje ← število
3. končamo z odgovorom največje
```

Pri zapisu algoritma v psevdokodi smo uporabili simbol ←, ki predstavlja prirejanje vrednosti na desni strani simbola spremenljivki, zapisani levo od simbola.

Zapis algoritma v programskem jeziku Java:

```
static int najdiNajvečjeŠtevilo(int[] seznam) {
  // Vrne največje število v nepraznem seznamu seznam
  int največje = seznam[0];
  for(int indeks = 1; indeks < seznam.length; indeks++) {
    int število = seznam[indeks];
    if(število > največje) {
      največje = število;
    }
  }
  return največje;
}
```

Zahtevnost algoritmov

Učinkovitost posameznega algoritma lahko formaliziramo s pojmom zahtevnost algoritma. Zahtevnost algoritma je določena kot relacija med velikostjo vhoda in količino nekega vira, ki ga pri danem vhodu potrebuje algoritem za izvedbo celotnega postopka. Dva vira, ki nas navadno zanimata pri analizi algoritmov, sta prostor in čas.

Pri prostorski zahtevnosti s številom pomnilniških lokacij merimo količino prostora, ki je potreben za izvedbo algoritma. Pri časovni zahtevnosti merimo količino časa, ki je potreben za izvedbo algoritma. V nadaljevanju se bomo osredotočili le na obravnavo časovne zahtevnosti, podrobnejša razlaga prostorske zahtevnosti, ki je v marsičem sorodna časovni, pa je izpuščena.

Pri formalni analizi algoritma časa, potrebnega za izvedbo algoritma, navadno ne izražamo v sekundah, temveč v korakih, potrebnih za izvedbo algoritma pri dani velikosti vhoda. Kaj predstavlja posamezne korake, ni natančno določeno, pomembno pa je, da je dejanski čas izvajanja posameznega koraka navzgor omejen z neko konstanto. Kot korake lahko upoštevamo operacije, ki so značilne za skupino algoritmov za reševanje nekega problema. Tako lahko na primer pri matematičnih algoritmih štejemo aritmetične operacije, pri iskalnih algoritmih pa štejemo operacije primerjanja.

Natančna zahtevnost algoritma, ki jo označimo s $T(n)$, ni odvisna le od samega algoritma, temveč tudi od njegove implementacije. Zaradi tega nas bolj kot natančna zahtevnost algoritma zanima velikostni red zahtevnosti, ki ga dobimo z asimptotično oceno zahtevnosti algoritma. Najpogosteje uporabljena asimptotična ocena zahtevnosti

je funkcija $O(\cdot)$, ki razkriva asimptotično zgornjo mejo rasti funkcije $T(n)$. Če velja $T(n) = O(g(n))$, potem je funkcija $T(n)$ za dovolj velike *n* navzgor omejena s funkcijo $cg(n)$, kjer je *c* poljubna pozitivna konstanta.

Asimptotično zgornjo mejo algoritma dobimo tako, da v izrazu $T(n)$, ki predstavlja zahtevnost algoritma, zanemarimo vse aditivne člene nižjega reda in obdržimo le tistega, ki najhitreje raste, ter vse konstantne multiplikativne faktorje.

Izračun časovne kompleksnosti algoritma si oglejmo na primeru dveh algoritmov za izračun potence nekega števila.

Število b^n lahko izračunamo z naslednjim preprostim algoritmom:

```
1. p ← 1
2. za i = 1, … ,n ponavljamo:
    2.1. p ← p krat b
3. končamo z odgovorom p
```

Navodilo `2.1`, v katerem se izvaja množenje, se pri velikosti vhoda *n* izvede *n*-krat. Število korakov, ki jih zgornji algoritem potrebuje za izračun potence v odvisnosti od vhoda *n*, je potemtakem ravno *n*, saj algoritem za izračun potrebuje *n* aritmetičnih operacij. Časovna kompleksnost preprostega algoritma za potenciranje je velikostnega reda *n*, kar zapišemo kot $O(n)$.

Število b^n lahko izračunamo tudi z naslednjim izboljšanim algoritmom:

```
1. p ← 1, q ← b, m ← n
2. dokler m > 0, ponavljamo:
    2.1. če je m lih, potem množimo p ← p krat q
    2.2. razpolovimo m (pozabimo na ostanek)
    2.3. množimo q ← q krat q
3. končamo z odgovorom p
```

Navodila `2.1` do `2.3` izvedejo skupaj največ 2 množenji. Ta navodila se ponavljajo, dokler ob razpolavljanju vrednosti *n* ta ne doseže vrednosti 0, torej $\lfloor log_2(n) \rfloor + 1$ krat. Največje število korakov, ki jih gornji algoritem potrebuje za izračun potence v odvisnosti od vhoda *n*, je največ $2(\lfloor log_2(n) \rfloor + 1) = 2\lfloor log_2(n) \rfloor + 2$. Pri izračunu učinkovitosti algoritma med korake nismo šteli operacije razpolavljanja števila *m*, saj je celoštevilsko deljenje z dve, glede na splošno celoštevilsko množenje, zelo enostavna in hitra operacija. Časovno kompleksnost izboljšanega algoritma za potenciranje dobimo z naslednjo redukcijo izraza $2\lfloor log_2(n) \rfloor + 2$:

Zanemarimo aditivne člene, razen najhitreje rastočega:

$$2\lfloor \log_2(n) \rfloor + 2 \quad \rightarrow \quad 2\lfloor \log_2(n) \rfloor$$

Odstranimo konstantne multiplikativne faktorje:

$$2\lfloor \log_2(n) \rfloor \quad \rightarrow \quad \lfloor \log_2(n) \rfloor$$

Zanemarimo zaokroževanje navzdol, saj velja $log_2(n) - \lfloor log_2(n) \rfloor < 1$, tako da zaokroževanje členu $log_2(n)$ odšteje vrednost (aditivni člen), ki je manjša od `1`:

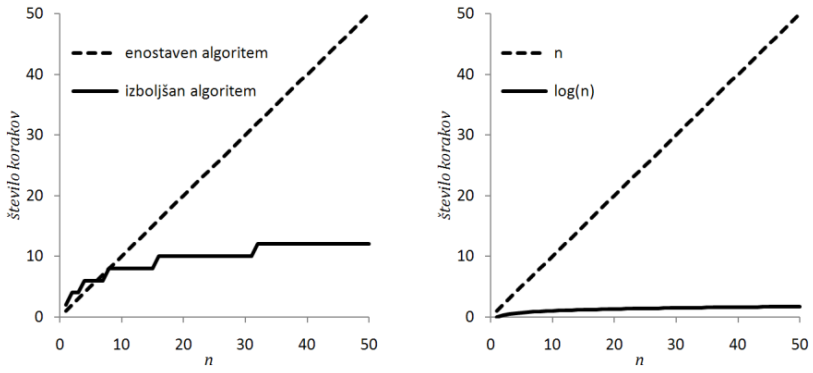
$$\lfloor \log_2(\mathrm{n}) \rfloor \quad \rightarrow \quad \log_2(\mathrm{n})$$

Zanemarimo bazo logaritma, saj velja $log_2(n) = log_2(b) log_b(n)$, kjer predstavlja člen $log_2(b)$ konstantni multiplikativni faktor:

$$\log_2(\mathrm{n}) \quad \rightarrow \quad \log(\mathrm{n})$$

Časovna kompleksnost izboljšanega algoritma za potenciranje je torej velikostnega reda $log(n)$, kar zapišemo kot $O\big(log(n)\big)$.

Slika 1.1 prikazuje število korakov, potrebnih pri izvedbi enostavnega in izboljšanega algoritma za potenciranje, v odvisnosti od velikosti vhoda *n*.



Slika 1.1: Časovna kompleksnost enostavnega in izboljšanega algoritma za potenciranje.

Prvi (levi) graf na sliki prikazuje izračunano časovno kompleksnost $T(n)$, drugi pa asimptotično zgornjo mejo rasti časovne kompleksnosti ($O(n)$ za enostaven algoritem in $O(log(n))$ za izboljšan algoritem).

2. Iterativno in rekurzivno reševanje problemov

Kot smo videli v prejšnjem poglavju, postopek za reševanje nekega problema ponavadi vključuje ponavljanje (ali iteracijo) dela navodil, s katerim se postopoma približujemo zadanemu cilju oz. rešitvi. Pri tem se rezultat vsake iteracije uporabi kot začetno stanje naslednje iteracije.

Tak postopek ponavljanja navodil lahko vključuje zanko, v kateri se ponavljajo določeni stavki in spreminja stanje. Tovrstno reševanje problemov imenujemo iterativno reševanje problemov.

Kadar ponavljanje in približevanje rešitvi dosežemo preko rešitve nekoliko manjšega problema, za katerega uporabimo enak postopek rešitve, tak način imenujemo rekurzivno reševanje problemov.

Kaj je rekurzija

Rekurzija je način definiranja neke funkcije, pri katerem je v definiciji uporabljena kar sama funkcija. Rekurzija se pogosto uporablja tako v matematiki kot tudi v računalništvu. Pri programskih jezikih pomeni rekurzivno klicanje funkcij možnost, da funkcija neposredno ali posredno kliče samo sebe.

Primer rekurzivne definicije iz vsakdanjega življenja je definicija prednikov neke osebe. Zapisali bi jo lahko takole:

- predniki neke osebe so starši te osebe,
- starši prednikov neke osebe so tudi predniki te osebe.

Vidimo, da se v drugem delu definicije prednikov sklicujemo kar na prednike, torej se funkcija pri definiciji sklicuje sama nase. Seveda bi bilo tako sklicevanje lahko neskončno, če ne bi podali tudi osnovnega primera, ki definira najpreprostejši primer brez rekurzivnega sklicevanja (prvi del definicije prednikov).

Rekurzivno reševanje problema poteka tako, da kompleksen začetni problem razdelimo na dele tako, da je del problema enostavno rešljiv, za del problema pa lahko uporabimo enak postopek kot za prvotni problem (le da je tokrat problem manjši). Če pa je podan problem dovolj enostaven, ga rešimo neposredno, brez delitve.

Rekurzivno reševanje problema tipično vključuje:

- opis rešitve problema za enostaven 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 in tudi enostavnih 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 (vključno) `n`. Postopek izračuna fakultete bi s psevdokodo lahko zapisali na naslednji način:

```
funkcija Fakulteta od n:
  1. fakulteta ← 1
  2. za števila od 1 do n naredimo:
      2.1. fakulteta ← fakulteta krat število
  3. končamo z odgovorom fakulteta
```

Primer za `n=0` bi sicer lahko obravnavali ločeno, a ni potrebe: v tem primeru se zanka namreč ne izvrši niti enkrat in postopek se konča s pravilno vrednostjo 1.

Zapišimo isti postopek še v programskem jeziku Java:

```
static int fakulteta(int n) {
  int f = 1;
  for(int 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.

Tak zapis je rekurziven, saj smo pri definiciji funkcije fakulteta uporabili kar funkcijo fakulteta. Seveda smo pri tem morali podati tudi enostaven primer, pri katerem se rekurzija zaključi. V našem primeru je to `0!`, ki vrne rezultat 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 psevdokodi:

```
funkcija Fakulteta od n:
  1. če je n enak 0, potem končamo z odgovorom 1
  2. končamo z odgovorom n krat Fakulteta od n-1
```

ali pa v jeziku Java:

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

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 funkcija `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).

Poglejmo še, kako je s časovno zahtevnostjo obeh algoritmov. Pri iterativni rešitvi imamo v zanki eno množenje, zanka pa se ponovi n-krat. Časovna zahtevnost iterativnega izračuna fakultete je torej $O(n)$.

Pri rekurzivni rešitvi pa imamo vsakič v funkciji le eno množenje, a se sama funkcija kliče n-krat. Torej imamo tudi pri rekurzivni rešitvi izračuna fakultete časovno zahtevnost $O(n)$.

Podrobnosti o izvajanju rekurzivnega programa

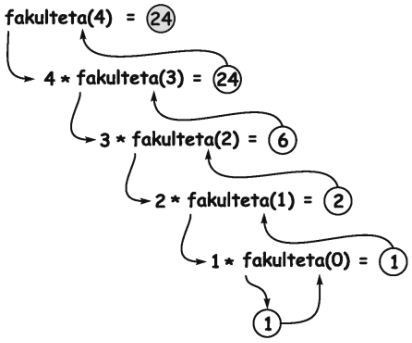
Poglejmo si, kako se izvaja rekurzivna funkcija `fakulteta`. Najlažje to prikažemo kar na primeru, tako da sledimo izvajanju kode. Za osnovo vzemimo kodo zgornje javanske rekurzivne metode.

Recimo, da želimo izračunati vrednost `4!` in zato pokličemo metodo `fakulteta(4)`. Koda v telesu metode najprej preveri, ali je metodi podan parameter `n` (v našem primeru število 4) enak 0. Ker pogoj ni izpolnjeni (4 ni enako 0), se izvrši naslednji stavek, ki sledi stavku `if`, to je `return n * fakulteta(n-1);`. Metoda torej vrne rezultat, ki je enak zmnožku števila 4 (naš `n`) in rezultata, ki ga vrne klic metode `fakulteta(3)`.

Preden lahko metoda izračuna in vrne rezultat, mora dobiti vrednost, ki jo vrne klic `fakulteta(3)`. To pomeni, da ponovno pokliče isto metodo `fakulteta`, le da tokrat z manjšim številom za parameter metode.

Postopek se ponavlja, dokler ne pride do klica `fakulteta(0)`. V tem primeru preverjanje `n==0` vrne resnično in se zato izvrši koda v telesu stavka `if`, torej stavek `return 1;`. Tako metoda vrne vrednost 1 in s tem se končajo tudi rekurzivni klici te metode (vrednost parametra 0 je izstopni pogoj iz rekurzije).

Sedaj se le še do konca izvršijo vsi predhodni klici metod, tako da se izračuna in vrne ustrezen rezultat. Celoten postopek (vrstni red klicev ter računanje in vračanje rezultatov) je prikazan na spodnji sliki.



Slika 2.1: Postopek računanja `fakulteta(4)`.

Pri vsakem ponovnem klicu metode se vsi podatki trenutnega klica metode shranijo na izvajalni sklad (*execution stack*, *call stack*), na vrh podatkov o prejšnjem klicu metode. Na izvajalni sklad se shranijo parametri metode, njene lokalne spremenljivke ter naslov vrnitve (*return address*), ki pove, kje je naslednji ukaz, ki naj se izvede, ko se metoda konča. Ti podatki se nalagajo na izvajalni sklad ob vsakem novem klicu metode. Ko se posamezna metoda zaključi, se z izvajalnega sklada poberejo podatki prejšnjega klica metode (vedno vzamemo podatke z vrha sklada). To se ponavlja do prvega klica metode.

Prav zaradi shranjevanja podatkov na izvajalni sklad ob vsakem klicu metode je delovanje rekurzivnega programa počasnejše od iterativnega (in zasede tudi več pomnilnika), čeprav imata oba algoritma za izračun fakultete enako asimptotično oceno časovne zahtevnosti izvajanja.

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:

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

Ker člene zaporedja računamo tako, da seštevamo predhodna dva člena zaporedja, je samo zaporedje definirano rekurzivno. Zaporedje lahko tako zapišemo z naslednjimi tremi formulami:

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

Zapis v psevdokodi bi bil naslednji:

```
funkcija Fibonacci od n:
  1. če je n enak 1, potem končamo z odgovorom 0
  2. če je n enak 2, potem končamo z odgovorom 1
  3. končamo z odg. Fibonacci od n-1 plus Fibonacci od n-2
```

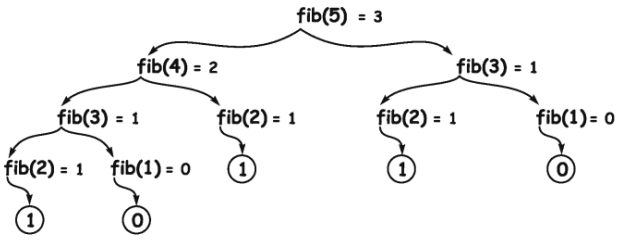
Javanska metoda pa je naslednja:

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

Koda je kratka in enostavno razumljiva, saj le sledi zgoraj zapisani rekurzivni definiciji zaporedja.

Poglejmo si še primer izvajanja rekurzivnega postopka za izračun `n`-tega Fibonaccijevega števila. Tokrat se bomo naslonili kar na samo rekurzivno definicijo zaporedja.

Ker funkcija `fib` dvakrat kliče samo sebe (tretja formula pri opisu zaporedja), lahko njeno delovanje (izvajanje) predstavimo z binarnim drevesom. Poglejmo si primer klica funkcije `fib(5)` za računanje petega Fibonaccijevega števila (ustrezno drevo klicev prikazuje slika 2.2).



Slika 2.2: Postopek računanja `fib(5)`.

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.

Čeprav je Fibonaccijevo zaporedje definirano rekurzivno, to ne pomeni, da ne moremo sestaviti tudi iterativne rešitve problema. Poiščimo torej še iterativno rešitev tega problema.

Rešitev je nekoliko daljša, a vseeno dovolj enostavna: v zanki sproti računamo zaporedne člene, dokler ne izračunamo `n`-tega člena. Takrat dobimo željen rezultat in lahko zaključimo z izračuni. Prva dva člena zaporedja sta podana (0 in 1 po vrsti), vsak naslednji člen zaporedja pa v zanki izračunamo tako, da seštejemo dva predhodna člena. Pri tem si moramo seveda zapomniti vrednosti obeh predhodnih členov.

Zapišimo naveden iterativni postopek še v psevdokodi:

```
funkcija Fibonacci od n:
  1. če je n enak 1, potem končamo z odgovorom 0
  2. če je n enak 2, potem končamo z odgovorom 1
  3. predhodni člen ← 1
  4. predpredhodni člen ← 0
  5. za števila od 3 do n naredimo:
      5.1. fibonacci ← predpredhodni plus predhodni člen
      5.2. predpredhodni člen ← predhodni člen
      5.3. predhodni člen ← fibonacci
  6. končamo z odgovorom fibonacci
```

Pa še javanska koda:

```
static int fibonacci(int n) {
  int f;
  if(n == 1)
    return 0;
  if(n == 2)
    return 1;
  int n1 = 1;
```

```
  int n2 = 0;
  for(int i = 3; i <= n; i++) {
    f = n2 + n1;
    n2 = n1;
    n1 = f;
  }
  return f;
}
```

In kako je s časovno zahtevnostjo obeh algoritmov?

Kot smo ugotovili že pri pregledu drevesa klicev funkcije `fib(5)` na sliki 2.2, rekurzivni algoritem na vsakem koraku dvakrat kliče samega sebe. Taka drevesna rekurzija ima eksponentno časovno zahtevnost $O(2^n)$. To pomeni, da se z za ena večjim vhodom *n* število zahtevanih operacij za izvedbo algoritma podvoji.

Nasprotno pa imamo pri iterativni rešitvi v zanki le eno operacijo seštevanja, zanka pa se *n*-krat ponovi. Iterativni algoritem za izračun *n*-tega Fibonaccijevega števila ima torej linearno časovno zahtevnost $O(n)$.

Čeprav je v primeru Fibonaccijevega zaporedja rekurzivna rešitev krajša in bolj razumljiva, je hkrati tudi časovno veliko bolj zahtevna in prostorsko potratna. Isti izračuni se po nepotrebnem izvedejo večkrat. Zato lahko brez večjega premisleka ugotovimo, da je iterativna rešitev tega problema primernejša od rekurzivne.

Eleganca in enostavnost ali učinkovitost

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

Rekurzija ne varčuje s pomnilnikom (navadno ga porabi več od iterativne rešitve, saj se veliko podatkov shrani na izvajalni sklad), niti ne deluje hitreje. Njena glavna prednost je, da je program pogosto napisan krajše, bolj jedrnato in bolj razumljivo. Slednje velja še posebej pri problemih, ki so rekurzivni po naravi in zato nimajo enostavne iterativne rešitve. Taki primeri, pri katerih s pridom uporabimo rekurzijo, so na primer drevesa ali problem Hanojskih stolpov.

Veliko problemov pa lahko rešimo na oba načina, iterativno ali rekurzivno, in obe rešitvi pa sta podobno kompleksni in razumljivi, z enako časovno zahtevnostjo. Pri takih problemih je zaradi omejitev izvajalnega sklada navadno bolje uporabiti iterativno rešitev.

Pretvorba rekurzije v iteracijo

Vsako rekurzivno rešitev problema lahko zapišemo tudi na iterativen način. Rekurzivno rešitev lahko torej vedno prevedemo na iterativno rešitev. V splošnem je pretvorba rekurzije v iteracijo precej zapletena, saj moramo v iterativno rešitev ročno vgraditi tudi

sklad, ki nadomesti izvajalni sklad, in sami poskrbeti za ustrezno polnjenje oz. praznjenje tega sklada.

Pri tem moramo opozoriti, da navadno pri rekurziji z novimi rekurzivnimi klici izvajalni sklad neprestano narašča. Obstaja pa tudi poseben tip rekurzije, imenujemo jo repna rekurzija, pri kateri izvajalni sklad ne narašča in ostaja konstanten. Zato je repna rekurzija v osnovi bolj podobna iteraciji in je njena pretvorba v iterativen postopek lahko zelo enostavna (pri njenem izvajanju namreč ne potrebujemo sklada).

V nadaljevanju si poglejmo postopek pretvorbe rekurzije v iteracijo; pri tem bomo repno rekurzijo obravnavali posebej.

Pretvorba repne rekurzije

Repna rekurzija (*tail recursion*) je poseben primer rekurzije, pri kateri je zadnja operacija v funkciji rekurzivni klic funkcije. Rezultat rekurzivnega klica je tudi rezultat funkcije, v kateri izvedemo klic. Taka rekurzija je zelo podobna iteraciji, zato jo zelo enostavno pretvorimo v iteracijo.

Tipičen primer repne rekurzije predstavlja funkcija `NSD` (največji skupni delitelj), ki jo bomo obravnavali v nadaljevanju in uporabili tudi kot primer pretvorbe rekurzivne rešitve v iterativno.

Bodite pozorni, da funkcija `fakulteta`, ki smo jo obravnavali v predhodnih razdelkih, ni repna rekurzija, čeprav je rekurzivni klic funkcije postavljen v zadnji stavek funkcije. Kje je razlika? Pri fakulteti moramo rezultat rekurzivnega klica še pomnožiti z `n`, preden lahko vrnemo rezultat funkcije. Zato rekurzivni klic ni zadnja operacija v funkciji, kar ima za posledico, da se mora na izvajalni sklad shraniti tudi vrednost spremenljivke `n`, da lahko po vrnitvi iz rekurzija naredimo potreben izračun. To pomeni, da se nalagajo vse odložene operacije, ki morajo biti izvedene še po zadnjem rekurzivnem klicu, kar ima za posledico rast izvajalnega sklada. Naša enostavna iterativna rešitev problema izračuna fakultete ni neposredna pretvorba rekurzivne rešitve v iterativno, temveč smo pri tem spremenili sam postopek izračuna (računamo od spodaj navzgor).

Vrnimo se na naš primer repne rekurzije in njene pretvorbe v iteracijo. Kot smo že omenili, bomo pretvorbo naredili na primeru funkcije `NSD` za iskanje največjega skupnega delitelja dveh naravnih števil (*greatest common divisor* ali *GCD*), za katerega izračun bomo uporabili Evklidov algoritem.

Največji skupni delitelj dveh naravnih števil je največje celo število, ki brez ostanka deli obe števili. Izračunamo ga tako, da obe števili razbijemo na prafaktorje in poiščemo tiste prafaktorje, ki so skupni obema številoma. Njihov zmnožek je ravno največji skupni delitelj teh dveh števil.

Poglejmo si primer na številih `360` in `1350`:

```
360 = 2 * 2 * 2 * 3 * 3 * 5
1350 = 2 * 3 * 3 * 3 * 5 * 5
```

Prafaktorji, ki so skupni obema številoma, so 2, 3, 3 in 5, njihov zmnožek pa je 90. Torej je največji skupni delitelj števil 360 in 1350 enak 90, kar lahko zapišemo tudi s funkcijo kot NSD(360,1350) = 90.

Učinkovitejšo metodo za izračun največjega skupnega delitelja pa je zapisal matematik Evklid, zato se po njem imenuje Evklidov algoritem. Pri tem je uporabil dejstvo, da največji skupni delitelj dveh števil deli tudi razliko teh dveh števil. Funkcijo NSD je definiral rekurzivno na naslednji način:

```
NSD(a, 0) = a
NSD(a, b) = NSD(b, a mod b), a >= b > 0
```

Kot je razvidno iz definicije funkcije, je odgovor novega izračuna NSD tudi odgovor originalnega problema. Ko se torej rekurzija zaključi, dobimo rezultat brez nadaljnjih preračunavanj (uporabljena je repna rekurzija).

Zapišimo rekurzivni algoritem še v psevdokodi:

```
funkcija NSD od a, b:
  1. če je b enak 0, potem končamo z odgovorom a
  2. končamo z odgovorom NSD od b, a mod b
```

in v Javi:

```
static int NSD(int a, int b) {
  if(b == 0)
    return a;
  return NSD(b, a%b);
}
```

Pri prvem klicu metode NSD(a,b) moramo paziti, da je a >= b, saj je postopek tako definiran. V nasprotnem primeru spremenljivki enostavno zamenjamo.

In kako bi naš rekurzivni problem pretvorili v iterativnega? Vidimo, da je rekurzivni klic zadnji stavek v funkciji in da je rezultat tega klica tudi rezultat, ki ga vrne naša funkcija. Ob rekurzivnem klicu se le spremenijo vrednosti spremenljivk a in b: spremenljivka a dobi vrednost spremenljivke b, v spremenljivko b pa se zapiše vrednost ostanka pri deljenju a z b. Bodite pozorni na to, da sprememba vrednosti obeh spremenljivk ohranja relacijo med njima: še vedno namreč velja a >= b.

Torej bi rekurzivni klic lahko nadomestili z zanko, ki se izvaja toliko časa, dokler je vrednost b večja od 0 (to je namreč izstopni pogoj iz rekurzije, zato bo v iterativni različici izstopni pogoj iz zanke, ki bo nadomestila rekurzijo). V zanki pa potem naredimo ustrezno spremembo vrednosti spremenljivk a in b, tako kot se je ta sprememba naredila ob rekurzivnem klicu. Rezultat izračuna največjega skupnega delitelja se ob končanju zanke (ko je b enak 0) nahaja v spremenljivki a (enako kot pri rekurzivni različici algoritma).

Vse povedano lahko zapišemo z naslednjo psevdokodo, ki opisuje iterativno rešitev iskanja največjega skupnega delitelja po Evklidovem algoritmu:

```
funkcija NSD od a, b:
  1. dokler je b večji od 0, ponavljamo:
      1.1. začasna ← b
      1.2. b ← a mod b
      1.3. a ← začasna
  2. končamo z odgovorom a
```

Iterativna javanska rešitev pa bi bila naslednja:

```
static int NSD(int a, int b) {
  int temp;
  while(b > 0) {
    temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}
```

Tudi tu velja, da mora biti na začetku `a >= b`; v nasprotnem primeru zamenjamo vrednosti obeh spremenljivk.

Splošna pretvorba rekurzije

V splošnem rekurzijo pretvorimo v iteracijo tako, da simuliramo obnašanje rekurzivne funkcije, pri tem pa izvajalni sklad nadomestimo s svojo podatkovno strukturo. Med izvajanjem funkcije moramo tako sami poskrbeti, da se na sklad, ki ga implementiramo v programu, shranijo ustrezne vrednosti: vsi parametri funkcije, vse lokalne spremenljivke ter tudi mesto povratka (to je mesto, kjer smo prekinili izvajanje funkcije z rekurzivnim klicem). V rešitev moramo torej vključiti tudi tiste akcije, za katere sicer med izvajanjem rekurzivnega programa poskrbi računalnik sam (vsa shranjevanja stanj in vrednosti spremenljivk ob rekurzivnih klicih).

Pa si poglejmo pretvorbo na stvarnem rekurzivnem primeru. Za osnovo vzemimo drevesno strukturo datotečnega sistema in implementirajmo iskanje določene datoteke v datotečnem sistemu. Datoteko bomo iskali po imenu, podana pa naj bo tudi mapa iskanja. Datoteko iščemo v podani mapi in vseh njenih podmapah (drevesna struktura datotečnega sistema).

V psevdokodi bi rekurzivno rešitev problema lahko zapisali takole:

```
funkcija poisci dat, ime:
  1. če je dat iskana navadna datoteka ime,
     končamo z odgovorom dat
  2. če je dat mapa, potem:
      2.1. za vsak vnos v mapi naredimo:
          2.1.1. najdena ← poišči vnos, ime
          2.1.2. če smo datoteko našli,
                 končamo z odgovorom najdena
  3. končamo z odgovorom ni datoteke
```

Za začetek smo zapisali enostavneje razumljivo (in krajšo) rekurzivno rešitev. Pri tem smo pojem datoteka uporabili splošno, tako za navadne datoteke kot tudi za mape.

Zapišimo isti postopek še v Javi:

```
public static File poisci(File dir, String ime) {
// iščemo datoteko z imenom ime
// v mapi dir in njenih podmapah
  if(dir.isFile() && dir.getName().equals(ime))
    return dir; // iskana datoteka je kar dir
  if(dir.isDirectory()) { // dir je mapa (direktorij)
    for(File dat: dir.listFiles()) { // prelistamo vsebino
      // rekurzivno iščemo za vsak vnos v mapi
      File najdena = poisci(dat, ime);
      if(najdena != null) // če datoteko najdemo, jo vrnemo
        return najdena;
    }
  }
  return null; // datoteke nismo našli
}
```

Pri javanski kodi smo dopisali nekaj komentarjev za splošno razumevanje kode, podrobnosti o razredu `File` in njegovih metodah (`isFile()`, `isDirectory()`, `getName()` in `listFiles()`) pa poiščite v dokumentaciji jezika.

Rešitev z uporabo rekurzije je enostavna in razumljiva. Iskanje datoteke v podmapah se izvaja rekurzivno, kar je bolj naravno, daj je tudi sama datotečna struktura definirana rekurzivno. Metoda `poisci` v zanki rekurzivno kliče metodo `poisci` za vsako podmapo v drevesni datotečni strukturi. Ko pregleda vse podmape trenutne mape, vrne kontrolo en nivo više (očetu), ki preveri naslednji vnos v mapi. Postopek se ponavlja, dokler ne najdemo iskane datoteke oz. pregledamo vse podmape. Pri tem nam ni potrebno vedeti, kako globoko bomo morali iskati (v koliko podmap bomo pogledali), niti si nam ni potrebno zapomniti, katere podmape smo že preiskali in katere še moramo. Za vse to samodejno poskrbi izvajalni sklad, na katerega se odlagajo vrednosti spremenljivk pred vsakim rekurzivnim klicem.

Pri iterativni rešitvi pa moramo za shranjevanje pomembnih spremenljivk poskrbeti sami, saj izvajanje rekurzije le simuliramo. Izvajalni sklad torej nadomestimo s svojo podatkovno strukturo, v katero shranjujemo vrednosti tistih spremenljivk, ki jih bomo še potrebovali pri nadaljnjih izračunih.

V splošnem pretvorbo naredimo tako, da vzpostavimo svoj sklad in vse vrednosti, ki se sicer pri rekurziji shranijo na izvajalni sklad ob rekurzivnem klicu, zapišemo na naš sklad v programu. Nato pa v zanki ponavljamo izračune nad podatki na skladu, dokler sklada ne spraznimo. Seveda lahko po potrebi na sklad odlagamo še dodatne podatke tudi med izvajanjem teh izračunov.

Podrobnosti o abstraktnem podatkovnem tipu sklad najdete v 3. poglavju. Tu naj povemo le to, da sklad hrani elemente, ki jih vedno dodajamo le na vrh sklada, odvzemamo pa tudi le z vrha sklada. Tako je vrstni red dodajanja elementov ravno obraten od vrstnega reda odvzemanja (zadnji dodan element je prvi na vrsti za odvzem).

Kako bi torej lahko zapisali iterativno različico naše funkcije za iskanje datoteke? Najprej začetno mapo iskanja postavimo na sklad ter vpeljemo zanko, ki se bo ponavljala toliko časa, dokler ne izpraznimo sklada. V zanki pa najprej vzamemo s sklada eno datoteko, nato pa zanjo podobno kot prej preverimo, ali ustreza iskalnim pogojem. V primeru, da je to kar iskana datoteka, jo vrnemo in s tem funkcijo zaključimo. Če pa je ta datoteka pravzaprav mapa, potem poiščemo vse njene vnose in jih po vrsti shranimo na sklad. Slednje simulira rekurzivne klice. Nato zanko ponovimo. V primeru, da se sklad sprazni, to pomeni, da nismo našli datoteke, ki bi ustrezala iskalnim pogojem (če bi jo našli, bi takrat že zaključili funkcijo).

Iterativno različico naše funkcije za iskanje datoteke bi v psevdokodi lahko zapisali takole:

```
funkcija poisci ime, datoteka:
  1. postavi datoteko na sklad
  2. dokler imamo še kaj na skladu, ponavljamo:
      2.1. datoteka ← vzemi element s sklada
      2.2. če je datoteka iskana navadna datoteka ime,
           končamo z odgovorom datoteka
      2.3. če je datoteka direktorij, potem:
          2.3.1. za vsak vnos direktorija naredimo:
              2.3.1.1. postavi vnos na sklad
  3. končamo z odgovorom ni datoteke
```

In v Javi:

```
public static File poisci(File dir, String ime) {
  Stack<File> sklad = new Stack<File>();
  sklad.push(dir);
  while(!sklad.empty()) {
    File dat = sklad.pop();
    if(dat.isFile() && dat.getName().equals(ime))
      return dat;
    if(dat.isDirectory())
      for(File d: obrni(dat.listFiles()))
        sklad.push(d);
  }
  return null;
}
```

Javanski razred `Stack`, ki smo ga uporabili za implementacijo našega sklada v metodi `poisci`, je opisan v razdelku *Vektor in sklad* v 5. poglavju. Tu omenimo le, da metoda `push()` potisne element na sklad, metoda `pop()` pa vrne element s sklada.

Pozoren bralec je v iterativni različici javanske kode lahko opazil, da smo v zanki `for` uporabili tudi metodo `obrni()`, ki vrne obrnjeno tabelo. Ta dodatek je sicer potreben le zaradi tega, da obe različici, rekurzivna in iterativna, pregledujeta datotečno drevesno strukturo na enak način: v globino in najprej po najbolj levih vejah.

3. Podatkovne strukture

Študij algoritmov je tesno povezan s študijem podatkovnih struktur. Zato si bomo v nadaljevanju pogledali, kaj so to podatkovne strukture in kaj abstraktni podatkovni tipi.

Podatkovne strukture

Podatkovna struktura je sistematičen način organiziranja in hranjenja podatkov, ki omogoča učinkovito rabo podatkov s strani množice uporabljenih algoritmov. Primeri pogosto uporabljenih podatkovnih struktur so vrsta, sklad, binarno drevo in razpršena tabela.

Nad podatkovnimi strukturami izvajamo operacije, ki uporabljajo podatke v podatkovni strukturi in z njimi manipulirajo. Operacije so izvedene z uporabo algoritmov, ki ustrezajo dani podatkovni strukturi. Tako nad podatkovno strukturo vrsta med drugim izvajamo operacijo dodaj, ki doda nov element na konec vrste, in operacijo odstrani, ki odstrani element z začetka vrste. Različne podatkovne strukture si pogosto delijo iste operacije, ki pa so zaradi razlik med podatkovnimi strukturami implementirane z različnimi algoritmi, kar ima za posledico razlike pri njihovih učinkovitostih. Zaradi tega so za uporabo določenih operacij nekatere podatkovne strukture bolj primerne od drugih podatkovnih struktur in obratno.

Če definiramo podatkovno strukturo posredno zgolj z operacijami, ki se lahko izvajajo nad njo, ter z lastnostmi teh operacij, potem tako definirani podatkovni strukturi pravimo abstraktna podatkovna struktura ali abstraktni podatkovni tip (*abstract data type* oz. *ADT*). Pri tem lastnosti operacij določajo učinke operacij in so pri formalni definiciji abstraktnega podatkovnega tipa opisane z matematičnimi orodji. Abstraktni podatkovni tip je lahko definiran dodatno tudi s tipom podatkov, ki jih vsebuje, lahko pa ni omejen na poseben tip podatkov, temveč je generičen. Abstraktni podatkovni tip predstavlja matematični model za nek razred pomensko podobnih podatkovnih struktur.

Abstraktni podatkovni tip implementiramo tako, da implementiramo vsako od abstraktnih operacij s svojim postopkom, pri čemer uporabimo ustrezen algoritem. Uporaba implementiranih operacij omogoča manipuliranje s konkretnimi podatkovnimi strukturami skladno z definicijo abstraktnega podatkovnega tipa. Navadno je možno posamezen abstraktni podatkovni tip implementirati z različnimi konkretnimi podatkovnimi strukturami.

Abstraktni podatkovni tipi

Nekatere abstraktne podatkovne tipe, ki uspešno nastopajo pri reševanju mnogih raznolikih problemov in jih zato lahko smatramo za standardne, smo opisali v nadaljevanju poglavja. Omeniti velja, da lahko nek abstraktni podatkovni tip definiramo z različnimi množicami operacij. To še posebej velja za seznam in drevo. V opisu, ki sledi, so zato pri definiciji vsakega abstraktnega podatkovnega tipa zajete predvsem tiste operacije, ki predstavljajo stalnico pri veliki večini njegovih definicij.

Seznam

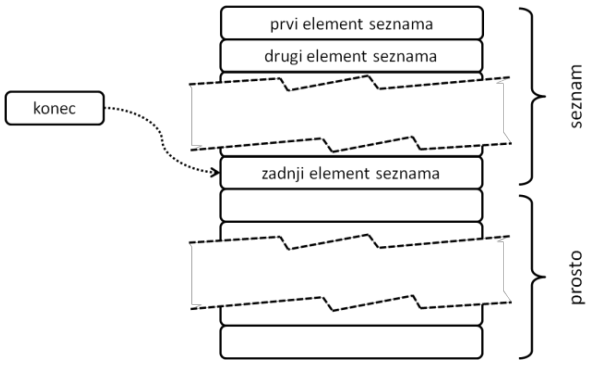
Seznam (*list*) je abstraktni podatkovni tip, ki predstavlja končno zaporedje elementov. Vrstni red elementov v seznamu je pomemben. Isti element se lahko v seznamu pojavi tudi več kot enkrat.

Osnovne operacije nad seznamom so naslednje:
- `ustvari()` vrne referenco na nov prazen seznam.
- `vstavi(p,e)` vstavi element `e` na položaj `p`.
- `nastavi(p,e)` element na položaju `p` zamenja z elementom `e`.
- `element(p)` vrne element, ki se v seznamu nahaja na položaju `p`.
- `briši(p)` iz seznama izbriše element, ki se nahaja na položaju `p`.
- `prazno()` vrne `true`, če je seznam prazen, ali `false`, če seznam ni prazen.

Abstraktni podatkovni tip seznam navadno implementiramo s poljem ali s povezanim seznamom.

Implementacija s poljem

Pri implementaciji s poljem so elementi seznama shranjeni v svojem zaporedju kot zaporedni elementi polja. Operaciji nastavi in beri sta podprti s hitrim dostopom do elementov polja in se zato izvedeta s časovno zahtevnostjo reda $O(1)$. Bolj zahtevno pa je vstavljanje elementa v seznam, saj je potrebno vse elemente seznama, ki sledijo vstavljenemu elementu, v polju premakniti za eno mesto. Podobno velja tudi za operacijo brisanja elementa iz seznama. Obe operaciji se zato izvedeta s časovno zahtevnostjo reda $O(n)$. Zaradi možnosti vstavljanja elementov v seznam mora biti dolžina polja pred vstavitvijo novega elementa večja od dolžine seznama. Zato je del polja lahko večji del uporabe neizkoriščen.

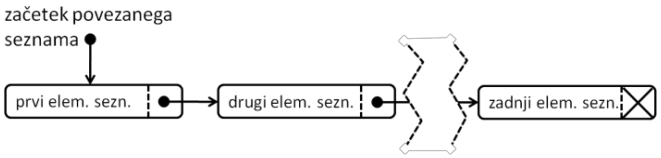


Slika 3.1: Implementacija seznama s poljem.

Pri implementaciji s poljem navadno uporabljamo dinamično polje. To omogoča, da se v primeru, če zaradi vstavljanja novih elementov v seznam zmanjka prostega prostora v polju, celotno polje prepiše v večji pomnilniški prostor.

Implementacija s povezanim seznamom

Povezan seznam je podatkovna struktura, ki jo sestavlja zaporedje vozlišč, ki so med seboj povezana z referencami. Posamezno vozlišče sestavljata dve komponenti: podatkovni del in referenca (ali usmerjena povezava) na naslednji element povezanega seznama, ki povezuje vozlišče z njegovim naslednikom. Referenca zadnjega vozlišča povezanega seznama vsebuje posebno vrednost `null`, ki predstavlja ničto referenco in s tem zaključuje povezan seznam. Elementi seznama so shranjeni vsak v svojem vozlišču, in sicer v njegovem podatkovnem delu v pravem vrstnem redu.



Slika 3.2: Implementacija seznama s povezanim seznamom.

Časovne zahtevnosti operacij pri implementaciji abstraktnega seznama s povezanim seznamom so odvisne od načina podajanja položaja v operacijah. Če je položaj podan kot referenca, potem se vse operacije, vključno z vstavljanjem in brisanjem, izvedejo s časovno zahtevnostjo reda $O(1)$. Če pa je položaj podan z zaporedno številko elementa v seznamu (kakor pri implementaciji s poljem), se operacije vstavi, nastavi, element in briši izvedejo s časovno zahtevnostjo reda $O(n)$, saj je dostop do vozlišča v povezanem seznamu implementiran s sledenjem povezav od prvega do želenega vozlišča.

Vrsta

Abstraktni podatkovni tip vrsta (*queue*) je posebna izvedba seznama, pri katerem lahko elemente dodajamo le na konec seznama, brišemo pa jih lahko le na začetku seznama. Operacija briši izbrisani element tudi vrne, zato jo pri vrsti poimenujemo odstrani. Vrsta predstavlja podatkovno strukturo tipa FIFO (iz angleškega *First In, First Out*), ki določa, da bodo elementi iz vrste odstranjeni v istem vrstnem redu, v katerem so bili v vrsto dodani.

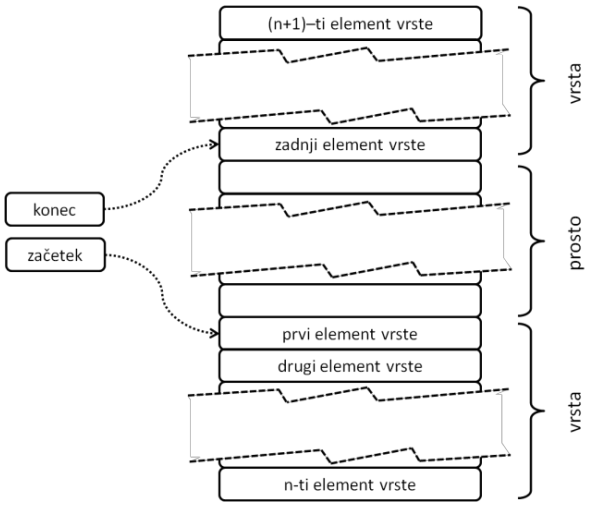
Osnovne operacije nad vrsto so naslednje:

- `ustvari()` vrne referenco na novo prazno vrsto.
- `dodaj(e)` doda element `e` na konec vrste.
- `odstrani()` izbriše in vrne element na začetku vrste.
- `prvi()` vrne element na začetku vrste, ne da bi ga pri tem tudi odstranil iz vrste.
- `prazno()` vrne `true`, če je vrsta prazna, ali `false`, če vrsta ni prazna.

Abstraktni podatkovni tip vrsta lahko, podobno kot seznam, implementiramo s poljem ali s povezanim seznamom. V prvem primeru zaradi večje učinkovitosti algoritmov pri dodajanju in brisanju uporabimo krožno polje.

Implementacija s krožnim poljem

Podatkovna struktura krožno polje je poseben primer polja, pri katerem se položaj začetka zaporedja, shranjenega v polju, spreminja. Zaporedje elementov, ki bi zaradi tega segalo preko konca polja, pa se nadaljuje spet na začetku polja.

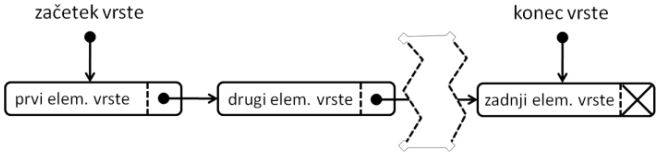


Slika 3.3: Implementacija vrste s krožnim poljem.

Opisani koncept omogoča implementacijo operacije odstrani s časovno zahtevnostjo reda $O(1)$, saj je pri izvedbi brisanje prvega elementa potrebno le spremeniti indikator začetka zaporedja. Prav tako se s časovno zahtevnostjo reda $O(1)$ skoraj vedno izvede tudi operacija dodaj, saj nov element dodajamo na konec vrste. Izjema je le primer, ko polje nima več prostora za nove elemente vrste. V takem primeru je potrebno polje povečati, kar vključuje tudi prepisovanje vsebine polja, zaradi česar se dodajanje izvede s časovno zahtevnostjo reda $O(n)$.

Implementacija s povezanim seznamom

Implementacija vrste s povezanim seznamom se od tovrstne implementacije seznama razlikuje v nekoliko poenostavljenih operacijah, saj se vstavljanje (operacija dodaj) in odstranjevanje vršita le na začetku in na koncu vrste. Ob pomoči dodatne reference na konec povezanega seznama, se obe operaciji izvedeta s časovno zahtevnostjo $O(1)$.



Slika 3.4: Implementacija vrste s povezanim seznamom.

Sklad

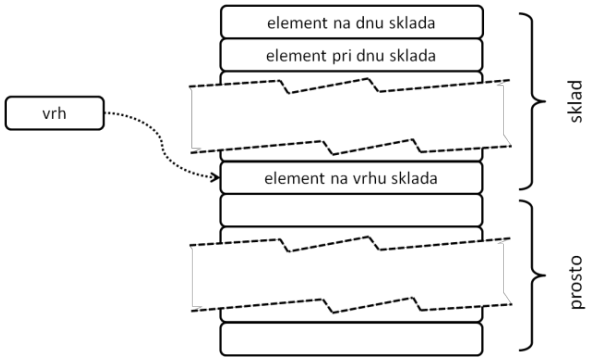
Abstraktni podatkovni tip sklad (*stack*) je posebna vrsta seznama, pri katerem lahko elemente dodajamo in brišemo le na začetku seznama. Tudi pri skladu, podobno kot pri vrsti, nastopa namesto operacije briši operacija odstrani, ki vrne izbrisan element. Sklad predstavlja podatkovno strukturo tipa LIFO (iz angleškega *Last In, First Out*), ki določa, da bodo elementi s sklada odstranjeni v obratnem vrstnem redu, v katerem so bili dodani na sklad. Pri skladu se začetek seznama navadno označuje z vrhom.

Osnovne operacije nad skladom so naslednje:
- `ustvari()` vrne referenco na nov prazen sklad.
- `push(e)` doda element `e` na vrh sklada.
- `pop()` odstrani (in vrne) element na vrhu sklada.
- `vrh()` vrne element na vrhu sklada, ne da bi ga pri tem tudi odstranil s sklada.
- `prazno()` vrne `true`, če je sklad prazen, ali `false`, če sklad ni prazen.

Abstraktni podatkovni tip sklad lahko, podobno kot seznam in vrsto, implementiramo s poljem ali s povezanim seznamom.

Implementacija s poljem

Sklad enostavno implementiramo s poljem tako, da dno sklada postavimo na začetek polja, pozneje dodani elementi sklada pa so v polju shranjeni na lokacijah z višjimi indeksi. Najvišji indeks ima vrh sklada, ki je bil tudi zadnji dodan na sklad.

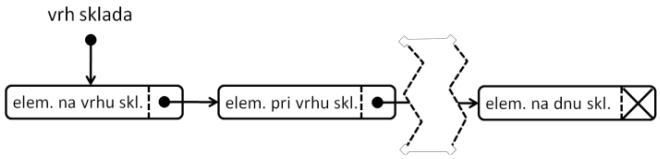


Slika 3.5: Implementacija sklada s poljem.

Ker pri skladu elemente dodajamo (operacija `push`) in odstranjujemo (operacija `pop`) le na vrhu sklada, se ti dve operaciji izvedeta s časovno kompleksnostjo $O(1)$. Podobno kot pri vrsti je izjema le primer, ko polje nima več prostora za nove elemente sklada. V takem primeru je potrebno polje povečati, zaradi česar se dodajanje na sklad izvede s časovno zahtevnostjo reda $O(n)$.

Implementacija s povezanim seznamom

Implementacija sklada s povezanim seznamom je še enostavnejša od tovrstne implementacije vrste. Dodajanje (operacija `push`) in odstranjevanje (operacija `pop`) se vršita le na vrhu sklada, to je na začetku povezanega seznama. Obe operaciji se izvedeta s časovno zahtevnostjo $O(1)$.



Slika 3.6: Implementacija sklada s povezanim seznamom.

Množica

Množica (*set*) je abstraktni podatkovni tip, ki predstavlja pojem matematične množice elementov. Za razliko od seznama, vrste in sklada vrstni red elementov v množici ni pomemben. Od seznama, vrste in sklada se množica razlikuje tudi v tem, da se isti element lahko v množici pojavi največ enkrat (množica nima podvojenih elementov).

Osnovne operacije nad množico so naslednje:
- `ustvari()` vrne referenco na novo prazno množico.
- `dodaj(e)` doda element `e`.
- `briši(e)` izbriše element `e`.
- `vsebuje(e)` vrne `true`, če množica vsebuje element `e`, ali `false`, če množica tega elementa ne vsebuje.
- `prazno()` vrne `true`, če je množica prazna, ali `false`, če množica ni prazna.

Abstraktna množica je pogosto definirana z širšim naborom operacij, med katere najpogosteje sodijo naslednje operacije:
- `popiši()` omogoči prehod preko vseh elementov množice po nekem vrstnem redu.
- `unija(s,t)` vrne unijo množic `s` in `t`.
- `presek(s,t)` vrne presek množic `s` in `t`.
- `razlika(s,t)` vrne razliko množic `s` in `t`.

Abstraktni podatkovni tip množica lahko implementiramo z različnimi podatkovnimi strukturami, med katere sodijo polje, povezan seznam, drevo in zgoščevalna tabela. Ker množica ne dopušča podvajanja elementov, je odločilni dejavnik pri implementaciji najpomembnejših operacij (dodaj, briši in vsebuje) izvedba iskanja elementa v množici.

Implementacija s poljem ali s povezanim seznamom

Množico lahko implementiramo s poljem ali s povezanim seznamom na enak način kot abstraktni seznam, pri tem pa poskrbimo za unikatnost vsebovanih elementov.

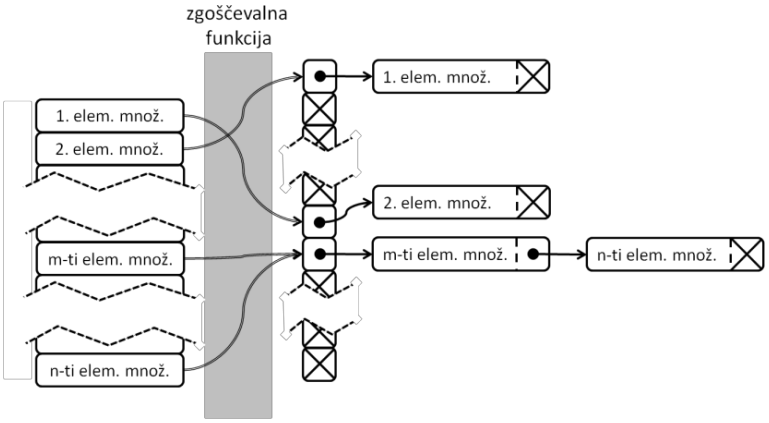
Če je množica implementirana s poljem ali s povezanim seznamom, se iskanje elementa v množici izvaja s časovno zahtevnostjo reda $O(n)$. Kot posledica se operacije dodaj, briši in vsebuje tudi izvajajo s časovno zahtevnostjo reda $O(n)$.

Implementacija z drevesom

Bolje od implementacij množice s poljem ali povezanim seznamom se lahko obnese implementacija množice z drevesom. Ta je ustrezna za urejene množice in v določenih primerih omogoča izvajanje osnovnih operacij dodaj, briši in vsebuje s časovno zahtevnostjo reda $O(log(n))$. Tej zahtevi ustrezajo implementacije binarnih iskalnih dreves, ki so opisana v razdelku *Drevo*.

Implementacija z zgoščeno tabelo

Zgoščena tabela (*hash table*), poznana tudi kot razpršena tabela, je podatkovna struktura, ki omogoča indeksiran dostop do elementov na podlagi vrednosti ključa, ki pripada vsakemu od elementov. Preslikavo med vrednostjo ključa in indeksom podatka opravi zgoščevalna funkcija. Elementi so shranjeni v polju na lokaciji, ki jo določa izračunani indeks.



Slika 3.7: Implementacija množice z zgoščeno tabelo.

Pri uporabi zgoščene tabele je bistvenega pomena izbira zgoščevalne funkcije. Najbolje bi bilo, če bi zgoščevalna funkcija preslikala vsak ključ v drug indeks. To zahtevo je v praksi zelo težko doseči, še posebej zato, ker navadno vnaprej ne poznamo uporabljene podmnožice domene ključev. Zgoščena tabela mora zato reševati problem sovpadanja, ko se dva ali več ključev preslika v isti indeks. Problem sovpadanja se v praksi rešuje na več načinov. Pogosto se uporabi pristop z odprto zgoščeno tabelo, ki v polju na vsaki lokaciji ne hrani samega elementa (ali reference na element), temveč referenco na seznam elementov, ki pripadajo indeksu lokacije.

Zgoščena tabela ob uporabi ustrezne zgoščevalne funkcije v povprečju omogoča izvajanje osnovnih operacij dodaj, briši in vsebuje s časovno zahtevnostjo reda $O(1)$.

Zavedati pa se moramo, da se lahko ob neustrezni zgoščevalni funkciji osnovne operacije izvajajo s časovno zahtevnostjo reda $O(n)$.

Množico lahko implementiramo z zgoščeno tabelo tako, da elementi množice predstavljajo ključe, medtem ko podatki, shranjeni v polju (ali v seznamih, ki jih posreduje polje), predstavljajo le zapise, ki nakazujejo, ali dani elementi pripada množici ali ne. Implementacija množice z zgoščeno tabelo je, za razliko od implementacije z drevesom, primerna tudi za neurejene množice.

Slovar

Abstraktni podatkovni tip slovar (*dictionary* ali *map*), poznan tudi kot preslikava ali asociativno polje, predstavlja zbirko parov (ključ, element), v kateri se lahko posamezen ključ pojavi največ enkrat in vrstni red parov v slovarju ni pomemben.

Osnovne operacije nad slovarjem so:
- `ustvari()` vrne referenco na nov prazen slovar.
- `dodaj(k,e)` doda par ključ `k` in element `e`.
- `odstrani(k)` odstrani par, ki vsebuje ključ `k`, in vrne pripadajoči element.
- `preslikaj(k)` vrne element, ki ustreza ključu `k`.

Implementacije slovarja so sorodne implementacijam množice, kar je razvidno že iz podobnosti njunih definicij. Pri tem operacija preslikaj ustreza operaciji vsebuje pri množici. Pri uporabi slovarja je še posebej pomembno učinkovito iskanje po slovarju, zato imajo pri slovarju prednost implementacije z učinkovito izvedbo operacije preslikaj.

Drevo

Abstraktni podatkovni tip drevo (*tree*) predstavlja hierarhično strukturirano zbirko elementov. Drevesa so primerna za predstavitev hierarhično organiziranih struktur, kot so na primer družinsko drevo, ki predstavlja rodovnik neke osebe, organizacijska struktura nekega podjetja ali matematični algebrski izraz.

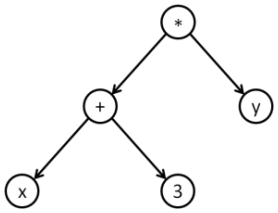
Posamezni elementi so vsebovani v vozliščih drevesa. Vsako vozlišče ima nič ali več sinov in največ enega očeta. Vrstni red otrok posameznega očeta je pomemben in se šteje od leve proti desni. Otrokom istega očeta pravimo brati. V nepraznem drevesu obstaja natanko eno vozlišče, ki nima očeta in mu pravimo koren drevesa. Nekatera vozlišča nimajo sinov. Takim vozliščem pravimo listi, ostalim pa notranja vozlišča.

Globina nekega vozlišča je dolžina poti od korena do tega vozlišča, izražena v številu vozlišč na poti. Višina drevesa je globina njegovega najglobljega lista. Prazno drevo ima višino enako 0.

Če obstaja pot od vozlišča `a` do vozlišča `b` in je globina vozlišča `a` manjša od vozlišča `b`, potem je vozlišče `a` prednik vozlišča `b` in vozlišče `b` je naslednik vozlišča `a`.

Poddrevo je del drevesa, ki sam po sebi tudi predstavlja neko drevo. Vsako vozlišče v drevesu je tudi koren nekega poddrevesa, ki ga sestavlja skupaj s svojimi nasledniki.

Na sliki 3.8 je prikazano drevo, ki predstavlja matematični izraz `(x+3)*y`. Koren tega drevesa je vozlišče `*`, višina drevesa pa je enaka 3. Listi drevesa so vozlišča `x`, `3` in `y`, vozlišče `+` pa je notranje vozlišče. V tem drevesu je vozlišče `+` prednik vozlišča `x`, vozlišče `x` pa je naslednik vozlišča `*`. Globina listov `x` in `3` je enaka 3, medtem ko je globina lista `y` enaka 2. Vozlišče `+` ima tudi globino 2. Koren `*` ima globino 1.



Slika 3.8: Matematični izraz `(x+3)*y`, predstavljen v drevesni strukturi.

Podobno kot pri seznamu lahko nad drevesom izvajamo raznolike operacije. Osnovne operacije nad splošnim drevesom so naslednje:

- `ustvari()` vrne referenco na novo prazno drevo.
- `ustvari(e,`t_1`,`t_2`,...,`t_i`)` vrne referenco na novo drevo z elementom `e` v korenskem vozlišču in z drevesi t_1, t_2, ... , t_i kot otroci. Če je `i` enak 0, potem se ustvari drevo z enim samim vozliščem (samo s korenom).
- `koren()` vrne koren drevesa.
- `oče(n)` vrne očeta vozlišča `n`.
- `leviOtrok(n)` vrne najbolj levega otroka vozlišča `n`.
- `desniBrat(n)` vrne desnega brata vozlišča `n`.
- `element(n)` vrne element v vozlišču `n`.

Podani splošni opis drevesa dopušča obstoj široke množice dreves. S splošnim opisom je na primer skladno tudi tako drevo, pri katerem ima vsako notranje vozlišče le enega samega otroka in je zato drevo izrojeno v seznam.

Ker v splošnem vsa drevesa ne nudijo prednosti posebnih izvedenk dreves, kakor je na primer iskanje elementa v drevesu s časovno zahtevnostjo reda $O(log(n))$, je smiselno posebne vrste dreves obravnavati posebej. Med temi so še posebej pomembna drevesa, ki sodijo med binarna iskalna drevesa.

Binarno iskalno drevo

Binarno (ali dvojiško) drevo je drevo, pri katerem velja, da ima vsako vozlišče največ dva otroka. Pri binarnem drevesu otroke nekega vozlišča ne obravnavamo kot zaporedje otrok od leve proti desni, temveč obravnavamo posebej morebitnega levega in morebitnega desnega otroka. To pomeni, da ima lahko vozlišče v binarnem drevesu tudi samo desnega otroka.

Binarno iskalno drevo je binarno drevo z naslednjimi lastnostmi:

- Levo poddrevo korena vsebuje le vozlišča z elementi, ki so manjši od elementa korena.
- Desno poddrevo korena vsebuje le vozlišča z elementi, ki so večji od elementa korena.
- Levo in desno poddrevo korena sta oba binarni iskalni drevesi.

Ena od bistvenih odlik binarnih iskalnih dreves je možnost hitrega izvajanja operacije iskanja elementa v drevesu. Zaradi tega se binarna iskalna drevesa uporabljajo pri implementaciji nekaterih abstraktnih podatkovnih tipov kot sta množica in slovar.

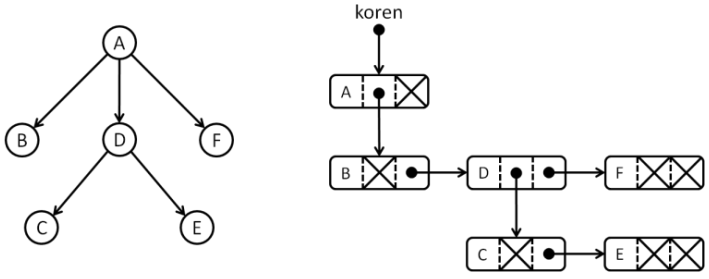
Iskanje elementa v binarnem iskalnem drevesu poteka tako, da najprej preverimo, ali koren vsebuje iskani element. Če temu ni tako, se na podlagi primerjave med vrednostjo elementa v korenu in iskanega elementa odločimo, ali bomo z iskanjem elementa nadaljevali v levem ali v desnem poddrevesu korena. Postopek nato nadaljujemo, dokler elementa ne najdemo ali pa ustrezno poddrevo, v katerem bi nadaljevali z iskanjem, ne obstaja.

Očitno je, da je potrebno operacijo primerjanja ponoviti največ h-krat, če je h višina drevesa, v katerem iščemo element. Operacija iskanja se bo torej izvajala hitreje pri nižjih drevesih kakor pri višjih. Višino drevesa lahko minimiziramo, če pri izgradnji drevesa poskrbimo za njegovo uravnovešenost. Drevo je uravnovešeno (ali poravnano), če se globini poljubnih dveh listov drevesa razlikujeta največ za 1. V tem primeru je višina drevesa minimalna in pri binarnih drevesih z n vozlišči znaša $\lfloor log_2(n) \rfloor + 1$.

Ugotovimo lahko torej, da je pri uravnovešenih binarnih iskalnih drevesih časovna zahtevnost izvajanja iskanja elementa v drevesu reda $O(log(n))$. V praksi uporabljena iskalna drevesa, ki omogočajo tovrstno časovno zahtevnost, navadno niso popolnoma uravnovešena, velja pa, da je njihova višina velikostnega reda $O(log(n))$. Primeri takih dreves so na primer rdeče-črna drevesa ali AVL drevesa.

Implementacija drevesa s povezano strukturo

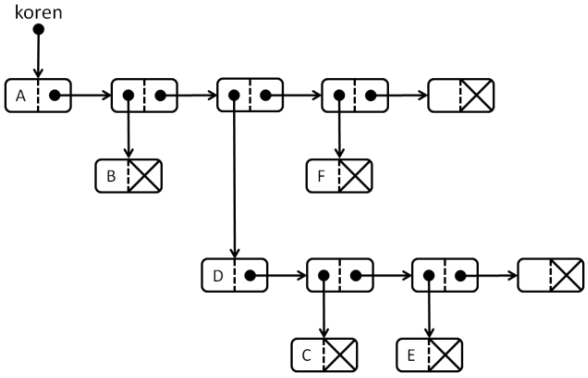
Podobno kot ostale opisane abstraktne strukture lahko tudi drevo implementiramo s poljem ali s povezano strukturo. Na tem mestu se bomo posvetili le implementaciji s povezano strukturo, ki je pogosto bolj primerna za uporabo.



Slika 3.9: Implementacija drevesa s povezano strukturo, pri kateri vsebuje vsako vozlišče referenco na svojega levega otroka in na svojega desnega brata.

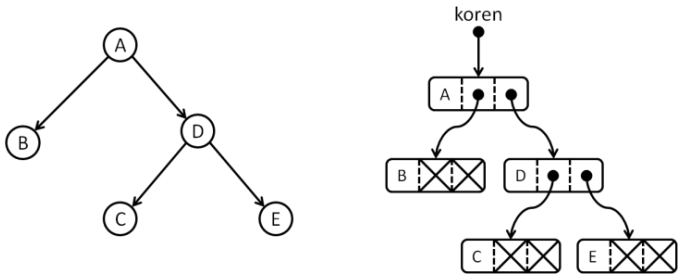
Primer take implementacije je na sliki 3.9, kjer je na levi strani prikazano drevo, na desni strani pa predstavitev implementacije tega drevesa s povezano strukturo.

Drevo navadno implementiramo s strukturo vozlišč, ki so med seboj povezana z referencami. Pri tem lahko uporabimo način predstavitve strukture drevesa, pri kateri vsebuje vsako vozlišče referenco na svojega levega otroka in na svojega desnega brata (slika 3.9), ali pa takega, kjer vsako vozlišče vsebuje reference na vse svoje otroke, kot ga prikazuje slika 3.10.



Slika 3.10: Implementacija drevesa s slike 3.9 s povezano strukturo, pri kateri vsebuje vsako vozlišče referenco na seznam vseh svojih otrok.

Kadar imamo v vsakem vozlišču reference na vse otroke tega vozlišča, je lahko referenca na otroke podana posredno z referenco na seznam otrok (ta primer prikazuje slika 3.10) ali pa kar neposredno z referencami na vsakega otroka posebej. Slednje je še posebej primerno za binarna drevesa, kjer je število otrok vnaprej omejeno na 2. Na sliki 3.11 je prikazano binarno drevo ter njegova implementacija s povezano strukturo, pri kateri imamo v vsakem vozlišču tudi neposredni referenci na oba otroka tega vozlišča.



Slika 3.11: Implementacija binarnega drevesa s povezano strukturo, pri kateri vsebuje vsako vozlišče neposredni referenci na svoja otroka.

DRUGI DEL
Programski jezik Java

4. Programiranje v jeziku Java

Izbira delovnega okolja za Javo je lahko poljubna. Jezik Java je namreč neodvisen od operacijskega sistema, saj se izvaja znotraj javanskega navideznega stroja (*Java Virtual Machine* ali JVM). Nekaj manjših razlik med različnimi operacijskimi sistemi najdemo le v podrobnostih, kot je na primer zapis datotečne poti, vendar tudi te lahko obvladamo programsko. Primere v tem gradivu smo izdelali v okolju Linux.

Priprava platforme za delo

Za razvoj javanskih programov priporočamo uporabo razvojnega kompleta *JDK* (*Java SE JDK*), ki vključuje tudi ustrezen prevajalnik, in preprostega urejevalnika besedil. V tem gradivu bomo uporabljali *Java SE JDK 6*, čeprav večina primerov deluje tudi v starejših različicah.

Java SE 6 JDK

Za razvoj javanskih programov torej potrebujemo razvojni komplet *Java SE 6 JDK* (SE pomeni *Standard Edition*, JDK pa je kratica za *Java Development Kit*), ki je tudi najprimernejše orodje za učenje Jave. Med drugim vključuje različna orodja JDK (kot je na primer prevajalnik), izvajalno okolje (*Java Runtime Environment* ali JRE) ter knjižnice. *Java SE JDK 6* lahko brezplačno prenesete s spletnega naslova http://java.sun.com/javase/downloads/index.jsp.

Dokumentacija

Obsežna dokumentacija JDK zajema podrobneje opisane razrede in njihovo uporabo. Predvsem je zanimiva specifikacija programskega vmesnika API (*API Specification*), v kateri so po paketih navedeni vsi javanski razredi (skupaj z vmesniki, izjemami, napakami ...), za vsak razred pa najdemo uvodni opis razreda, pregled vseh njegovih atributov, konstruktorjev in metod ter njihov podrobnejši opis. Dokumentacijo *Java SE 6 Documentation* lahko pregledujemo preko spleta (java.sun.com/javase/6/docs) ali pa jo v obliki stisnjene ZIP datoteke prenesemo s spletnih strani *Java SE Downloads* (java.sun.com/javase/downloads/index.jsp) ter jo razširimo v poljubno mapo v našem računalniku.

Nastavitev poti do orodij

Orodja JDK se namestijo v podmapo `bin` tiste mape, v katero smo namestili JDK. Za enostaven dostop do teh orodij moramo nastaviti pot do omenjene mape. To naredimo tako, da spremenimo sistemsko spremenljivko `PATH` in ji dodamo pot do mape `bin`. Način spreminjanja spremenljivke `PATH` je odvisen od operacijskega sistema.

Druga sistemska spremenljivka, ki jo nastavimo po potrebi, se imenuje `CLASSPATH`. Uporablja jo izvajalno okolje JRE, spremenljivka pa določa pot do prevedenih razredov (`.class` datotek). Za naše primere zadostuje, da spremenljivke `CLASSPATH` nimamo nastavljene (spremenljivka sploh ne obstaja v sistemskem okolju). Če spremenljivka obstaja in je napačno nastavljena, imamo lahko težave pri izvajanju programov. Zato je priporočljivo, da v pot `CLASSPATH`, če le-ta obstaja, dodamo tudi pot do tekočega direktorija (`.`).

Razvojna orodja

Čeprav je za učenje jezika Java in razvoj programov iz tega gradiva povsem primeren razvojni komplet JDK, nam pri razvoju obsežnejših projektov ta ne zadostuje več. V tem primeru lahko uporabimo različne razvojne pripomočke (brezplačne ali plačljive), ki ponavadi vključujejo urejevalnik, prevajalnik, izvajalno okolje z možnostjo razhroščevanja in obsežno dokumentacijo ter tako omogočajo hitrejši in enostavnejši razvoj programske kode. Primer takih integriranih razvojnih orodij, ki jih lahko dobimo na spletu, so *Eclipse* (www.eclipse.org), *NetBeans* (www.netbeans.org), *BlueJ* (www.bluej.org) ali *JBuilder* (www.codegear.com/products/jbuilder).

Pisanje, prevajanje in izvajanje programov

Najprej bomo za ogrevanje ponovili, kako napišemo enostaven program v programskem jeziku Java in kaj vse moramo narediti, preden si lahko ogledamo rezultate (delovanje) našega programa.

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

Pisanje izvorne kode programa

Vsak program začnemo s pisanjem izvorne kode, ki jo lahko napišemo v poljubnem urejevalniku besedila. Pomembno je le, da nam urejevalnik besedila omogoča shranjevanje vsebine kot navadno besedilo (*plain text*). Navadno je preprost urejevalnik že sestavni del operacijskega sistema, kot sta na primer *Beležnica* (*Notepad*) v Windows okolju ali *vi* (oziroma izboljšana različica *vim*) v Linux/Unix okolju. Boljša izbira pa sta uporabniku prijaznejša urejevalnika *Notepad++* (notepad-plus.sourceforge.net) v Windows okolju ali *gedit* v Linux/Unix okolju. Seveda je uporaba določenega urejevalnika stvar okusa in navad vsakega posameznika. Pomembno je le, da je urejevalnik enostaven za uporabo in da ga dovolj dobro poznamo.

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

```
public class Primer {

  public static void main(String[] args) {
    System.out.println("Pozdravljeni!\n");
  }

}
```

Vsebino shranimo kot navadno besedilo v datoteko z imenom `Primer.java` (pri tem pazimo, da je ime datoteke res tako, kot smo ga podali tu). Imena datotek, ki vsebujejo izvorno kodo Java, morajo vedno imeti podaljšek `.java`, samo ime datoteke pa mora biti enako imenu razreda, v katerem je definirana metoda `main`. Pri tem moramo paziti, da se ime datoteke popolnoma ujema z imenom razreda (velika začetnica), saj Java loči velike in male črke.

Prevajanje programa

Program prevedemo v ukazni lupini (odpremo terminalsko okno oziroma ukazno vrstico), kjer za prevajanje uporabimo javanski prevajalnik, ki ga pokličemo z ukazom `javac`, kateremu sledi ime datoteke z izvorno kodo.

Naš program, katerega izvorna koda se nahaja v datoteki `Primer.java`, prevedemo z ukazom:

```
javac Primer.java
```

Rezultat prevajanja (če seveda med prevajanjem ni prišlo do napake zaradi napačno napisane kode) je javanska vmesna koda (*Java bytecode*), ki se zapiše v datoteko, katere osnovno ime je enako imenu datoteke z izvorno kodo, podaljšek pa je `.class`. V našem primeru se ob prevajanju ustvari datoteka `Primer.class`.

Izvajanje programa

Tudi izvajanje programa poteka v ukazni lupini. Vmesno kodo, ki smo jo ustvarili v prejšnjem koraku, lahko izvajamo v posebnem, izvajalnem okolju. Zaženemo ga z ukazom `java`, kateremu sledi ime prevedene datoteke brez podaljška. V našem primeru bi program pognali z ukazom:

```
java Primer
```

In kaj naredi naš program? Najprej izpiše:

```
Pozdravljeni!
```

in nato skoči v novo vrstico ter konča.

5. Zbirke podatkovnih struktur

Zbirka (*collection*) je objekt, ki združuje več elementov. Zbirko uporabimo za shranjevanje elementov, omogoča pa tudi vračanje elementov ter upravljanje z njimi. Najpreprostejša zbirka, ki jo že poznate, je tabela elementov, ki pa programerju nudi le malo možnosti za lažje delo in upravljanje elementov, ki so shranjeni v njej.

Ogrodje zbirk (*collection framework*) zajema celotno strukturo razredov (skupaj z vmesniki in abstraktnimi razredi), ki določajo delo z zbirkami podatkovnih struktur. Java nudi v ta namen temeljni vmesnik `Collection`, ki povzema značilnosti vseh zbirk. Iz njega so izpeljani vrsta `Queue` (vstavlja v vrsto elemente, ki čakajo na obdelavo; primer je FIFO vrsta), seznam `List` (zaporedje elementov) in množica `Set` (brez podvojenih elementov). Slednja je lahko tudi urejena, kar določa vmesnik `SortedSet`.

Vse zbirke omogočajo določene temeljne operacije, kot so:
- dodajanje elementa v zbirko (`add()`),
- odstranjevanje elementa iz zbirke (`remove()`),
- preverjanje, ali zbirka vsebuje določen element (`contains()`),
- preverjanje, ali je zbirka prazna (`isEmpty()`),
- preverjanje velikosti zbirke (`size()`) ali
- pretvorbo zbirke v polje elementov (`toArray()`).

Preko zbirke se sprehodimo z uporabo iteratorjev (vmesnik `Iterator`) ali pa s konstruktom `for-each`, ki omogoča enostaven sprehod po vseh elementih zbirke z uporabo zanke `for`.

Poseben vmesnik, ki ni izpeljan iz vmesnika `Collection`, nudi tudi slovar `Map` (povezuje ključe in vrednosti) ter iz njega izpeljan urejen slovar `SortedMap`, pri katerem so ključi urejeni.

Navedeni vmesniki omogočajo, da uporabljamo zbirke neodvisno od podrobnosti njihovih predstavitev.

Osnovne implementacije zbirk v Javi so zgoščena tabela (*hashtable*), prilagodljivo polje (*resizable array*), uravnoteženo drevo (*balanced tree*), povezan seznam (*linked list*) ter zgoščen seznam (*hashtable linked list*). Vsaki implementaciji pripadajo določeni razredi; nekatere od njih bomo spoznali v nadaljevanju.

Iz opisanega vidimo, da ima Java že vgrajeno podporo glavnim abstraktnim podatkovnim tipom in njihovim implementacijam, ki smo jih spoznali v 3. poglavju.

Seznami

Seznami so zbirke elementov v znanem zaporedju, torej ima vsak element točno določeno mesto v seznamu. V tem so seznami pomensko podobni poljem, saj vsakemu

elementu pripada natančno določen položaj (indeks). Vendar pa seznami nimajo vnaprej določene dolžine in se njihova dolžina lahko dinamično spreminja (polja tega ne omogočajo). Seznami omogočajo hranjenje podvojenih elementov.

Sezname lahko izvedemo s prilagodljivim poljem `ArrayList` ali pa s povezanim seznamom `LinkedList`.

Poglejmo si uporabo seznama `ArrayList` na primeru. Seznam bomo uporabili za shranjevanje naključnega števila celih števil, prikazali uporabo operacij za dodajanje elementa v seznam in za odstranjevanja elementa iz seznama ter za izpis vseh elementov seznama.

Ogrodje zbirk (vsi razredi in vmesniki) je opisano v paketu `java.util`, zato moramo na začetki programa napovedati uporabo tega paketa:

```
import java.util.*;
```

Seznam deklariramo kot objekt razreda `ArrayList`, kateremu v oklepajih <> podamo tip elementov, ki jih hrani (tako uporabo imenujemo generična koda). Tip mora biti določen z razredom, saj seznam `ArrayList` lahko hrani le objekte. V našem primeru, ko gre za cela števila (tip `int`) smo torej podali razred `Integer`:

```
ArrayList<Integer> sez;
```

Klic konstruktorja ustvari prazen seznam z začetno kapaciteto desetih elementov:

```
sez = new ArrayList<Integer>();
```

Nov element dodamo v seznam s pomočjo metode `add()`, ki doda element na konec seznama. Objekt, ki ga dodajamo, podamo kot argument metode. Zato moramo pred tem iz primitivnega tipa `int` ustvariti objekt `Integer` z isto vrednostjo. Če želimo dodati vrednost `5`, to naredimo v dveh korakih:

```
Integer i = new Integer(5);
sez.add(i);
```

ki pa ju lahko tudi združimo:

```
sez.add(new Integer(5));
```

Če metodi `add()` dodamo še en argument, to je pozicijo novega elementa, dosežemo vrivanje elementa v seznam na točno določeno mesto. Število `5` dodamo na prvo mesto v seznamu z naslednjim stavkom:

```
sez.add(0, new Integer(5));
```

Pri dodajanju novih elementov v seznam pa lahko uporabimo tudi samodejno pretvarjanje primitivnih tipov (*boxing/unboxing*). To pomeni, da programerju ni potrebno skrbeti za pretvorbo primitivnega tipa `int` v ovijalni tip `Integer`, temveč za

ustrezno pretvorbo poskrbi kar sam prevajalnik. Tako lahko oba stavka za dodajanje elementa zapišemo krajše kot:

```
sez.add(5);
sez.add(0, 5);
```

Element na določeni poziciji odstranimo iz seznama z metodo `remove()`, kateri kot argument podamo pozicijo elementa v seznamu. Prvi element seznama torej odstranimo s stavkom:

```
sez.remove(0);
```

Seveda je pred tem smiselno preveriti, ali je v seznamu sploh kakšen element. Metoda `isEmpty()` vrne `true`, če je seznam prazen:

```
if(!sez.isEmpty())
  sez.remove(0);
```

Kadar želimo izbrisati iz seznama vse elemente, lahko uporabimo metodo `clear()`, ki počisti seznam:

```
sez.clear();
```

Za izpis seznama bomo napisali svojo metodo, ki se bo sprehodila preko vseh elementov seznama in izpisovala njihove vrednosti. Naj se metoda imenuje `izpisiSeznam`, kot argument pa prejme referenco na seznam, katerega elemente želimo izpisati. V našem primeru gre za seznam `ArrayList` objektov `Integer`:

```
public static void izpisiSeznam(ArrayList<Integer> s)
```

Telo metode sestavlja `for` zanka, ki gre preko vseh indeksov v seznamu (od `0` do `size()`) ter za vsakega od indeksov pokliče metodo `get()`, ki vrne element s podanim indeksom. Vrednost tako dobljenega elementa pretvorimo v niz in jo izpišemo:

```
for(int i=0; i<s.size(); i++)
  System.out.print(s.get(i).toString() + "  ");
```

Seveda lahko tudi to zanko nekoliko poenostavimo. Kot prvo izkoristimo samodejno ovijanje primitivnih tipov in metodo `printf`, ki omogoča lažje formatiranje izpisa:

```
for(int i=0; i<s.size(); i++)
  System.out.printf(" %d ", s.get(i));
```

Še enostavnejša pa je uporaba konstrukta `for-each`, ki omogoča enostavnejši (in tudi bolj pregleden) prehod preko vseh elementov zbirke. Tako zanko zapišemo tudi z rezervirano besedo `for`, kateri v oklepajih sledi tip elementa, znak `:` in ime zbirke. V našem primeru bi zanka izgledala takole:

```
for(int i: s)
  System.out.printf(" %d ", i);
```

Zapisano sintakso bi lahko prebrali kot "za vsak element `i`, ki je tipa `int`, v zbirki `s`". Pri tem moramo opozoriti, da na tak način lahko elemente zbirke le pregledujemo, ne moremo pa jih spreminjati.

Še korak naprej k poenostavitvi izpisa elementov seznama po lahko naredimo, če uporabimo prekrito metodo `toString()` razreda `ArrayList` (ta jo podeduje od razreda `AbstractCollection`). Metoda vrne niz vseh elementov seznama po vrsti, ločenih z vejico, v oglatih oklepajih. Taka predstavitev seznama je za nas povsem primerna, zato jo lahko uporabimo pri našem izpisu:

```
System.out.println("Seznam vsebuje elemente: " + s);
```

Razred `ArrayList` ima na zalogi še cel kup drugih metod, ki jih tukaj nismo omenili, lahko pa si jih pogledate v JDK dokumentaciji.

Sestavimo sedaj program, ki bo prikazoval delo s seznamom. V programu bomo najprej s pomočjo generatorja naključnih števil izbrali število med `0` in `MAX` (ki je konstanta) ter napolnili seznam po vrsti s števili od `0` do izbranega (naključnega) števila.

```
int meja = (int) (Math.random()*MAX);
for(int i=0; i<=meja; i++)
  sez.add(i);
```

Potem uporabimo različne operacije nad seznamom, ki smo jih opisali zgoraj: odstranimo prvi element seznama (če le-ta ni prazen), dodamo novo število na začetek seznama ter na koncu seznam še izpraznimo (odstranimo vse elemente). Po vsaki operaciji tudi izpišemo vsebino seznama in trenutno število elementov v seznamu s pomočjo metode `izpisiSeznam`.

Datoteka `Seznam.java` vsebuje izvorno kodo programa, kjer smo upoštevali tudi že vse omenjene poenostavitve kode.

Vektor in sklad

Razred `Vector` je starejša različica po velikosti spremenljivega polja objektov, saj izhaja iz JDK 1.0. Z uvedbo ogrodja zbirk so nekoliko spremenili tudi razred `Vector`, tako da implementira vmesnik `List`. Najpogostejše operacije nad vektorjem, kot so metode za dodajanje, brisanje in vračanje elementov `insertElementAt(e,i)`, `addElement(e)`, `setElementAt(e,i)`, `removeElementAt(i)` in `elementAt(i)`, so tako dobile tudi nekoliko krajša imena, ki so skladna z metodami vmesnika `List`: `add(i,e)`, `add(e)`, `set(i,e)`, `remove(i)` in `get(i)`, po vrsti.

Uporaba vektorjev je podobna kot uporaba ostalih vrst seznamov. Ustvarimo ga s klicem konstruktorja, ki mu določimo tudi tip elementov:

```
Vector<Integer> vektor = new Vector<Integer>();
```

Zgoraj navedene metode bi lahko na ustvarjenem vektorju uporabili na naslednji način:

```
vektor.addElement(10);
vektor.insertElementAt(15, 0);
vektor.setElementAt(5, 0);
int stevilo = vektor.elementAt(0);
vektor.removeElementAt(0);
```

oziroma enakovredno z uporabo nam bolj domačih metod vmesnika `List`:

```
vektor.add(10);
vektor.add(0, 15);
vektor.set(0, 5);
int stevilo = vektor.get(0);
vektor.remove(0);
```

V splošnem je razred `Vector` ekvivalenten razredu `ArrayList`, le da je sinhroniziran. To pomeni, da so vse metode razreda `Vector` nitno varne (*thread safe*) in jih lahko uporabljamo v več nitih, ne da bi pri tem morali sami skrbeti za sinhronizacijo. Seveda pa uporaba sinhronizacije vpliva tudi na performanse in je zato uporaba vektorja nekoliko počasnejša. Zato se v primerih, ko ne potrebujemo sinhronizacije (npr. ko je seznam lokalna spremenljivka ali pa ga spreminjamo v eni sami niti), priporoča uporaba razreda `ArrayList`, saj je hitrejši v primerjavi z razredom `Vector`. Razred `ArrayList` namreč ni sinhroniziran, podobno kot tudi vsi drugi razredi, ki so prišli z ogrodjem zbirk.

Včasih želimo elemente seznama obravnavati na poseben način, tako da jih odlagamo v seznam po vrsti, iz seznama pa jih jemljemo v obratnem vrstnem redu. Tako uporabo seznama imenujemo sklad, saj elemente nalagamo na kup, jemljemo pa jih vedno z vrha kupa (zadnji vstavljen element je tudi prvi odstranjen s sklada).

V Javi je sklad predstavljen z razredom `Stack`, ki je izpeljan iz razreda `Vector`. Razširja ga z metodami, ki omogočajo uporabo vektorja kot sklad. Razred `Stack` tako nudi nekaj posebnih metod za delo s skladom: metoda `push(e)` postavi element `e` na sklad (na vrh sklada), metoda `pop()` vrne element na vrhu sklada in ga hkrati tudi odstrani s sklada, metoda `peek()` pa vrne prvi element na skladu, ne da bi pri tem sklad kakorkoli spremenila.

Še primer ustvarjanja sklada in uporabe omenjenih metod:

```
Stack<Integer> sklad = new Stack<Integer>();
int element;
sklad.push(10);
element = sklad.peek();
element = sklad.pop();
```

Množice

Množica je primer zbirke, ki ne more vsebovati podvojenih elementov. Opisuje jo vmesnik `Set`, ki med drugim določa tudi standardne operacije nad množicami, kot so unija, presek, podmnožica ter razlika dveh množic.

Če imamo množici `m1` in `m2`, lahko nad njima izvedemo standardne operacije na naslednji način:

- `m1.containsAll(m2)` vrne vrednost `true`, če je množica `m2` podmnožica množice `m1` (množica `m1` vsebuje vse elemente, ki so v množici `m2`);
- `m1.addAll(m2)` spremeni množico `m1` v unijo množic `m1` in `m2` (unija je množica, ki vsebuje vse elemente, ki so vsebovani v kateri od obeh množic);
- `m1.retainAll(m2)` spremeni množico `m1` v presek množic `m1` in `m2` (presek je množica, ki vsebuje le tiste elemente, ki so vsebovani v obeh množicah);
- `m1.removeAll(m2)` spremeni množico `m1` v razliko množic `m1` in `m2` (razlika množic `m1` in `m2` je množica, ki vsebuje vse elemente množice `m1`, ki niso vsebovani v množici `m2`).

Množico opisuje vmesnik `Set`, izvedemo pa jo lahko z zgoščeno tabelo (razred `HashSet`), z uravnoteženim (rdeče-črnim) drevesom (razred `TreeSet`) ali z zgoščenim seznamom (razred `LinkedHashSet`). Prva izvedba je primernejša za neurejeno množico za splošno rabo, druga za urejeno množico elementov, tretja pa množico uredi po zaporedju dodajanja elementov. V našem primeru bomo uporabili neurejeno množico.

Iz nizov, ki so podani kot argumenti programa, bomo sestavili množico besed. Ker se podvojene besede ne shranjujejo v množico, bomo na koncu z izpisom elementov množice dobili seznam vseh različnih besed.

Množico deklariramo kot objekt razreda `HashSet`, kateremu v oklepajih <> podamo tip elementov, ki jih hrani (generična koda). Tip mora biti določen z razredom, saj množica `HashSet` lahko hrani le objekte. Naši objekti bodo tipa `String`, saj bomo hranili besede (podane argumente):

```
HashSet<String> m = new HashSet<String>();
```

Za vstavljanje elementov v množico uporabimo metodo `add()`. Ta metoda doda podani element k elementom množice, če elementa še ni v množici. V primeru uspešnega dodajanja metoda vrne vrednost `true`. V zanki se sprehodimo preko polja nizov `args` ter v množico dodajamo vsak argument posebej:

```
for(int i=0; i<args.length; i++)
  if(!m.add(args[i]))
    System.out.println("Podvojena: " + args[i]);
```

V primeru, da element ni bil uspešno dodan v množico (kar pomeni, da gre za podvojen element), smo ta podvojeni element (besedo) tudi izpisali.

Na koncu želimo izpisati še vse elemente tako zgrajene množice. Za izpis množice smo se tudi tukaj oprli na prekrito metodo `toString()` razreda `HashSet` (tudi ta jo podeduje od razreda `AbstractCollection`), ki poskrbi za primeren izpis elementov množice. Metoda `size()` tudi tukaj vrne število elementov v množici:

```
System.out.println("Skupaj je bilo " + m.size() +
                   " različnih besed: " + m);
```

Datoteka `Mnozica.java` hrani izvorno kodo tega primera. Pri izvajanju programa ne smemo pozabiti podati argumentov (poljubne besede), kajti brez njih bo množica vedno prazna. Primer klica programa:

```
java Mnozica prva druga prva tretja beseda druga konec
```

Vrste

Vrste so zbirke elementov, ki so na nek način urejeni. Elemente dodajamo urejeno v vrsto, odvzemamo pa jih vedno na začetku vrste (pri glavi). Tipično se uporabljajo za hranjenje elementov, ki čakajo na obdelavo (kot na primer vrsta ljudi pred blagajno v trgovini).

Vrste imajo lahko različne urejenosti. Najbolj tipična urejenost vrste je FIFO (*First In First Out*) način, pri katerem elemente dodajamo na konec vrste (prvi vstavljen element je tako tudi prvi odvzet). Zelo uporabne so tudi prioritetne vrste, ki uredijo elemente glede na njihovo prioriteto (glede na podano primerjalno funkcijo ali pa na naravno ureditev elementov). Tudi sklad, ki smo ga že omenili v tem poglavju, predstavlja poseben tip vrste, in sicer vrsto LIFO (*Last In First Out*), pri kateri elemente vstavljamo na začetek vrste.

Vsak tip vrste torej uporablja svoja pravila za dodajanje oz. vstavljanje elementov. Ne glede na urejenost elementov v vrsti pa je vedno glava vrste tisti element, ki ga odstranimo iz vrste ob klicu metode `remove()` ali `poll()`.

Vrste so po obnašanju in uporabi precej podobne seznamom. Glavna razlika med njimi je, da iz seznama lahko odvzamemo poljuben element, pri vrsti pa je na voljo le prvi element vrste.

Vrsta `Queue` je vmesnik (*interface*), ki omogoča dodajanje elementov v vrsto in odvzemanje elementov iz glave vrste. Zato ta vmesnik nudi nekaj posebnih metod za delo z vrsto: za vstavljanje, odvzemanje in predogled elementa v vrsti. Vsaka od navedenih metod ima dve izvedbi, ena ob neuspeli operaciji sproži izjemo, druga pa vrne posebno vrednost (odvisno od operacije je to `null` ali `false`).

Za vstavljanje lahko uporabimo metodi `add(e)` ali `offer(e)`, za odvzemanje `remove()` ali `poll()` ter za predogled elementa `element()` ali `peek()`. Metode `add(e)`, `remove()` in `element()` ob neuspehu prožijo izjeme, ostale tri pa vračajo vnaprej določeno vrednost.

Vrste največkrat izvedemo s povezanim seznamom `LinkedList` ali prioritetno vrsto `PriorityQueue`, ki temelji na prioritetni kopici.

Poglejmo si na kratkem primeru uporabo prioritetne vrste. V vrsto bomo vstavljali (naključno generirana) števila in jih potem po vrsti odvzemali iz vrste. Prioriteto elementa naj določa kar njegova vrednost, torej lahko za prioriteto uporabimo kar privzeto naravno urejenost števil.

Vrsto ustvarimo s klicem konstruktorja:

```
Queue<Integer> vrsta = new PriorityQueue<Integer>();
```

S pomočjo generatorja naključnih števil najprej izberemo število elementov v vrsti (med 0 in `MAX`, ki je konstanta v programu), nato pa v zanki generiramo vrednosti vstavljenih elementov.

```
int meja = (int) (Math.random()*MAX);
for(int i=0; i<meja; i++) {
  int st = (int) (Math.random()*MAX);
  vrsta.add(st);
}
```

Elemente dodajamo v vrsto s pomočjo metode `add(e)`, ki doda element `e` v vrsto na ustrezno mesto (v bistvu gradimo naraščajoče urejen seznam števil).

Ko v vrsto vstavimo vse elemente, lahko s pomočjo metode `size()` izpišemo velikost naše vrste:

```
System.out.printf("V vrsti je %d elementov %n",
                  vrsta.size());
```

Za ogled elementa, ki je naslednji na vrsti, lahko uporabimo metodo `peek()`. Metoda vrne prvi element v vrsti, ne da bi ga pri tem odstranila iz vrste. V primeru prazne vrste metoda vrne `null`.

```
System.out.printf("Prvi element v vrsti je %d%n",
                  vrsta.peek());
```

Posamezne elemente lahko jemljemo iz vrste s pomočjo metode `poll()`. Ta metoda iz vrste odvzame prvi element in ga vrne; v primeru prazne vrste vrne `null`. Elemente lahko jemljemo iz vrste v zanki in sproti izpisujemo vrednosti odvzetih elementov:

```
Integer stevilo;
while((stevilo = vrsta.poll()) != null)
  System.out.printf("Na vrsti je %d ... %n", stevilo);
```

Celoten program je v datoteki `Vrsta.java`.

Slovarji

Poseben primer zbirke je tudi tako imenovan slovar (*map*), ki vsebuje pare `<ključ,vrednost>`. Ključ ne more biti podvojen, vsak ključ se preslika v največ eno vrednost. Ključ tako enolično določa element v zbirki, medtem ko je vrednost pri vsakem ključu poljubna (lahko je tudi enaka za več različnih ključev). V tem pogledu so slovarji podobni funkcijam v matematiki.

Slovar opisuje vmesnik `Map`, izvedemo pa ga lahko z zgoščeno tabelo (razred `HashMap`), z uravnoteženim drevesom (razred `TreeMap`) ali z zgoščenim seznamom (razred `LinkedHashMap`). Pri prvi izvedbi dobimo neurejen slovar, pri drugi urejenega,

pri tretji pa je slovar urejen po vrstnem redu dodajanja elementov. V našem primeru uporabe slovarja bomo uporabili urejen slovar.

Napišimo program, ki bo izpisal frekvenco pojavljanja (število pojavitev) besed, ki so podane kot argumenti programa. Besede skupaj z njihovimi frekvencami naj program izpiše urejene po abecedi.

Program mora torej pomniti elemente, ki jih sestavljata dve medsebojno povezani vrednosti: posamezna beseda in število ponovitev te besede. Besede so poljubne, a se v zbirki ne smejo ponavljati. Zato bodo besede ključi v našem slovarju. Pripadajoče frekvence besed pa naj predstavljajo vrednosti v slovarju.

Ker mora biti naš slovar urejen (abecedni izpis), bomo za slovar uporabili objekt razreda `TreeMap`, kateremu v oklepajih <> podamo tip ključa in tip vrednosti, ki ju slovar hrani. V našem primeru so ključi besede, torej tipa `String`, vrednosti pa pripadajoče frekvence (`Integer`):

```
TreeMap<String,Integer> slovar =
    new TreeMap<String,Integer>();
```

Preden se lotimo pisanja kode, si poglejmo še nekaj osnovnih metod slovarja, ki jih bomo potrebovali pri reševanju našega primera.

Za vstavljanje elementov v slovar uporabimo metodo `put(k,v)`. Ta metoda doda podani element (par `<k,v>`) k elementom slovarja. Če slovar že vsebuje ključ `k`, se vrednost pri tem ključu zamenja z vrednostjo `v`, metoda pa vrne staro vrednost pri tem ključu.

Metoda `get(k)` vrne vrednost, ki je shranjena v slovarju pod ključem `k`; če ključa `k` ni v slovarju, metoda vrne `null`.

Z metodama `containsKey(k)` in `containsValue(v)` preverimo, ali je v slovarju element s podanim ključem oziroma s podano vrednostjo. Obe metodi vrneta vrednost tipa `boolean`.

Če želimo vse argumente programa shraniti v slovar, se moramo najprej sprehoditi preko polja nizov `args`. Vsak argument (beseda) predstavlja ključ v našem slovarju. Sprehod preko argumentov naredimo v preprosti zanki:

```
for(String kljuc: args)
```

Za vsako besedo (ključ) moramo nato preveriti, ali je že vsebovana v slovarju:

```
if(slovar.containsKey(kljuc))
```

V primeru, da besede še ni v slovarju, jo dodamo v slovar, njena pripadajoča frekvenca pa je 1 (če besede še ni v slovarju, pomeni, da se je sedaj pojavila prvič):

```
slovar.put(kljuc, 1);
```

Če pa beseda v slovarju že obstaja, moramo najprej ugotoviti, kakšna je njena trenutna frekvenca, potem pa to vrednost povečati za 1 ter jo zapisati nazaj v slovar pod isti ključ (s tem spremenimo vrednost pri tem ključu):

```
int frekvenca = slovar.get(kljuc);
frekvenca++;
slovar.put(kljuc, frekvenca);
```

Zgornje tri stavke lahko krajše zapišemo z enim samim:

```
slovar.put(kljuc, slovar.get(kljuc)+1);
```

Na koncu moramo le še izpisati vse elemente urejenega slovarja, kar lahko naredimo s pomočjo prekrite metode `toString()` razreda `TreeMap`, ki poskrbi za primeren izpis elementov slovarja:

```
System.out.println(slovar);
```

Datoteka `Slovar.java` hrani izvorno kodo tega primera. Tudi tukaj moramo pri izvajanju programa podati argumente (poljubne besede), kajti brez njih bo slovar vedno prazen. Primer klica programa:

```
java Slovar prva druga prva tretja beseda druga prva
```

Iteratorji

Vmesnik `Collection` nudi tudi iteratorje, ki jih uporabljamo za prehod preko zbirke. Iterator (*iterator*) je objekt, ki omogoča prehod preko vseh elementov zbirke po vrsti.

Iterator določene zbirke dobimo s klicem metode `iterator()` te zbirke.

Vmesnik `Iterator` nudi tri metode, od katerih je ena opcijska. Metoda `hasNext()` vrne `true`, če so v iteraciji še elementi. Metoda `next()` vrne naslednji element iteracije. Metoda `remove()` pa izvede opcijsko operacijo brisanja zadnjega vrnjenega elementa iz zbirke. Med iteracijo zbirke ne moremo spreminjati, izjema je le brisanje posameznih elementov z metodo `remove()`.

Uporabo iteratorjev si je najbolje pogledati kar na enostavnem primeru. Imamo na primer seznam nizov, definiran kot:

```
List<String> seznam = new ArrayList<String>();
```

in bi želeli izpisati vse posamezne elemente seznama. Preko seznama se bomo sprehodili s pomočjo iteratorja. Najprej moramo pridobiti referenco na iterator tega seznama:

```
Iterator<String> it = seznam.iterator();
```

Nato pa v zanki, ki se ponavlja toliko časa, dokler je še kakšen element v iteraciji, pridobimo naslednji element in ga izpišemo:

```
while(it.hasNext()) {
  String element = it.next();
  System.out.println(element);
}
```

Implicitno iteratorje uporablja tudi `for-each` zanka (a so skriti), ki je za programerja bolj enostavna za uporabo. Prav zaradi enostavnosti je bolj priporočljiva uporaba `for-each` zanke za prehod preko zbirke, razen seveda v primerih, ko te zanke ne moremo uporabiti: ko želimo odstraniti trenutni element zbirke (z uporabo metode `remove`), ali pa želimo vzporedne prehode preko več zbirk hkrati.

6. Grafika

Platforma JFC (*Java Foundation Classes*) vključuje tudi dva paketa knjižnic za gradnjo grafičnih uporabniških vmesnikov (*Graphical User Interface* ali GUI), to sta AWT (*Abstract Windowing Toolkit*) in Swing. Prvi je starejši in je hkrati osnova grafičnih gradnikov. Drugi pa je novejši in nudi grafične elemente, katerih videz je neodvisen od okolja operacijskega sistema. Zato bomo pri gradnji grafičnih vmesnikov raje uporabljali Swing komponente.

Osnovni gradniki Swing so izpeljani iz gradnikov AWT, na primer razred `Frame` iz AWT je osnova razreda `JFrame` iz Swing (imena razredov Swing se začnejo s črko `J`).

Programi z GUI so dogodkovno vodeni (*event driven*), saj se odzivajo na akcije uporabnika (kot je na primer klik na gumb). Za razliko od postopkovnega programiranja (*procedural programming*) pri dogodkovno vodenih programih pišemo poleg kode za izgradnjo grafičnega vmesnika le kodo, ki pomeni odziv na različne dogodke. V Javi uporabljamo za to poslušalce (*listeners*), ki si jih bomo ogledali enem naslednjih razdelkov.

Preprosto okno

Začnimo z najpreprostejšim programom, ki na zaslonu prikaže grafično okno. Program je zelo enostaven in nima (še) nobene praktične uporabnosti.

Osnova javanskega programa z grafičnim uporabniškim vmesnikom je samostojno okno, objekt razreda `JFrame`, ki ima vlogo vsebnika, v katerega lahko postavimo druge gradnike.

Razred `JFrame` se nahaja v paketu `javax.swing`, zato najprej napovemo uporabo tega paketa (kar vseh razredov iz paketa):

```
import javax.swing.*;
```

Kot vsi javanski programi mora tudi ta program definirati nek osnovni razred, v našem primeru razred `Okno`, v katerem moramo definirati metodo `main`, saj se program začne izvajati v tej metodi.

V `main` metodi najprej ustvarimo nov objekt razreda `JFrame`:

```
JFrame okno = new JFrame();
```

nato pa ta objekt prikažemo na zaslonu (naredimo viden) s klicem metode `setVisible(true)`.

Program `Okno.java` v prvi različici izgleda takole:

```
import javax.swing.*;

public class Okno {
  public static void main(String[] args) {
    JFrame okno = new JFrame();
    okno.setVisible(true);
  }
}
```

Ko program `Okno` poženemo, se zgoraj levo na zaslonu prikaže okno, podobno tistemu na sliki 6.1. Ker oknu nismo določili velikosti, se izriše najmanjše možno okno.

Slika 6.1: Najpreprostejše okno.

Po izrisu okna je program stopil v zanko, v kateri čaka na dogodek. Oknu lahko spremenimo velikost, ga minimiziramo, maksimiziramo ali zapremo.

Gumb za zapiranje okna sicer deluje (zapre okno), a ob zapiranju okna ne konča našega programa, ker za to nismo napisali ustrezne kode. Zato moramo po zaprtju okna še prekiniti delovanje programa (v ukazni lupini pritisnemo `Ctrl+c`).

Problem rešimo enostavno, če dodamo naslednjo vrstico kode:

```
okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Metoda `setDefaultCloseOperation()` določi, kot pove že njeno ime, privzeto operacijo, ki se izvede ob zaprtju okna. Kot argument ji podamo celo število, ki določa to operacijo. Najlažje je, če uporabimo kar že definirane konstante, kot je na primer `EXIT_ON_CLOSE` iz razreda `JFrame`, ki določa, da se ob zaprtju okna program konča z uporabo metode `exit` razreda `System`.

Okno lahko tudi poimenujemo (določimo naslov okna), če konstruktorju kot argument podamo niz z imenom:

```
JFrame okno = new JFrame("Moje okno");
```

Že ustvarjenemu oknu pa lahko določimo (ali spremenimo) ime s pomočjo metode `setTitle(ime)`.

Oknu spremenimo oziroma določimo velikost s klicem metode `setSize()`, ki ji podamo širino in višino okna v pikah:

```
okno.setSize(200, 100);
```

Program `Okno1.java` v nekoliko dopolnjeni različici je naslednji:

```
import javax.swing.*;

public class Okno1 {
  public static void main(String[] args) {
    JFrame okno = new JFrame("Moje okno");
    okno.setSize(200, 100);
    okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    okno.setVisible(true);
  }
}
```

Rezultat prikazuje spodnja slika.

Slika 6.2: Nekoliko popravljeno okno.

Grafične komponente

Program z grafičnim vmesnikom lahko vključuje vrsto elementov oz. gradnikov (*components*), kot so okno (*window*), gumb (*button*), stikalo (*checkbox* in *radio button*), vnosno polje (*text field*), seznam (*choice* in *list*), oznaka (*label*), menujska vrstica (*menu bar*), orodna vrstica (*tool bar*) in drugi. Zanimiv gradnik je vsebnik (razred `Container`), ki lahko vsebuje druge gradnike.

Vsebniki so lahko osnovni ali vmesni. Osnovni vsebniki, kot sta na primer `JFrame` ali `JApplet`, določajo samostojna okna in tako predstavljajo osnovni gradnik, zato jih ne moremo vključevati v druge vsebnike. Vsak grafični program mora vsebovati vsaj en osnovni vsebnik. Vmesni vsebniki, kot je na primer `JPanel`, pa ponujajo površino, na katero polagamo gradnike, ter s tem omogočajo združevanje gradnikov v skupine tako, da jih lahko uporabljamo kot en sam gradnik.

Ostali gradniki, kot so `JButton`, `JTextField` ali `JLabel`, ne morejo vsebovati drugih komponent in morajo biti vsebovani v katerem od vsebnikov.

Vsak gradnik je lahko vsebovan v največ enem vsebniku. Poleg tega mora biti vsak gradnik del hierarhije vsebnikov, da se prikaže na zaslonu. Hierarhija vsebnikov (*containment hierarchy*) je pravzaprav drevo gradnikov z osnovnim vsebnikom kot korenom.

Vsak osnovni vsebnik ima tudi tako imenovano delovno površino (*content pane*), ki (neposredno ali posredno) vsebuje vse vidne komponente grafičnega vmesnika tega osnovnega vsebnika. Gradnike torej razvrščamo na delovno površino, ki jo ponuja vsebnik.

Hierarhijo razredov Swing si lahko ogledate v dokumentaciji JDK.

Okno s komponentami

Dodajmo v naše okno še en nov gradnik, recimo, da je to nov gumb. Gumb ustvarimo s klicem konstruktorja, s katerim hkrati določimo tudi vsebino napisa na gumbu:

```
JButton gumb = new JButton("Klikni me!");
```

Potem moramo gumb tudi dodati vsebini okna. To naredimo s pomočjo metode `add(gumb)`, kateri kot argument podamo gumb, ki ga dodajamo.

Vendar pa gradnikov (pri Swing vsebnikih) ne moremo dodati neposredno v osnovni vsebnik, temveč jih lahko dodamo le na delovno površino vsebnika. Tako moramo najprej pridobiti referenco na delovno površino vsebnika, kar naredimo s klicem metode `getContentPane()`, nato pa z metodo `add` dodamo novi gumb[1]:

```
okno.getContentPane().add(gumb);
```

Dopolnjen program je v datoteki `Okno2.java`. Dodani gumb pri tem zasede celotno površino okna, kot to prikazuje spodnja slika.

Slika 6.3: Okno z gumbom.

Z dodajanjem novih gradnikov na ta način postaja koda vedno bolj nepregledna, zato je bolje, da uvedemo nov razred, ki je izpeljan iz razreda `JFrame`, gradnike postavimo za atribute tega razreda, v konstruktorju pa poskrbimo, da se gradniki dodajo vsebini okna na ustrezno mesto.

V primeru našega zadnjega programa bi definirali nov razred `Okvir`, ki je izpeljan iz razreda `JFrame`. Atributi razreda so poleg gumba tudi tri spremenljivke, ki določajo velikost okna (širino in višino) in njegov naslov. V konstruktorju poskrbimo, da se nastavi naslov okna na ustrezen niz, določi njegova velikost in privzeta operacija ob zaprtju okna ter da se doda gumb k obstoječi vsebini okna:

```
public class Okvir extends JFrame {
  private final int SIRINA = 200;
  private final int VISINA = 150;
  private String naslov = "Moje okno";
  private JButton gumb = new JButton("Klikni me!");

  public Okvir() {
    setTitle(naslov);
    setSize(SIRINA, VISINA);
```

[1] Pravzaprav je v Java 6 zaradi enostavnosti metoda `add` prekrita, tako da po potrebi sama poskrbi za dodajanje komponente na delovno površino. To pomeni, da bi lahko v našem primeru uporabili enostavnejši stavek `okno.add(gumb);` in gradnik `gumb` bi se dodal na delovno površino vsebnika `okno`.

```
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        getContentPane().add(gumb);
    }
}
```

Pri klicu metod (kot je na primer `setTitle()`) nismo rabili reference na objekt `JFrame`, saj metode, gre za podedovane metode tega razreda, kličemo znotraj konstruktorja. Enakovredno bi lahko uporabili tudi določilo `this` (na primer `this.setTitle()`).

V metodi `main` nato ustvarimo objekt razreda `Okvir` in ga prikažemo na zaslonu (razred `Okvir` je izpeljan iz razreda `JFrame`, zato tudi deduje metodo `setVisible`):

```
Okvir okno = new Okvir();
okno.setVisible(true);
```

Ta različica programa je v datoteki `Okno3.java`. Program se pri izvajanju v ničemer ne razlikuje od našega prejšnjega programa `Okno2.java`. Preizkusite ga sami!

Pa dodajmo v obstoječe okno še nekaj gradnikov. Kot primer preprostega grafičnega vmesnika v Javi bi lahko napisali program, kjer osnovno okno zajema dve oznaki, vnosno polje za vpis besedila ter dva gumba. Prvi gumb zbriše vsebino vnosnega polja, drugi gumb pa v drugo oznako izpiše vsebino vnosnega polja.

Program zgradimo postopoma. Najprej dodamo vse potrebne gradnike (oznaki, gumba in vnosno polje), ki jih zapišemo kot atribute razreda:

```
private JLabel oznaka1 = new JLabel("Vpiši besedilo: ");
private JLabel oznaka2 = new JLabel(" ..... ");
private JTextField tekst =  new JTextField(5);
private JButton gumb1 = new JButton("Zbriši");
private JButton gumb2 = new JButton("Potrdi");
```

Pri kreiranju oznak smo konstruktorju kot argument podali besedilo oznake, konstruktor vnosnega polja pa kot argument prejme privzeto velikost vnosnega polja.

Zaradi enostavnosti dodajmo še en vmesni vsebnik, ki bo združeval vse navedene gradnike v skupino. Zato ustvarimo nov objekt razreda `JPanel`, ki bo igral vlogo tega vmesnega vsebnika:

```
private JPanel ozadje = new JPanel();
```

V konstruktorju razreda `Okvir` najprej poskrbimo za to, da vse oznake, gumbe in vnosno polje združimo (dodamo) v vmesnem vsebniku `ozadje`:

```
ozadje.add(oznaka1);
ozadje.add(tekst);
ozadje.add(oznaka2);
ozadje.add(gumb1);
ozadje.add(gumb2);
```

Pri tem je vrstni red dodajanja elementov pomemben, saj se elementi dodajajo po vrsti od leve proti desni, vrstico za vrstico. Zakaj je tako, bomo videli v naslednjem razdelku.

Na koncu (še vedno znotraj konstruktorja razreda `Okvir`) pa dodamo vmesni vsebnik na delovno površino okna:

```
getContentPane().add(ozadje);
```

Celoten program je v datoteki `Okno4.java`, rezultat pa prikazuje slika 6.4.

Slika 6.4: Okno z oznakama, vnosnim poljem in dvema gumboma.

Delovna površina vsebnika, referenco nanjo dobimo s klicem metode `getContentPane()`, je objekt razreda `Container`. Torej bi lahko delovno površino okna (objekt `vsebina`) opisali takole:

```
Container vsebina = getContentPane();
```

Stavek `getContentPane().add(ozadje);` pa bi lahko zapisali z zaporedjem stavkov:

```
Container vsebina = getContentPane();
vsebina.add(ozadje);
```

Ker je razred `Container` opisan v paketu `java.awt`, moramo v tem primeru napovedati tudi uporabo paketa `java.awt` s stavkom:

```
import java.awt.*;
```

Včasih želimo referenco na delovno površino okna shraniti kot atribut razreda. Tako spremenjen program je v datoteki `Okno5.java`.

Razporejevalniki

V prejšnjem primeru so se gradniki na delovni površini prikazali po vrsti od leve proti desni, kot smo jih dodajali na površino. Taka razporeditev seveda ni vedno primerna, zato lahko uporabimo ustrezen razporejevalnik (*layout manager*), ki omogoča za naš primer najboljšo in najenostavnejšo razporeditev gradnikov.

Z uporabo razporejevalnikov dosežemo, da gradniki ohranjajo svojo obliko ne glede na velikost posameznih gradnikov. Tako se ohranjajo medsebojna razmerja med gradniki, tudi če spremenimo velikost vsebnika. Videz grafičnega vmesnika torej v celoti določa izbrani način razvrščanja in vrstni red dodajanja gradnikov.

AWT ima pet razredov, ki jih lahko uporabimo za razporejevalnike, poleg tega pa še vmesnik `LayoutManager`, ki omogoča izdelavo lastnega razporejevalnika. Najpogosteje se uporabljajo razporejevalniki `BorderLayout`, `FlowLayout` ter `GridLayout`, katere si bomo pogledali na primerih v nadaljevanju. Razporejevalnik `CardLayout` je uporaben v posebnih primerih (za izdelavo večličnih obrazcev), saj vsebnik razdeli na več strani, od katerih je naenkrat vidna le ena. Razporejevalnik `GridBagLayout` pa je sicer zelo zmogljiv, a nekoliko bolj zapleten za uporabo, saj vsebnik razdeli na mrežo različno velikih celic.

Seveda lahko gradnike postavljamo tudi brez razporejevalnika (*null layout*), če podamo njihov položaj glede na koordinate zaslona. Na način ni preveč priporočljiv, saj nastane velik problem pri spreminjanju velikosti gradnikov (*resize*).

Ker so razporejevalniki opisani v paketu `java.awt`, ob uporabi razporejevalnikov ne smemo pozabiti napovedati uporabe tega paketa.

V nadaljevanju si najprej poglejmo tri primere razvrščanja gumbov v oknu, nato pa se bomo vrnili k našemu primeru (`Okno5.java`) in s pomočjo primernega razporejevalnika bolje razvrstili gradnike v oknu.

Robno razvrščanje (`BorderLayout`)

`BorderLayout` je privzet razporejevalnik za okna. Gradnike lahko dodajamo na štiri robove vsebnika, preostalo površino pa zapolni peti gradnik. Ustvarimo ga s klicem konstruktorja `new BorderLayout()`, vsebniku pa določimo razporejevalnik s klicem metode `setLayout()`:

```
BorderLayout raz = new BorderLayout();
vsebnik.setLayout(raz);
```

Ker navadno niti ne potrebujemo reference na sam razporejevalnik, lahko zgornja dva stavka enostavno združimo:

```
vsebnik.setLayout(new BorderLayout());
```

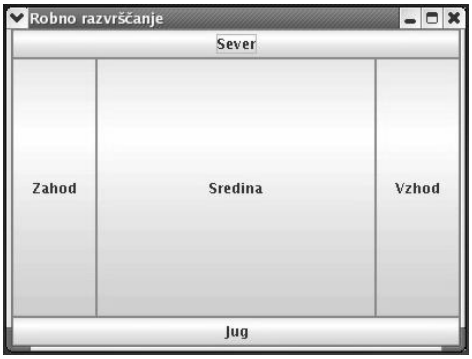
Posamezne gradnike dodajamo z metodo `add(gradnik, poz)`, kjer je `poz` položaj gradnika v vsebniku: zgoraj, spodaj, levo, desno ali v sredini. Za položaj lahko uporabimo tudi konstante razreda `BorderLayout`, ki so poimenovane po straneh neba (`NOTRH`, `SOUTH`, `WEST`, `EAST` in `CENTER`, po vrsti).

Tako lahko na sredino okna dodamo en velik gumb z naslednjim stavkom:

```
vsebnik.add(new JButton("Sredina"), BorderLayout.CENTER);
```

Velikost gradnikov se spremeni tako, da le-ti zavzamejo ves razpoložljiv prostor.

Primer robnega razvrščanja je opisan v programu `Robno.java`, katerega rezultat prikazuje slika 6.5.



Slika 6.5: Robno razvrščanje v oknu.

Tekoče razvrščanje (`FlowLayout`)

`FlowLayout` je privzet razporejevalnik za vsebnike razreda `Panel`. Gradnike dodajamo po vrsti od leve proti desni, vrstico za vrstico. Pri tem gradniki ohranijo svojo velikost (v našem primeru se velikost gumbov prilagodi velikosti napisa na gumbu). Razporejevalnik `FlowLayout` ustvarimo s klicem ustreznega konstruktorja ter ga določimo za razporejevalnik vsebnika:

```
FlowLayout raz = new FlowLayout();
vsebnik.setLayout(raz);
```

Oba stavka lahko tudi združimo:

```
vsebnik.setLayout(new FlowLayout());
```

Program `Tekoce.java` izriše okno z gumbi, kot prikazuje spodnja slika.

Slika 6.6: Tekoče razvrščanje v oknu.

Razvrščanje v mrežo (`GridLayout`)

`GridLayout` razdeli vsebnik v mrežo enakih celic, v katere po vrsti razvrsti gradnike. Tako so vsi gradniki enako veliki, ne glede na njihovo vsebino. Ustvarimo ga s klicem konstruktorja, kateremu podamo velikost mreže (število vrstic in število stolpcev):

```
vsebnik.setLayout(new GridLayout(2,3));
```

Primer razvrščanja v mrežo prikazuje slika 6.7, koda tega programa pa je v datoteki `Mreza.java`.

Slika 6.7: Razvrščanje v mrežo enako velikih celic.

Preizkusite opisane primere razvrščanja ter opazujte njihovo obnašanje pri spreminjanju velikosti okna (z vlečenjem miške okno povečamo ali pomanjšamo). Vsi gradniki okna se primerno razporedijo po razpoložljivem prostoru, skladno s pravili, ki jih določa razporejevalnik.

Naš program

Če se ponovno vrnemo k našemu programu `Okno5.java`, ugotovimo, da smo v njem uporabljali razporejevalnik, čeprav ga nismo eksplicitno ustvarili. Osnovne gradnike (oznaki, vnosno polje in gumba) smo namreč združili v vsebniku razreda `JPanel`, za katerega velja privzet razporejevalnik `FlowLayout`. Zato se je vseh pet osnovnih gradnikov razporedilo po vrsti od leve proti desni, tako kot smo jih dodajali v vsebnik. Sam vsebnik smo nato postavili na delovno površino okna, za katerega je privzet razporejevalnik `BorderLayout`. Ker pri dodajanju nismo navedli, na katero stran želimo postaviti vsebnik, se je upoštevala privzeta vrednost, to je dodajanje v sredino. Sedaj tudi lažje razumemo, zakaj smo dobili takšen izgled in obnašanje okna (kot ga prikazuje slika 6.4).

Še večjo prilagodljivost pa lahko dosežemo, če posamezne gradnike najprej združimo v vmesne vsebnike ter na delovno površino postavimo te vsebnike. Najbolj smiselno je združevati gradnike, ki na nek način (na primer pomensko) spadajo skupaj. V našem primeru bi lahko združili prvo oznako in vnosno polje v enem vsebniku ter oba gumba v

drugem vsebniku. Tretji vsebnik bi tako vseboval le drugo oznako. Zato moramo ustvariti tri vsebnike, ki so lahko definirani kot atributi razreda:

```
private JPanel vnos = new JPanel();
private JPanel izpis = new JPanel();
private JPanel gumbi = new JPanel();
```

Vsebnik vnos združuje oznako in vnosno polje, oboje postavljeno po vrsti eno za drugim:

```
vnos.setLayout(new FlowLayout());
vnos.add(oznaka1);
vnos.add(tekst);
```

Prva vrstica (določitev razporejevalnika `FlowLayout`) ni potrebna, saj je `FlowLayout` tako ali tako privzet razporejevalnik za vsebnike razreda `JPanel`.

Drugemu vsebniku, `izpis`, dodamo le drugo oznako (tudi tu uporabimo kar privzet razporejevalnik `FlowLayout`):

```
izpis.add(oznaka2);
```

Čeprav v vsebnik `izpis` postavimo le eno oznako, smo nov vsebnik uporabili zato, da oznaka `oznaka2` ohrani svojo privzeto velikost in da je postavljena na sredino (sredinska poravnava je privzeta poravnava pri tekočem razvrščanju).

Oba gumba združimo v vsebniku `gumbi`, za katerega tudi uporabimo privzet razporejevalnik `FlowLayout`:

```
gumbi.add(gumb1);
gumbi.add(gumb2);
```

Vse tri vsebnike potem postavimo na ustrezna mesta na delovno površino. Tu smo izbrali razporejevalnik `GridLayout`, ki ima tri vrstice, en stolpec ter 20 pik vodoravnega razmaka med stolpci in 5 pik navpičnega razmaka med vrsticami:

```
vsebina.setLayout(new GridLayout(3,1,20,5));
vsebina.add(vnos);
vsebina.add(izpis);
vsebina.add(gumbi);
```

Pri tem smo tudi malo povečali okno, ki je sedaj višine 150 pik, da imamo dovolj prostora za vse gradnike. Celoten program je v datoteki `Okno6.java`, njegov rezultat pa je prikazan na sliki 6.8.

Slika 6.8: Okno z uporabo razporejevalnikov.

Naš dosedanji program sicer odpre okno in v njem izriše vse gradnike, med njimi tudi dva gumba, a nima nobene funkcionalnosti. Če kliknemo na gumb, se ne zgodi nič. V naslednjem razdelku bomo pogledali, kako programu dodamo tudi funkcionalnost.

Dogodki in poslušalci

Programi z grafičnim vmesnikom so dogodkovno vodeni, kar pomeni, da se odzivajo na akcije uporabnika. Preko mehanizma dogodkov (*events*) lahko izvajamo nadzor množice dejavnih elementov, ki se lahko kadarkoli odzovejo na dejanje uporabnika.

Izvajanje programa z grafičnim vmesnikom je odvisno od povzročenih dogodkov. Dogodek je sporočilo v obliki objekta `AWTEvent`, ki ga javanski sistem posreduje metodi `processEvent()` gradnika, kadar se v izvajanju programa zgodi kaj takega, kar bi lahko zanimalo ta gradnik (kot je na primer sprememba stanja, interakcija z uporabnikom in podobno).

Seveda nas ponavadi izmed množice sproženih dogodkov zanima le majhen del. Zato ima AWT vzpostavljen dogodkovni model, s katerim gradnik o dogodku obvesti poslušalce (*listeners*). Tako lahko vsakemu gradniku prijavimo ali odjavimo poslušalca (metodi `addXListener()` in `removeXListener()`), ta pa mora nositi izvedbo predpisanega vmesnika `XListener` (`X` označuje poljubnega poslušalca).

Paket `java.awt.event` ponuja več različnih vmesnikov, vsak je namenjen določenemu tipu dogodkov. Med drugimi naj omenimo vmesnik `ActionListener`, ki opazuje akcijske dogodke, ter vmesnik `WindowListener` za dogodke v zvezi z okni. Zanimivi so tudi `MouseListener` in `MouseMotionListener`, ki opazujeta mišje dogodke, ter `KeyListener` za dogodke v zvezi s tipkovnico. Ostale vmesnike in opis metod posameznih vmesnikov si lahko ogledate v JDK dokumentaciji.

V splošnem se rokovanja z dogodki lotimo tako, da za posamezno vrsto dogodkov napišemo ustreznega poslušalca (izpolnimo zahtevan vmesnik) in ga prijavimo gradniku. V primeru dogodka izvajalno okolje pokliče ustrezno metodo na vmesniku poslušalca, kar izkoristimo za pisanje odzivov. Pri tem so o dogodku obveščeni vsi prijavljeni poslušalci, istega poslušalca pa lahko prijavimo na različne vire dogodkov.

Poslušalec kot notranji razred

Pa si oglejmo obravnavo dogodkov na našem primeru. Programu `Okno6.java` dodajmo tudi odziv na dogodke, ki se zgodijo ob aktivaciji (na primer ob kliku) katerega od

gumbov. Zato moramo najprej definirati ustreznega poslušalca, ki se odziva na akcije gumbov. Tako definiramo nov razred `GumbPoslusalec`, ki implementira vmesnik `ActionListener`, ter v njem metodo `actionPerformed()`, ki je tudi edina metoda, ki jo zahteva ta vmesnik:

```
private class GumbPoslusalec implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    ...
  }
}
```

Razred definiramo kot notranji privatni razred, saj ni potrebe, da bi ga uporabljali še kjerkoli drugje. Odločitev za notranji razred nam bo tudi poenostavila napisano kodo, kar bomo spoznali v nadaljevanju na koncu tega razdelka.

Pri tem ne smemo pozabiti na napoved uporabe paketa `java.awt.event` na začetku programske kode, saj so v njem definirani z dogodki povezani vmesniki in razredi.

Potem obema gumboma dodamo nov izvod poslušalca (poslušalca prijavimo gumbu) na naslednji način:

```
gumb1.addActionListener(new GumbPoslusalec());
gumb2.addActionListener(new GumbPoslusalec());
```

Oba stavka postavimo v konstruktor razreda `Okvir`.

Sedaj moramo le še definirati metodo `actionPerformed()`, ki se pokliče ob dogodku. V njej torej določimo odziv našega programa na ta dogodek. Ker smo istega poslušalca prijavili obema gumboma, moramo najprej ugotoviti, kateri gumb je bil izvor dogodka. Sam dogodek `e` dobimo kot argument metode, s pomočjo metode `getSource()` pa dobimo referenco na objekt, na katerem se je dogodek dejansko zgodil. Potem lahko enostavno preverimo, kateri od obeh gumbov se ujema z izvorom dogodka, in ustrezno ukrepamo:

```
public void actionPerformed(ActionEvent e) {
  Object izvor = e.getSource();
  if(izvor == gumb1)
    tekst.setText("");
  if(izvor == gumb2)
    oznaka2.setText(tekst.getText());
}
```

Če je izvor dogodka `gumb1`, potem s klicem metode `setText("")` nastavimo vsebino vnosnega polja na prazen niz (kar pomeni, da jo zbrišemo). Če pa je izvor dogodka `gumb2`, vsebino vnosnega polja, ki jo dobimo s klicem metode `getText()`, zapišemo v oznako `oznaka2` (slednje dosežemo z uporabo metode `setText()`).

Celoten program je v datoteki `Okno7.java`. Preverite njegovo delovanje.

Poslušalec ni njuno nov razred

Obstaja pa tudi krajši in zato nekoliko enostavnejši način obravnave dogodkov. Tako ni potrebno definirati novega razreda za poslušalca dogodkov, ampak lahko vmesnik `ActionListener` implementira kar naš razred `Okvir`:

```
public class Okvir extends JFrame implements ActionListener
```

Ker je razred `Okvir` tudi poslušalec dogodkov, moramo v njem definirati metodo `actionPerformed()`, ki jo zahteva ta vmesnik. Metoda je enaka kot v prejšnjem primeru.

Tako definiranega poslušalca dodamo gumbu s stavkom:

```
gumb1.addActionListener(this);
```

kjer se določilo `this` nanaša na ta isti razred, torej razred `Okvir`, ki je hkrati tudi poslušalec.

Koda tega programa je v datoteki `Okno8.java`. V delovanju obeh programov ni razlik.

Poslušalec kot anonimni razred

Kadar želimo napisati preprost odziv na en sam gradnik, lahko to naredimo tudi v obliki anonimnega notranjega razreda, ki ga definiramo kar v argumentu metode za prijavo poslušalca. Če bi v našem primeru želeli narediti odziv le na gumb `gumb2`, bi to lahko zapisali takole:

```
gumb2.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
         oznaka2.setText(tekst.getText());
       }
});
```

Kadar pa želimo dogodke obravnavati bolj splošno in metode uporabiti za različne gradnike, vedno napišemo samostojni razred. Ta je sicer lahko tudi notranji razred.

Uporabimo `XAdapter` namesto `XListener`

Poglejmo si še en primer. Program `Okno7.java` dopolnimo še s poslušalcem za dogodke, povezane z okni, ter ga uporabimo za odziv na dogodek zapiranja okna.

Naj se nov razred, ki sprejema okenske dogodke, imenuje `OknoPoslusalec`. Tako namesto metode:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

dodamo oknu nov izvod poslušalca, ki smo ga napisali, da bdi nad okenskimi dogodki:

```
addWindowListener(new OknoPoslusalec());
```

Vmesnik `WindowListener`, ki naj bi ga izdelal naš razred `OknoPoslusalec`, vsebuje naslednje metode: `windowActivated()`, `windowClosed()`, `windowClosing()`, `windowDeactivated()`, `windowDeiconified()`, `windowIconified()` in `windowOpened()`. Ta podatek smo dobili z vpogledom v dokumentacijo JDK.

Od vseh omenjenih metod vmesnika pa nas zanima le metoda `windowClosing()`, ki se pokliče ob zapiranju okna.

Če bi razred `OknoPoslusalec` implementiral vmesnik `WindowListener`, bi moral implementirati tudi vseh sedem metod tega vmesnika (vsak razred, ki implementira vmesnik, mora implementirati tudi vse metode tega vmesnika). Čeprav nas zanima ena sama metoda, to je metoda `windowClosing()`, bi morali definirati še preostalih šest metod, ki bi pač ostale prazne.

Na srečo pa obstaja tudi druga, krajša pot. Namesto da razred implementira vmesnik `WindowListener`, naj razred raje razširja abstraktni razred `WindowAdapter`. To omogoča, da nam ni potrebno definirati vseh metod vmesnika (ker so le-te že definirane v abstraktnem razredu kot prazne metode), temveč lahko prekrijemo le tisto metodo, ki nas zanima (torej samo metodo `windowClosing()`):

```
public class OknoPoslusalec extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

S pomožnimi razredi `XAdapter`, ki nosijo prazne izvedbe vseh metod vmesnika poslušalca `XListener`, si pomagamo v primeru, ko nas zanima le nekaj metod iz vmesnika poslušalca. Poslušalca napišemo tako, da razširimo ustrezen adapter in s svojo izvedbo prekrijemo le tiste metode, ki nas zanimajo.

Opisana koda je v datoteki `Okno9.java`. Preverite, da se njeno izvajanje v ničemer ne razlikuje od programa `Okno7.java`.

Seveda bi bilo v tem zadnjem primeru enostavneje uporabiti kar anonimen notranji razred, saj gre za enostaven odziv okna na dogodek. Razred lahko definiramo kar v argumentu metode za prijavo poslušalca:

```
addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent e) {
         System.exit(0);
       }
});
```

Prenos reference na objekt

Verjetno ste opazili, da smo v zadnjem primeru (`Okno9.java`) uporabili kot poslušalce objekte dveh različnih razredov: `GumbPoslusalec` in `OknoPoslusalec`. Medtem ko je slednji navaden javni razred, pa je razred `GumbPoslusalec` definiran kot notranji razred razreda `Okvir`. Zakaj? No, za tako rešitev smo imeli zelo tehten razlog. Metoda `actionPerformed` tega razreda se namreč sklicuje na različne objekte (`gumb1`, `gumb2`, `oznaka2`, `tekst`), ki so deklarirani kot privatne objektne spremenljivke razreda `Okvir`. Da so te spremenljivke vidne (v dosegu) tudi v metodi `actionPerformed`, mora biti ta metoda definirana znotraj razreda `Okvir` (kot metoda razreda `Okvir` ali kot metoda razreda `GumbPoslusalec`, ki je notranji razred razreda `Okvir`). Obe rešitvi smo prikazali v prejšnjih primerih (`Okno8.java` in `Okno7.java`, po vrsti).

Ali lahko sestavimo tudi rešitev, kjer je metoda `actionPerformed` definirana v razredu `GumbPoslusalec`, ki pa ni definiran kot notranji razred? Seveda lahko.

Razred `GumbPoslusalec` lahko najavimo tudi kot javni razred, njegova edina metoda pa ostane nespremenjena:

```
public class GumbPoslusalec implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    Object izvor = e.getSource();
    if(izvor == gumb1)
      tekst.setText("");
    if(izvor == gumb2)
      oznaka2.setText(tekst.getText());
  }
}
```

Vendar pa tu nastane manjši problem, ki ga moramo rešiti. Ker se v metodi `actionPerformed` sklicujemo na spremenljivke `gumb1`, `gumb2`, `tekst` in `oznaka2`, morajo biti te spremenljivke tudi deklarirane v tem razredu, saj jih drugače ne moremo uporabljati. Deklariramo jih lahko kot privatne atribute razreda:

```
private JButton gumb1, gumb2;
private JTextField tekst;
private JLabel oznaka2;
```

Seveda pa želimo, da se te spremenljivke sklicujejo na tiste (v naši kodi istoimenske) objekte, ki smo jih že ustvarili znotraj razreda `Okvir` (kjer so deklarirane kot privatni atributi razreda). Zato ob kreiranju novega objekta `GumbPoslusalec` (ki ga ustvarimo v razredu `Okvir`), prenesemo tudi reference na te štiri spremenljivke v objekt razreda `GumbPoslusalec`. Za to najenostavneje poskrbimo kar v konstruktorju razreda, saj se ta zagotovo pokliče ob vsakem kreiranju novega izvoda tega razreda. Konstruktorju ob klicu kot argumente podamo reference na vse štiri objekte, ki se na ta način prenesejo v novo ustvarjen objekt:

```
public GumbPoslusalec(JButton g1, JButton g2,
                      JTextField t, JLabel o) {
  this.gumb1 = g1;
  this.gumb2 = g2;
```

```
    this.tekst = t;
    this.oznaka2 = o;
  }
```

Seveda moramo sedaj pri klicu samega konstruktorja podati tudi vse štiri argumente:

```
new GumbPoslusalec(gumb1, gumb2, tekst, oznaka2);
```

Celoten program je v datoteki `Okno10.java`. Tudi delovanje tega programa se v ničemer ne razlikuje od programa `Okno7.java`. Preverite sami!

Lastnosti sistema

Za predstavitev dejanske implementacije AWT uporabljamo razred `Toolkit`. Preko objekta tega razreda lahko tako pridobimo informacije o dejanskem sistemu, ki je v uporabi. Metoda `getDefaultToolkit()` vrne referenco na objekt razreda `Toolkit`. Metoda `getScreenSize()` vrne velikost zaslona v pikah v obliki objekta razreda `Dimension`. Tako bi na primer dejansko velikost zaslona ugotovili z naslednjimi stavki:

```
Toolkit tk = Toolkit.getDefaultToolkit();
Dimension zaslon = tk.getScreenSize();
System.out.printf("Velikost zaslona je %d x %d\n",
                  zaslon.width, zaslon.height);
```

Potem lahko velikost okna v programu prilagodimo dejanski velikosti zaslona:

```
JFrame okno = new JFrame("Četrtinsko okno");
okno.setSize(zaslon.width/2, zaslon.height/2);
```

Risanje

Za risanje in slikanje lahko uporabimo tudi javanske metode. Programi lahko rišejo neposredno na osnovni vsebnik (torej na objekte razreda `JApplet` ali `JFrame` ter njihove izpeljanke) ali pa na vmesni vsebnik `JPanel`. Ponavadi pri risanju uporabljamo objekt razreda `JPanel` kot risalno površino.

Vsak javanski gradnik ima svoj grafični kontekst (*graphic context*), ki ga predstavlja objekt razreda `java.awt.Graphics`. Uporabljamo ga za risanje na ta gradnik, saj vsebuje različne metode za risanje. Objekta `Graphics` ne moremo ustvariti neposredno, temveč ga dobimo od gradnika (kot je na primer `JPanel`) s pomočjo metode `getGraphics()`, ki vrne referenco na grafični kontekst tega gradnika.

Grafični gradniki (tako v AWT kot v Swing) se izrišejo avtomatično, kadar se za to izkaže potreba (kadar se spremeni vsebina gradnika, njegova lega, velikost ali vidnost). Za njihov izris poskrbi metoda `paint()` razreda `java.awt.Component`, ki jo samodejno pokliče izvajalni sistem. Ob tem se metodi avtomatično posreduje referenca na objekt razreda `Graphics` za risanje na ta gradnik.

Ponoven izris gradnika (oziroma obnovitev vsebine gradnika) pa lahko zahtevamo tudi programsko s klicem metode `repaint()`, ki potem pokliče metodo `paint()`. V programu ponavadi ne kličemo metode `paint()` neposredno, temveč vedno preko metode `repaint()`.

Pri Swing gradnikih metoda `paint()` pokliče tri ločene metode v naslednjem vrstnem redu: `paintComponent()` za izris gradnika, `paintBorder()` za izris obrobe in `paintChildren()` za izris njegovih otrok (podgradnikov). Kadar se torej ponovno izriše vsebnik, se ponovno izrišejo tudi vsi gradniki, ki jih vsebuje.

Kodo za risanje postavimo v programu znotraj prekrite metode za izris gradnika. Za izris uporabljajo metodo `paint()` vsi gradniki, ki neposredno izhajajo iz razreda `java.awt.Component` (torej vsi AWT gradniki ter `JApplet` in `JFrame`). Razred `javax.swing.JComponent` in njegovi nasledniki pa uporabljajo za svoj izris metodo `paintComponent()`. Zato pri AWT gradnikih prekrijemo metodo `paint()`, pri Swing gradnikih pa metodo `paintComponent()`.

Razredi, izpeljani iz `java.awt.Container`, ki prekrijejo metodo `paint()`, morajo v njej na začetku poklicati `super.paint()`, da zagotovijo pravilen izris otrok. Podobno velja za razrede, izpeljane iz razreda `JComponent` (vključno z razredom `JPanel`); ti morajo tipično znotraj svoje prekrite metode `paintComponent()` najprej poklicati metodo `super.paintComponent()`.

Za komponente v Swingu je privzeto dvojno pomnjenje (*double buffering*), kar omogoča bolj gladko izrisovanje. Izris se najprej naredi v predpomnilnik (*offscreen buffer*) in šele ko je končan, se v celoti prikaže na zaslonu.

Risanje z grafičnimi primitivi

Ključni element pri programsko podprtem izrisovanju grafičnih primitivov (kot so črte, liki, besedilo) in pri upodabljanju bitnih slik na grafični površini je objekt razreda `Graphics`, ki predstavlja grafični kontekst gradnika. Ustrezen grafični kontekst je avtomatično podan kot argument metode `paint()` oziroma metode `paintComponent()`.

Za risalno površino programskega risanja ponavadi izberemo gradnik `JPanel`. Tako napišemo svoj razred, ki razširja razred `JPanel`, in prekrijemo metodo `paintComponent()`, v kateri pokličemo ustrezne metode za izris želenih grafičnih primitivov:

```
public class Risalnica extends JPanel {
  public void paintComponent(Graphics g) {
    // tukaj kličemo metode za izris
  }
}
```

Grafični kontekst uporablja enostaven koordinatni sistem, kjer je vsaka pika predstavljena z dvema koordinatama: *x* in *y*. Izhodišče koordinatnega sistema je v levem zgornjem kotu gradnika. Koordinate *x* naraščajo od leve proti desni, koordinate *y* pa od

zgoraj navzdol. Koordinatni sistem lahko tudi prestavimo v katerokoli drugo točko z metodo `translate(dx, dy)`, kjer vrednosti *dx* in *dy* označujeta spremembo koordinat izhodišča.

Razred `Graphics` nudi več metod za risanje grafičnih primitivov. Tako lahko črto narišemo z uporabo metode `drawLine()`, kateri kot argumente podamo koordinati (*x* in *y*) začetne točke in koordinati končne točke.

Metoda `drawRect()` izriše pravokotnik, argumenti metode pa so *x* in *y* koordinati zgornjega levega oglišča, dolžina pravokotnika ter višina pravokotnika.

Elipso izrišemo z metodo `drawOval()`, kateri kot argumente podamo *x* in *y* koordinati zgornjega levega oglišča, dolžina pravokotnika ter višina pravokotnika, elipsa pa se izriše znotraj tako določenega pravokotnika. Za izris kroga določimo isto velikost dolžine in širine (izris elipse v kvadratu).

Vse metode `drawX()` imajo, kjer je to smiselno, tudi sorodno metodo `fillX()`, ki izriše polnjen lik. Tako na primer `fillRect()` izriše polnjen pravokotnik, `fillOval()` pa polnjeno elipso.

Ostale metode za risanje si oglejte v JDK dokumentaciji.

Za primer si poglejmo še program, ki izriše graf funkcije sinus (sin *x*). Program naj se imenuje `Sinus.java` in ga najdete med primeri tega poglavja. Velikost grafa (velikost enote) se prilagaja trenutni velikosti okna, tako da graf vedno zavzame celo površino okna.

Program mora prikazati enostavno okno, v katerega postavimo risalno površino, na njej pa se izriše graf. Za risalno površino lahko uporabimo kar razred `Sinus`, če je le-ta izpeljan iz razreda `JPanel`. V `main()` metodi razreda `Sinus` najprej ustvarimo novo okno, ki je objekt razreda `JFrame`:

```
JFrame okno = new JFrame("Graf sin(x)");
```

Pri tem smo uporabili konstruktor, ki nastavi tudi naslov okna. Potem ustvarimo risalno površino (objekt razreda `Sinus`) in jo postavimo na delovno površino okna:

```
Sinus graf = new Sinus();
okno.getContentPane().add(graf);
```

Velikost okna prilagodimo velikosti risalne površine (metoda `getSize()` vrne velikost gradnika v obliki objekta `Dimension`, katerega atribut `width` določa širino, `height` pa višino gradnika):

```
okno.setSize(graf.getSize().width, graf.getSize().height);
```

Na koncu oknu določimo privzeto operacijo ob zapiranju okna in ga prikažemo:

```
okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
okno.setVisible(true);
```

Razred `Sinus` ima tudi nekaj privatnih atributov, ki določajo privzeto širino in višino gradnika (velikost gradnika se nastavi ob klicu konstruktorja), ter rob, ki določa razpoložljivo risalno površino. Sedaj moramo le še prekriti metodo `paintComponent()`, v kateri bomo poskrbeli, da se graf dejansko izriše. Zaenkrat smo sestavili naslednjo kodo:

```
public class Sinus extends JPanel {
  private final int SIRINA = 800;
  private final int VISINA = 400;
  private final int ROB = 10;

  public static void main(String[] args) {
    JFrame okno = new JFrame("Graf sin(x)");
    Sinus graf = new Sinus();
    okno.getContentPane().add(graf);
    okno.setSize(graf.getSize().width,
                 graf.getSize().height);
    okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    okno.setVisible(true);
  }

  public Sinus() {
    setSize(SIRINA, VISINA);
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    // poskrbimo za izris grafa
  }
}
```

V metodi `paintComponent()` najprej pokličemo istoimensko metodo razreda `JPanel`, torej `super.paintComponent()`, ki med drugim poskrbi za pravilen izris ozadja (tudi izbris obstoječe vsebine) in naredi gradnik neprozoren. Nato na prazno risalno površino izrišemo koordinatni osi (klic metode `narisiOsi(g)`) in pokličemo še metodo za izris vrednosti funkcije `narisiSinus(g)`. Pred klicem zadnjih dveh metod smo prestavili izhodišče koordinatnega sistema na sredino risalne površine, kar nam poenostavi risanje grafa. V celoti metoda `paintComponent()` izgleda takole:

```
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  Dimension velikost = this.getSize();
  g.setClip(ROB, ROB, velikost.width-2*ROB,
            velikost.height-2*ROB);
  g.translate(velikost.width/2, velikost.height/2);
  narisiOsi(g);
  narisiSinus(g);
}
```

Z metodo `setClip()` smo določili območje gradnika, kjer je risanje dovoljeno (*clip region*). Grafa namreč ne rišemo preko celotne risalne površine, ampak okoli risalne plošče pustimo primeren rob, katerega velikost določa konstanta `ROB`.

In kako izrišemo koordinatni osi? Ne pozabimo, da smo izhodišče koordinatnega sistema že prestavili na sredino. Ker je velikost enote odvisna od velikosti risalne površine, najprej ugotovimo velikost tega gradnika (metoda `getSize()`). V odvisnosti od širine risalne površine določimo tudi velikost izrisanih črtic, ki označujejo enote (1 in -1 na *y* osi ter mnogokratnik PI/4 na *x* osi), in velikost koraka, to je razmika med izrisanimi črticami. Osi *x* in *y* izrišemo preko celotnega območja risanja tako, da se sekata v središču koordinatnega sistema. Izris koordinatnih osi ni zapleten, potrebno je le temeljito premisliti, si skicirati na papir in morda tudi nekajkrat poizkusiti in po potrebi ustrezno popraviti kodo.

```
private void narisiOsi(Graphics g) {
  Dimension velikost = this.getSize();
  int hMeja = velikost.width/2;
  int vMeja = velikost.height/2;
  int crtica = velikost.width/200;
  g.setColor(Color.black);
  g.drawLine(-hMeja, 0, hMeja, 0);
  int korak = hMeja/6;
  for(int i = korak; i <= hMeja; i += korak) {
    g.drawLine(i, crtica, i, -crtica);
    g.drawLine(-i, crtica, -i, -crtica);
  }
  g.drawLine(0, vMeja, 0, -vMeja);
  g.drawLine(-crtica, -2*vMeja/3, crtica, -2*vMeja/3);
  g.drawLine(-crtica, 2*vMeja/3, crtica, 2*vMeja/3);
}
```

Z metodo `setColor()` smo nastavili barvo risanja na črno barvo, torej se koordinatni sistem izriše s črnimi črtami.

Podobno se lotimo tudi izrisa same funkcije. Tukaj moramo najprej določiti razmerje med vrednostjo *x* in radiani (širina risalne površine naj bo enaka vrednosti 3 PI). Na *y* osi postavimo vrednost 1 na 2/3 izrisane koordinatne osi. Oboje smo pravzaprav določili že v prejšnji metodi ob izrisu koordinatnega sistema, ko smo izrisali oznake enot (črtice). Izrisa grafa se lotimo tako, da za vsako vrednost *x* izračunamo vrednost $y = \sin(x)$ (seveda z ustrezno pretvorbo enot) in dobljeno točko izrišemo kot majhen rdeč krogec.

```
private void narisiSinus(Graphics g) {
  Dimension velikost = this.getSize();
  int hMeja = velikost.width/2;
  int vMeja = velikost.height/2;
  double razmerjeX = 3*Math.PI/(2*hMeja);
  double razmerjeY = 2*vMeja/3;
  int korak = hMeja/6;
  g.setColor(Color.red);
  for(int x = -hMeja; x <= hMeja; x++) {
    int y = (int) (razmerjeY * Math.sin(x*razmerjeX));
    y = -y;
    g.fillOval(x-1, y-1, 3, 3);
  }
}
```

Bodite pozorni na stavek:

```
y = -y;
```

Ker koordinata *y* v koordinatnem sistemu gradnika narašča od zgoraj navzdol, moramo izračunano vrednost *y* najprej obrniti (preslikati preko osi *x*), da jo pravilno izrišemo.

Bitne slike

Bitne slike (kot so slike v datotekah *gif* ali *jpg*) so predstavljene z objekti razreda `java.awt.Image`.

Če imamo bitno sliko zapisano v datoteki (v formatu GIF, JPEG ali PNG), jo lahko preberemo z metodo `getImage()` in jo naložimo v pomnilnik. Omenjena metoda vrne objekt `Image`, ki je referenca na bitno sliko v pomnilniku.

Pred tem pa moramo s klicem metode `getDefaultToolkit()` dobiti tudi ustrezen izvod razreda, ki izvede vmesnik `Toolkit`, saj metoda `getImage()` pripada temu razredu:

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image slika = tk.getImage("slika.jpg");
```

Bitno sliko potem lahko prikažemo na zaslonu s pomočjo metode `drawImage()` razreda `Graphics`.

Kot primer si poglejmo program `Slika.java`, ki iz tekočega direktorija na disku iz datoteke *slika.jpg* prebere sliko in jo prikaže na zaslonu (na risalni površini v oknu).

Okno in risalno površino določimo podobno kot v prejšnjem primeru (za opis glejte primer `Sinus.java` v razdelku *Risanje z grafičnimi primitivi*). Risalno površino predstavlja objekt razreda `Slika`, ki ga izpeljemo iz razreda `JPanel`. Prekrita metoda `paintComponent()` poskrbi za izris bitne slike, če le-ta obstaja:

```
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  if(bitna != null)
    g.drawImage(bitna, ROB, ROB, this);
}
```

Slika je shranjena v spremenljivki `bitna`, ki je privatni atribut razreda:

```
private Image bitna = null;
```

Ker slike na začetku še nimamo, nastavimo vrednost tega atributa na `null`.

Za nalaganje slike poskrbimo že v konstruktorju razreda `Slika`. Konstruktor lahko zapišemo kot:

```
public Slika() {
  bitna=Toolkit.getDefaultToolkit().getImage("slika.jpg");
  this.sirina = bitna.getWidth(this) + 2*ROB;
  this.visina = bitna.getHeight(this) + 2*ROB;
}
```

Na koncu (zadnja dva stavka) smo poskrbeli, da se velikost risalne površine prilega velikosti prebrane slike (z upoštevanjem nekaj roba okoli slike). Metodi `getWidth()` in `getHeight()` namreč vrneta širino in višino slike oziroma `-1`, če velikost slike še ni poznana. V praksi lahko ugotovimo, da se obe metodi izvršita veliko prej, kot se naloži cela slika v pomnilnik, zato vedno vrneta vrednost `-1`, ker velikost slike še ni poznana. Metoda `getImage()` namreč vrne le referenco na sliko, ki dobi podatke iz podane datoteke, ne poskrbi pa v celoti za prenos vseh teh podatkov iz datoteke in za pripravo slike na izris na gradniku. Zato moramo pred določitvijo velikosti risalne površine počakati, da se iz datoteke preberejo vsi podatki o sliki oziroma dokler ne dobimo pozitivnih vrednosti za širino in višino slike. Zato dodamo zanko, katera se v prazno izvaja toliko časa, dokler podatki o velikosti slike niso znani:

```
while((bitna.getWidth(this)<0)||(bitna.getHeight(this)<0))
  ;
```

Zanka se torej zaključi, ko obe metodi `getWidth()` in `getHeight()` vrneta pozitivni vrednosti.

Program `Slika.java` prikazuje, kako lahko enostavno izrišemo bitno sliko. Pri tem se nismo obremenjevali s tem, ali datoteka s sliko sploh obstaja in ali jo program lahko prebere. V primeru napak pri branju slike bi naš program obtičal v neskončni zanki (saj velikost neobstoječe slike ni nikoli znana) in se ne bi nikoli končal. Problem lahko enostavno rešimo tako, da v zanko vpeljemo števec `i` in izvajanje zanke prekinemo, ko števec `i` preseže neko vnaprej določeno veliko vrednost.

Risanje na bitno sliko v ozadju

Poglejmo si še en primer, kako lahko bitno sliko uporabimo pri risanju na gradnik. V našem naslednjem primeru bomo v okno postavili risalno površino in gumb. Na risalni površini se bodo ob vsakem miškinem kliku okoli točke klika narisali krogi. S klikom na gumb pa bomo izbrisali celo risalno površino.

Program bomo napisali tako, da se risanje ne bo izvajalo neposredno na risalno površino, temveč bo potekalo na sliko risalne površine v pomnilniku in šele po zaključenem izrisu bomo z metodo `paint` poskrbeli za prikaz te slike risalne površine na zaslonu. Tak postopek imenujemo dvojno pomnjenje (*double buffering*).

Swing ima vgrajeno podporo za dvojno pomnjenje in vsi gradniki Swing privzeto uporabljajo to tehniko.

Za začetek napišemo razred `Krogi`, ki vsebuje `main` metodo. V njej ustvarimo `okno`, objekt razreda, izpeljanega iz `JFrame`, ter ga prikažemo na zaslonu:

```
public class Krogi {
  public static void main(String[] args) {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}
```

Razred `Okvir`, ki je izpeljan iz razreda `JFrame`, ima med drugim dva atributa. Eden je gumb `JButton`, drugi pa določa risalno površino, to je objekt razreda `Risalnica`, izpeljanega iz razreda `JPanel`. V konstruktorju razreda `Okvir` poskrbimo za primerno postavitev gradnikov in druge, sedaj nam že domače podrobnosti:

```
class Okvir extends JFrame {
  private Container vsebina = null;
  private Risalnica platno = new Risalnica();
  private JButton brisi = new JButton("Briši");

  public Okvir() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ...
    vsebina = this.getContentPane();
    vsebina.add(brisi, BorderLayout.SOUTH);
    vsebina.add(platno, BorderLayout.CENTER);
  }
}
```

Risalna površina se mora tudi odzivati na miškine klike. Zato ji moramo dodati ustreznega poslušalca, to je `MouseListener`, ki poskrbi za primeren odziv. Če uporabimo vmesnik `MouseListener`, moramo prekriti vse štiri njegove metode, čeprav nas zanima le ena metoda, to je `mouseClicked()`. Lahko bi sicer namesto vmesnika uporabili razred `MouseAdapter`, a ker je razred `Okvir` že izpeljan iz razreda `JFrame`, ne more istočasno razširjati tudi razreda `MouseAdapter`. Ena rešitev je, da definiramo povsem nov razred za poslušalca. A ker gre le za enostaven odziv na dogodek, je krajša in bolj enostavna uporaba notranjega anonimnega razreda:

```
platno.addMouseListener(new MouseAdapter() {
  public void mouseClicked(MouseEvent e) {
    platno.narisiKrog(e.getX(), e.getY());
    platno.repaint();
  }
});
```

Z metodama `getX()` in `getY()` smo dobili koordinati (x,y) tiste točke v gradniku, v kateri se je zgodil miškin klik. Obe vrednosti posredujemo metodi `narisiKrog()`.

Gumbu smo določili oranžno barvo ozadja (klic metode `setBackground()`) in črno barvo ospredja oziroma barvo napisa (metoda `setForeground()`). Ker se gumb tudi odziva na klik, mu moramo dodati poslušalca za ta dogodek (`ActionListener`). Tudi tu smo se odločili za anonimen notranji razred, saj gre le za enostaven odziv na dogodek:

```
brisi.setBackground(Color.orange);
brisi.setForeground(Color.black);
brisi.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent d) {
    platno.brisiRisalnico();
    platno.repaint();
  }
});
```

V razredu `Risalnica` definiramo obe metodi (za risanje krogov in za brisanje risalne površine). V razredu imamo privatno spremenljivko `drugaSlika`, ki predstavlja sliko risalne površine v pomnilniku:

```
private BufferedImage drugaSlika = null;
```

Prekrijemo metodo `paintComponent()`, v kateri pa nimamo nobene kode za risanje, temveč le prikažemo sliko risalne površine (če ta obstaja). Seveda pred tem poskrbimo za pravilen izris in čiščenje ozadja:

```
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  if(drugaSlika != null)
    g.drawImage(drugaSlika, 0, 0, this);
}
```

Tudi metoda `brisiRisalnico()` je enostavna. Spremenljivko `drugaSlika` postavi na `null` in s tem se obstoječa slika izbriše:

```
public void brisiRisalnico() {
  drugaSlika = null;
}
```

Vso kodo v zvezi z risanjem tako postavimo v metodo `narisiKrog()`, ki poskrbi za izris krogov. Če slika `drugaSlika` ne obstaja, se najprej kreira nova slika enake velikosti kot je risalna površina. To dosežemo s klicem konstruktorja `BufferedImage`, ki mu kot argumente podamo velikost slike in njen tip. Slednji je kar ena od konstant tega razreda; v našem primeru smo izbrali 8-bitni RGBA barvni model. Potem s klicem metode `getGraphics()` dobimo referenco na objekt `Graphics`, to je grafični kontekst te slike, s pomočjo katerega lahko rišemo na sliko. Naključno izberemo barvo risanja izmed različnih barv v tabeli `barve` ter v zanki izrišemo pet krogov, vsakega z malo večjim polmerov, središče vseh pa je v točki (x, y); x in y sta parametra metode:

```
public void narisiKrog(int x, int y) {
  if(drugaSlika == null)
    drugaSlika =
      new BufferedImage(getSize().width, getSize().height,
                        BufferedImage.TYPE_4BYTE_ABGR);
  Graphics g = drugaSlika.getGraphics();
  g.setColor(barve[(int) (Math.random()*barve.length)]);
  for(int r=5; r<26; r+=5)
    g.drawOval(x-r, y-r, 2*r, 2*r);
}
```

Celotna izvorna koda tega programa je v datoteki `Krogi.java`.

Apleti

Apleti so javanski programčki, ki se ne izvajajo samostojno, saj je njihov namen razširitev zmogljivosti njihovega izvajalnega okolja. Aplet je torej javanska koda, ki se izvaja v pregledovalniku *Applet Viewer* ali znotraj javansko osveščenega brskalnika.

Telo apleta mora biti razred, ki je izpeljan iz že pripravljenega razreda `Applet` (iz paketa `java.applet`) ali pa iz njega izpeljanega razreda `JApplet`, kadar gre za Swing komponento (paket `javax.swing`). Oba razreda ponujata metode za vzpostavitev, zagon, zaustavitev in uničenje apleta (metode `init()`, `start()`, `stop()` in `destroy()`, po vrsti).

Za razliko od samostojnih javanskih programov, ki morajo imeti vstopno metodo `main()`, kjer se začne izvajati koda, imajo apleti metodo `init()`, v kateri se začne izvajati koda. Zato ponavadi v apletu prekrijemo metodo `init`, ostale štiri metode pa le po potrebi.

Najpreprostejši aplet, ki izpiše niz “Pozdravljeni!”, bi torej lahko izgledal takole:

```
import javax.swing.*;

public class MojAplet extends JApplet {
  public void init() {
    this.getContentPane().add(new JLabel("Pozdravljeni!"));
  }
}
```

Niz “Pozdravljeni!” smo zapisali v oznako (gre za grafični program!), ki smo jo postavili na delovno površino apleta.

Za pregled apletov moramo napisati tudi html datoteko, kjer med oznakama `<APPLET>` in `</APPLET>` določimo datoteko s prevedeno javansko kodo in (opcijsko) velikost okna, v katerem se bo izvajala. V našem primeru gre za javansko vmesno kodo iz datoteke `MojAplet.class`, velikost pa je 300 pik po širini in 50 pik po višini:

```
<APPLET CODE="MojAplet.class" WIDTH=300 HEIGHT=50></APPLET>
```

Obe datoteki, `MojAplet.java` in `MojApletTest.html`, najdete med primeri tega poglavja.

Delovanje primera lahko preizkusimo tako, da v ukaznem oknu pokličemo pregledovalnik apletov, ki mu kot argument podamo html datoteko (seveda je potrebno program pred tem prevesti):

```
appletviewer MojApletTest.html
```

Druga možnost pa je, da html datoteko `MojApletTest.html` odpremo neposredno v spletnem brskalniku.

Za konec pa si poglejmo še primer apleta, ki podan znesek preračuna iz tolarjev v evre. Ker smo vse prvine tega programa (gradnike oznaka, vnosno polje, gumb, vsebnike, razporejevalnike ter odziv gumba na akcijo) spoznali in opisali že na začetku tega poglavja, jih tu ne bomo ponovno razlagali, temveč zapišimo le kodo tega apleta:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Menjava extends JApplet {
  private final double tecaj = 239.64;
  private JLabel znesek=new JLabel("Vpiši znesek v SIT: ");
  private JLabel rez = new JLabel("");
  private JTextField vnos = new JTextField("0", 6);
  private JButton gumb = new JButton("Pretvori v EUR");

  public void init() {
    JPanel izracun = new JPanel();
    izracun.add(znesek);
    izracun.add(vnos);
    JPanel gumbi = new JPanel();
    gumbi.add(gumb);
    JPanel izpis = new JPanel();
    izpis.add(rez);
    Container vsebina = getContentPane();
    vsebina.add(izracun, BorderLayout.NORTH);
    vsebina.add(gumbi, BorderLayout.CENTER);
    vsebina.add(izpis, BorderLayout.SOUTH);
    gumb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent d) {
          rez.setText(vnos.getText() + "  SIT  =  " +
Math.round(100*Double.parseDouble(vnos.getText())/tecaj)/
100.0 + "  EUR");
        }
    });
  }
}
```

Pripadajočo html datoteko pa lahko zapišemo takole:

```
<HTML>
<HEAD><TITLE>Primer apleta</TITLE></HEAD>
<BODY>
  <H2>Pretvorba SIT v EUR</H2><BR>
  <APPLET CODE="Menjava.class" WIDTH=300 HEIGHT=100>
  </APPLET>
</BODY>
</HTML>
```

Omenjeno kodo najdete v datotekah `Menjava.java` in `menjava.html`.

Apleti in aplikacije

Kako izbrati, ali naj program napišemo, da bo deloval kot aplet ali kot samostojen program (aplikacija)? In zakaj ne bi bil naš program kar oboje hkrati?

Seveda lahko zapišemo kodo tudi tako, da naš program po potrebi deluje kot samostojen program (aplikacija) ali pa znotraj spletnega brskalnika (aplet)[1]. Izhajamo iz kode apleta:

```
public class Aplet extends JApplet {
  public void init() {
    // telo apleta
  }
}
```

Če želimo ta aplet poganjati kot samostojen program, mu moramo za začetek dodati metodo `main`, saj se vsak program začenja v tej metodi (podobno kot se vsak aplet začenja v metodi `init` in zažene z metodo `start`). Torej imamo:

```
public class Aplet extends JApplet {
  public void init() {
    // telo apleta
  }

  public void main(String[] args) {
    // poskrbimo za ustrezen zagon apleta
  }
}
```

V metodi `main` moramo torej poskrbeti za ustvarjanje primerno velikega okna, kateremu na delovno površino postavimo objekt razreda `Aplet`. Ne smemo pozabiti na klica obeh metod, s katerima poskrbimo za pravilen zagon apleta, torej `init` in `start`, ter seveda tudi na prikaz samega okna.

Pa pojdimo lepo po vrsti. Najprej ustvarimo objekt razreda `Applet`:

```
Aplet aplet = new Aplet();
```

Ustvarimo tudi novo okno, ki mu določimo ime, velikost in poskrbimo za ustrezno reakcijo ob zapiranju okna:

```
JFrame okno = new JFrame("Okno");
okno.setSize(400,200);
okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Nato postavimo objekt `aplet` v središče delovne površine okna (spomnimo se, da je privzet razporejevalnik za okna `BorderLayout`):

```
okno.getContentPane().add(aplet, BorderLayout.CENTER);
```

[1] Pri tem seveda nismo upoštevali tistih omejitev, ki jih imajo apleti zaradi varnostnih razlogov, kot tudi ne posebnih zmožnosti, ki so specifične le za aplete.

Na koncu moramo poskrbeti še za pravilen zagon apleta:

```
aplet.init();
aplet.start();
```

ter za prikaz okna na zaslonu:

```
okno.setVisible(true);
```

Metodo `main` bi torej lahko zapisali takole:

```
public static void main(String[] args) {
  Aplet aplet = new Aplet();
  JFrame okno = new JFrame("Okno");
  okno.setSize(400,200);
  okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  okno.getContentPane().add(aplet, BorderLayout.CENTER);
  aplet.init();
  aplet.start();
  okno.setVisible(true);
}
```

Program `ApletProgram.java` prikazuje tak dvoličen program, ki ga lahko izvajamo samostojno ali pa znotraj spletnega brskalnika. Program ne naredi nič posebnega, prikaže le eno oznako z vnaprej določenim besedilom.

Prikaz oznake smo vključili v metodo `init`:

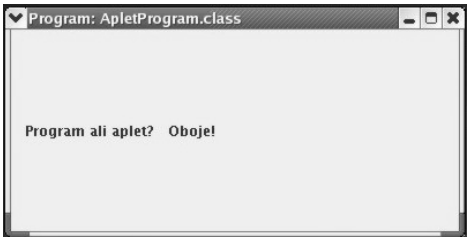
```
this.getContentPane().add(new JLabel("..."));
```

Seveda ne smemo pozabiti na pripadajočo html kodo (`ApletProgramTest.html`), ki jo uporabimo v primeru prikazovanja znotraj spletnega brskalnika ali programa *appletviewer*.

Program prevedemo, nato pa ga lahko izvajamo na dva načina. Z ukazom

```
java ApletProgram
```

poženemo samostojen program, katerega rezultat je okno, prikazano na sliki 6.9.

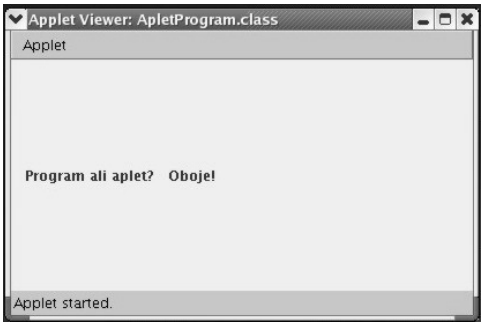


Slika 6.9: Program poženemo kot aplikacijo.

Podoben rezultat dobimo tudi z ukazom

```
appletviewer ApletProgramTest.html
```

pri čemer se naš program požene kot aplet znotraj html dokumenta (slednje je podobno, kot če html dokument odpremo v spletnem brskalniku). Rezultat je okno, ki ga prikazuje slika 6.10.



Slika 6.10: Program poženemo kot aplet.

V obeh primerih (sliki 6.9 in 6.10) se torej odpre okno, v katerem se prikaže aplet, znotraj njega pa se v oznaki izpiše neko besedilo.

In kako lahko ugotovimo, ali se program izvaja samostojno ali kot aplet? Če pozorno pogledamo obe sliki (6.9 in 6.10), opazimo, da se izgled samega okna nekoliko razlikuje. Poleg tega pa smo v primeru samostojnega programa (v metodi `main`) naslov okna določili sami (izbrali smo naslov "Program: ApletProgram.class"), medtem ko na naslov okna pri uporabi ukaza *appletviewer* nimamo vpliva (spletni brskalnik pa apleta sploh ne prikaže v samostojnem oknu, temveč ga vključi neposredno v vsebino strani).

Grafični programi in niti

Naslednja dva razdelka obravnavata večnitnost pri gradnji grafičnih programov. Ker je za ta razdelek pomembno vsaj osnovno poznavanje koncepta niti, priporočamo, da si prej pogledate poglavje o nitih (9. poglavje).

Pri pisanju grafičnih programov je zelo pomembna skrbna uporaba niti, saj jih dobro napisan program uporabi za ustvarjanje odzivnega uporabniškega vmesnika. Tak program se vedno odziva na uporabnikovo interakcijo (program ne »zmrzne«), ne glede na druga opravila v izvajanju. Zato je pri pisanju grafičnih programov zelo pomembno, da razumemo, kako Swing uporablja niti.

Grafični program v Javi deluje v treh vrstah niti: zagonskih nitih (*initial threads*), dogodkovnih nitih (*event dispatch threads*) in delovnih nitih (*worker threads*). Zagonske niti so tiste, v katerih se izvaja začetna koda aplikacije. V dogodkovnih nitih se izvaja vsa koda, ki skrbi za obravnavo dogodkov. Delovne niti pa skrbijo za časovno potratne naloge, ki se izvajajo v ozadju (opravila, ki tečejo v delovni niti, so objekti razreda `SwingWorker`, katerega opis najdete tudi v dokumentaciji JDK). Vse navedene

niti nudi že samo ogrodje Swing, zato ni potrebno, da jih eksplicitno ustvarimo. Naša naloga je le, da te niti pravilno uporabimo za ustvarjanje odzivnega in vzdržljivega grafičnega programa.

Ko program zaženemo, se vedno začne izvajati v metodi `main` (oziroma v metodi `init` pri apletih). Ob vstopu v metodo `main` se začne izvajati zagonska nit, ki se zaključi ob izstopu iz te metode. Dogodkovna nit teče vseskozi v ozadju in skrbi za obravnavanje dogodkov. Ta nit se zaključi ob zaprtju glavnega okna programa.

Ker metode Swing komponent večinoma niso nitno varne (*thread safe*), bi njihovo klicanje iz različnih niti lahko povzročilo napake v delovanju. Zato je pomembno, da program vse te metode kliče iz iste niti, in sicer tiste, ki obravnava tudi Swing dogodke, torej dogodkovne niti[1]. Tako vse grafične objekte ustvarjamo, izrisujemo in posodabljamo le v dogodkovni niti[2].

Za izgradnjo grafičnega uporabniškega vmesnika in izdelavo grafike v Javi torej uporabimo dogodkovno nit[3]. Pri grafičnih programih tako v metodi `main` (v zagonski niti) le pokličemo statično metodo `invokeLater()` razreda `SwingUtilities` (iz paketa `javax.swing`), ki poskrbi za zagon v dogodkovni niti. Kaj želimo zagnati, podamo kot parameter metodi. Ta parameter mora biti objekt razreda, ki implementira vmesnik `Runnable`, v svoji metodi `run()` pa ima zapisano kodo, ki naj se izvede v dogodkovni niti. Metodo `invokeLater()` tako pokličemo z:

```
MojRazred nit = new MojRazred();
SwingUtilities.invokeLater(nit);
```

kjer `MojRazred` lahko definiramo kot:

```
public class MojRazred implements Runnable {
  public void run() {
    // koda, ki se izvede v dogodkovni niti
  }
}
```

Metoda `run` niti `nit` se izvede, ko se (ponovno) prične izvajati dogodkovna nit.

Ker objekta `nit` sicer ne potrebujemo (ustvarimo ga le za potrebe klica metode `invokeLater`), lahko uporabimo kar anonimen razred, ki mu le definiramo metodo `run`: v njej zaradi enostavnosti in preglednosti samo pokličemo drugo metodo, ki poskrbi za izgradnjo grafičnega vmesnika.

Celotno kodo lahko povzamemo v naslednjih vrsticah:

[1] Nekatere metode Swing komponent so v API specifikaciji označene kot nitno varne in te lahko kličemo iz katerekoli niti.
[2] Če navedenega pri pisanju programa ne upoštevamo, se bo program večino časa sicer lahko obnašal pravilno, a občasno se lahko pojavijo nepredvidljive napake, ki jih tudi težko ponovimo (kar seveda zelo oteži razhroščevanje).
[3] Naših programov v prejšnjih razdelkih nismo gradili na tak način, saj še nismo znali uporabiti niti. Delovanje programov pa je bilo za ponazoritev pisanja grafičnih programov in uporabe grafičnih komponent povsem zadovoljivo, čeprav pri resnejših projektih to ne bi bilo sprejemljivo.

```
import javax.swing.*;

public class GraficniProgram {
  private static void ustvariGUI() {
    //
    // koda za ustvarjanje okna
    //
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        ustvariGUI();
      }
    });
  }
}
```

Ko grafični uporabniški vmesnik enkrat ustvarimo, delovanje programa vodijo predvsem dogodki grafičnega vmesnika (torej metode za obravnavanje dogodkov, kot je na primer `actionPerformed`), od katerih vsak povzroči izvedbo kratkega opravila v dogodkovni niti. Vsa opravila v dogodkovni niti se morajo hitro zaključiti, saj se v nasprotnem primeru nakopičijo neobdelani dogodki in uporabniški vmesnik postane neodziven (za daljša opravila pa uporabimo delovno nit).

Zgoraj omenjen način izdelave grafičnega programa je primeren za aplikacije, pri katerih se izvajanje začne v metodi `main`. Pri apletih, kjer se izvajanje programa začne v metodi `init`, pa moramo ustrezno kodo postaviti v omenjeno metodo.

Poleg tega moramo pri apletih za razporeditev opravila izgradnje grafičnega vmesnika uporabiti metodo `invokeAndWait` namesto metode `invokeLater`. Obe metodi sta sicer zelo podobni in obe prejmeta en sam parameter, to je objekt `Runnable`, ki definira novo opravilo. Edina razlika med njima je, da metoda `invokeLater` le razporedi opravilo za izvajanje in takoj konča, medtem ko metoda `invokeAndWait`, kot pove že njeno ime, tudi počaka, da se opravilo izvede do konca in šele nato konča. To je zelo pomembno pri apletih, saj bi se ob uporabi metode `invokeLater` lahko metoda `init` zaključila, še preden bi bil ustvarjen grafični uporabniški vmesnik, kar bi lahko povzročilo probleme v spletnem brskalniku, ki je zagnal ta aplet. Nasprotno pa je pri aplikacijah razporeditev opravila, ki ustvari grafični uporabniški vmesnik, ponavadi zadnja stvar, ki jo naredi zagonska nit, zato je pravzaprav vseeno, ali uporabimo metodo `invokeLater` ali metodo `invokeAndWait`.

Pri apletih bi torej uporabili naslednjo kodo:

```
private void ustvariGUI() {
  //
  // sestavimo uporabniški vmesnik
  //
}

public void init() {
```

```
    try {
      SwingUtilities.invokeAndWait(new Runnable() {
        public void run() {
          ustvariGUI ();
        }
      });
    }
    catch(Exception e) {
      System.err.println("Napaka pri ustvarjanju GUI.");
    }
  }
```

Ker metoda `invokeAndWait` lahko sproži tudi izjemo (podrobnosti o tem najdete v dokumentaciji JDK), to izjemo prestrežemo in ustrezno obravnavamo.

Animacije

Za konec pa napišimo še program, pri katerem bomo uporabili znanje, ki smo ga pridobili v tem poglavju. Želeli bi narediti animacijo žogice, ki se giblje znotraj določenega igrišča in se odbija od njegovih sten.

Začnemo z osnovo, pri kateri bomo žogico premikali ročno, s klikom na gumb. Ker so vse uporabljene prvine že razložene v drugih razdelkih tega poglavja, naj tu le na splošno opišemo rešitev in opozorimo na nekaj podrobnosti. Celoten program je v datoteki `RocnaAnimacija.java`.

V metodi `main` najprej poskrbimo, da se v dogodkovni niti ustvari in izriše okno, v katerega smo postavili objekt `igrisce` (razreda `Igrisce`), ki predstavlja površino, po kateri se bo žogica gibala, in gumb `gumb`, s katerim bomo kontrolirali premikanje žogice. Razred `Okno` smo hkrati uporabili še za poslušalca dogodkov nad gumbom: ob vsakem kliku na gumb se nad objektom `igrisce` izvede metoda `zazeni`.

```
public class RocnaAnimacija {

  private static void ustvariGUI() {
    Okno okno = new Okno();
    okno.setVisible(true);
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        ustvariGUI();
      }
    });
  }
}


class Okno extends JFrame implements ActionListener {
```

```
  private final int SIRINA = 400;
  private final int VISINA = 500;
  private Igrisce igrisce;
  private JButton gumb;

  public Okno() {
    setTitle("Animacija žogice");
    setSize(SIRINA, VISINA);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    igrisce = new Igrisce();
    gumb = new JButton("Žoga!");
    gumb.addActionListener(this);
    Container c = this.getContentPane();
    c.add(igrisce, BorderLayout.CENTER);
    c.add(gumb, BorderLayout.SOUTH);
  }

  public void actionPerformed(ActionEvent e) {
    Object izvor = e.getSource();
    if(izvor != gumb)
      return;
    if(!gumb.getText().equals("Korak ..."))
      gumb.setText("Korak ...");
    igrisce.zazeni();
  }
}
```

Razred `Igrisce` vsebuje eno žogico (objekt razreda `Zoga`) in tudi poskrbi za izris te žogice v svoji metodi `paintComponent` preko klica metode `narisi` razreda `Zoga`. Poleg tega definira metodo `zazeni`, ki poskrbi za premik žogice (klic metode `premakni`, ki je definirana v razredu `Zoga`) in ponoven izris trenutnega stanja (klic metode `repaint`).

```
class Igrisce extends JPanel {
  private Zoga zogica;

  public Igrisce() {
    setBackground(Color.white);
    zogica = new Zoga(this);
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    zogica.narisi(g);
  }

  public void zazeni() {
    zogica.premakni();
    repaint();
  }
}
```

Razred `Zoga` je tisti, ki opisuje našo žogico, njeno premikanje in odboje od sten igrišča. Definira dve metodi, ki smo ju že omenili: metoda `narisi` izriše žogico na podan

grafični kontekst, metoda `premakni` pa premakne žogico in hkrati preveri, ali je žogica prišla do kateregakoli roba in ji v tem primeru ustrezno spremeni smer gibanja. Razred ima tudi več atributov, od katerih sta `BARVA` in `RADIJ` konstanti, ki po vrsti določata barvo narisane žogice in njen polmer. Vrednosti ostalih atributov se nastavijo v konstruktorju: `x` in `y` določata trenutni položaj (središča) žogice na igrišču (začetna vrednost je zgornji levi kot), `hitrostX` in `hitrostY` pa hitrost premikanja žogice v *x* in *y* smeri po vrsti (začetni vrednosti sta nastavljeni naključno med 1 in 10). Zadnji atribut je referenca na samo igrišče, v katerem se žogica premika. To potrebujemo, da lahko ugotovimo, kdaj je žogica prišla do roba igrišča (tudi če se velikost igrišča med delovanjem spreminja, to ne vpliva na pravilno zaznavanje robov igrišča).

```
class Zoga {
  private final Color BARVA = Color.red;
  private final int RADIJ = 20;
  private Igrisce igr;
  private int hitrostX, hitrostY;
  private int x, y;

  public Zoga(Igrisce i) {
    this.igr = i;
    this.x = RADIJ;
    this.y = RADIJ;
    this.hitrostX = (int)(Math.random()*10+1);
    this.hitrostY = (int)(Math.random()*10+1);
  }

  public void narisi(Graphics g) {
    g.setColor(BARVA);
    g.fillOval(x-RADIJ, y-RADIJ, 2*RADIJ, 2*RADIJ);
  }

  public void premakni() {
    x += hitrostX;
    y += hitrostY;
    if(x <= RADIJ) {
      x = RADIJ;
      hitrostX = -hitrostX;
    }
    else if(x >= igr.getWidth()-RADIJ) {
      x = igr.getWidth()-RADIJ;
      hitrostX = -hitrostX;
    }
    if(y <= RADIJ) {
      y = RADIJ;
      hitrostY = -hitrostY;
    }
    else if(y >= igr.getHeight()-RADIJ) {
      y = igr.getHeight()-RADIJ;
      hitrostY = -hitrostY;
    }
  }
}
```

Zgornji program sicer deluje in prikazuje odbijanje žogice po igrišču, a bi za popoln učinek animacije želeli samodejno premikanje žogice.

To lahko dosežemo tako, da posamezne premike, ki se sedaj zgodijo ob vsakem kliku na gumb, postavimo v zanko, ki se samodejno izvaja, dokler je ne prekinemo. Za zagon animacije in njeno prekinitev bomo uporabili kar pripravljen gumb, zato moramo spremeniti kodo, ki se izvede ob kliku na gumb:

```
public void actionPerformed(ActionEvent e) {
  Object izvor = e.getSource();
  if(izvor != gumb)
    return;
  if(!gumb.getText().equals("Ustavi")) {
    gumb.setText("Ustavi");
    igrisce.zazeni(); //začetek animacije(zaženi animacijo)
  }
  else {
    gumb.setText("Poženi");
    igrisce.ustavi(); //konec animacije (ustavi animacijo)
  }
}
```

Na gumbu se sedaj ob vsakem kliku izmenjujeta napisa *Ustavi* in *Poženi*. Za zagon animacije smo uporabili metodo `zazeni`, za ustavitev animacije pa metodo `ustavi`. Obe metodi sta definirani v razredu `Igrisce` in ju moramo še napisati (oz. ustrezno spremeniti).

V metodi `zazeni` moramo sedaj celotno kodo postaviti v zanko, ki se izvaja toliko časa, dokler ne bo uporabnik s klikom na gumb prekinil izvajanje animacije. Za to kontrolo, ali se animacija izvaja ali ne, bomo uvedli novo spremenljivko `ustavi`, ki naj bo kar atribut razreda `Igrisce`. Ob zagonu animacije bo ta spremenljivka dobila vrednost `false`, zanka pa se bo izvajala toliko časa, dokler je vrednost spremenljivke enaka `false` (njeno vrednost bomo spremenili z metodo `ustavi`, kot je opisano v nadaljevanju).

```
public void zazeni() {
  ustavi = false;
  while(!ustavi) {
    zogica.premakni();
    repaint();
  }
}
```

Metoda `ustavi` je enostavna, saj zahteva le spremembo vrednosti atributa `ustavi`, ki mora postati `true`.

```
public void ustavi() {
  ustavi = true;
}
```

Preizkusimo delovanje tako popravljenega programa. Katastrofa! Začetno stanje se sicer izriše, a takoj po zagonu animacije se program neha odzivati (premiki žogice se ne izrisujejo, klik na gumb ne deluje več, okna pa tudi ne moremo zapreti – program moramo prekiniti z uporabo `Ctrl+c`).

V čem je problem? Pozabili smo na pomembno pravilo: opravila v dogodkovni niti morajo biti kratka, saj se v nasprotnem primeru dogodki nakopičijo in uporabniški vmesnik postane neodziven. In kaj smo naredili mi? Kodo za animacijo izvajamo v zanki, dokler se vrednost atributa `ustavi` ne spremeni. Ta zanka je del odziva na klik na gumb, torej eden izmed dogodkov v dogodkovni niti. Vrednost atributa `ustavi` lahko spremenimo le s klikom na gumb, torej z novim dogodkom, ki pa ne more biti obdelan, dokler se ne konča obdelava prvega dogodka (torej dokler se naša zanka ne zaključi). Tako smo ustvarili neskončno zanko in neodziven program. In ker je ponovno izrisovanje žogice tudi eden v vrsti dogodkov (metoda `repait` ob vsakem klicu sproži nov dogodek), se tudi izrisi premaknjenih žogic ne morejo osveževati in tako animacija ni vidna.

In kako lahko kodo popravimo, da bo delovala? Rešitev je, da napišemo kodo za odziv na dogodek klika na gumb dovolj kratko, da se hitro zaključi. Zato za izvajanje same animacije (ponavljajoča zanka) uporabimo svojo nit, v metodi `zazeni`, ki se mora dovolj hitro izvesti, pa to nit le ustvarimo in poženemo. Tako smo poskrbeli, da je odziv na dogodek kratek, časovno bolj potratna animacija pa teče potem neodvisno v svoji niti. Program lahko tako hkrati izvaja animacijo in se odziva na dogodke.

Popravimo najprej kodo za animacijo. V razred `Igrisce` dodamo nov atribut `nit`, ki predstavlja nit, v kateri se izvaja animacija.

```
private Thread nit;
```

Koda, ki se bo izvajala v niti, mora biti zapisana v metodi `run` objekta `Runnable`. Mi bomo to metodo `run` definirali kar v razredu `Igrisce`, ki mora zato implementirati vmesnik `Runnable`:

```
class Igrisce extends JPanel implements Runnable
```

V metodo `run` postavimo zanko, v kateri se izvaja animacija:

```
public void run() {
  while(!ustavi) {
    zogica.premakni();
    repaint();
  }
}
```

Da se animacija ne izvaja prehitro, je smiselno v zanko vključiti še ukaz, s katerim izvajanje niti za krajši čas prekinemo. Klic metode `sleep` prekine izvajanje niti za določeno število milisekund, ki ga podamo kot parameter metode. Metodo `sleep` moramo postaviti v `try/catch` blok, saj lahko sproži izjemo v primeru, če katerakoli druga nit prekine obstoječo nit. Ker nas to niti ne zanima, izjemo le prestrežemo in ne naredimo nič (`catch` del je prazen).

```
public void run() {
  while(!ustavi) {
    try {
      nit.sleep(25);
    }
    catch(InterruptedException e) {
    }
    zogica.premakni();
    repaint();
  }
}
```

Sedaj moramo na ustreznem mestu le še ustvariti in zagnati nit z animacijo. Ta koda se mora izvesti takrat, ko želimo izvajati animacijo, torej kot odgovor na klik na gumb. Metodo `zazeni` torej spremenimo tako, da najprej ustvari novo nit (v niti naj se izvaja metoda `run` razreda `Igrisce`), nato pa to nit tudi zažene:

```
public void zazeni() {
  ustavi = false;
  nit = new Thread(this);
  nit.start();
}
```

S tem so naši popravki zaključeni. Izvajanje programa je sedaj tekoče in program se lepo odziva na interakcijo z uporabnikom (kliki na gumb, zapiranje okna ...). Koda celotnega programa je v datoteki `Animacija.java`.

7. Datoteke in tokovi

Tokovi (*streams*) so objektna predstavitev vhodno/izhodnih pogovorov in omogočajo branje oz. zapisovanje podatkov. Prednost uporabe tokov je v tem, da različne zunanje enote (kot so zaslon, datotečni sistem, tiskalnik, omrežna vtičnica, idr.) prikažejo kot razred, ki omogoča njihovo preprosto obravnavo pri izmenjavi podatkov. Tako lahko tokove uporabljamo tudi za dostop do datotek.

Izraz tok se nanaša na zaporedni dostop do podatkov, kjer vsebino preberemo kot zaporedje bajtov. Razrede za delo s tokovi ponuja paket `java.io`.

Tokove lahko v grobem razdelimo na znakovne in binarne tokove. Vsaka od skupin pa se deli tudi na vhodne in izhodne tokove. Iz tega izhajajo tudi štirje temeljni razredi, ki jih v Javi uporabljamo pri delu s tokovi. Za binarne tokove (tok bajtov) sta temeljna razreda `InputStream`, ki predstavlja vhodne tokove za branje, in `OutputStream`, ki zajema izhodne tokove za pisanje. Pri znakovnih tokovih pa paket `java.io` nudi temeljna razreda `Reader` (za branje) in `Writer` (za pisanje). Ponavadi v praksi uporabljamo razrede, ki so izpeljani iz teh temeljnih razredov.

Nov tok ustvarimo s klicem konstruktorja. Tok lahko zapremo eksplicitno s klicem metode `close()` ali pa počakamo, da se zapre implicitno, ko ga pospravi smetar (ko se nanj ne sklicuje nihče več). Ker pa so tokovi pogosto vezani na pomembna sistemska sredstva, je zelo priporočljivo, da jih vedno zapiramo z metodo `close()`, takoj ko je to mogoče.

Preden pa se lotimo konkretnih primerov dela z datotekami in tokovi, si poglejmo še izjeme in njihovo obravnavo, saj se bomo pri delu z datotekami zagotovo srečali z njimi.

Izjeme

Javanska koda, če je napisana pravilno, je lahko zelo zanesljiva in trdoživa. K temu pripomore tudi mehanizem izjem, ki je vgrajen v jezik in izvajalno okolje. Tako lahko v programu predvidimo tudi neobičajne okoliščine delovanja in nanje ustrezno reagiramo.

Kaj so izjeme

Med procesom izvajanja programa se lahko pripetijo različni neobičajni dogodki. Obravnava problematičnih okoliščin, ki jih imenujemo tudi izjemni dogodki ali izjeme (*exceptions*), je pri Javi vgrajena v sam jezik.

Izjeme so torej nepričakovani dogodki ali napake, ki se pojavijo med izvajanjem programa, zaradi katerih se prekine normalen potek programskih ukazov, saj nadaljnje izvajanje ni mogoče. Primeri izjem so deljenje z nič, branje iz neobstoječe datoteke, nedostopno omrežje, napačen URL naslov, dostop do elementa z indeksom izven meja tabele ...

Vsaka metoda lahko poleg običajne vrnjene vrednosti vrne tudi objekt, ki opisuje izjemni dogodek. Temu rečemo, da metoda sproži izjemo (*throws an exception*). Izvajanje metode se zaključi v trenutku, ko sproži izjemo. Takrat se ustvari nov objekt, ki opisuje to izjemo, in ta potuje nazaj po klicnem skladu (*method call stack*), dokler ne najde primernega prestreznika izjem (*exception handler*).

Vrste izjem

Posebno obnašanje izjem in njihovo potovanje po klicnem skladu omogoča razred `Throwable`, ki je skupni predhodnik vseh izjemnih dogodkov. Iz njega sta izpeljani dve veji, razred `Error` in razred `Exception`. Prvi je namenjen opisu hudih napak, zaradi katerih program ne more več nadaljevati (na primer izčrpanje ključnega vira, kot je pomnilnik). V tem primeru ne moremo narediti veliko, zato ponavadi takih napak sploh ne obravnavamo. Drugi pa opisuje izjeme, katerih obravnavo ponavadi vključimo v pravilno napisan program.

Razred `RuntimeException` je neposredno izpeljan iz razreda `Exception` in opisuje izjeme, ki so posledica napake v programu (na primer prekoračitev obsega polja ali neprimerna pretvorba tipov). Izjemam te vrste bi se s previdnejšim kodiranjem lahko izognili, zato jih ponavadi v kodi niti ne razglašamo niti ne prestrezamo.

Vsi ostali potomci razreda `Exception` pa opisujejo izjeme, ki jih v programu težko zaznamo in preprečimo (na primer nedostopno omrežje ali nedovoljen dostop do datoteke). Zato zanje poskrbimo s programsko obravnavo (razglašanjem ali prestrezanjem) teh izjem.

Izjeme, ki jih določata razreda `Error` in `RuntimeException` ter njuni podrazredi, imenujemo nepreverjene izjeme (*unchecked exceptions*). Zanje velja, da jih ponavadi ne obravnavamo v kodi.

Vse ostale izjeme, ki jih določajo razred `Exception` in vsi njegovi podrazredi, razen podrazreda `RuntimeException`, pa so preverjene izjeme (*checked exceptions*) in zanje ponavadi poskrbimo z ustrezno programsko obravnavo.

Proženje izjem

Kadar med delovanjem metode lahko pride do izjeme, tako metodo posebej označimo, da metoda lahko sproži izjemo.

Če želimo razglasiti, da med delovanjem metode lahko pride do izjeme, zapišemo v glavi metode tudi tip možne izjeme. Pri tem uporabimo rezervirano besedo `throws`:

```
public int pisiDat(String imeDat) throws IOException
```

Metoda `pisiDat` zapiše podatke v datoteko `imeDat`. Pri tem lahko pride do izjeme `IOException`, ki jo metoda sproži. Tako prevajalnik ve, da metoda sicer vrača

vrednost `int`, a se ob napaki lahko tudi predčasno konča in sproži izjemo tipa `IOException`. Ista metoda lahko razglasi tudi več izjem:

```
public int pisiDat(String imeDat)
  throws FileNotFoundException, IOException
```

Ko v metodi zaznamo neobičajno stanje ali dogodek, ki onemogoči pravilno delovanje metode, znotraj metode sprožimo izjemo:

```
throw new IOException("Datoteka ne obstaja");
```

Ustvarili smo nov objekt `IOException` z opisom izjeme (objekt mora biti tipa `Throwable`). S stavkom `throw` prekinemo izvajanje metode ter posredujemo izjemo (novo ustvarjeni objekt) po klicnem skladu do ustreznega prestreznika izjem.

Kjer kličemo metode, ki razglašajo izjeme, imamo tri možnosti za obravnavo izjem:
- izjemo lahko spustimo nazaj po klicnem skladu,
- izjemo prestrežemo in jo ustrezno obdelamo ali pa
- izjemo prestrežemo in sprožimo drugo izjemo (novo izjemo bomo morali obravnavati kje drugje).

Kadar se odločimo za možnost, da izjemo spustimo nazaj po klicnem skladu, moramo v glavi metode le razglasiti njen tip (na primer `throws IOException`). S tem povemo, da naša metoda ne bo obravnavala izjem, ampak jih bo le posredovala naprej po klicnem skladu.

Prestrezanje in obravnavo izjem pa opisuje naslednji razdelek.

Obravnava izjem

Obravnava izjem (*exception handling*) pomeni prestrezanje sproženih izjem in ustrezno reakcijo nanje.

Lasten prestreznik izjem zapišemo s stavčnim blokom `try`/`catch`/`finally`, ki ga simbolično lahko zapišemo takole:

```
try {
  // stavki, ki lahko sprožijo izjemo
}
catch (XException e) {
  // obravnava izjeme XException
}
catch (YException e) {
  // obravnava izjeme YException
}
finally {
  // stavki za varen zaključek dela
}
```

Kodo, pri kateri lahko pride do izjeme, postavimo v `try` blok. Vsako posamezno izjemo obravnava lasten prestreznik `catch`. Pri tem lahko izjeme tudi združujemo in jih več obravnavamo v istem prestrezniku. Ker so vse izjeme izpeljane iz razreda `Exception`, bi univerzalni prestreznik za poljubne izjeme lahko zapisali takole:

```
catch (Exception e) {
  // obravnava vseh izjem
}
```

Ena od možnih obravnav izjem je izpis podrobnega sporočila izjeme (metoda `getMessage()`) skupaj z izpisom klicnega sklada (metoda `printStackTrace()`):

```
catch (Exception e) {
  System.out.println(e.getMessage());
  e.printStackTrace();
}
```

Stavki znotraj bloka `finally` se izvedejo v vsakem primeru, če je do izjeme prišlo ali tudi če do nje ni prišlo. Izvedejo se tudi, če pride do izjeme v bloku `catch`. Blok `finally` je opcijski, zato ni nujno, da ga podamo.

Tak način obravnave izjem, kjer pišemo ločeno kodo za njihovo prestrezanje, prispeva k jasnosti kode in posredno tudi k manj napakam v programih.

Tok bajtov

Za prenašanje binarne vsebine (slike, zvok, zaporedna predstavitev objektov) uporabljamo razreda `InputStream` (za branje) in `OutputStream` (za pisanje) ter njune potomce, saj z njimi obdelujemo posamezne bajte.

Razred `FileInputStream` je namenjen branju binarnih podatkov (bajtov) iz datotek v datotečnem sistemu. Podobno lahko za pisanje binarnih podatkov v datoteke uporabimo razred `FileOutputStream`. Oba razreda imata konstruktor, ki kot argument prejme niz znakov, ki določa ime datoteke. Nov tok za branje lahko torej odpremo s klicem konstruktorja:

```
InputStream vhodniTok = new FileInputStream(ime);
```

Za ustvarjanje toka za pisanje spet kličemo ustrezen konstruktor:

```
OutputStream izhodniTok = new FileOutputStream(drugoIme);
```

Oba razreda pa imata tudi konstruktor, ki kot argument prejme objekt `File`, ki predstavlja datoteko. Z uporabo tega konstruktorja lahko nov tok za branje iz datoteke odpremo v dveh korakih. Najprej ustvarimo nov objekt, ki predstavlja našo datoteko, nato pa s pomočjo tega objekta ustvarimo še vhodni tok:

```
File datoteka = new File(ime);
InputStream vhodniTok = new FileInputStream(datoteka);
```

Za branje enega bajta iz datoteke uporabimo metodo `read()`, ki vrne prebrani bajt oziroma vrednost `-1`, če smo že prišli do konca datoteke. Branje datoteke tako izvajamo v zanki, dokler je prebrana vrednost različna od `-1`:

```
int bajt;
while((bajt = vhodniTok.read()) != -1)
  ...
```

Za zapisovanje enega bajta v datoteko pa uporabimo metodo `write()`:

```
izhodniTok.write(bajt);
```

Po zaključku dela z datotekami moramo datoteke tudi zapreti, za kar uporabimo metodo `close()`:

```
vhodniTok.close();
izhodniTok.close();
```

Seveda pri delu s tokovi ne smemo pozabiti napovedati uporabe paketa z ustreznimi razredi:

```
import java.io.*;
```

Pri delu z datotekami lahko pride tudi do nepredvidenih dogodkov, povezanih predvsem z V/I operacijami (torej do vhodno/izhodnih izjem). V JDK dokumentaciji so pri opisih posameznih metod navedene tudi vrste izjem, ki jih posamezna metoda razglaša (to pomeni, da ta metoda lahko sproži ali posreduje to vrsto izjeme). Tako na primer pri klicu konstruktorjev `FileInputStream(ime)` ali `FileOutputStream(ime)` lahko pride do izjeme `FileNotFoundException` (ki je posebna vrsta izjeme `IOException`), pri branju iz datoteke ali pisanju v datoteko pa se lahko sproži izjema `IOException`. Slednja je možna tudi pri zapiranju datoteke.

Seveda moramo v dobro napisanem programu poskrbeti tudi za ustrezno obravnavo možnih izjem.

Vse navedeno lahko povzamemo na enostavnem primeru, kjer vsebino poljubne datoteke prepišemo v drugo datoteko (imeni obeh datotek podamo kot argumenta programa). Ker gre za poljubno binarno datoteko, bo prepisovanje potekalo po bajtih.

Pri obravnavi izjem (možne so različne izjeme, a so vse vrste `IOException`) smo se zaradi enostavnosti kode odločili, da v programu izjem ne bomo obravnavali, ampak jih bomo posredovali v izvajalno okolje. Zato v `main` metodi razglasimo izjemo `IOException`:

```
public static void main(String[] args) throws IOException
```

Izvorna koda tega programa je v datoteki `Prepisi.java`.

Če bi želeli v našem programu obravnavati tudi izjeme, do katerih lahko pride pri odpiranju datotek ali pri branju oziroma pisanju, bi morali ustrezne kritične dele

postaviti v blok `try` in ustrezne izjeme ujeti v bloku `catch`. V blok `finally` bi postavili zapiranje obeh datotek, saj naj bi program kljub izjemi poskrbel za sprostitev virov. Še vedno pa mora `main` metoda razglasiti izjemo, saj v primeru, ko do nje pride pri zapiranju datotek (v bloku `finally`), te ne ulovimo, temveč jo le posredujemo naprej po klicnem skladu.

Tako dopolnjen program smo zapisali v datoteko `Prepisi1.java`. Ima nekoliko več vrstic, a koda še vedno ostaja dovolj pregledna.

Tok znakov

Razreda `Reader` (za branje) in `Writer` (za pisanje) ter njune potomce uporabljamo za prenašanje besedila, saj z njimi lahko prenašamo posamezne znake. Neposredno iz razreda `Reader` je izpeljan razred `InputStreamReader`, ki je neke vrste most med bitnim tokom in znakovnim tokom (bere bajte in jih dekodira v znake skladno s kodno tabelo). Iz njega pa je izpeljan razred `FileReader`, ki ga lahko uporabimo za branje znakovnih datotek. Podobno tudi za pisanje znakovnih datotek lahko uporabimo razred `FileWriter`, ki je preko razreda `OutputStreamWriter` izpeljan iz razreda `Writer`.

Za branje znakovnih datotek tako navadno uporabljamo razred `FileReader`, razred `FileWriter` pa za pisanje znakovnih datotek. Ustrezen tok odpremo s klicem konstruktorja, ki mu podamo ime datoteke:

```
FileReader vhodniTok = new FileReader(ime);
FileWriter izhodniTok = new FileWriter(ime);
```

Spremenimo naš prejšnji program tako, da bo prepisoval datoteko po znakih. Zato odpremo primeren vhodni in izhodni tok, medtem ko sta preobloženi metodi za branje in pisanje podobni (`read` in `write`, po vrsti). Kodo spremenjenega programa najdete v datoteki `PrepisiZnake.java`.

Ovijanje tokov

V prejšnjih razdelkih smo spoznali, kako beremo iz datotečnega toka temeljne informacije, to je bajte oziroma znake. Velikokrat pa bi želeli iz datoteke prebrati druge tipe podatkov, kot so na primer cela ali realna števila. V takih primerih uporabimo druge razrede, ki jih nudi paket `java.io`, in jih po potrebi povežemo z datotečnimi tokovi. Takemu sestavljanju tokov pravimo tudi ovijanje tokov.

Tokove ovijamo tako, da konstruktorju novega toka podamo tok, ki ga želimo oviti. Tako na primer tok `FileInputStream` bere binarne podatke iz datoteke. Če želimo iz datoteke prebirati realna števila, bomo obstoječi datotečni tok ovili z razredom `DataInputStream`, ki zna prebrane bajte združiti v primitivni tip `double` (realna števila):

```
FileInputStream tok = new FileInputStream(ime);
DataInputStream podatki = new DataInputStream(tok);
```

Oba stavka lahko združimo v naslednji zapis:

```
DataInputStream podatki =
    new DataInputStream(new FileInputStream(ime));
```

In kako najlažje poiščemo ustrezen razred za ovijanje? Poiščemo tisti razred, ki nam nudi ustrezne metode za branje želenih podatkov. V našem primeru je to metoda `readDouble()`, ki jo najdemo v razredu `DataInputStream`:

```
double stevilo = podatki.readDouble();
```

Seveda lahko tokove po potrebi poljubno ovijamo. Tako bi lahko pri branju iz datoteke uporabili še medpomnenje (*buffering*), kar omogoča razred `BufferedInputStream`:

```
DataInputStream podatki = new DataInputStream(
    new BufferedInputStream(
        new FileInputStream(ime)));
```

Vedno pa velja, da je zadnji tok v verigi ovijanja tisti, katerega metode nas zanimajo.

Pa si poglejmo ovijanje tokov še na enem primeru. Spremenimo naš prejšnji program tako, da bo datoteko prepisoval po vrsticah. Iščemo torej razreda, katerih metode omogočajo branje cele vrstice naenkrat in tudi zapis tako prebrane vrstice. Datotečni tok, ki bere znake, lahko ovijemo z razredom `BufferedReader`, ki omogoča branje preko medpomnilnika. Metodo `readLine()` za branje cele vrstice znakov najdemo med metodami tega razreda. Bodite pozorni, da metoda vrne vsebino vrstice kot niz znakov (brez znaka za novo vrstico) oziroma vrednost `null`, ko doseže konec toka.

```
FileReader tok = new FileReader(ime);
BufferedReader podatki = new BufferedReader(tok);

String niz = podatki.readLine();
```

Podoben razred poiščemo tudi za pisanje:

```
FileWriter tok = new FileWriter(ime);
BufferedWriter podatki = new BufferedWriter(tok);
```

Ta razred nudi tudi metodi `write()` za zapis niza znakov in `newLine()` za zapis znaka za novo vrstico.

```
podatki.write(niz);
podatki.newLine();
```

Koda tega primera je v datoteki `PrepisiVrstice.java`.

Datoteke z naključnim dostopom

Pri naključnem dostopu do datotek lahko v datoteko pišemo ali pa iz nje beremo vsebino. Naključni dostop v Javi omogoča razred `RandomAccessFile`, ki

implementira vmesnika `DataInput` (vmesnik omogoča branje bajtov iz binarnega toka in njihovo sestavljanje v katerikoli javanski primitivni tip) in `DataOutput` (pretvori javanski primitivni tip v zaporedje bajtov in jih zapiše v binarni tok).

Način dostopa do datoteke podamo kot argument konstruktorja. Tako lahko datoteko `dat.dat` odpremo za branje (`r`) ali pa za branje in pisanje (`rw`) z naslednjima klicema konstruktorja:

```
RandomAccessFile b = new RandomAccessFile("dat.dat","r");
RandomAccessFile bp = new RandomAccessFile("dat.dat","rw");
```

Če odpiramo datoteko za branje in pisanje in če ta datoteka še ne obstaja, se ustvari nova datoteka.

V tako odprti datoteki se lahko s pomočjo metode `seek()` postavimo na poljuben položaj v datoteki, trenutni položaj datotečnega kazalca pa vrne metoda `getFilePointer()`. Z metodo `length()`, ki vrne dolžino datoteke, lahko preverimo, ali je v datoteki že kaj zapisano.

Branje podatkov lahko poteka preko metode `read()`, ki prebere en bajt, ali pa uporabimo metode, ki sestavijo prebrane bajte v podatke primitivnih tipov, kot so `readDouble()`, `readInt()`, `readChar()`, ali v celo vrstico (metoda `readLine()`).

Podobno lahko za pisanje podatkov uporabimo metodo `write()`, ki zapiše en bajt, ali pa višjenivojske metode, kot so `writeDouble()`, `writeInt()`, `writeChar()` ali `writeChars()`.

Podrobnejši opis metod najdete v JDK dokumentaciji.

Za primer dela z datoteko z naključnim dostopom si poglejmo program v datoteki `NakljucenDostop.java`. Program odpre datoteko, ki je podana kot argument, za hkratno branje in pisanje (če ime ni podano, vzame privzeto ime `datoteka.dat`). Najprej preveri dolžino datoteke in izpiše opozorilo o prepisu datoteke, če datoteka ni prazna. Sledi zapis naključnega števila celih števil od 0 naprej v datoteko, nato pa naključno izbere neko pozicijo v datoteki in se postavi nanjo. Prebere število, ki je na izbrani poziciji, in ga izpiše skupaj s trenutno pozicijo datotečnega kazalca (pred branjem in po branju). Na koncu se postavi spet na začetek datoteke (klic metode `datoteka.seek(0)`) in po vrsti bere in izpisuje na zaslon vsa števila iz datoteke.

Za branje števil iz datoteke uporabljamo metodo `readInt()`, ki prebere predznačeno 32-bitno celo število. Metoda torej prebere naslednje štiri bajte iz datoteke in jih tolmači kot tip `int`. V primeru, ko je datoteke konec, preden uspe prebrati vse štiri bajte, metoda sproži izjemo `EOFException`, ki je vrsta `IOException`. Zato moramo branje iz datoteke postaviti v `try/catch` blok, v katerem potem ujamemo izjemo `EOFException`.

V datoteki zapisana cela števila so zapisana po bajtih, kjer vsako število zavzame štiri bajte. To moramo upoštevati pri nastavljanju datotečnega kazalca, saj se le-ta za vsako naslednje število poveča za vrednost štiri. Prvo število je zapisano na lokaciji `0`. Tako v

primeru, da želimo prebrati `n`-to število po vrsti, nastavimo vrednost datotečnega kazalca na `(n-1)*4`:

```
datoteka.seek((n-1)*4);
```

Bodite pozorni na delovanje programa pri datotekah, ki že imajo zapisano neko vsebino. Pri pisanju števil v datoteko se namreč datoteka (delno) prepiše. Pri tem lahko v njej poleg novo vpisanih števil ostanejo tudi stari podatki. Vsi ti podatki se pri izpisu datoteke interpretirajo kot cela števila, zato lahko v tem primeru dobimo zelo nenavaden in nepričakovan izpis vsebine datoteke.

Pisanje in branje objektov

Java omogoča tudi zapisovanje in branje objektov (v prejšnjih razdelkih smo brali/pisali le primitivne podatkovne tipe). Temu sta namenjena tokova `ObjectOutputStream` in `ObjectInputStream`, ki vsak objekt spremenita v zaporedje bajtov, tako da ga lahko zapišemo v poljuben tok in ga kasneje znamo tudi prebrati. Ta postopek pretvorbe objekta v bajte imenujemo serializacija objektov (*object serialization*).

Nov tok za zapisovanje objektov ustvarimo s klicem konstruktorja, kateremu podamo izhodni tok, v katerega želimo zapisovati:

```
OutputStream tok = new FileOutputStream(ime);
ObjectOutputStream objTok = new ObjectOutputStream(tok);
```

Metoda `writeObject()` poskrbi za zapis objekta, ki ji ga podamo kot argument:

```
objTok.writeObject(objekt);
```

Nov objekt tipa `Date` bi na primer lahko zapisali takole:

```
objTok.writeObject(new Date());
```

Ker metoda `writeObject()` objekt najprej pretvori v zaporedje bajtov, morajo biti ti objekti take vrste, da omogočajo serializacijo (vsi objekti niso taki). To pomeni, da morajo ti objekti izdelati vmesnik `Serializable` (vmesnik nima nobenih atributov ali metod ter služi le za označitev primernosti za serializacijo). Če temu ni tako, metoda sproži izjemo `NotSerializableException`. Izjema `InvalidClassException` pa se sproži v primeru napake razreda. Obe izjemi sta potomca bolj splošne izjeme `IOException`.

Na koncu zapisovanja, preden tok zapremo, moramo poklicati še metodo `flush()`, ki poskrbi za splakovanje toka (vsi bajti iz medpomnilnika se zapišejo v tok). Pisanje torej zaključimo z:

```
objTok.flush();
objTok.close();
```

Za branje in obnovo objekta iz toka poskrbi metoda `readObject()` razreda `ObjectInputStream`. Seveda moramo najprej tok za branje objektov odpreti s klicem konstruktorja, ki mu podamo vhodni tok, iz katerega želimo brati:

```
InputStream tok = new FileInputStream(ime);
ObjectInputStream objTok = new ObjectInputStream(tok);

Date d = (Date) objTok.readObject();
```

Ker metoda `readObject()` bere poljubne objekte, vrača tip `Object`. Zato moramo prebranemu objektu nastaviti ustrezen tip s prirejanjem.

Za demonstracijo zapisovanja in branja objektov v oziroma iz datoteke napišimo lasten razred, katerega izvode bomo zapisali v datoteko in jih nato tudi prebrali iz datoteke.

Naj naš nov razred opisuje naše prijatelje. Razred `Prijatelj` je povsem preprost, vsebuje nekaj atributov, dva konstruktorja in metodo za pretvorbo v niz (ta zajema vrednosti vseh atributov):

```
public class Prijatelj {
  private String priimek;
  private String ime;
  private int starost;
  private String telefon;

  public Prijatelj() {
    this("", "", 0, "");
  }

  public Prijatelj(String i, String p, int s, String t) {
    this.priimek = p;
    this.ime = i;
    this.starost = s;
    this.telefon = t;
  }

  public String toString() {
    return "Prijatelj: " + this.ime + " " + this.priimek +
           ", " + this.starost + " let, tel.: " +
           this.telefon;
  }
}
```

Razredu lahko dodamo še metodi za branje in zapis objekta. Za zapis objekta v datoteko bomo napisali metodo `pisiDat`, ki prejme želeni izhodni tok kot argument:

```
public void pisiDat(FileOutputStream tok)
    throws IOException {
  ObjectOutputStream objIzhTok =
      new ObjectOutputStream(tok);
  objIzhTok.writeObject(this);
  objIzhTok.flush();
}
```

Ker pri zapisovanju lahko pride do proženja izjem, mora metoda razglasiti izjemo. V telesu metode najprej odpremo tok za zapis objekta, poskrbimo za zapis tega objekta (določilo `this` se nanaša na objekt, ki ga zapisujemo) ter splaknemo tok.

Zapisani objekt lahko preberemo z metodo `beriDat`, ki prejme kot argument vhodni tok. Tudi tu najprej ustvarimo tok za branje objektov in iz njega preberemo en objekt:

```
public void beriDat(FileInputStream tok)
    throws IOException, ClassNotFoundException {

  ObjectInputStream objVhTok = new ObjectInputStream(tok);
  Prijatelj stari = (Prijatelj) objVhTok.readObject();

  this.priimek = stari.priimek;
  this.ime = stari.ime;
  this.starost = stari.starost;
  this.telefon = stari.telefon;
}
```

Tudi v metodi `beriDat()` vse morebitne izjeme le posredujemo naprej.

Prebranemu objektu priredimo tip `Prijatelj` ter nastavimo vrednosti atributov objekta (nanj se nanaša določilo `this`) na vrednosti atributov prebranega objekta.

Da lahko objekte pišemo oziroma beremo, jih moramo serializirati, zato mora naš razred implementirati vmesnik `Serializable`:

```
public class Prijatelj implements Serializable
```

In kako lahko uporabljamo obe metodi razreda? Recimo, da imamo podatke o svojih prijateljih shranjene v polju prijatelji:

```
Prijatelj[] prijatelji = new Prijatelj[MAX];
```

Potem lahko vse podatke o prijateljih zapišemo v datoteko v zanki, ki gre preko vseh elementov polja in vsakega zapiše v datoteko. Seveda moramo najprej odpreti ustrezno datoteko za zapis podatkov in jo na koncu tudi zapreti. Vse stavke postavimo tudi v `try`/`catch` blok, da prestrežemo sprožene izjeme:

```
try {
  FileOutputStream izhTok = new FileOutputStream(imeDat);
  for(int i=0; i<MAX; i++)
    prijatelji[i].pisiDat(izhTok);
  izhTok.close();
}
catch(IOException e) {
  System.out.println("Napaka: " + e.getMessage());
}
```

Pri branju postopamo podobno, le da tokrat odpremo datoteko za branje podatkov:

```
try {
  FileInputStream vhTok = new FileInputStream(imeDat);
  i = 0;
  while(i < MAX) {
    prijatelji[i] = new Prijatelj();
    prijatelji[i++].beriDat(vhTok);
  }
  vhTok.close();
}
catch(IOException e) {
  System.out.println("Napaka: " + e.getMessage());
}
```

V datoteki `Objekti.java` je izvorna koda programa, ki prikazuje delo z objekti. V tem primeru ustvarimo polje petih objektov razreda `Prijatelj` in v datoteko zapišemo vseh pet objektov. Nato iz datoteke preberemo vse zapisane objekte in jih izpišemo na zaslon.

Standardni tokovi

Java podpira tri standardne tokove: standardni vhod, do katerega dostopamo preko objekta `System.in`, standardni izhod, do katerega dostopamo preko objekta `System.out`, ter standardni izhod za napake, dostop do katerega omogoča objekt `System.err`. Vsi trije tokovi se ustvarijo samodejno in jih ni potrebno posebej odpirati. Standardni tokovi privzeto berejo s tipkovnice (vhod) ter izpisujejo na zaslon (izhod).

Standardni tokovi v Javi so binarni tokovi (razlog je zgodovinski, saj Java 1.0 ni poznala znakovnih tokov). `System.out` in `System.err` sta definirana kot objekta razreda `PrintStream` (ta razred sicer predstavlja binarni tok, a dobro podpira delo z znaki), medtem ko je `System.in` objekt razreda `InputStream` (osnovni binarni vhodni tok). Če želimo standardni vhod obravnavati kot znakovni tok, ga moramo oviti z `InputStreamReader`:

```
InputStreamReader vhTok = new InputStreamReader(System.in);
```

Še bolj učinkovito in uporabno pa je, da ta vhodni tok ovijemo še z `BufferedReader`, ki omogoča branje preko medpomnilnika. Tako lahko vsebino na vhodu tudi popravljamo in šele ob pritisku na tipko Enter se le-ta dejansko prebere (branje vrstic).

```
BufferedReader vhod = new BufferedReader(
                          new InputStreamReader(System.in));
```

Za primer si poglejmo kratek program, ki bere standardni vhod ter ga zapisuje na standardni izhod. Program smo poimenovali `Echo`, saj kot odmev zapiše vsako prebrano vrstico. V spodnji kodi spremenljivke `izhod` niti ne potrebujemo, saj lahko na njenem mestu uporabljamo kar `System.out`. Vendar smo jo zaradi prikaza bolj splošne uporabe izhodnega toka vseeno ohranili (spremenljivka `izhod` bi namreč lahko predstavljana poljuben izhodni tok).

```
import java.io.*;

public class Echo {
  public static void main(String[] args)
                       throws IOException {

    BufferedReader vhod =
      new BufferedReader(new InputStreamReader(System.in));
    PrintStream izhod = System.out;
    String vrstica;
    while((vrstica = vhod.readLine()) != null)
      izhod.println(vrstica);
  }
}
```

Ko program zaženemo, čaka na naš vhod. Preko tipkovnice vtipkamo vrstico in jo zaključimo z Enter. Vnos vrstic zaključimo s kombinacijo tipk `Ctrl+z`, ki pomeni konec datoteke (EOF).

Formatirano branje in pisanje v praksi: enostavno delo z datotekami

V predhodnih razdelkih smo si ogledali različne primere uporabe tokov in dela z datotekami v Javi. Tokovi so sicer zelo močno orodje, a hkrati tudi precej zahtevni za uporabo. Njihova prilagodljivost in moč nam nista vedno v pomoč, saj bi velikokrat raje dali prednost enostavnejši uporabi. V praksi namreč pogosto potrebujemo enostavno branje iz tekstovne datoteke, ki omogoča branje nizov in drugih primitivnih tipov podatkov, ter enostaven zapis teh podatkov v neko tekstovno datoteko.

Omenjeni možnosti nam nudita razreda `Scanner` in `PrintWriter`, ki omogočata formatirano branje oz. zapis. Oba razreda si bomo pogledali v nadaljevanju. Na koncu razdelka pa smo pripravili še en primer uporabe obeh razredov.

Razred `Scanner`

Razred `Scanner` (najdemo ga v paketu `java.util`) predstavlja preprost bralnik besedila, ki besedilo razčleni na posamezne dele s pomočjo regularnih izrazov ter zna iz njega izluščiti primitivne podatkovne tipe in nize znakov. Bralnik razbije vhod na posamezne elemente z uporabo ločitvenega vzorca (privzeto je to prazen prostor ali *whitespace*), kateri je podan z regularnim izrazom.

Pred prvo uporabo moramo bralnik odpreti, za kar poskrbi klic konstruktorja. Konstruktorju moramo kot parameter podati vir podatkov za branje, ki je lahko datoteka (objekt tipa `File`), tok (objekt tipa `InputStream`) ali pa poljuben niz (objekt tipa `String`). V zadnjem primeru niz ne določa imena datoteke, kot je v navadi pri klicih konstruktorjev tokov, temveč vir podatkov za bralnik (bralnik bere elemente iz podanega niza).

Bralnik, ki bere iz standardnega vhoda, tako lahko odpremo s stavkom:

```
Scanner bralnik = new Scanner(System.in);
```

Če želimo brati iz datoteke, uporabimo drug konstruktor:

```
Scanner bralnik = new Scanner(new File("ime.dat"));
```

Po končani uporabi bralnik zapremo s klicem metode `close()`.

Bralnik lahko pri branju elementov le-te tudi pretvarja v različne tipe, za kar uporabimo različne metode za branje. Poljuben naslednji element prebere metoda `next()`, ki vrne prebran niz znakov. Metoda `nextLine()` prebere celo vrstico (vse znake do konca vrstice), medtem ko za branje števil in drugih enostavnih podatkovnih tipov uporabljamo metode `nextInt()`, `nextDouble()`, `nextBoolean()` ...

S pomočjo metode `hasNext()` preverimo, ali je na vhodu še kakšen element. Če nas zanimajo elementi točno določenega tipa, uporabimo sorodne metode `hasNextInt()`, `hasNextDouble()`, `hasNextBoolean()`, `hasNextLine()` ...

```
String s;
if(bralnik.hasNext())
  s = bralnik.next();

int n;
while(bralnik.hasNextInt())
  n = bralnik.nextInt();
```

Kot smo že omenili, je privzeto ločilo med posameznimi elementi vhoda prazen prostor (*whitespace*), ki zajema znak za presledek (`' '`), tabulator (`'\t'`), novo vrstico (`'\n'`), return (`'\r'`) ter prehod na novo stran (`'\f'`). Trenutno vrednost ločitvenega vzorca izvemo s klicem ustrezne metode:

```
Pattern vzorec = bralnik.delimiter();
```

Seveda lahko ločitveni vzorec (in s tem ločilo med elementi na vhodu) poljubno nastavimo na vrednost, ki ustreza našim zahtevam:

```
String vzorec = ":";
bralnik.useDelimiter(vzorec);
```

Niz `vzorec` tukaj določa ločitveni vzorec (v našem primeru je to znak `:`), ki je v splošnem podan z regularnim izrazom. Podrobnosti o sestavljanju vzorca so opisane v JDK dokumentaciji pri opisu razreda `Pattern`. Tukaj bomo navedli le nekaj izmed možnosti:

- posamezen znak ali niz znakov (npr. `":"` ali `"xyz"`),
- ali znak a ali znak b (`"a|b"`),
- prazen prostor (`"\\s"`),
- znak za novo vrstico (`"\\n"`),
- tabulator (`"\\t"`),

- črka ali številka (`"\\w"`),
- katerikoli znak, razen črke ali številke (`"[^\\w]"`),
- katerikoli znak, razen črke ali številke, ki se pojavi vsaj enkrat (`"[^\\w]+"`) ...

Če želimo ločitveni vzorec ponastaviti nazaj na privzeto vrednost, to za nas opravi metoda `reset()`.

Razred `PrintWriter`

Razred `PrintWriter` (najdemo ga v paketu `java.io`) omogoča zapisovanje formatirane predstavitve objekta v tekstovni izhodni tok. Razred implementira vse metode za izpis (`print`, `println`, `printf`) razreda `PrintStream`, ki jih znamo uporabljati že od izpisovanja na standardni izhod preko `System.out`.

Izhodni tok odpremo s pomočjo konstruktorja, ki mu lahko podamo različne argumente, objekte tipa `File`, `OutputStream`, `Writer` ali `String` (v tem primeru niz predstavlja ime datoteke). Pri tem lahko vključimo tudi samodejno izpiranje toka (*line flushing*). Po končani uporabi tok zapremo s klicem metode `close()`.

Primer uporabe obeh razredov

Pa si poglejmo še primer uporabe razredov `Scanner` in `PrintWriter`. Recimo, da imamo v tekstovni datoteki `studenti.txt` zapisane rezultate zadnjega izpitnega roka pri programiranju. V njej so v vsaki vrstici zapisani podatki enega študenta, in sicer vpisna številka, ime (eno ali več), priimek (eden ali več), število doseženih točk na vajah ter na koncu še število točk na pisnem izpitu. Vsi navedeni podatki so ločeni s podpičjem. Primer datoteke je naslednji:

```
63070279;Frodo;Baggins;30;70
63070264;Samwise;Gamgee;25;60
63060043;Meriadoc Merry;Brandybuck;23;67
63060053;Aragorn;Rightful king of Arnor and Gondor;15;58
63070284;Peregrin Pip/Pippin;Took;5;10
63060047;Sauron;Dark Lord;16;36
```

Napišimo program, ki prebere podatke o ocenah študentov ter v novo datoteko, katere ime je podano kot argument ukazne vrstice, zapiše le vpisne številke študentov in njihovo skupno število točk pri predmetu. Tudi v tej datoteki naj bo vsak študent zapisan v svoji vrstici, med vpisno številko in številom točk pa naj bo en presledek. Izhodna datoteka programa za zgornjo vhodno datoteko bi bila naslednja:

```
63070279 100
63070264 85
63060043 90
63060053 73
63070284 15
63060047 52
```

Za branje podatkov o študentih uporabimo bralnik, kateremu nastavimo ločilo med posameznimi elementi na znak podpičje (ločilo med posameznimi podatki enega študenta) ali novo vrstico (ločilo med podatki različnih študentov):

```
Scanner bralnik = new Scanner(new File("studenti.txt"));
bralnik.useDelimiter(";|\\n|\\r\\n");
```

Kot znak za novo vrstico smo upoštevali tako znak `'\n'` (Linux sistemi) kot tudi kombinacijo znakov `'\r'` in `'\n'` (Windows sistemi).

Za zapis podatkov v izhodno datoteko odpremo ustrezen izhodni tok, ki je vezan na datoteko, katere ime podamo kot argument programa:

```
PrintWriter izhod = new PrintWriter(args[0]);
```

Ker izjem ne bomo lovili, mora funkcija `main` deklarirati proženje izjem.

```
public static void main(String[] args) throws Exception
```

V zanki nato beremo podatke iz datoteke, dokler imamo še kakšen podatek na vhodu:

```
while(bralnik.hasNext())
```

Ustrezne podatke beremo z `bralnik.next()` za nize (to je za vpisno številko, ime in priimek) oziroma z `bralnik.nextInt()` za cela števila (to so točke vaj in točke izpita):

```
String vpisna = bralnik.next();
bralnik.next(); // ime ignoriramo
bralnik.next(); // priimek ignoriramo
int vaje = bralnik.nextInt();
int izpit = bralnik.nextInt();
```

Prebrane podatke sproti zapisujemo v izhodno datoteko z uporabo metode `printf()`:

```
izhod.printf("%s %d%n", vpisna, vaje+izpit);
```

Bralnik in izhodni tok moramo na koncu še zapreti z ukazom `close()`:

```
bralnik.close();
izhod.close();
```

Kodo tega programa najdete v datoteki `Izpiti.java`.

8. Delo z mrežo

Javanski razredi iz paketa `java.net` ponujajo visokonivojski vmesnik za dostop do TCP/IP omrežij. Z uporabo že pripravljenih razredov lahko enostavno napišemo program za delo z vtičnicami (*sockets*), izdelamo porazdeljen program tipa odjemalec/strežnik (*client/server*), delamo z URL (*Universal Resource Locator*) naslovi, vzpostavimo omrežno povezavo različnih protokolov ali obravnavamo prejeto vsebino. Omogočajo nam tako priklapljanje na obstoječe omrežne storitve kot vzpostavitev lastnega strežnika.

Različne omrežne vire lahko dosegamo preko različnih protokolov. V svetovnem spletu ponavadi uporabljamo `http` (*HyperText Transfer Protocol*) protokol, ki uporablja URL naslove za lociranje podatkov. Vzpostavljene povezave z omrežnimi viri so podobne tokovom, ki smo jih obravnavali v poglavju *Datoteke in tokovi*.

Razred `URL`

Vire v internetu v Javi opisujejo izvodi razreda `URL`. Objekte `URL` ustvarimo s klicem konstruktorjev, katerim lahko podamo različne argumente. Primera:

```
URL vir1 =
     new URL("http://lgm.fri.uni-lj.si/pa/index.html");
URL vir2 =
     new URL("http","lgm.fri.uni-lj.si/pa/","index.html");
```

Prvi konstruktor prejme en sam argument, to je niz znakov, ki opisuje cel URL naslov. Drugi konstruktor pa prejme tri ločene nize, ki skupaj sestavljajo URL naslov, to so protokol, gostitelj (*host*) in ime datoteke.

Če je URL napačno sestavljen, konstruktor sproži izjemo `MalformedURLException`, ki jo moramo ujeti ali pa posredovati naprej.

Ko objekt `URL` enkrat ustvarimo, ga ne moremo več spreminjati. Seveda pa ga vedno lahko zavržemo in ustvarimo novega.

Posamezne elemente URL naslova lahko dobimo preko različnih metod tega razreda. Navedimo le nekatere od njih (vse se nanašajo na URL objekt):

- `getFile()` vrne ime datoteke,
- `getHost()` vrne ime gostitelja,
- `getPath()` vrne tisti del URL naslova, ki določa pot,
- `getPort()` vrne številko vrat (*port number*),
- `getProtocol()` vrne ime protokola,
- `getRef()` vrne sidro (*anchor* ali *reference*).

Seveda vse metode niso primerne za vse vrste URL naslovov (na primer, URL naslov morda sploh ne vsebuje sidra).

Ko imamo vzpostavljeno povezavo z virom (ko smo uspešno ustvarili objekt `URL`), lahko s klicem metode `openStream()` odpremo vhodni tok tipa `InputStream`, ki ga uporabimo za branje vira. Potem postopamo podobno kot pri branju datotek.

Branje datoteke, podane z URL

Za primer si poglejmo program, ki na zaslon izpiše vsebino datoteke, ki je podana z URL naslovom. Datoteko bomo brali (in izpisovali) po vrsticah, zato bomo okoli vhodnega toka `InputStream` ovili najprej `InputStreamReader`, ki omogoča branje po znakih, okoli njega pa še `BufferedReader`, s katerim lahko naenkrat preberemo celo vrstico:

```
URL vir = new URL(naslov);
InputStream tok = vir.openStream();
InputStreamReader vhod = new InputStreamReader(tok);
BufferedReader branje = new BufferedReader(vhod);
```

Za branje vhoda uporabimo kar metodo `readLine()`, ki naenkrat prebere celo vrstico:

```
String vrstica;
while((vrstica = branje.readLine()) != null)
  System.out.println(vrstica);
```

V `while` zanki vsako prebrano vrstico sproti izpišemo. Na koncu vzpostavljen tok tudi zapremo:

```
branje.close();
```

Vzpostavljanje povezave z virom in vhodnega toka ter branje iz vira postavimo v `try/catch` blok, saj pri tem lahko pride tudi do izjeme, ki jo ujamemo in obravnavamo.

Na začetku programa moramo tudi napovedati uporabo obeh paketov (za delo s tokovi in z mrežo):

```
import java.io.*;
import java.net.*;
```

Datoteka `PreberiURL.java` hrani izvorno kodo tega programa.

Pri zagonu programa moramo kot argument podati veljaven URL naslov in na zaslon se izpiše vsebina te lokacije. Primer klica programa:

```
java PreberiURL http://lgm.fri.uni-lj.si/pa/index.html
```

Namesto da vsebino na podanem URL naslovu izpisujemo na zaslon, bi jo lahko enostavno zapisali kar v datoteko na lokalnem disku. Naslednji primer prikazuje program, ki na lokalni disk v tekoči direktorij shrani poljubno datoteko (na primer tekstovno datoteko ali sliko). Datoteka je podana z URL naslovom.

Izhajamo iz prejšnjega primera, kjer smo vzpostavili povezavo z virom in prebrali njegovo vsebino. Ker želimo brati poljubno datoteko (tekstovno ali binarno), bomo okoli vhodnega toka ovili le razred `BufferedInputStream`, ki omogoča branje preko medpomnilnika:

```
URL vir = new URL(naslov);
BufferedInputStream vhod =
    new BufferedInputStream(vir.openStream());
```

Seveda moramo ustvariti še en izhodni tok, ki bo prebrane podatke zapisal v datoteko na disku. `BufferedOutputStream` je primeren ovoj okoli datotečnega izhodnega toka:

```
BufferedOutputStream izhod =
  new BufferedOutputStream(new FileOutputStream(ime));
```

Spremenljivka `ime` določa ime datoteke na disku. Najbolj smiselno je, da se to ime ujema z imenom datoteke spletnega vira. Slednje nam vrne metoda `getFile()`:

```
String ime = izvor.getFile();
```

Ker pa to ime lahko vključuje tudi znak `/` kot ločilo v poti, bomo za ime datoteke upoštevali le tiste znake, ki sledijo zadnji pojavitvi znaka `/` v nizu (če se znak '`/`' v nizu ne pojavi, je rezultat spodnjega stavka kar nespremenjen niz):

```
ime = ime.substring(ime.lastIndexOf('/')+1);
```

Ker gre tokrat za branje in pisanje binarnih datotek, uporabimo metodi `read()` in `write()` za branje oziroma zapis bajtov:

```
int bajt;
while((bajt = vhod.read()) != -1)
  izhod.write(bajt);
```

Na koncu oba toka tudi zapremo:

```
vhod.close();
izhod.close();
```

Izvorna koda tega programa je v datoteki `Shrani.java`.

Prikaz slike, podane z URL

Za konec pa si oglejmo še primer, kako v oknu prikažemo sliko, ki je podana z URL naslovom. Pri izdelavi programa se naslonimo na primer `Slika.java` iz razdelka *Bitne slike* v poglavju *Grafika*. Primer je zelo podoben, le da smo tam datoteko s sliko prebrali z lokalnega diska, ne iz spletnega vira.

Za nalaganje slike tudi tokrat poskrbimo že v konstruktorju razreda (razred smo poimenovali `Prikazi`). Za branje bitne slike uporabimo metodo `getImage()`, ki pa ji

kot argument podamo objekt `URL`, ki predstavlja URL naslov slike za prikaz. Zato najprej ustvarimo `URL` objekt, preko katerega se povežemo na spletni vir:

```
URL url = new URL(naslov);
Image slika = Toolkit.getDefaultToolkit().getImage(url);
```

Metoda vrne objekt `Image`, ki je referenca na bitno sliko v pomnilniku. Pri ustvarjanju objekta `URL` lahko pride do izjeme, če niz `naslov` ne predstavlja poznanega protokola. Zato ta stavek postavimo v `try` blok, izjemo pa prestrežemo in obravnavamo v `catch` bloku.

Vse povedano sestavimo v kodo, ki je zapisana v datoteki `Prikazi.java`.

Ob klicu programa `Prikazi` moramo podati tudi URL datoteke z bitno sliko. Primer klica programa je:

```
java Prikazi http://lgm.fri.uni-lj.si/pa/primeri/slika.jpg
```

Če podana slika obstaja in če jo program uspe prebrati, jo prikaže v oknu na zaslonu.

9. Niti

Java ima vključeno tudi podporo za niti. Nit (*thread*) je tok izvajanja programskih stavkov, ki se odvija znotraj programa hkrati z ostalimi nitmi. Niti so sicer podobne procesom, a se od njih razlikujejo predvsem po načinu delitve virov (kot je na primer pomnilnik). Procesi so navadno medsebojno neodvisni in imajo ločene naslovne prostore, medtem ko niti delijo isti pomnilnik in druge vire.

Večnitnost je model, ki omogoča več nitim, da sobivajo znotraj enega procesa, si delijo njegove vire, a se (lahko) izvajajo neodvisno. Ta mehanizem omogoča hitro izmenjavo posameznih opravil na procesorju, kar daje vtis hkratnega izvajanja opravil.

Vsaka nit ima svojo prioriteto, ki določa, kolikšen delež celotnega procesorskega časa bo dodeljen posamezni niti. Javanski izvajalni sistem vedno uporablja več niti: najmanj eno nit za programski proces in eno nit za smetarja (*garbage collector*), ki je nit z najnižjo prioriteto.

Za delo z nitmi v Javi uporabljamo razred `Thread` oziroma vmesnik `Runnable`. Oba sta v paketu `java.lang`, torej sta na voljo vsem razredom brez posebne napovedi.

Razred `Thread`

Eden od načinov izdelave večnitnega programa je, da izdelamo razred, ki je izpeljan iz razreda `Thread`. Razred lahko zapišemo takole:

```
public class Nit extends Thread {
  Nit() {
    // konstruktor
  }

  public void run() {
    // telo niti
  }
}
```

Metoda `run` je tista, ki določa delovanje niti. Podedujemo jo od razreda `Thread`, a ker želimo sami določiti obnašanje niti, to metodo ponavadi prekrijemo. Novo nit (nov izvod našega razreda `Nit`) ustvarimo s klicem konstruktorja in jo poženemo s klicem metode `start`. S tem nit preide v delujoče stanje; izvajati se začne njena metoda `run`:

```
Nit nitka = new Nit();
nitka.start();
```

Pa si poglejmo izdelavo niti natančneje na primeru. Kot primer vzemimo program, v katerem izdelamo pet izvodov našega razreda (pet niti) in jih poženemo hkrati. Vsak izvod razreda naj v zanki izpisuje svoje ime, ki mu ga podamo kot argument konstruktorju. Tako bomo ob izvajanju programa lahko opazovali, kako se izvajajo posamezne niti.

Kot smo že rekli, naš razred izpeljemo iz razreda `Thread`, ki omogoča večnitnost:

```
public class Niti extends Thread
```

V razredu napišemo tudi konstruktor, ki sicer le pokliče konstruktor razreda `Thread` in hkrati določi tudi ime niti. Vsaka nit ima namreč svoje ime, ki ga vrne metoda `getName()`. Če imena niti ne podamo eksplicitno, dobi nit privzeto ime "Thread-*x*", kjer je *x* zaporedna številka ustvarjene niti.

```
Niti(String ime) {
  super(ime);
}
```

Eno nit lahko potem ustvarimo s klicem konstruktorja, ki mu kot argument podamo ime te niti:

```
Niti nit = new Niti("Nova Nitka");
```

Nit nato poženemo s klicem metode `start`, ki prične izvajanje te niti (metodo `start` smo podedovali od razreda `Thread`):

```
nit.start();
```

Metoda poskrbi za vzpostavitev nove niti in pokliče metodo `run` te niti. Ko se metoda `run` konča, nit umre. Ko umrejo vse niti, se zaključi tudi izvajanje programa.

Metoda `run`, ki jo vsebuje razred `Thread`, določa izvajanje niti. V izpeljanem razredu prekrijemo to metodo in s tem določimo, kaj naj nit dela. Metodo `run` bi lahko zapisali takole:

```
public void run() {
  for(int i=0; i<20; i++) {
    System.out.print(getName());
  }
  System.out.println("Konec niti: " + getName());
}
```

V zanki, ki se dvajsetkrat ponovi, izpisujemo ime niti ter na koncu izpišemo tudi, da se je nit končala.

Da se posamezne niti ne izvajajo prehitro, bomo po vsakem izpisu imena niti v metodi `run` dodali še začasno zaustavitev niti. S klicem metode `sleep(n)` lahko nit spravimo k počitku za `n` milisekund (začasno zaustavimo izvajanje te niti). Dolžina zaustavitve, ki jo določa atribut `pocakaj`, naj bo naključna (med 0 in 5 sekundami) in jo nastavimo že v konstruktorju.

Ob klicu metode `sleep` moramo ujeti izjemo `InterruptedException`, ki jo metoda sproži v primeru, ko jo prekine druga nit. Ker je tak dogodek pričakovan, bomo izjemo le prestregli, ukrepali pa ne bomo.

Pet izvodov tega razreda naredimo kar v `main` metodi, ki jo lahko vključimo v razred `Niti`. V njej ustvarimo tabelo petih niti (ime vsake niti je kar zaporedna številka te niti) ter vse ustvarjene niti tudi poženemo.

Cel program, katerega koda je zajeta v datoteki `Niti.java`, je torej naslednji:

```
public class Niti extends Thread {
  private int pocakaj;

  public Niti(String ime) {
    super(ime);
    pocakaj = (int) (Math.random()*5000);
    System.out.println("Ime niti: " + getName() +
                       ", cas pocivanja: " + pocakaj);
  }

  public void run() {
    for(int i=0; i<20; i++) {
      System.out.print(getName());
      try {
        sleep(pocakaj);
      }
      catch (InterruptedException e) {
      }
    }
    System.out.println("Konec niti: " + getName());
  }

  public static void main(String[] args) {
    final int MAX = 5;

    Niti[] niti = new Niti[MAX];
    for(int i=0; i<MAX; i++)
      niti[i] = new Niti(" " + i + " ");
    System.out.println("Zagon vseh niti ...");
    for(int i=0; i<MAX; i++)
      niti[i].start();
  }
}
```

Ob izvajanju programa vidimo, da je rezultat odvisen tako od posameznih časov počivanja niti kot od zaporedja izvajanja niti (slednje najlažje preverimo, če nastavimo čas počivanja vseh niti na isto vrednost). Ker pa se niti izvajajo neodvisno, lahko dobimo ob ponovnem zagonu programa povsem drugačno zaporedje izpisov.

Vmesnik `Runnable`

Drug način izdelave niti pa je, da razred implementira vmesnik `Runnable`. Vmesnik ponuja le metodo `run`, ki jo izdelamo kot telo niti, to metodo pa mora implementirati tudi naš razred, ki ta vmesnik izdela. Ta način je uporaben predvsem v primeru, ko je naš razred že izpeljan iz nekega drugega razreda in ga zato ne moremo izpeljati še iz razreda `Thread` (Java ne pozna večkratnega dedovanja).

```
public class Nit implements Runnable {

  public void run() {
    // tu zapišemo, kaj naj nit dela
    ...
  }

}
```

Za delovanje same niti pa v našem razredu še vedno potrebujemo objekt razreda `Thread`. Ponavadi je to kar privatni atribut razreda, lahko pa ga podamo kot spremenljivko v metodi `main`:

```
public static void main (String[] args) {
  Thread nitka = new Thread(new Nit());
  nitka.start();
}
```

Ob ustvarjanju nove niti smo konstruktorju razreda `Thread` podali en argument, to je ime objekta, katerega `run` metodo naj ta nit uporabi za svoje telo. Seveda moramo zatem nit tudi zagnati.

Če objekt razreda `Thread` v razred vključimo kot privatni atribut, ga lahko deklariramo na naslednji način:

```
private Thread nit;
```

Razredu potem dodamo tudi metodo `pozeni`, ki poskrbi za ustvarjanje in zagon niti, če ta še ne obstaja:

```
public void pozeni() {
  if(nit == null) {
    nit = new Thread(this, ime);
    nit.start();
  }
}
```

Če nit še ne obstaja (spremenljivka `nit` je enaka `null`), ustvarimo novo nit in ji določimo, čigavo metodo `run` naj uporabi kot svoje telo (`this`, torej tega razreda) in podamo ime niti (`ime`). Potem ustvarjeno nit tudi zaženemo.

Tako bi naš prejšnji program lahko zapisali tudi na drugačen način, ki ga prikazuje program `Niti1.java`. Delovanje obeh programov je identično.

Stanja niti, prioriteta

Življenje niti sestavljajo štiri stanja, nekatera med njimi se lahko tudi ponavljajo, med stanji pa prehajamo s pomočjo metod. Nit se tako vedno nahaja v enem izmed naslednjih stanj:

- ustvarjena nova nit (*new thread*),
- aktivno stanje (*runnable*),
- zaustavljeno izvajanje (*not runnable*, *blocked*),
- končano izvajanje, nit umre (*dead*).

Prehode med posameznimi stanji omogočajo različne metode:

- `start()`: prične izvajanje niti in jo tako spravi v aktivno stanje (kliče se njena metoda `run`),
- `sleep()`: začasno zaustavi izvajanje niti,
- `wait()`: blokira izvajanje niti, dokler ni izpolnjen določen pogoj,
- `notify()`: nadaljuje izvajanje niti, ko je izpolnjen določen pogoj,
- `yield()`: prepusti procesorski čas drugim nitim.

Vsaka nit ima tudi svojo prioriteto, ki pomeni njeno razvrstitev glede na prednostni dostop do virov operacijskega sistema (kot je na primer procesorski čas). Če niti prioritete ne podamo posebej, prevzame prioriteto svojega starša. Prioriteta je predstavljena s celim številom med 1 (najmanjša, konstanta `Thread.MIN_PRIORITY`) in 10 (največja, konstanta `Thread.MAX_PRIORITY`), privzeta pa je 5 (normalna prioriteta, konstanta `Thread.NORM_PRIORITY`).

Prioriteto niti lahko izvemo s klicem metode `getPriority()`, nastavimo oziroma spremenimo pa jo z metodo `setPriority()`.

Uporabo prioritete niti si poglejmo še na primeru. Program `Niti.java` spremenimo tako, da bo vsaka od niti začasno zaustavljena za enak, vnaprej določen čas (pol sekunde).

```
pocakaj = 500;
```

Vsaki niti nastavimo tudi prioriteto in sicer na naključno vrednost med 1 in najvišjo prioriteto.

```
setPriority((int) (Math.random()*Thread.MAX_PRIORITY)+1);
```

Sedaj je izvajanje posameznih niti odvisno od njihovih prioritet. Nit z najvišjo prioriteto se bo končala najhitreje, nit z najnižjo prioriteto pa zadnja. Tako spremenjen program je v datoteki `Niti2.java`.

Uporaba niti

In zakaj sploh potrebujemo večnitno izvajanje programa? Eden glavnih razlogov za večnitnost je ustvarjanje odzivnih uporabniških vmesnikov. Poglejmo si torej tak primer programa, ki demonstrira praktično uporabo niti.

Napišimo program, ki deluje kot števec (preprosta štoparica). Naj bo to grafični program, ki v oknu prikazuje trenutno vrednost števca, zraven pa nam trije gumbi omogočajo upravljanje s števcem (zagon, ustavitev, ponastavitev).

Izdelava grafičnega vmesnika nam ne bi smela delati težav, saj vse prvine poznamo že iz poglavja *Grafika*. V okno postavimo tri gumbe za upravljanje števca in oznako za izpis vrednosti števca, dodamo poslušalce za te gumbe in ustrezno reakcijo na dogodek. Kodo bi za začetek lahko zapisali takole (podrobnosti in razlago te kode si poglejte v poglavju *Grafika*; zaenkrat v programu zaradi poenostavitve kode nismo uporabili posebne niti za izgradnjo uporabniškega vmesnika):

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Stevec extends JFrame
                    implements ActionListener {
  private String naslov = "Števec";
  private Container vsebina = null;
  private JPanel kontrole = new JPanel();
  private JPanel plosca = new JPanel();
  private JButton start = new JButton("Start");
  private JButton stop = new JButton("Stop");
  private JButton reset = new JButton("Reset");
  private JLabel izpis = new JLabel("...");

  public Stevec() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle(naslov);
    setSize(300,100);
    start.addActionListener(this);
    stop.addActionListener(this);
    reset.addActionListener(this);
    kontrole.add(start);
    kontrole.add(stop);
    kontrole.add(reset);
    plosca.add(izpis);
    vsebina = getContentPane();
    vsebina.add(kontrole, BorderLayout.NORTH);
    vsebina.add(plosca, BorderLayout.CENTER);
  }

  public void actionPerformed(ActionEvent e) {
    if(e.getSource() == start) {
      // poženi števec
    }
    if(e.getSource() == stop) {
      // zaustavi števec
    }
    if(e.getSource() == reset) {
      // postavi števec na 0
    }
  }

  public static void main(String[] args) {
    Stevec stevec = new Stevec();
    stevec.setVisible(true);
  }
}
```

Seveda v tej kodi manjka najpomembnejša stvar: naš števec in upravljanje z njim. Za števec uporabimo kar celoštevilsko spremenljivko `st`, ki je atribut razreda, njena začetna vrednost pa je nič. Potrebujemo tudi eno logično spremenljivko `pocakaj`, ki določa, ali števec trenutno teče ali stoji.

```
private int st = 0;
private boolean pocakaj = false;
```

Potem lahko v metodi `actionPerformed` definiramo naslednje odzive na posamezne gumbe. Ob kliku gumba `stop` števec ustavimo tako, da postavimo spremenljivko `pocakaj` na `true`:

```
if(e.getSource() == stop) {
  pocakaj = true;
}
```

Števec ponastavimo tako, da mu določimo vrednost 0 in njegovo vrednost prikažemo v oznaki:

```
if(e.getSource() == reset) {
  st = 0;
  izpis.setText(Integer.toString(st));
}
```

Zagon števca pa vključuje nastavitev spremenljivke `pocakaj` in klic metode, v kateri poteka štetje:

```
if(e.getSource() == start) {
  pocakaj = false;
  delaj();
}
```

Metoda `delaj` bi bila lahko naslednja:

```
public void delaj() {
  while(true) {
    if(!pocakaj)
      izpis.setText(Integer.toString(st++));
  }
}
```

Naredimo neskončno zanko, v kateri povečujemo vrednost števca in jo izpisujemo v oznaki, če seveda števec ni zaustavljen (vrednost `pocakaj` mora biti `false`).

Ideja sicer zveni dobro, a je neuporabna. Problem nastane v sami metodi `delaj`, kjer smo ustvarili zanko, ki se nikoli ne izteče (neskončno zanko). Dokler teče ta zanka, je procesor zaseden in ne more obdelovati drugih akcij, kot je na primer obnavljanje okna ali odziv ob kliku na gumb. Dokler se zanka ne konča, se ne konča niti metoda `delaj`, pa tudi metoda `actionPerformed`, kjer smo poklicali metodo `delaj`, se zato ne konča. Rezultat je, da se program ne odziva več na druge dogodke (zamrzne). Po drugi strani pa moramo povečevanje števca postaviti v zanko, ki se izvaja, dokler teče program, saj je to tisto, kar naj bi program delal.

Kako lahko rešimo ta problem, da ohranimo povečevanje števca v neskončni zanki in hkrati zagotovimo odzivnost programa? Tu nam pomagajo niti. Če postavimo povečevanje števca v novo nit, obnavljanje vsebine okna in odzivi na klikanje gumbov pa ostanejo v osnovni niti, se obe niti lahko izvajata istočasno (delita si procesorski čas).

Za števec moramo torej ustvariti novo nit, neskončno zanko pa postaviti za telo te nove niti. Naš razred `Stevec` je že izpeljan iz razreda `JFrame`, zato ne more biti izpeljan tudi iz razreda `Thread`. Razred mora torej izdelati vmesnik `Runnable`:

```
public class Stevec extends JFrame
                    implements ActionListener, Runnable
```

Tako moramo v razredu napisati tudi metodo `run`, ki prevzame delo te niti:

```
public void run() {
  while(true) {
    if(!pocakaj)
      izpis.setText(Integer.toString(st++));
  }
}
```

Za delovanje niti pa potrebujemo še objekt razreda `Thread`, ki naj bo kar privatni atribut razreda:

```
private Thread stejNit = null;
```

Novo nit kreiramo ob prvem kliku na gumb `start`. Če nit še ne obstaja (`stejNit==null`), jo ustvarimo s klicem konstruktorja, ki mu kot argument podamo objekt, katerega metodo `run` bo uporabljala ta nit (v našem primeru je to referenca `this`, saj uporabimo `run` metodo tega razreda):

```
stejNit = new Thread(this);
```

Seveda moramo potem nit tudi zagnati:

```
stejNit.start();
```

Ob kliku na gumb `start` se torej izvede naslednja koda:

```
if(e.getSource() == start) {
  pocakaj = false;
  if(stejNit == null) {
    stejNit = new Thread(this);
    stejNit.start();
  }
}
```

Povečevanje vrednosti števca v metodi `run` je zelo hitro, saj v zanki ne delamo skoraj nič drugega. Štoparica bi delovala bolje, če bi se števec povečeval na primer vsako stotinko ali vsako desetinko sekunde, saj je hitrejše izpisovanje števca nesmiselno. Zato

v metodi `run` dodamo znotraj zanke tudi klic metode `sleep(100)`, ki začasno zaustavi nit za 100 milisekund:

```
try {
  stejNit.sleep(100);
}
catch(InterruptedException e) {
}
```

Tako se števec poveča in izpiše le enkrat na desetinko sekunde, pa že tem izpisom le težko sledimo.

Izvorna koda programa, ki smo ga tako sestavili, je v datoteki `Stevec.java`. V metodo `main` smo dodali tudi klic metode `SwingUtilities.invokeLater()`, ki poskrbi, da se uporabniški vmesnik (grafika) izriše v dogodkovni niti (za podrobnosti glej razdelek *Grafični programi in niti* v poglavju *Grafika*).

Sinhronizacija niti

Pri večnitnih programih lahko nastopijo težave, kadar želi več niti hkrati dostopati do istih virov (na primer spreminjati iste podatke). Te težave, ki izvirajo iz nedoločljivosti izvajanja niti, lahko rešujemo s časovno uskladitvijo ali sinhronizacijo posameznih niti.

V dosedanjih primerih smo uporabljali le neodvisne, neusklajene niti (*asynchronous threads*), kjer je vsaka nit vsebovala lastne kopije podatkov, ki jih je potrebovala pri izvajanju. Pa si poglejmo še primer, ko več niti dostopa do zunanjih (skupnih) podatkov.

Recimo, da imamo trgovino, ki prodaja en sam izdelek (na primer koncertne vstopnice) in seveda vodi evidenco o trenutnem številu izdelkov na zalogi. Razred `Trgovina` naj ima en sam atribut, to je `zaloga`, ki določa število izdelkov, ki so na zalogi v trgovini:

```
private int zaloga = 0;
```

V konstruktorju nastavimo vrednost zaloge na neko naključno vrednost (število vstopnic, ki so v prodaji):

```
public Trgovina() {
  zaloga = (int) (Math.random()*500) + 10;
}
```

Napišimo še metodo, ki vrača trenutno vrednost zaloge v trgovini:

```
public int zaloga() {
  return zaloga;
}
```

Nakup `n` izdelkov izvedemo s pomočjo metode `nakup`, v kateri najprej preverimo, ali je na zalogi dovolj izdelkov ter ustrezno zmanjšamo zalogo (opravimo nakup). Pri tem

metoda vrne število kupljenih izdelkov oziroma `-1`, če nakupa nismo mogli opraviti zaradi premajhne zaloge.

```
public int nakup(int n) {
  if(zaloga >= n) {
    zaloga -= n;
    return n;
  }
  return -1;
}
```

Pred nakupom moramo preveriti, ali je na zalogi toliko izdelkov, kot bi jih želeli kupiti. Metoda `prosto` vrne resnično, če je na zalogi najmanj `n` izdelkov:

```
public boolean prosto(int n) {
  if(zaloga >= n)
    return true;
  else
    return false;
}
```

V trgovini lahko kupuje več kupcev (tudi sočasno), zato za vsakega kupca izdelamo svojo nit. Razred `Kupec` je tako izpeljan iz razreda `Thread`, saj vsak kupec nakupuje neodvisno od ostalih v svoji niti:

```
public class Kupec extends Thread
```

Razred ima en sam atribut, to je referenca na trgovino, v kateri kupuje:

```
private Trgovina trgovina;
```

Vrednost atributa `trgovina` nastavimo v konstruktorju. Dobimo jo preko argumentov konstruktorja skupaj z imenom kupca (imenom niti):

```
public Kupec(String i, Trgovina t) {
  super(i);
  trgovina = t;
}
```

Srce razreda je metoda `run`, v kateri izvajamo nakupovanje, dokler ima trgovina še kakšen izdelek na zalogi:

```
public void run() {
  while (trgovina.zaloga() > 0) {
    // nakupuj
  }
  System.out.println(getName() +
                     " je zaključil nakupovanje");
}
```

Samo nakupovanje pa bi lahko izgledalo takole. Najprej se odločimo, koliko izdelkov bomo kupili (število izdelkov `stevilo` izberemo naključno med 1 in 10):

```
int stevilo = (int) (Math.random()*10) + 1;
```

Nato preverimo, ali je v trgovini dovolj izdelkov na zalogi za naš nakup:

```
if(trgovina.prosto(stevilo)) {
  // opravi nakup
}
else
  System.out.println("Ni dovolj zaloge.");
```

Samo transakcijo ob nakupu simuliramo tako, da nit za nekaj časa zaustavimo (interval izberemo naključno):

```
try {
  sleep((int) (Math.random()*500+50));
}
catch(InterruptedException e) {
}
```

Potem opravimo nakup in pri tem preverimo, ali se je uspešno izvedel:

```
if(trgovina.nakup(stevilo) == stevilo)
  System.out.println("Uspešen nakup.");
else
  System.out.println("Napaka pri nakupu.");
```

V metodi `main` najprej ustvarimo novo trgovino, nato pa še nekaj kupcev (njihovo število podamo kot argument programa). Nato izpišemo trenutno zalogo in poženemo kupce v nakupovanje.

```
public static void main(String[] args) {
  Trgovina trg = new Trgovina();
  Kupec[] kupci = new Kupec[Integer.parseInt(args[0])];
  for(int i=0; i<kupci.length; i++)
    kupci[i] =
        new Kupec("Kupec " + Integer.toString(i+1), trg);
  System.out.println("Zaloge je " + trg.zaloga());
  for(int i=0; i<kupci.length; i++)
    kupci[i].start();
}
```

Program je v datoteki `Trgovina.java`, poskusite ga izvajati z različnim številom kupcev in opazujte njegovo delovanje (število kupcev morate programu podati kot argument).

Glede na to, da pred vsakim nakupom preverimo, ali je na zalogi dovolj izdelkov, bi si mislili, da je preverjanje uspešno izvedenega nakupa povsem odveč, saj je nakup neuspešen le takrat, ko na zalogi ni dovolj izdelkov. A ob zagonu programa vidimo, da ni tako. Kljub temu, da pred nakupom preverjamo stanje zaloge in opravimo nakup le v primeru zadostne zaloge, nam program občasno izpiše tudi napako pri nakupu (torej je ob klicu metode nakup stanje zalog manjše od želenega nakupa).

Pri enem samem kupcu program lepo deluje. Težave pa lahko nastanejo že pri dveh kupcih. Zakaj?

Ta primer prikazuje glavno težavo pri uporabi niti - nikoli namreč ne vemo, kdaj se nit izvaja oziroma kdaj bo njeno izvajanje prekinjeno. Tako se lahko zgodi, da en kupec (ena nit) preveri zalogo za želen nakup, preden pa mu uspe nakup do konca izvesti, nakupovanje prevzame drug kupec (izvajanje niti se prekine in procesorski čas se nameni drugi niti). Ta drugi kupec pokupi vso zalogo izdelkov ter preda nakupovanje spet prvemu kupcu. Prvi kupec, ki se te prekinitve pri nakupovanju sploh ne zaveda, nadaljuje z nakupovanjem tam, kjer je prej ostal. Zalogo je že preveril, torej izvede sam nakup. Ker pa se je medtem zaloga brez njegove vednosti spremenila, je nakup neuspešen.

Kadar lahko različne niti dostopajo do istih podatkov, lahko pride do nepredvidljivih rezultatov, zato moramo te podatke na nek način zaščititi pred deljenim dostopom. Poskrbeti moramo, da več niti ne dostopa istočasno do istega vira vsaj v kritičnih delih. Takim navzkrižjem se lahko izognemo tako, da vir zaklenemo, ko ga uporablja ena nit. Dokler je zaklenjen, ga druge niti ne morejo uporabiti. Ko nit vira ne potrebuje več, ga ponovno odklene in s tem omogoči drugim nitim, da ga uporabijo.

Na ta način damo deljene objekte v izključno rabo le eni niti naenkrat. To pomeni, da ima le ena nit dostop do objekta, vse ostale pa čakajo, da ta nit zaključi svojo nalogo. Šele potem lahko druga nit dobi dostop do objekta. Ta proces imenujemo sinhronizacija niti (*thread synchronization*).

Java ima že vgrajen mehanizem, ki preprečuje sočasen dostop do podatkov objekta. Ker ponavadi do njih dostopamo preko metod, lahko metodo označimo kot sinhronizirano (*synchronized*). Samo ena nit naenkrat lahko kliče sinhronizirano metodo določenega objekta (čeprav lahko ta nit kliče hkrati več sinhroniziranih metod tega objekta).

Vsak objekt privzeto vsebuje eno ključavnico, ki jo imenujemo tudi semafor, ki je avtomatično del objekta. Ob klicu katerekoli sinhronizirane metode se objekt zaklene in nobena druga sinhronizirana metoda tega objekta ne more biti poklicana (iz druge niti), dokler se prva ne zaključi in ponovno odklene objekt. Eno samo ključavnico si torej delijo vse sinhronizirane metode posameznega objekta. Vsaka metoda, ki dostopa do kritičnih virov, mora biti sinhronizirana.

V Javi sinhronizacijo napovemo z rezervirano besedo `synchronized`. Najpogostejši način sinhronizacije je sinhronizacija cele metode:

```
public synchronized int imeMetode()
```

Sinhroniziramo pa lahko tudi le kritični del kode znotraj metode, ki jo imenujemo kritični odsek (*critical section*). Tu uporabimo blok `synchronized`, kjer v oklepaju podamo objekt, katerega ključavnico uporabimo za sinhronizacijo kode v bloku:

```
synchronized(objekt) {
  ...
}
```

Če se vrnemo na naš zadnji primer, moramo najprej ugotoviti, kje so težave pri usklajevanju dostopa. Problem predstavlja sočasen dostop do atributa `zaloga`, torej bomo morali uporabiti ključavnico objekta `trgovina`.

Kritični del kode je sama akcija nakupa:

```
if(trgovina.prosto(stevilo)) {
  try {
    sleep((int) (Math.random()*500+50));
  }
  catch(InterruptedException e) {
  }
  if(trgovina.nakup(stevilo) == stevilo)
    System.out.println("Uspešen nakup");
  else
    System.out.println("Napaka pri nakupu ...");
}
else
  System.out.println("Ni dovolj zaloge");
```

Zato ta del kode postavimo v blok `synchronized`, za ključavnico pa uporabimo objekt `trgovina`:

```
synchronized(trgovina) {
  // kritični del kode
}
```

Katere pa so metode znotraj objekta trgovina, ki morajo biti sinhronizirane? To sta dve metodi, prva za izvedbo nakupa in druga za preverjanje razpoložljive zaloge:

```
public synchronized int nakup(int n)
public synchronized boolean prosto(int n)
```

Ustrezno popravljen program (datoteka `TrgovinaS.java`) sedaj deluje pravilno, saj smo preprečili več nitim sočasen dostop do atributa `zaloga`. Njegovo delovanje preverite z različnim številom kupcev.

Proizvajalec in potrošnik

Včasih pa moramo delo dveh niti tudi časovno uskladiti. Tak primer sta dve niti, od katerih ena pripravlja podatke (proizvajalec ali *producer*), druga pa te podatke uporablja (potrošnik ali *consumer*). Pri tem ni dovolj, da obe niti ne dostopata istočasno do podatkov, temveč moramo njuno delo uskladiti tako, da druga nit ne dobi podatka, ki še ni pripravljen, hkrati pa tudi ne sme dvakrat prebrati istega podatka.

Oglejmo si preprost primer. Proizvajalec naj v naključnih časovnih intervalih generira števila med 1 in 10 ter jih shranjuje v odložišču. Potrošnik pa naj iz odložišča (ki je isti objekt, kot ga uporablja proizvajalec za odlaganje števil) jemlje števila, takoj ko so ta na voljo. Svoje delo zaključi, ko dobi zadnje število (to je 10).

Napišimo najprej razred, ki predstavlja odložišče. Razred ima en privatni atribut, to je trenutna vsebina odložišča:

```
public class Odlozisce {
  private int vsebina;

}
```

V razredu imamo tudi dve metodi, s pomočjo katerih beremo oz. spreminjamo vrednost atributa. Metoda `postavi` nastavi vrednost atributa `vsebina` na podano vrednost (postavi število v odložišče):

```
public void postavi(int v) {
  vsebina = v;
  System.out.println("Postavljeno: " + v);
}
```

Metoda `vzemi` pa vrne vrednost atributa `vsebina` (branje vsebine odložišča); ko je vsebina prebrana, se njena vrednost nastavi na 0:

```
public int vzemi() {
  int v = vsebina;
  vsebina = 0;
  System.out.println("Vzeto: " + v);
  return v;
}
```

Proizvajalca bomo opisali z razredom `Proizvajalec`, ki ga izpeljemo iz razreda `Thread`, saj želimo svojo nit za generiranje števil.

```
public class Proizvajalec extends Thread
```

Metodo, ki določa delovanje te niti, lahko zapišemo takole: v zanki postavljamo na odložišče števila od 1 do 10 po vrsti, pred vsako ponovitvijo zanke pa počakamo nekaj časa (interval je izbran naključno med 0 in 100 milisekundami).

```
public void run() {
  for(int i=1; i<=10; i++) {
    polica.postavi(i);
    try {
      sleep((int)(Math.random()*100));
    }
    catch (InterruptedException e) {
    }
  }
}
```

Objekt, ki predstavlja odložišče, smo poimenovali `polica`, objektna spremenljivka je atribut razreda.

```
private Odlozisce polica;
```

Ker naj bi bilo odložišče skupno proizvajalcu in potrošniku, saj preko njega izmenjujeta podatke (števila), moramo referenco na ta objekt podati ob klicu konstruktorja proizvajalca.

```
public Proizvajalec(Odlozisce o) {
  this.polica = o;
}
```

Razred `Potrosnik` opisuje potrošnika, ki iz odložišča jemlje števila. Tudi potrošnik deluje v svoji niti:

```
public class Potrosnik extends Thread
```

Njegovo delovanje določa metoda `run`. V zanki jemlje števila iz odložišča, dokler ne pride do zadnjega števila, to je števila z vrednostjo 10. Takrat se zanka zaključi in z njo tudi metoda `run` ter s tem tudi sama nit. Ob vsaki ponovitvi zanke počakamo nekaj časa (tudi tu je interval izbran naključno med 0 in 100 milisekundami), da se zanka ne izvaja prehitro.

```
public void run() {
  int vrednost = 0;
  do {
    try {
      sleep((int)(Math.random()*100));
    }
    catch (InterruptedException e) {
    }
    vrednost = polica.vzemi();
  } while(vrednost < 10);
}
```

Tudi v potrošniku smo definirali atribut `polica`, ki je referenca na odložišče, torej na isti objekt, kot ga uporablja proizvajalec za hranjenje števil. Referenco na ta objekt smo podali on klicu konstruktorja potrošnika.

```
private Odlozisce polica;

public Potrosnik(Odlozisce o) {
  this.polica = o;
}
```

Na koncu napišimo še razred, katerega metoda `main` poskrbi za ustvarjanje odložišča, proizvajalca in potrošnika ter za zagon obeh niti:

```
public class PPAsinhrono {
  public static void main(String[] args) {
    Odlozisce odl = new Odlozisce();
    Proizvajalec pro = new Proizvajalec(odl);
    Potrosnik pot = new Potrosnik(odl);
    pro.start();
    pot.start();
  }
}
```

Obe niti, proizvajalec in potrošnik, sta v tem primeru nesinhronizirani, zato smo razred tudi tako poimenovali. Cel program je v datoteki `PPAsinhrono.java`. Preizkusite delovanje programa in preverite, ali rešitev ustreza problemu, kot smo si ga zamislili.

Proizvajalec in potrošnik delita podatke preko objekta razreda `Odlozisce`. Čeprav naj bi potrošnik prebral vsako vrednost natanko enkrat, pa pri izvajanju programa opazimo, da ni vedno tako. Pojavita se lahko dve vrsti problemov:

- Proizvajalec je hitrejši od potrošnika in generira dve števili, preden uspe potrošnik prebrati prvo število. V tem primeru potrošnik izpusti število.
- Potrošnik je hitrejši od proizvajalca in že drugič prebere število, preden uspe proizvajalec zgenerirati novo število. Tako potrošnik prebere število 0, čeprav le-to ni med števili proizvajalca.

Obema opisanima problemoma se lahko izognemo, če proizvajalčevo shranjevanje novega števila v odložišče časovno uskladimo s porabnikovim branjem tega števila. Program popravimo tako, da bo delovanje obeh niti usklajeno.

Najprej moramo zagotoviti, da obe niti ne dostopata istočasno do vrednosti v odložišču. Sočasen dostop lahko preprečimo z uporabo zaklepanja objekta. V našem primeru sta kritični metodi `vzemi` in `postavi` v razredu `Odlozisce`. Obe torej označimo kot sinhronizirani:

```
public synchronized int vzemi() {
  ...
}

public synchronized void postavi(int v) {
  ...
}
```

Ko vstopimo v sinhronizirano metodo, nit, ki je klicala to metodo, zaklene objekt, katerega metodo je poklicala. Tako ostale niti ne morejo klicati nobene sinhronizirane metode istega objekta, dokler je ta objekt zaklenjen. Nit, ki je zaklenila objekt, lahko poljubno kliče tudi ostale sinhronizirane metode tega objekta, saj ima že kontrolo nad njegovo ključavnico.

Vendar pa s tem problema nismo rešili, saj lahko proizvajalec še vedno prehiteva potrošnika ali obratno. Zato moramo obe niti tudi časovno uskladiti. Proizvajalec mora na nek način obvestiti potrošnika, da je nova vrednost na voljo, potrošnik pa mora na nek način obvestiti proizvajalca, da lahko pripravi novo vrednost. Za to lahko v sinhronizirani metodi uporabimo metodi `wait` in `notify` (ali `notifyAll`). Prva metoda omogoča niti, da počaka, dokler ni izpolnjen določen pogoj, druga metoda pa obvesti čakajoče niti (eno ali vse), ko se ta pogoj spremeni. Vse omenjene metode so metode razreda `Object`, zato jih podedujejo vsi objekti v Javi.

V razred `Odlozisce` moramo torej dodati še en privatni atribut, ki pove, ali je nova vrednost na voljo (resnično, ko je nova vrednost nastavljena in še ne prebrana, in neresnično, kadar je vrednost prebrana in še ni ponovno nastavljena):

```
private boolean naVoljo = false;
```

Program želimo napisati tako, da bo potrošnik čakal, dokler proizvajalec ne bo postavil nove vrednosti v odložišče in o tem obvestil potrošnika. Podobno pa naj tudi proizvajalec čaka, dokler potrošnik ne prebere vrednosti iz odložišča in o svoji aktivnosti obvesti proizvajalca. Šele takrat lahko proizvajalec postavi novo vrednost v odložišče.

Metodi `vzemi` in `postavi` moramo spremeniti tako, da čakata in obveščata druga drugo o svojih aktivnostih.

Koda metode `vzemi` se tako začne z zanko, katera se ponavlja, dokler v odložišču ni na voljo nove vrednosti (dokler proizvajalec ne zgenerira nove vrednosti). V zanki kliče metodo `wait`, ki povzroči čakanje te metode, dokler je proizvajalec ne obvesti, da je zaključil svoje delo (metoda `wait` povzroči, da potrošnik sprosti ključavnico nad objektom odložišče, ki jo pridobi ob vstopu v sinhronizirano metodo, ter tako omogoči proizvajalcu, da zaklene ta objekt in nastavi vrednost v odložišču). Ko proizvajalec nastavi vrednost v odložišču, s klicem metode `notify` o tem obvesti potrošnika. Metoda, ki ima izključen nadzor nad deljenim objektom, lahko namreč pokliče metodo `notify` ali `notifyAll`, ki sporoči čakajočim nitim (eni ali vsem), da lahko prekinejo s čakanjem. Potrošnik nato prekine svoje stanje čakanja (vrednost spremenljivke `naVoljo` se je med tem tudi spremenila na `true`), izstopi iz `while` zanke ter vrne vrednost iz odložišča. Še pred tem pa postavi spremenljivko `naVoljo` ponovno na `false`, saj po odvzemu vrednosti iz odložišča le-ta ni več na voljo.

```
public synchronized int vzemi() {
  while(naVoljo == false) {
    try {
      wait();
    }
    catch (InterruptedException e) {
    }
  }
  int v = vsebina;
  vsebina = 0;
  System.out.println("Vzeto: " + v);
  naVoljo = false;
  notify();
  return v;
}
```

Metoda `wait` lahko sproži izjemo `InterruptedException`, kadar jo prekine druga nit, zato jo postavimo v `try`/`catch` blok. Izjemo prestrežemo, a ne ukrepamo, saj je tak dogodek pričakovan.

Podobno spremenimo tudi kodo metode `postavi`, kjer na začetku v zanki čakamo, da potrošnikova nit prebere trenutno vrednost iz odložišča ter obvesti proizvajalca, da lahko v odložišče postavi novo vrednost. Tu gre za primer, ko proizvajalec čaka na potrošnika, da opravi svoje delo, zato se `while` zanka ponavlja, dokler je vrednost `naVoljo` resnična. Tudi tu moramo takoj po postavitvi nove vrednosti v odložišče nastaviti spremenljivko `naVoljo`, tokrat na vrednost `true`.

```
public synchronized void postavi(int v) {
  while(naVoljo == true) {
    try {
      wait();
    }
    catch (InterruptedException e) {
    }
  }
  vsebina = v;
  System.out.println("Postavljeno: " + v);
  naVoljo = true;
  notify();
}
```

Tako dopolnjena koda je v datoteki `PPSinhrono.java`, ki smo jo tako poimenovali zaradi usklajenega dela proizvajalca in potrošnika. Preizkusite delovanje programa in preverite, ali obe niti res delujeta usklajeno in v pravilnem zaporedju (vrstni red izpisov bi moral biti vedno enak).

TRETJI DEL
Programski jezik C

10. Programiranje v jeziku C

Za razliko od Jave, kjer po prevajanju dobimo javansko vmesno kodo, ki jo izvajamo v posebnem izvajalnem okolju, je pri jeziku C rezultat prevajanja izvršljiva koda, ki je odvisna od računalniškega sistema. Preveden program tako v splošnem ni prenosljiv na druge sisteme. Izbira platforme za delo (Windows ali Linux ali kaj tretjega) je v tem primeru bolj pomembna, saj so nanjo vezana tudi potrebna orodja za razvoj programov (predvsem prevajalnik).

V nadaljevanju si poglejmo, 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

Najprej moramo napisati izvorno kodo programa. Le-to lahko napišemo v poljubnem urejevalniku besedila, ki pa nam mora omogočati shranjevanje vsebine kot navadno besedilo. Uporabimo lahko isti urejevalnik kot za pisanje javanskih programov (nekaj besed o izbiri urejevalnika smo zapisali že v razdelku *Pisanje izvorne kode programa* v 4. poglavju).

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`. Imena datotek, ki vsebujejo izvorno kodo C, naj imajo vedno podaljšek `.c` (kot na primer naš program `prvi.c`). Tako lahko po imenu (mi in tudi prevajalnik, ki tako ime zahteva) vedno prepoznamo datoteko z izvorno kodo jezika C.

Prevajanje programa

Za prevajanje in povezovanje programov v tem gradivu uporabljamo standardni *GNU C/C++* prevajalnik. 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 (posebej označiti pri namestitvi orodij). Alternativa na Windows platformi je tudi prevajalnik *MinGW* (www.mingw.org). Pravzaprav je izbira prevajalnika dokaj poljubna, paziti moramo le, da podpira standardni ANSI C jezik (ISO C99).

Program prevedemo v ukazni lupini (odpremo terminalsko okno oziroma *Cygwin Bash Shell* ali ukazno vrstico v okolju Windows). Prevajalnik *GNU C/C++ Compiler* pokličemo z ukazom `gcc`, kateremu sledi ime datoteke z izvorno kodo in lahko tudi dodatne 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/Windows okolju), ki vsebuje izvajalno kodo.

Pri klicu prevajalnika je priporočljivo dodati tudi opcijo `-o`, ki določi ime izhodne (izvršljive) datoteke (v našem primeru naj bo 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*), zbirnje (*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 večinoma uporabljali vse štiri hkrati, kot smo to naredili v zgornjem primeru.

Za razvoj programov lahko uporabimo tudi katero izmed številnih razvojnih okolij, kot sta na primer *Eclipse CDT* (www.eclipse.org/cdt/) ali *Pelles C for Windows* (www.christian-heffner.de); slednji uporablja prevajalnik *lcc*.

Izvajanje programa

Tudi izvajanje programa poteka v ukazni lupini. Ko smo s prevajanjem ustvarili izvršljivo datoteko, v našem primeru je to `program` (oz. `program.exe` v Cygwin/Windows 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 tudi tekoči direktorij, to je `./`).

Kakšen pa je rezultat? Program izpiše:

```
Pozdravljeni!
```

in nato skoči v novo vrstico ter konča.

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.

11. Hiter pregled jezika C (za Java programerje)

Za začetek si na kratko poglejmo osnovno sintakso jezika, ki je v grobem precej podobna javanski sintaksi. Zato se ne bomo spuščali v podrobnosti, temveč le povzeli osnovne konstrukte jezika C in nakazali razlike med obema jezikoma.

Rezervirane besede

Nabor rezerviranih besed jezika ANSII C je kratek; besede so po abecedi zapisane v spodnji tabeli.

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

Tabela 11.1: Rezervirane besede.

Večino rezerviranih besed in njihov pomen (uporabo) poznamo že iz Jave. Vse navedene rezervirane besede pa bomo vsaj na kratko obravnavali tudi v nadaljevanju tega poglavja.

Spremenljivke in konstante

Spremenljivke v jeziku C deklariramo podobno kot v Javi: najprej napovemo tip spremenljivke, temu sledi ime spremenljivke (ali z vejico ločen seznam spremenljivk), stavek pa se zaključi s podpičjem.

```
int zacetek, konec, korak;
```

Ob deklaraciji lahko spremenljivko tudi inicializiramo (ji nastavimo začetno vrednost).

```
int zacetek = 1;
```

Spremenljivke navadno deklariramo tam, kjer jih potrebujemo. Če spremenljivko uporabljamo v različnih funkcijah, jo lahko deklariramo kot globalno spremenljivko tako, da je njena deklaracija nad funkcijo `main` (izven nje). Taka spremenljivka je potem dostopna v vseh funkcijah programa (v dani datoteki). Tudi globalnim spremenljivkam lahko ob deklaraciji določimo začetno vrednost.

```
int globalna = 11;

int main() {
  ...
}
```

Konstante

Konstante imajo vrednosti, ki jih ne moremo spremeniti, kot so na primer število `123`, znak `'a'` ali niz `"abc"`. Uporabimo jih za nastavitev vrednosti spremenljivke, na primer:

```
int i = 123;
char znak = 'a';
char *niz = "abc";
```

Konstanta je tudi spremenljivka, katere vrednosti ne moremo spremeniti (lahko jo le beremo). Tako spremenljivko deklariramo s pomočjo rezervirane besede `const`:

```
const float pi = 3.14;
```

Ob deklaraciji moramo spremenljivki tudi nastaviti vrednost, saj njene vrednosti kasneje ne moremo spreminjati (torej tudi ne inicializirati).

Najpogosteje pa za konstante v programu uporabljamo kar predprocesorski ukaz za zamenjavo `#define`, ki je podrobneje opisan v 17. poglavju.

Določila spremenljivk

Spremenljivki lahko posebej določimo tudi način pomnjenja. Za to uporabimo določila pomnilniških razredov `auto`, `register`, `static` in `extern`.

Privzeto ima vsaka lokalna spremenljivka (t.j. spremenljivka znotraj funkcije) način `auto`. Ker je to privzeta vrednost, rezervirano besedo navadno izpustimo. Tako sta naslednja dva stavka znotraj funkcije enakovredna:

```
auto int i;
int i;
```

Spremenljivki se ob definiciji samodejno rezervira prostor, ki se tudi samodejno sprosti, ko spremenljivka ne obstaja več (po zaključku funkcije ali bloka, v katerem je bila definirana).

Kadar neko spremenljivko pogosto uporabljamo, ji dodamo določilo `register`, kar je namig prevajalniku, da naj spremenljivko hrani v registru (hitrejše delovanje). To sicer ne pomeni, da bo na koncu spremenljivka tudi res shranjena v registru. Velikost take spremenljivke je omejena z velikostjo registra in nad njo ne moremo uporabiti unarnega operatorja `&`, saj nima lokacije v pomnilniku in zato tudi ne naslova. Določilo `register` se dandanes le redko uporablja, saj sodobni prevajalniki kodo dobro optimizirajo že sami.

Če spremenljivki v funkciji dodamo določilo `static`, prevajalnik poskrbi, da se prostor za to spremenljivko ohrani do konca programa. Na ta način dosežemo, da si funkcija zapomni vrednost spremenljivke od prejšnjega klica te funkcije.

Kadar pa kot statično označimo globalno spremenljivko, je njena vrednost veljavna do konca izvorne datoteke (spremenljivka je lokalna v podani datoteki). Tako določilo `static` pravzaprav določa tako stalnost kot tudi stopnjo privatnosti spremenljivke.

Če je globalna spremenljivka definirana v neki drugi datoteki, z določilom `extern` povemo prevajalniku, da je ta spremenljivka definirana nekje drugje. Deklaracija z oznako `extern` ne rezervira prostora za spremenljivko, saj je bil prostor zanjo že rezerviran ob njeni definiciji.

Spremenljivke z določili `auto`, `register` in `static` lahko inicializiramo že ob njihovem kreiranju. Globalne spremenljivke in spremenljivke z določilom `static` dobijo začetno vrednost 0, če jih nismo eksplicitno inicializirali. Spremenljivke `auto` in `register`, ki jim ne določimo začetne vrednosti, pa imajo nedefinirano vrednost (njihova vrednost je odvisna od tega, kaj je v tistem trenutku zapisano v pomnilniku na lokacijah, rezerviranih za spremenljivko). Zato jim moramo pred uporabo vedno nastaviti vrednost.

Posebno določilo, ki ga lahko dodamo spremenljivki pri deklaraciji, je tudi `volatile`. Ta pove prevajalniku, da se lahko vrednost spremenljivke spreminja tudi v procesih, ki tečejo v ozadju (kot je na primer spreminjanje globalne spremenljivke v prekinitvi servisne rutine ali v večnitni aplikaciji) in je torej ni dovoljeno optimizirati.

Podatkovni tipi, funkcije

Podatkovni tipi določajo množico vrednosti, ki jih podpira jezik, ter operacije nad njimi. Poleg osnovnih podatkovnih tipov nudi jezik C tudi sestavljene tipe ter možnost definiranja lastnih podatkovnih tipov.

Osnovni podatkovni tipi

Jezik C nudi standardni nabor osnovnih podatkovnih tipov: `char` predstavlja en znak (8 bitov), `int` celo število, `float` realno število, `double` pa realno število v dvojni natančnosti.

Navedenim osnovnim tipom lahko dodamo tudi modifikatorje `short`, `long`, `signed` in `unsigned`, ki določajo velikost prostora v pomnilniku, ki ga zaseda spremenljivka.

```
short int a;
signed char b;
unsigned int c;
long double d;
```

Seveda je ta prostor v pomnilniku odvisen od same implementacije jezika (od platforme), a v splošnem veljata naslednji pravili: `short int` <= `int` <= `long int` ter `float` <= `double` <= `long double`.

K osnovnim podatkovnim tipom bi lahko prišteli tudi tip `void`. Ta se najpogosteje uporablja za označitev tipa funkcije, ki ne vrača vrednosti.

Sestavljeni podatkovni tipi

Sestavljeni podatkovni tipi so, kot pove že ime, sestavljeni iz več drugih podatkovnih tipov. Tabela ali polje (*array*) je primer sestavljenega podatkovnega tipa, ki vsebuje več elementov istega tipa. Pri deklaraciji polja moramo navesti, koliko elementov bo največ v polju, da se lahko rezervira ustrezno velik prostor v pomnilniku za to polje. Za razliko od Jave informacija o velikosti polja ni del samega polja (kot je atribut `length` v Javi), zato mora programer sam poskrbeti za to, da je velikost polja znana tam, kjer jo potrebuje (shrani jo v spremenljivko). Splošna deklaracija polja je naslednja:

```
tip ime[velikost];
```

V splošnem se uporaba polja v jeziku C nekoliko razlikuje od polj v Javi; te razlike so podrobneje opisane v 13. poglavju.

Kadar želimo skupaj grupirati več spremenljivk različnih tipov, lahko uporabimo strukturo (*structure*). Splošna deklaracija strukture je naslednja:

```
struct ime_strukture {
  tip1 ime_elementa1;
  tip2 ime_elementa2;
  tip3 ime_elementa3;
};
```

Število elementov v strukturi je poljubno. Več o strukturah najdete v razdelku *Strukture, kazalci na strukture in rekurzivne strukture* v 20. poglavju.

Tudi unije (*union*), podobno kot strukture, lahko vsebujejo elemente različnih tipov. Vendar si elementi unije delijo isti prostor v pomnilniku, zato lahko sočasno nastavimo vrednost le enemu od elementov. Torej lahko naenkrat uporabljamo le en element unije. Deklaracija unije je podobna deklaraciji strukture:

```
union ime_unije {
  tip1 ime_elementa1;
  tip2 ime_elementa2;
  tip3 ime_elementa3;
};
```

Tudi spremenljivko tipa unija deklariramo podobno kot pri strukturah:

```
union ime_unije ime_spr_unija;
```

Tudi do elementov unije dostopamo podobno kot do elementov strukture, to je s pomočjo operatorja pika:

```
ime_spr_unija.ime_elementa;
```

Lastni podatkovni tipi

V jeziku C lahko definiramo tudi lastne podatkovne tipe. Če ne želimo biti omejeni le na vrednosti osnovnih podatkovnih tipov, lahko ustvarimo nov podatkovni tip, ki sprejme poljubne vrednosti, ki jih podamo ob definiciji tega novega tipa. Taki novi tipi se imenujejo naštevni tipi (*enumerations*) in vsebujejo seznam konstant, na katere se sklicujemo kot na cela števila. Prva vrednost je 0, druga 1 in tako naprej. Tip ustvarimo s pomočjo rezervirane besede `enum`:

```
enum ime_nov_tip {vrednost1, vrednost2, vrednsot3, ...};
```

Ob definiciji tipa lahko navedemo tudi, čemu ustreza določena vrednost:

```
enum ime_nov_tip {vrednost1=1, vrednost2, vrednsot3, ...};

enum ime_nov_tip {vrednost1=v1, vrednost2=v2,
                  vrednsot3=v3, vrednsot4=v4};
```

V prvem primeru se številčenje vrednosti začne z 1 in ne z 0, kot je privzeto (vrednosti `vrednost2` potem ustreza 2 in tako dalje). V drugem primeru pa posamezne vrednosti označimo s podanimi vrednostmi `v1`, `v2`, `v3` in `v4` namesto s celimi števili.

Kot primer si poglejmo, kako lahko določimo naštevni tip, ki predstavlja slovenske registrske oznake, ter kako uporabljamo spremenljivko takega tipa:

```
enum registrske {CE,GO,KK,KP,KR,LJ,MB,MS,NM,PO,SG};
enum registrske oznaka;
...
oznaka = KK;
while(oznaka != SG)
  oznaka++;
```

Poljuben nov podatkovni tip lahko definiramo s pomočjo rezervirane besede `typedef`, sintaksa pa je naslednja:

```
typedef obstojeci_tip ime_novega_tipa;
```

Tu je `ime_novega_tipa` sinonim (in navadno krajši zapis) določenega obstoječega tipa, ki ga lahko uporabljamo kot katerikoli vgrajen tip jezika. Primer:

```
enum logic {FALSE, TRUE};
typedef enum logic boolean;
...
boolean konec = FALSE;
while(!konec) {
  ...
  konec = TRUE;
}
```

V tem primeru smo definirali nov tip `boolean`, ki prejme vrednosti `FALSE` (pripadajoče število 0) in `TRUE` (pripadajoča 1). Potem lahko tip `boolean` uporabljamo za deklaracijo spremenljivke na enak način, kot bi bil to eden izmed osnovnih podatkovnih

tipov. Spremenljivko lahko nadalje uporabljamo v izrazih ter ji prirejamo vrednosti (`TRUE` ali `FALSE`). Bodite pozorni na vrstni red vrednosti v naštevnem tipu, saj mora logična vrednost neresnično ustrezati vrednosti 0, resnično pa od nič različni vrednosti (kar je lahko 1).

Pretvorba med tipi

Včasih želimo spremenljivki prirediti določen izraz oz. vrednost, katere tip ne ustreza tipu spremenljivke. Problem rešimo tako, da tip prirejanja pretvorimo v ustrezen tip spremenljivke. Tudi operatorji zahtevajo operande določenega tipa in pri tem lahko pride do pretvorbe vrednosti operandov iz enega tipa v drugega.

Pretvorbe med podatkovnimi tipi so lahko implicitne (samodejne) ali eksplicitne (zahtevane). Implicitne so tiste pretvorbe, ki jih prevajalnik lahko izvede samodejno. Pri eksplicitnih pretvorbah pa samodejna pretvorba ni mogoča in jo moramo eksplicitno zapisati v kodo. To nam omogoča operator za pretvorbo tipa `()` (*cast operator*). Pretvorbe med tipi so v jeziku C podobne kot pretvorbe v Javi.

```
int a;
float b = 3.14;
char c;

a = b;   //samodejna pretvorba float v int
c = a;   //samodejna pretvorba int v char
b = a/2; //samodejna pretvorba int v float, vrednost b je 1

b = (float)a/2; //zahtevana pretvorba int v float (spr. a),
                //vrednost b je 1.5
```

Funkcije

V splošnem ima funkcija naslednjo obliko:

```
tip ime(parametri) {
  stavki;
}
```

Funkcijo sestavlja glava (prva vrstica) in telo med zavitimi oklepaji. V glavi je najprej naveden tip, ki ga funkcija vrača. Če ga izpustimo, je tip funkcije privzeto `int`. Če funkcija nič ne vrača, je njen tip `void`. Imenu funkcije sledijo njeni parametri, ki so navedeni v oklepaju in ločeni z vejico. Za vsak parameter navedemo njegov tip in ime.

V jeziku C se vsi parametri funkcije prenašajo po vrednosti. To pomeni, da se ob klicu funkcije ustvari kopija vsakega parametra in funkcija uporablja te kopije. Vrednost posameznega parametra se torej prekopira v ustrezno spremenljivko funkcije. Zato funkcija ne more spreminjati vrednosti parametrov, ki so ji podani. Kadar želimo vrednosti parametrov spremeniti, lahko to dosežemo s pomočjo kazalcev – funkciji namesto spremenljivke podamo naslov te spremenljivke (podrobneje je to razloženo v *Kazalci in argumenti funkcij* v 13. poglavju).

Funkcija se zaključi, ko se izvede zadnji stavek v njenem telesu (v tem primeru funkcija ne vrne nobene vrednosti). Funkcijo lahko zaključimo na poljubnem mestu z uporabo stavka `return`.

```
return izraz;
return;
```

V prvem primeru funkcija vrne vrednost izraza, ki se mora po tipu ujemati s tipom funkcije. V drugem pa se funkcija samo zaključi in ne vrača nobene vrednosti (tip funkcije mora biti `void`).

Vsako funkcijo (razen `main`) moramo deklarirati, preden jo uporabimo v programu. Če je funkcija definirana pred njeno prvo uporabo v programu, je temu pogoju že zadoščeno. Sicer pa moramo pred uporabo funkcijo deklarirati (zapišemo le tip in ime funkcije ter v oklepaju tipe njenih parametrov), definiramo (zapišemo glavo funkcije skupaj z njenim telesom) pa jo lahko na poljubnem mestu v programu.

```
int f(int);  // deklaracija funkcije

int main() {
  int rez;
  ...
  rez = f(5);  // uporaba (klic) funkcije v funkciji main
  ...
}

int f(int x) {  // definicija funkcije
  int r = 0;
  ...
  return r;
}
```

Če funkcijo `f` iz primera uporabljamo le v funkciji `main`, bi jo lahko tam tudi deklarirali. Tudi v tem primeru mora biti funkcija nekje definirana.

```
int main() {
  int rez, f(int);  // deklaracija funkcije f
  ...
  rez = f(5);
  ...
}

int f(int x) {  // definicija funkcije f
  ...
}
```

Izrazi in operatorji

Operatorji skupaj z operandi tvorijo izraz (*expression*). Vsak izraz ima svoj tip in vrednost. Tako je na primer izraz `1+2` sestavljen iz enega operatorja `+` ter dveh operandov `1` in `2`. Tip tega izraza je `int`, vrednost pa `3`.

Navadno operatorji prejmejo dva operanda. Če operator prejme en sam operand, se imenuje unarni operator. Nekateri operatorji pa delujejo tudi nad več operandi; primer je pogojni operator, ki prejme tri operande.

Operatorje v C-ju lahko razdelimo v več skupin glede na njihov značaj. V nadaljevanju so operatorji vsake skupine povzeti v tabeli z opisom in primerom uporabe.

Aritmetični operatorji

Operator	Opis	Primer
+	seštevanje	x+y
-	odštevanje	x-y
*	množenje	x*y
/	deljenje	x/y
%	ostanek pri deljenju	x%y
++	inkrement	++x y++
--	dekrement	--x y--
-	unarni minus	-x
+	unarni plus	+x

Tabela 11.2: Aritmetični operatorji.

Operatorja inkrement in dekrement uporabljamo za povečanje oz. zmanjšanje vrednosti spremenljivke za 1. Oba operatorja lahko uporabimo v prefiksni (++x) ali postfiksni (x++) obliki. Razlika med njima je v tem, da se v prvem primeru vrednost poveča za ena preden se izračuna izraz, medtem ko se v drugem primeru vrednost poveča za ena šele po izračunu izraza.

Primer izraza, v katerem nastopata oba operatorja je na primer naslednji:

```
a = ++x * y--;
```

V zgornjem primeru se tako spremenljivki x prišteje 1, njena nova vrednost se pomnoži z vrednostjo spremenljivke y ter rezultat shrani v spremenljivko a. Nato pa se vrednost spremenljivke y še zmanjša za 1. Zgornjemu izrazu je torej ekvivalentno naslednje zaporedje stavkov:

```
x++;
a = x * y;
y--;
```

Opozoriti velja tudi na posebnost operatorja za deljenje /, ki lahko predstavlja celoštevilsko deljenje (rezultat je celo število) ali pa navadno deljenje (rezultat je realno število). Celoštevilsko deljenje se samodejno izvede takrat, kadar sta oba operanda celoštevilska. Če pa je katerikoli izmed operandov realno število, se izvede navadno deljenje.

Zato moramo pri deljenju vedno preveriti, ali so operandi res ustreznega tipa za pričakovan rezultat ter po potrebi eksplicitno spremeniti tip operandov.

```
int x = 9/2;     // x dobi vrednost 4
float y = 9/2;   // y dobi vrednost 4.0
float z = 9.0/2; // z dobi vrednost 4.5
float w = 9/2.0; // w dobi vrednost 4.5
```

Logični operatorji

Operator	Opis	Primer
`&&`	logični in	`x&&y`
`||`	logični ali	`x||y`
`!`	logični ne	`!x`

Tabela 11.3: Logični operatorji.

Logični operatorji kot rezultat vrnejo resnično (`1`) ali neresnično (`0`). Za resnično se šteje katerakoli od nič različna vrednost. Tako je na primer rezultat izraza `5&&6` enak `1` (resnično).

Logični ne (`!`) obrne logično vrednost operanda in vrne `0` ali `1`. Tako je na primer `!-5` enako `0`, `!5` je tudi enako `0`, `!0` pa je enako `1`.

V splošnem se izraz izračunava od leve proti desni in izračun se ustavi takoj, ko je rezultat znan. To lahko s pridom uporabimo pri pisanju pogojev:

```
if(x != 0 && y/x > 5)
```

V zgornjem primeru bi, ko je `x` enak `0`, imeli deljenje z nič, a do izračuna izraza `y/x` sploh ne pride, saj se izračun ustavi že v prvem delu, ker je izraz `x!=0` neresničen in tako celoten izraz v pogoju stavka `if` dobi vrednost `0`.

Primerjalni operatorji

Operator	Opis	Primer
`==`	je enak	`x==y`
`!=`	ni enak	`x!=y`
`<`	je manjši	`x<y`
`>`	je večji	`x>y`
`<=`	je manjši ali enak	`x<=y`
`>=`	je večji ali enak	`x>=y`

Tabela 11.4: Primerjalni operatorji.

Tudi primerjalni operatorji (imenovani tudi relacijski operatorji) kot rezultat vračajo resnično (`1`) ali neresnično (`0`).

Bitni operatorji

Operator	Opis	Primer
`&`	bitni in	`x&y`
`|`	bitni ali	`x|y`
`^`	bitni ekskluzivni ali	`x^y`
`<<`	pomik v levo	`x<<1`
`>>`	pomik v desno	`x>>1`
`~`	eniški komplement	`~x`

Tabela 11.5: Bitni operatorji.

Bitni operatorji delujejo nad posameznimi biti operandov (pri tem operandi ne smejo biti tipa `float` ali `double`). Tako se pri bitnem in (*and*) posamezen bit zapiše v rezultat, če je ta bit 1 v obeh operandih. Pri bitnem ali (*or*) pa mora biti bit enak 1 v vsaj enem izmed operandov, da se zapiše v rezultat. Bitni ekskluzivni ali (*xor*) da rezultat 1 le, kadar je en operand 1, drugi pa 0; kadar sta oba operanda enakih vrednosti, je rezultat 0.

Poglejmo si delovanje bitnega operatorja in na primeru. Izraz `10&12` lahko izračunamo tako, da najprej oba operanda pretvorimo v dvojiški zapis. Številu `10` ustreza `00001010`, številu `12` pa `00001100`. Če sedaj pri obeh številih pogledamo istoležne bite in upoštevamo, da operator `&` vrne `1` le v primeru, ko sta oba operanda enaka `1`, dobimo rezultat `00001000`, kar je v desetiškem zapisu enako `8`. Vrednost izraza `10&12` je torej enaka `8`.

Pri bitnih operatorjih in (`&`) ter ali (`|`) moramo paziti, da jih ne mešamo z logičnimi operatorji in (`&&`) ter ali (`||`). Bitna operatorja sicer lahko vračata enak rezultat kot logična, a le kadar operiramo samo z vrednostma 0 in 1.

```
1&1  // vrne 1
1&&1 // vrne 1

1&0  // vrne 0
1&&0 // vrne 0

1&2  // vrne 0
1&&2 // vrne 1

5|2  // vrne 7
5||2 // vrne 1
```

Eniški komplement (*one's complement*) dobimo tako, da vse ničle spremenimo v enke in obratno. Tako je `~10` enako `245` (`~00001010` je `11110101`), če operiramo z osmimi biti (ustrezen tip bi bil lahko `unsigned char`).

Operator pomika bitov (*bit shift left/right*) premakne bite levega operanda v levo oz. desno za število mest, podano v desnem operandu. Pri tem se prazni biti (na desni oz. na levi) nastavijo na nič. Ker gre za pomike bitov binarnega števila, je operacija pomika v levo za ena pravzaprav enaka množenju z 2, operacija pomika v desno za ena pa celoštevilskemu deljenju z 2. Tako je `11<<2` enako `00001011<<2` (število `11` v

binarnem zapisu), kar pomeni, da številu `00001011` na desni dodamo dve ničli, prva dva bita na levi pa zanemarimo (odrežemo). Dobimo rezultat `00101100`, kar je `44` (`11` krat `2` krat `2`). S pomikom v desno `11>>2` pa številu `00001011` dodamo dve ničli na levi strani, zanemarimo pa zadnja dva bita na desni. Tokrat dobimo `00000010`, kar je `2` (`11` deljeno z `2` je `5`, `5` deljeno z `2` pa je `2`). Operacije pomika bitov so veliko hitrejše od operacij množenja in deljenja, zato jih je smiselno uporabljati namesto slednjih dveh povsod tam, kjer želimo hitro računanje.

Prireditveni operatorji

Osnovni prireditveni operator je enačaj (=), ki vrednost izraza na desni priredi spremenljivki na levi.

```
x = 5;
y = 5 + 11;
```

Ostali prireditveni operatorji pa poleg prireditve izvedejo tudi aritmetično ali bitno operacijo nad spremenljivko, rezultat pa shranijo nazaj v to spremenljivko. Tako sta na primer spodnja izraza enakovredna glede končnega rezultata, le da je drugi izraz učinkovitejši:

```
x = x + 5;
x += 5;
```

Prireditveni operator zapišemo tako, da znaku za aritmetično oz. bitno operacijo sledi še enačaj. Na levi strani operatorja je podana spremenljivka, ki nastopa v vlogi prvega operanda in hkrati tudi prejemnika rezultata operacije, na desni pa drugi operand.

Operator	Opis	Primer
`=`	prireditev	`x=5`
`+=`	prištevanje in prireditev	`x+=5`
`-=`	odštevanje in prireditev	`x-=5`
`*=`	množenje in prireditev	`x*=2`
`/=`	deljenje in prireditev	`x/=2`
`%=`	ostanek pri deljenju in prireditev	`x%=2`
`<<=`	pomik v levo in prireditev	`x<<=1`
`>>=`	pomik v desno in prireditev	`x>>=1`
`&=`	bitni in ter prireditev	`x&=y`
`|=`	bitni ali ter prireditev	`x|=y`
`^=`	bitni ekskluzivni ali ter prireditev	`x^=y`

Tabela 11.6: Prireditveni operatorji.

Pri prireditvenem operatorju (=) moramo biti pozorni, da ga pomotoma ne uporabimo namesto primerjalnega operatorja ==. Tudi operator = namreč vrača vrednost, ki je enaka vrednosti, ki jo prirejamo (to je vrednost izraza na desni). Tako je stavek `if(i=j)` povsem legalen in pogoj je izpolnjen vedno, kadar je vrednost `j` različna od nič (vrednost `j` se priredi spremenljivki `i` ter se vrne kot rezultat izraza). Seveda je to nekaj povsem drugega kot `if(i==j)`.

Posebni operatorji

Operator	Opis	Primer
`?:`	pogojni operator	`x<0?-x:x`
`,`	zaporedje	`x,y`
`&`	naslov	`&x`
`*`	kazalec	`*x`
`sizeof()`	velikost objekta/tipa	`sizeof(int)`
`.`	element strukture	`s.x`
`->`	el. str. preko kazalca	`k->x`
`[]`	element polja	`p[10]`
`()`	klic funkcije	`f(3)`
`(tip)`	pretvorba tipa	`(float)x`

Tabela 11.7: Posebni operatorji.

Pogojni operator poznamo že iz Jave. Izraz `P?A:B` najprej izračuna vrednost izraza `P` in če je ta enaka resnično (različna od nič), potem izračuna vrednost izraza `A` in jo vrne kot rezultat celotnega izraza. Če pa je vrednost izraza `P` enaka neresnično (enaka nič), potem se izračuna vrednost izraza `B` ter vrne kot rezultat celotnega izraza.

Operator vejica nastopa najpogosteje kot ločilo med parametri funkcije ali ločilo med posameznimi podatkovnimi tipi:

```
int a, b=5, c;
funkcija(a, b, c, 1);
```

Vejica je tudi operator zaporedja, ki ga npr. najdemo v stavku `for` (izvedejo se vse operacije po vrsti):

```
for(i=1, j=1, a=10; i<=a; i++, j+=5)
  ;
```

V splošnem ima operator vejica obliko `izraz1, izraz2`. Pri tem se izračunata oba izraza, celoten izraz pa dobi vrednost in tip desnega izraza.

Prioritete operatorjev

Vsak operator ima določeno tudi prioriteto, ki določa vrstni red združevanja operatorjev. Operatorji z višjo prioriteto imajo prednost pred operatorji z nižjo. Operatorji z enako prioriteto se združujejo po vrsti od leve proti desni (razen pri nekaj izjemah, ko je vrstni red od desne proti levi).

Če želimo ta privzet vrstni red združevanja operatorjev spremeniti, uporabimo okrogle oklepaje `()`. Te lahko uporabimo tudi takrat, ko nismo prepričani o prioriteti ali pa ko želimo jasno poudariti prioritete v izrazu in ga s tem narediti bolj preglednega in lažje berljivega.

V spodnji tabeli so zapisani operatorji po prioritetah. V prvi vrstici tabele so operatorji z najvišjo prioriteto (ti najbolj vežejo), v naslednjih vrsticah pa prioriteta pada do zadnje vrstice, v kateri je operator z najnižjo prioriteto. Posamezni operatorji, ki so navedeni v isti vrstici, imajo enako prioriteto. Pri njih je vrstni red združevanja od leve proti desni, če v tabeli ni drugače navedeno.

Operatorji	Vrstni red združevanja; komentar
`() [] -> .`	
`! ~ + - * & sizeof (tip) ++ --`	z desne na levo; vsi operatorji so unarni
`* / %`	
`+ -`	
`<< >>`	
`< <= >= >`	
`== !=`	
`&`	
`^`	
`|`	
`&&`	
`||`	
`?:`	z desne na levo
`= += -= *= /= %= &= ^= |= <<= >>=`	z desne na levo
`,`	

Tabela 11.8: Prioritete operatorjev.

V splošnem jezik C ne predpisuje, v kakšnem vrstnem redu se izračunajo operandi operatorja in tudi vrstni red izračuna izrazov je nedoločen. Vrstni red izračunov je natančno določen le pri izrazih z logičnimi operatorji. Sicer pa prevajalnik sam izbere najbolj optimalen vrstni red. Tako se lahko na primer izrazi z asociativnimi in komutativnimi operatorji pred izračunom poljubno preuredijo, ne glede na postavljene oklepaje. Seštevanje `a+(b+c)` se lahko na primer izračuna tudi kot `(a+b)+c`. V večini primerov je to tako ali tako vseeno. Kadar pa želimo točno določen vrstni red izračuna, moramo uporabiti pomožne spremenljivke.

Podobno tudi ni določen vrstni red izračuna posameznih funkcijskih parametrov. Zato lahko v primerih, kot so spodnji, dobimo nepredvidljive rezultate (vrstni red vrednotenja izrazov je odvisen od prevajalnika in je lahko povsem drugačen od pričakovanega).

```
//ali se x poveča pred izračunom x*x ali po njem?
printf("%d %d \n", ++x, x*x);

//ali se prva kliče funkcija f ali funkcija g?
y = f() + g();
```

Kadar smo v dvomih, je bolje zapisati kodo nekoliko drugače, a nedvoumno:

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

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

Stavki

Izvajanje programa se vedno začne v funkciji `main`, v kateri se stavki izvajajo zaporedno, kot so zapisani. Med najenostavnejše stavke spadajo prireditveni stavki in klici funkcij, ki jih lahko zapišemo kot:

```
izraz;
```

Stavek je lahko tudi prazen:

```
;
```

Več stavkov lahko sestavimo v blok stavkov, če jih obdamo z zavitimi oklepaji:

```
{
  stavek1;
  stavek2;
}
```

Potek programa kontrolirajo in usmerjajo odločitveni stavki in ponavljalni stavki.

Odločitveni stavki

Podobno kot Java pozna tudi jezik C odločitvena stavka `if` in `switch`.

```
if(pogoj)
  stavek;

if(pogoj)
  stavek1;
else
  stavek2;

switch(izraz) {
  case konst1: stavek1;
  case konst2: stavek2;
  ...
  default: stavekn;
}
```

Stavek `switch` deluje podobno kot v Javi. Odvisno od vrednosti izraza `izraz` (ta mora biti tipa `int`) se izvrševanje programa nadaljuje pri stavku, ki je označen s konstantno oznako, katere vrednost je enaka izrazu `izraz`. V primeru, da take oznake ni, se nadaljuje pri oznaki `default`, če ta v stavku obstaja. Ker same konstantne oznake ne spreminjajo toka programa, navadno uporabimo stavek tudi `break`.

Zanke

Tudi ponavljalni stavki ali zanke v jeziku C so podobni javanskim. Ločimo tri vrste zank, to so `for`, `while` in `do-while`. Zanki `for` in `while` sta popolnoma enakovredni.

Zanka `do-while` pa se od zanke `while` razlikuje po tem, da se pogoj za ponovitev zanke preverja na koncu, ko so se stavki v telesu zanke vsaj enkrat že izvedli.

Najenostavnejša je zanka `while`, pri kateri se stavek ponavlja, dokler je izračunana vrednost pogoja različna od nič (resnična).

```
while(pogoj)
  stavek;
```

Zanko `for` določajo trije izrazi (ki pa so lahko tudi prazni). Prvi je `inicializacija` in se izvede pred vstopom v zanko. Drugi je `pogoj`, katerega vrednost se preveri pred vsako ponovitvijo zanke (tudi pred prvo). Zanka se ponavlja, dokler je pogoj izpolnjen (njegova vrednost je različna od nič). Tretji izraz je `korak` in se izvrši po vsaki ponovitvi zanke (a še pred ponovnim preverjanjem pogoja).

```
for(inicializacija; pogoj; korak)
  stavek;
```

Zgornjo zanko `for` bi lahko enakovredno zapisali z zanko `while`:

```
inicializacija;
while(pogoj) {
  stavek;
  korak;
}
```

Pri zanki `do-while` se stavek v telesu zanke vsaj enkrat izvrši, saj se pogoj za ponovitev zanke izračuna šele na koncu zanke. Zanka se ponavlja, dokler je izračunana vrednost pogoja različna od nič.

```
do {
  stavek;
} while(pogoj);
```

V povezavi z zankami pogosto uporabljamo stavke `break`, `continue` in `goto`, ki vplivajo na izvajanje zanke. Stavek `break` takoj prekine najbolj notranjo zanko (oziroma `switch` stavek). Stavek `continue` povzroči naslednjo ponovitev zanke. Stavek `goto` je pa brezpogojni skok na označeno mesto znotraj iste funkcije. Slednji je manj zaželen zaradi nepreglednosti programa ob njegovi uporabi, poleg tega pa problem lahko največkrat elegantneje rešimo z uporabo zank, odločitvenih stavkov ter stavkov `break` in `continue`.

12. Prvi preprosti programi

V prejšnje poglavju smo spoznali osnove jezika C in sedaj lahko poskusimo napisati tudi kakšen program. 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 zanko `while` našega programa, dobimo naslednjo zanko:

```
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 ponovno opozorimo 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* (|).

Koda tako spremenjenega programa je v datoteki `prepis2.c`.

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 številke, 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).

```
int stc = 0;
```

Ob vsakem prehodu zanke (ko preberemo en znak) pa vrednost spremenljivke povečamo za ena. V zanko torej dodamo tudi povečanje števca prebranih znakov:

```
stc++;
```

Na koncu, ko se zanka izteče, dodamo še stavek `printf`, ki izpiše vrednost števca `stc`.

```
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.

```
int stm, stv;
stm = stv = 0;
```

Oba števca ustrezno povečujemo v zanki. 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.

```
if((c>='A') && (c<='Z')) {  // ali je velika crka
  putchar(c + 'a'- 'A');
  stv++;  // stejemo velike crke
}
```

Več dodatnega 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 številka 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.

```
else {
  if((c>='a') && (c<='z')) // ali je mala crka
    stm++;  // stejemo male crke
  putchar(c);
}
```

Na koncu moramo seveda še dopolniti izpis in izpisati vrednosti obeh števcev črk.

```
printf("Malih crk je %d, velikih pa %d.\n", stm, stv);
```

Celoten program je shranjen v datoteki `prepis3.c`.

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
```

Opozoriti pa moramo še na eno stvar. 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 `' '`).

```
int zadnjic;
zadnjic = 0;
```

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.

```
zadnjic = c;
```

Če vse povedano zapišemo skupaj, dobimo program v datoteki `presledki.c`.

13. 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 (izven funkcije). 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 (kazalci) na ustrezne spremenljivke. V takem primeru funkciji kot argument podamo le naslov spremenljivke, preko katerega je ta spremenljivka dostopna tudi v sami funkciji.

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 kazalec `n`, vrednost 0. Z drugimi besedami, ta stavek postavi na nič vrednost spremenljivke `stevilo`. Ko se funkcija zaključi, spremenljivka `n` sicer 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, preko nove spremenljivke `p`, ki hrani vrednost naslova spremenljivke `stevilo` (torej je `p` kazalec na spremenljivko `stevilo`). Naslednji stavki so enakovredni zgornjima dvema:

```
int stevilo;
int *p;

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

Delovanje opisane funkcije si lahko pogledamo na naslednjem primeru (program `kazfun.c`), kjer spremenljivki `stevilo` priredimo vrednost 5, nato pa pokličemo funkcijo `init`, ki spremenljivki spremeni vrednost na 0. Vsakokratno vrednost spremenljivke preverimo z izpisom:

```
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; navadno 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 `strcpy(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 ne 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:

```
strcpy(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'`. Če je prvi znak niza shranjen v `niz[0]`, potem je zadnji znak niza shranjen v `niz[strlen(niz)-1]`. Ker je vsak niz zaključen z znakom `'\0'`, je vrednost `niz[strlen(niz)]` vedno enaka 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.

Uporaba nizov je prikazana tudi na primeru (datoteka `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.

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 15. poglavju).

14. 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 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).

```
int n = 0;
float stevilo = 0.0;
char niz[20];

scanf("%s in %d %f", niz, &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`).

Pri branju nizov znakov s funkcijo `scanf` moramo biti previdni, saj pri formatu "`%s`" preberemo celoten niz znakov (torej vse znake do prvega presledka oz. praznega znaka) ne glede na njegovo dolžino. Prebran niz se shrani v ustrezno spremenljivko, ki je tipa kazalec na znak (`char *`) in kaže na polje znakov, ki mora biti dovolj veliko, da lahko vanj shranimo prebran niz skupaj z zaključnim znakom `'\0'`. Za pravo velikost polja znakov moramo seveda poskrbeti sami. To pa je v nekaterih primerih zelo težko, saj ne moremo vedeti, kako dolg niz lahko uporabnik programa dejansko vpiše. Zato je pri uporabi funkcije `scanf` za branje nizov varneje uporabiti še določilo, ki pove največje število prebranih znakov niza. To je celo število, ki ga v formatu zapišemo med znaka `%` in `s` (npr. "`%8s`" za največ 8 znakov). V tem primeru se branje ustavi po določenem številu prebranih znakov, vsi ostali že vpisani znaki pa pridejo na vrsto pri naslednjem branju.

Primer branja niza z omejitvijo dolžine prebranega niza na 50 znakov prikazuje naslednja koda:

```
char niz[51];
scanf("%50s", niz);
```

Za prebran niz smo rezervirali prostor za skupaj 51 znakov, saj potrebujemo prostor za 50 prebranih znakov ter še za zaključni znak niza (znak `'\0'`).

Primer uporabe formatiranega branja in izpisa

Poglejmo si še en primer uporabe branja podatkov in izpisa rezultata. Program naj prebere dve celi števili in izpiše njuno vsoto (datoteka `sestej.c`). Čeprav je program zelo enostaven, predstavlja uporabo obeh v tem poglavju opisanih funkcij v praksi. Kot vidimo v primeru, so lahko argumenti funkcije `printf` tudi izrazi.

```
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);
```

Pisanje (branje) formatiranih podatkov v niz

Funkcijama `printf` in `scanf` sta zelo podobni tudi funkciji `sprintf` in `sscanf`, s to razliko, da slednji delata z nizi namesto s standardnim izhodom oziroma s standardnim vhodom. Zato jima moramo kot prvi parameter podati niz (oziroma kazalec na začetek niza), v katerega se zapiše rezultat pri `sprintf` oziroma iz katerega se preberejo podatki pri `sscanf`.

Primer uporabe funkcije `sprintf` prikazuje naslednja koda:

```
int dan = 11;
int mesec = 2;
int leto = 2009;
char datum[11];
sprintf(datum, "%d. %d. %02d", dan, mesec, leto%100);
```

S pomočjo funkcije `sprintf` smo v prikazanem primeru sestavili ustrezno formatiran niz `datum`, ki ima naslednjo obliko `"11. 2. 09"` (izpis `09` smo dosegli tako, da smo izpisali le zadnji dve števki spremenljivke `leto`, pri njenem formatu pa smo določili izpis na dve mesti z vodilnimi ničlami). Za niz `datum` smo rezervirali prostor za skupaj 11 znakov: največ dva znaka za vsako od treh števil, dva znaka za piki, dva znaka za presledka za pikama ter še en znak za konec niza.

Pri uporabi funkcije `sprintf` moramo vedno paziti, da je za niz rezervirano dovolj prostora, da lahko vanj zapišemo celoten rezultat formatiranja podatkov (pri pisanju na standardni izhod tega problema nismo imeli).

Kot vidimo, lahko funkcijo `sprintf` uporabljamo tudi za pretvarjanje števila v niz znakov. Nasprotno pa za pretvorbo niza znakov v število lahko uporabimo funkcijo `sscanf`, ki ima podoben učinek kot funkciji `atoi` ali `atof`, ki ju bomo spoznali v naslednjem poglavju. Niz, ki predstavlja realno število, lahko v število pretvorimo na naslednji način:

```
float pi;
sscanf("3.1415", "%f", &pi);
```

Spodnji primer pa prikazuje, kako lahko iz formatiranega niza izluščimo le želena tri cela števila:

```
int dan, mesec, leto;
char danes[] = "20.1.2010";
sscanf(danes, "%d.%d.%d", &dan, &mesec, &leto);
```

Podobno kot pri branju s standardnega vhoda tudi funkciji `sscanf` podamo naslove prebranih števil.

15. 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]);
}
```

Program prevedemo v izvršljivo datoteko `argumenti`:

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

in ga nato poženemo z ukazom:

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

Ob klicu programa smo podali tri argumente, to so `prvi`, `drugi` in `tretji` po vrsti.

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 (imenovane tudi stikala), 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 vsebino trenutnega direktorija. 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 `argv[i][j]` moramo preveriti, 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 lahko zadovoljimo s tem, da izpišemo vsako od ugotovljenih opcij. Program najdete v datoteki `opcije.c`.

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 argument (na prvem mestu je ime programa, ki ga izpustimo):

```
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 kazalec `s` (znak `*s`).

Celoten program najdete v datoteki `opcije1.c`.

16. Knjižnice

Jezik C je zelo majhen. Veliko funkcij, ki jih nudijo drugi programski jeziki, ni del osnovnega jezika C, ampak jih dobimo v bogatem naboru knjižnic. Tako večina implementacij jezika C vključuje standardne knjižnice funkcij, ki dopolnjujejo jezik z dodatnimi uporabnimi funkcijami in jih praktično lahko štejemo za del jezika (čeprav so navadno odvisne tudi od platforme). V njih najdemo funkcije za delo z nizi, matematične funkcije, vhodno-izhodne funkcije in še celo vrsto drugih.

Uporaba knjižnic

Kot smo že omenili v 4. poglavju, prevajanje z `gcc` obsega štiri faze: predprocesiranje (izhod je izvorna koda C brez komentarjev), samo prevajanje (izhod je zbirna koda), zbiranje (izhod je objektna koda) in povezovanje (sestavi datoteke z objektno kodo, tudi tiste iz knjižnic, v izvršljiv program). Te faze vedno potekajo v navedenem vrstnem redu. Program `gcc` pravzaprav ne prevaja, temveč le kliče ustrezne programe, ki poskrbijo za izvedbo posameznih faz.

Če uporabljamo knjižnice, jih moramo ob prevajanju tudi povezati z našim programom (to navadno ne velja za standardno knjižnico C, ki je shranjena v `/usr/lib/libc.a`, in za matematično knjižnico `/usr/lib/libm.a`, saj se ti knjižnici privzeto povežeta). Za povezovanje knjižnic v naš program uporabimo opcijo `l` (*link*) prevajalnika `gcc`:

```
gcc program.c –lIME –o program
```

V zgornjem primeru povemo prevajalniku `gcc`, da naj prevede datoteko `program.c` in v fazi povezovanja v `program` poveže tudi knjižnico z imenom `IME`. To knjižnico, katere datoteka se imenuje `libIME.a`, poišče na standardnih mestih (seznam standardnih sistemskih direktorijev za knjižnice; na sistemih Linux sta to navadno direktorija `/usr/lib` in `/lib`). Knjižnice so navadno arhivske datoteke (zato imajo končnico `.a`), katere sestavljajo objektne datoteke (slednje imajo končnico `.o`). Vrstni red objektnih datotek in knjižnic, ki jih navedemo programu `gcc`, je zelo pomemben, saj jih povezovalnik dodaja po vrsti, kot so zapisane. Po vrsti namreč pregleduje vhodno vrstico in uporablja, kar potrebuje: vsakič, ko naleti na objektno datoteko, jo doda v program, ko pa naleti na knjižnico, doda v program le tiste njene objektne datoteke, ki jih na tem mestu potrebuje. Zato navadno knjižnice navedemo za vsemi ostalimi objektnimi datotekami.

Da lahko v našem programu uporabimo sistemski klic ali funkcijo iz knjižnice, moramo v program vključiti ustrezno zaglavno datoteko (*header file*), kot so na primer nam že znane `string.h`, `math.h` ali `stdio.h`, v kateri je deklariran prototip te funkcije. Potem lahko funkcijo iz knjižnice uporabljamo na enak način, kot da bi jo napisali sami v programu. V zaglavnih datotekah so navadno zapisani le prototipi funkcij, definicije podatkovnih tipov in predprocesorski ukazi.

Poglejmo si še primer uporabe knjižnice funkcij za krmiljenje zaslona. Te se nahajajo v knjižnici `curses`, ki zajema funkcije za terminalsko neodvisno pisanje (risanje) na zaslon. Kot primer napišimo enostaven program `tipke.c`, ki najprej zbriše zaslon, nato izpiše navodilo v peto vrstico z začetkom v drugem stolpcu ter sprejema uporabnikov vnos, dokler ne dobi črke k. Pri tej nalogi smo uporabili kar nekaj funkcij, ki se nahajajo v paketu `curses` (`initscr` za inicializacijo zaslona, `clear` za brisanja zaslona, `move` za premik kurzorja in podobno; kratki opisi funkcij so v komentarju programa). Zato moramo v program vključiti tudi zaglavno datoteko `curses.h`, v kateri so definirani prototipi teh funkcij, ki določajo njihovo uporabo (število in tip argumentov ter tip vrednosti, ki jo funkcija vrača).

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

int main() {
  initscr(); //inicializacija zaslona
  noecho(); //ne izpisuj vtipkanih znakov
  clear(); //briši zaslon
  move(5,2); //prestavi kurzor v 5. vrstico, 2. stolpec
  printw("Pritisni tipko k za konec ... "); //izpis niza
  while(getch() != 'k') //preberi znak
    ;
  endwin(); // zaključek curses
  printf("Konec krmiljenja zaslona\n");
  return 0;
}
```

Seveda mora biti v program vključena tudi zaglavna datoteka `stdio.h`, ki definira prototip funkcije `printf`.

Program poskusimo prevesti:

```
gcc tipke.c -o tipke
```

Pri tem nam `gcc` vrne več napak (nedefinirane reference na funkcije), saj funkcije, ki smo jih uporabili za krmiljenje zaslona (t.j. `initscr`, `noecho` ...), niso definirane niti v našem programu niti v standardni knjižnici `libc.a`, ki se privzeto poveže v izvajalni program. Bodite pozorni, da podobne napake ne javi za funkcijo `printf`, saj ta funkcija je definirana v standardni knjižnici.

Torej moramo prevajalniku `gcc` eksplicitno povedati, naj v izvajalni program `tipke` poveže tudi knjižnico `curses`. Ta se tipično (na sistemih Linux) nahaja v datoteki `/usr/lib/libcurses.a`, pripadajoče prototipne deklaracije funkcij te knjižnice pa so navedene v zaglavni datoteki `/usr/include/curses.h`.

Pri prevajanju programa moramo torej uporabiti opcijo `-lcurses`, ki določa, naj se skupaj s programom poveže tudi navedena knjižnica:

```
gcc tipke.c -lcurses -o tipke
```

Zgornji zapis je pravzaprav krajši (in zato pogosteje uporabljan) zapis za:

```
gcc tipke.c /usr/lib/libcurses.a -o tipke
```

Izdelava lastnih knjižnic

Knjižnica je zbirka že prevedenih objektnih datotek, ki jih po potrebi lahko vključimo v naš program. Navadno je shranjena v posebni arhivski datoteki, ki ima končnico `.a`, s katero je določena statična knjižnica. Tako knjižnico lahko izdelamo iz objektnih datotek tudi sami, pri tem pa uporabimo orodje `ar` (*GNU archiver*).

Datoteke, ki jih shranimo v knjižnico, se sicer ne razlikujejo od navadnih datotek z izvorno kodo C, razen da ne vsebujejo funkcije `main`, saj se programi ne začenjajo v njih. Z datotekami knjižnice le razdelimo kodo na več manjših delov, ki jih nato lažje vzdržujemo, beremo in (ponovno) uporabljamo.

Pri izdelavi knjižnice moramo najprej pridobiti objektne datoteke, ki jih bomo vključili vanjo. Če imamo kodo za našo knjižnico v izvornih datotekah `prva.c`, `druga.c` in `tretja.c`, potem s prevajalnikom `gcc` z uporabo opcije `-c` izdelamo ustrezne objektne datoteke:

```
gcc -c prva.c
gcc -c druga.c
gcc –c tretja.c
```

ali

```
gcc -c prva.c druga.c tretja.c
```

Opcija `-c` pove, da se navedena datoteka z izvorno kodo C prevede in zbere, vendar ne poveže (izvedejo se le prve tri faze prevajanja z `gcc`). Rezultat je objektna datoteka, ki ustreza podani datoteki z izvorno kodo (datoteka dobi končnico `.o`).

Rezultat zgornjega prevajanja so torej datoteke `prva.o`, `druga.o` in `tretja.o`, katere lahko potem združimo v knjižnico:

```
ar rvs mojaknj.a prva.o druga.o tretja.o
```

Pri uporabi programa za izdelavo arhivov `ar` smo kot prvi argument podali določila `r`, `v` in `s`, od katerih ima vsako svoj pomen. Sledi ime arhivske datoteke, na koncu pa so navedene datoteke, ki jih dodajamo v arhiv. Opcija `r` določa dodajanje navedenih datotek v arhiv in jih prepiše (zamenja), če v arhivu že obstajajo. Opcija `v` določa, da program sproti izpisuje, kaj dela (za vsako dodano datoteko na primer izpiše, ali jo je dodal ali zamenjal obstoječo). Opcija `s` pa določa, da program `ar` ustvari tudi simbolno tabelo, ki jo potrebuje `gcc` pri uporabi knjižnice. To opcijo moramo uporabiti vedno, kadar ustvarjamo knjižnice objektnih datotek.

Našo knjižnico `mojaknj.a` nato uporabimo tako, da jo enostavno podamo prevajalniku `gcc` kot katerokoli drugo objektno datoteko:

```
gcc program.c mojaknj.a -o program
```

Naši knjižnici pa lahko določimo tudi posebno ime, jo shranimo na standardno mesto ter pri prevajanju nato uporabimo krajši način z uporabo opcije `-l`. Tak način uporabljamo navadno le za sistemske knjižnice (kot je na primer `libc.a`). Če torej želimo ustvariti novo sistemsko knjižnico, moramo ime knjižnice izbrati tako, da se začne z besedo `lib` (na primer `libknj.a`), kar določa sistemsko knjižnico. Nato datoteko shranimo v `lib` direktorij, kjer se nahajajo sistemske knjižnice (`/usr/lib/` na sistemih Linux). Pri prevajanju programa z `gcc` pa našo novo knjižnico enostavno povežemo z uporabo stikala `-lknj`:

```
gcc program.c -lknj -o program
```

Poglejmo si izdelavo lastne knjižnice še na primeru. Pripravimo knjižnico, ki bo nudila funkciji za pretvorbo tolarjev v evre in obratno.

Prva funkcija naj se imenuje `siteur`, druga pa `eursit`. Obe prejmeta kot parameter znesek, ki ga želimo pretvoriti, ter vrneta ustrezno pretvorjen znesek. Ker se tečaj tolarja napram evru ne spreminja več, lahko ustrezen menjalni tečaj zapišemo kot konstanto.

Obe funkciji bi lahko zapisali v datoteko `pretvorba.c` na naslednji način:

```
#include"pretvorba.h"
#define TECAJ 239.64

float siteur(float sit) {
  return(sit/TECAJ);
}

float eursit(float eur) {
  return(eur*TECAJ);
}
```

V program moramo vedno vključiti zaglavno datoteko naše knjižnice. Ta vsebuje prototipe vseh funkcij naše knjižnice, ki jih lahko uporabljamo.

Zaglavna datoteka se imenuje `pretvorba.h`, v njej pa je zapisano naslednje:

```
extern float siteur(float sit);
extern float eursit(float eur);
```

Pred prototip vsake funkcije zapišemo rezervirano besedo `extern`, ki pove, da je ta funkcija definirana v neki drugi datoteki.

Knjižnico `pretvorba.a` pripravimo, kot smo že zapisali na začetku tega razdelka: najprej izvorno datoteko prevedemo v objektno datoteko, to pa potem vstavimo v novo arhivsko datoteko.

```
gcc -c pretvorba.c
ar rvs pretvorba.a pretvorba.o
```

Zapišimo še izvorno kodo programa, v katerem bomo uporabili napisano knjižnico. Ta naj bo shranjena v datoteki `pret.c`:

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

int main() {
  float znesek;
  char valuta[4];

  printf("Vpisi znesek in valuto: ");
  scanf("%f %3s", &znesek, valuta);
  if((strcmp(valuta,"eur") == 0) ||
     (strcmp(valuta,"EUR") == 0))
    printf("%.2f EUR = %.2f SIT\n",znesek,eursit(znesek));
  else if((strcmp(valuta,"sit") == 0) ||
          (strcmp(valuta,"SIT") == 0))
    printf("%.2f SIT = %.2f EUR\n",znesek,siteur(znesek));
  else
    printf("Nepoznana valuta %s\n", valuta);
  return 0;
}
```

Seveda smo morali v program vključiti zaglavno datoteko naše knjižnice, da smo lahko uporabljali funkciji iz knjižnice.

Pri prevajanju programa moramo podati tudi knjižnico, ki jo program uporablja:

```
gcc pret.c pretvorba.a -o pret
```

17. Predprocesor

Predprocesor je program, ki obdela izvorno kodo C-jevega programa pred njenim prevajanjem. Ta program se pokliče kot prvi v verigi prevajanja (predprocesiranju sledijo še prevajanje, zbiranje in povezovanje). Predprocesor vzame izvorno kodo programa, iz nje odstrani komentarje ter interpretira posebne predprocesorske ukaze, ki so označeni z znakom `#` na začetku vrstice. Dva primera teh ukazov, ki smo ju spoznali že v predhodnih poglavjih, sta `#include` in `#define`. Prvi v program vključi vsebino navedene datoteke, drugi pa definira simbolično ime ali konstanto. Na ta način predprocesor pravzaprav definira svoj jezik, ki ga lahko koristno uporabimo pri programiranju, saj omogoča lažji razvoj programov, programi so lažje berljivi, enostavneje jih je spreminjati, koda pa je bolj prenosljiva med različnimi strojnimi arhitekturami.

Predprocesorske vrstice (označene z znakom `#`) lahko zapišemo kjerkoli v programu, čeprav jih navadno postavimo na začetek izvorne datoteke. Veljajo od mesta zapisa pa do konca datoteke. Bodite pozorni na to, da se predprocesorske vrstice ne zaključijo s podpičjem.

Zamenjava

Ukaz `define` uporabljamo za definiranje konstant ali katerihkoli makro zamenjav.

```
#define ime zamenjava
```

Povzroči zamenjavo imena `ime` z nizom `zamenjava` povsod v datoteki, kjer je to ime zapisano.

Tako lahko na primer v programu uporabimo simbolične konstante, ki predstavljajo vrednosti resnično in neresnično:

```
#define FALSE 0
#define TRUE !FALSE
```

Imenu lahko v oklepaju podamo tudi argumente in dobimo makro zamenjavo. Na tak način lahko definiramo tudi enostavne funkcije. Primer prikazuje izračun absolutne vrednosti števila:

```
#define abs(x) ((x) < 0 ? -(x) : (x))
```

Kot smo že omenili, velja z ukazom `define` definirana zamenjava od mesta ukaza do konca izvorne datoteke. Če želimo doseg zamenjav skrajšati (da ne velja do konca datoteke), uporabimo ukaz `undef`. Tako `#undef ime` pove predprocesorju, da se prej definirana zamenjava za `ime` ne uporablja več.

Vključitev datotek

Ukaz `include` povzroči vstavitev vsebine podane datoteke.

```
#include "datoteka"
#include <datoteka>
```

Če ime datoteke podamo v narekovajih, predprocesor išče datoteko najprej na tekočem direktoriju in če je ne najde, dalje na standardnih mestih (kjer ima sistem shranjene zaglavne datoteke). Če pa ime datoteke podamo med znakoma <>, predprocesor išče datoteko le na standardnih mestih (izpusti iskanje po tekočem direktoriju).

Na tak način ponavadi v program vključimo zaglavne datoteke (*header files*) raznih knjižnic, ki jih uporabljamo v programu.

Pogojno prevajanje

Predprocesorske ukaze lahko uporabimo tudi za kontrolo nad vključevanjem dela kode v program v času prevajanja. Za to uporabimo if-else konstrukt.

```
#if izraz
  // ti stavki v C se vključijo, če je izraz resničen
#else
  // ti stavki v C se vključijo, če je izraz neresničen
#endif
```

Ukaz `if` ovrednoti podan konstantni celoštevilski izraz `izraz` in preveri, ali je njegova vrednost različna od nič. Če je izraz resničen (vrednost različna od nič), se v program vključijo vsi stavki, ki sledijo ukazu `if` (do ukaza `else`, `elif` ali, če ni nobenega od teh dveh, do `endif`), stavki med `else` in `endif` (če ta del obstaja) pa se ignorirajo. Ukaz `if` se mora vedno zaključiti z ukazom `endif`. Uporaba ukazov `else` in `elif` (else if) je opcijska.

Podoben ukaz je `ifdef`, pri katerem predprocesor preveri, ali je podano ime že definirano z ukazom `define`. Podobno `ifndef` preveri, ali podano ime še ni definirano:

```
#ifdef ime
  // stavki v C se vključijo v program,
  // če je ime že definirano
#endif

#ifndef ime
  // stavki v C se vključijo v program,
  // če ime še ni definirano
#endif
```

Poglejmo si še dva praktična primera uporabe navedenih ukazov. V prvem primeru bomo definirali konstanto `DEBUG`, ki nam pove, ali program izvajamo v razhroščevalnem načinu delovanja (vrednost 1) ali v normalnem delovanju (vrednost 0).

V prvem primeru bomo v program dodali več kontrolnih izpisov, ki nam bodo pomagali pri iskanju napak v kodi. V drugem primeru pa naj program ne bi vseboval navedenih kontrolnih izpisov.

```
#define DEBUG 1
```

Potem lahko v programu preverimo, ali je konstanta `DEBUG` že definirana in jo v nasprotnem primeru definiramo z vrednostjo 0:

```
#ifndef DEBUG
  #define DEBUG 0
#endif
```

Nadalje lahko kodo programa zapišemo takole:

```
#if DEBUG
  printf("Kontrolni izpis pri razhroščevanju\n");
  printf("Izpis vrednosti spremenljivk ...\n");
#endif
```

Drugi primer uporabe predprocesorskih ukazov pa prikazuje uporabo konstante `VELIKOST_TABELE` za določitev velikosti tabele celih števil. Vrednost te konstante je že določena nekje v programu, a želimo, da ta vrednost ne bi bila večja od 500 ali manjša od 100, če pa je med obema navedenima vrednostima, pa naj bo 300. Za ustrezne vrednosti poskrbimo s ponovnim definiranjem konstante:

```
#if VELIKOST_TABELE>500
  #undef VELIKOST_TABELE
  #define VELIKOST_TABELE 500
#elif VELIKOST_TABELE <100
  #undef VELIKOST_TABELE
  #define VELIKOST_TABELE 100
#else
  #undef VELIKOST_TABELE
  #define VELIKOST_TABELE 300
#endif

int tabela[VELIKOST_TABELE];
```

18. 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 (po potrebi skupaj z njeno potjo). 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 (in tako 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 nje in podobno.

Zgoraj navedeni načini odpiranja datotek veljajo za tekstovne datoteke, v katerih so zapisani znaki, urejeni po vrsticah. Primer take datoteke je datoteka z izvorno kodo C, ki jo lahko odpremo v urejevalniku besedil in preberemo njeno vsebino. V splošnem pa so datoteke lahko tudi binarne; v njih so podatki zapisani kot zaporedje bajtov, ki se ponavadi interpretirajo drugače kot znaki. Primer binarne datoteke je datoteka z izvajalno kodo programa v C. Kadar želimo delati z binarnimi datotekami, moramo pri odpiranju datoteke dodati znak `b` na konec niza, ki določa način odpiranja. Tako v funkciji `fopen` kot način odpiranja binarne datoteke podamo niz `"rb"` za binarno branje, `"wb"` za binarno pisanje ter `"ab"` za binarno dodajanje. Seveda lahko v primeru, ko naš sistem ne razlikuje med tekstovnimi in binarnimi datotekami, oboje obravnavamo enako, to je kot tekstovne datoteke.

Poleg tega pa lahko pri načinu odpiranja datoteke k nizu dodamo še znak `+`, ki določa, da datoteko odpremo sočasno za branje in pisanje. Tako z `"r+"` odpremo tekstovno datoteko za branje in pisanje (tok je postavljen na začetek datoteke), z `"w+"` ustvarimo

tekstovno datoteko za branje in pisanje (če datoteka obstaja, se njena vsebina zbriše) ter z `"a+"` ustvarimo ali odpremo tekstovno datoteko za dodajanje (tok je postavljen na konec datoteke). Podobno velja tudi pri odpiranju binarnih datotek (`"r+b"`, `"w+b"` ter `"a+b"`).

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

```
fclose(fp);
```

Branje tekstovne datoteke oziroma pisanje v tekstovno 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 znakov `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 14. 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).

Za branje oz. pisanje večje količine podatkov naenkrat pa lahko uporabimo tudi funkciji `fread` in `fwrite`:

```
fread(polje, velikost, število, fp);
fwrite(polje, velikost, število, fp);
```

Funkcija `fread` prebere iz datoteke `fp` določeno število elementov (število elementov določa parameter `število`, parameter `velikost` pa pove velikost posameznega elementa) ter jih zapiše v pomnilnik, začenši z lokacijo `polje` (spremenljivka `polje` je torej kazalec na podatke v pomnilniku). Funkcija vrne število prebranih elementov, kar je ob napaki lahko tudi manj kot vrednost `število`.

Podobno funkcija `fwrite` zapiše `število` elementov velikosti `velikost` iz polja, na katerega kaže `polje`, v datoteko `fp` ter vrne število uspešno zapisanih elementov.

Če želimo na primer iz datoteke `fp` prebrati 100 znakov naenkrat (ne glede na znake za novo vrstico), lahko to enostavno naredimo na naslednji način:

```
char znaki[100];
fread(znaki,1,100,fp);
```

V zgornjem primeru smo v funkciji določili, da želimo prebrati 100 elementov (znakov), kjer je vsak element velikosti 1 znak (1 bajt), ter jih zapisati v pomnilnik v polje `znaki`.

Ker lahko s funkcijo `fread` (`fwrite`) preberemo (zapišemo) zaporedje poljubnih elementov, jo lahko uporabimo za branje (zapisovanje) poljubnih podatkovnih tipov, tudi struktur. V spodnjem primeru v datoteko najprej zapišemo vrednost spremenljivke `bruc`, to je enega elementa `struct student` (velikost tega elementa določimo kar z uporabo funkcije `sizeof`), nato pa iz datoteke preberemo ta element ter ga shranimo v spremenljivko `absolvent`. Pri tem bodite pozorni, da funkcijama `fread` in `fwrite` podamo naslova spremenljivk `bruc` in `absolvent`, saj obe funkciji kot prvi argument zahtevata naslov, na katerem se začnejo brati oz. zapisovati podatki.

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

struct student bruc;
...
fwrite(&bruc, sizeof(struct student), 1, fp);

...

struct student absolvent;
fread(&absolvent, sizeof(struct student), 1, fp);
```

S pomočjo funkcije `fread` pa lahko v pomnilnik preberemo tudi celo datoteko naenkrat. Seveda moramo najprej ugotoviti velikost te datoteke (spremenljivka `vel`), da lahko v pomnilniku rezerviramo ustrezno velik prostor za vse prebrane podatke. Potem

za branje cele datoteke zadostuje en sam klic funkcije `fread`, s katerim preberemo `vel` znakov (velikost vsakega znaka je 1) ter jih shranimo na zanje rezerviran prostor, na katerega kaže `vsebina`:

```
long vel;
char *vsebina;

fseek(fp, 0, SEEK_END);
vel = ftell(fp);
rewind(fp);

vsebina = (char*) malloc(sizeof(char)*vel);

// preberemo vel znakov (velikosti 1)
fread(vsebina, 1, vel, fp);
```

V zadnjem primeru smo za ugotovitev velikosti datoteke uporabili funkcijo `fseek` v kombinaciji s funkcijo `ftell`. Prva funkcija nastavi pozicijo v datoteki na določeno mesto, v našem primeru gre za odmik 0 od konca datoteke (`SEEK_END`), kar pomeni nastavitev pozicije na konec datoteke. Druga funkcija pa vrne trenutno pozicijo v datoteki, kar je v primeru, ko je pozicija na koncu datoteke, ravno velikost te datoteke. Pozicija v datoteki je določena kar s številom znakov od začetka datoteke. S funkcijo `rewind` se ponovno postavimo nazaj na začetek datoteke, kjer bomo začeli z branjem.

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 12. 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`).

```
while((c=fgetc(fp)) != EOF)
  fputc(c, stdout);
```

Ko zaključimo z izpisom datoteke, jo moramo zapreti.

```
fclose(fp);
```

Če se program uspešno zaključi, to sporočimo z vrnjeno vrednostjo 0 (za razliko od vrednosti 1 oz. 2 ob napaki). Celoten program je zapisan v datoteki `izpisi.c`.

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. Ustrezno dopolnite program `izpisi.c`, da bo njegovo delovanje bolj prijazno.

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.

```
if((fp2=fopen(argv[2],"w")) == NULL) {
  fclose(fp1);  // fp1 je bila uspešno odprta
  exit(1);
}
```

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 podano datoteko v drugo.

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 program v datoteki `izpisi1.c`.

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 je število v datoteki zapisano napačno in ne ustreza realnemu (ali celemu) številu, potem naj program na mestu napake zaključi z branjem.

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;
```

Seveda moramo na začetku programa deklarirati tudi spremenljivko, ki bo hranila vsoto prebranih števil in katere začetna vrednost je 0:

```
double vsota = 0;
```

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 smo dobili vpogled tudi v branje posameznih števil.

```
while(fscanf(fp, "%g", &stevilo) > 0) {
  printf("prebrano stevilo: %g\n", stevilo);
  vsota += stevilo;
}
```

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.

```
printf("Vsota stevil iz %s je %g\n", argv[1], vsota);
```

Program je zapisan v datoteki `vsota.c`. Pri zagonu programa ne pozabite podati datoteke s števili:

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

19. Rekurzija

Programski jezik C dopušča tudi rekurzivno klicanje funkcij, kar pomeni, da lahko funkcija neposredno ali posredno kliče samo sebe. Sam princip rekurzije je opisan v 2. poglavju, tu pa si bomo pogledali še nekaj primerov v jeziku C.

Za začetek ponovimo nekaj osnovnih principov rekurzije. Rekurzivno reševanje problema poteka tako, da se pri rešitvi kompleksnega računskega problema uporabi rešitev enakega, a enostavnejšega problema. To pomeni, da se pri reševanju nekega problema ponovijo isti izračuni, vendar nad manjšim obsegom. Postopek ponavljamo, dokler se obseg problema ne zmanjša do očitne rešitve.

Rekurzivno reševanje problema tipično vključuje:
- opis rešitve problema za enostaven primer, ki ima očitno rešitev, 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 v jeziku C ter primerjali rekurzivno in iterativno rešitev istega problema.

Rekurzivno in iterativno reševanje problema

Poskusimo v 2. poglavju opisan algoritem za izračun fakultete naravnega števila zapisati še v jeziku C. Spomnimo se, da je funkcija definirana na naslednji način:

```
n! = n * (n-1) * (n-2) * ... * 2 * 1
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, f = 1;
  for(i=1; i<=n; i++)
    f = f * i;
  return(f);
}
```

Rekurzivna rešitev je nekoliko krajša in ne vključuje zanke, saj za ponavljanje poskrbi rekurzivni klic. Ta postopek lahko zapišemo tudi v jeziku C:

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

Funkcijo lahko še nekoliko izboljšamo z upoštevanjem, da je tudi `1!` enako `1`, kar vključimo v pogoj stavka `if`:

```
int fakulteta(int n) {
  if(n < 2)
    return(1);
  return(n * fakulteta(n-1));
}
```

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

Ne pozabite, da mora imeti rekurzivna rešitev vedno primeren izstopni pogoj iz rekurzije. V našem primeru je to izračun `0!` (oziroma `1!`), torej ko je `n` enako 0 (oziroma manjše od 2); to je tudi najenostavnejši primer izračuna fakultete. V primeru `n=0` (oziroma `n<2`) namreč ne kličemo več rekurzivno funkcije `fakulteta`, temveč funkcija vrne rezultat 1 in tako zaključi z rekurzivnimi klici.

Rekurzivno definirani problemi

Nekateri problemi, kot je na primer Fibonaccijevo zaporedje, so rekurzivni že po definiciji. Tako je `n`-ti člen Fibonaccijevega zaporedja definiran kot vsota dveh predhodnih členov zaporedja. Pri tem sta prva dva člena zaporedja 0 in 1 po vrsti. Zaporedje lahko zapišemo tudi z naslednjimi formulami, kjer je `n` naravno število:

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

Ker je problem izračuna `n`-tega Fibonaccijevega števila podan rekurzivno, lahko enostavno sestavimo rekurzivno funkcijo, ki izračuna to število za podan `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, pravzaprav smo definicijo le prepisali v programski jezik C. Celoten program najdete v datoteki `fib-rek.c`.

Pa poiščimo še iterativno rešitev tega problema. Rešitev je malo daljša, a vseeno dovolj enostavna: v zanki sproti računamo vsak naslednji člen zaporedja (spremenljivka `f`) tako, da seštejemo dva predhodna člena (spremenljivki `n1` in `n2`), dokler ne izračunamo `n`-tega člena. Pri tem sta prva dva člena zaporedja podana, njuni vrednosti sta 0 in 1.

```
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`.

Po naravi rekurzivni problemi

Veliko problemov lahko rešimo na oba načina, rekurzivno ali iterativno (za podrobnosti glej razdelek *Pretvorba rekurzije v iteracijo* v 2. poglavju). Nekateri problemi so taki, da sta obe rešitvi 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 (implementacija binarnih v jeziku C je opisana v 20. poglavju), problem Hanojskih 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? Namig: poglejte si razdelek o pretvorbi rekurzije v iteracijo v 2. poglavju.

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));
```

ali pa takole:

```
if(palindrom(beseda, strlen(beseda)))
  printf("Beseda %s je palindrom.\n", beseda);
else
  printf("Beseda %s ni palindrom.\n", 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));
}
```

Pri prvem klicu funkcije podamo kot parameter celoten niz, ki ga preverjamo, in njegovo dolžino:

```
int jePalindrom = palindrom("cepec", 5);
```

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`.

```
void izpis() {
  int i;
  for(i=0; i<n; i++)
    printf("%d ", a[i]);
  printf("\n");
}
```

Pri tu zapisani rešitvi smo zaradi enostavnosti predpostavili, da je polje `a` deklarirano kot globalna spremenljivka (skupaj s spremenljivko `n`, ki določa njegovo dolžino) in je zato dostopno tudi v vseh funkcijah.

Funkcija `perm` je sicer malo daljša, a v primerjavi z iterativno rešitvijo vseeno bolj enostavna in razumljiva.

```
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, ki ga najdete v datoteki `permutacije.c` med primeri tega poglavja, je nekoliko daljši, saj poleg funkcij za izpis polja in izračun permutacij zajema tudi 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. Tu polje tudi ni več deklarirano kot globalno, zato moramo funkciji `perm` kot parametra podati tudi polje `a` in njegovo velikost `n`. Funkciji smo dodali tudi števec vseh narejenih (in izpisanih) permutacij elementov.

Izris vzorca iz pik

V naslednjem primeru bomo rekurzijo uporabili namesto ene od zank pri izrisu podanega znakovnega vzorca. Problem bi bil enostavneje rešljiv z iteracijo, a ga sprejmimo kot izziv in dobro vajo za pisanje rekurzivnih rešitev.

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 (program najdete v datoteki `pike.c`):

```
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);
```

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!

20. Povezani seznami in drevesa

V tem poglavju si bomo ogledali še nekatere implementacije abstraktnih podatkovnih tipov v jeziku C. Omejili se bomo na neurejene in urejene povezane sezname ter binarna iskalna drevesa. Tu opisane implementacije naj služijo kot zgled tudi za druge implementacije abstraktnih podatkovnih tipov v jeziku C.

Povezani seznami in drevesa so dinamične podatkovne strukture. Predstavljajo enega od načinov za dinamično shranjevanje podatkov v pomnilniku. Ne glede na število podatkov zavzame povezan 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 z naslednjo kodo (`stara_vel` je tu velikost prvotnega polja `stari`, `vel` pa želena nova velikost polja):

```
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 povezan seznam, ki ga v jeziku C pogosto imenujemo tudi 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 takega seznama temelji na strukturah, zato najprej ponovimo 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. Z njimi lahko implementiramo vse glavne abstraktne podatkovne tipe, ki smo jih spoznali v 3. poglavju. Seveda moramo pri tem upoštevati ustrezna pravila pri implementaciji njihovih operacij (kot je na primer poseben način dodajanja elementa na sklad).

V nadaljevanju si bomo ogledali splošne operacije pri delu s seznami, ki v glavnem zajemajo tudi vse posebne operacije posameznih abstraktnih podatkovnih tipov (dodajanje na sklad lahko na primer enačimo z dodajanjem na začetek seznama, če predpostavimo, da začetek seznama predstavlja vrh sklada).

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 sezname 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 (zato se taki seznami tudi imenujejo kazalčni seznami, saj so elementi povezani s kazalci). Podatkovni del je poljuben, sestavljen je lahko iz enega ali več podatkovnih tipov, ki so lahko tudi polja, strukture ali drugi sestavljeni tipi.

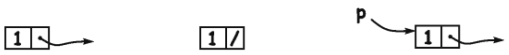
V nadaljevanju bomo zaradi enostavnosti podatkovni del omejili le na eno celoštevilsko vrednost, kar zadostuje za ponazoritev dela s seznami. Element seznama tako opišemo s strukturo `struct el` z dvema komponentama. Prva je `x`, ki predstavlja celoštevilsko vrednost elementa (podatkovni del). Druga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

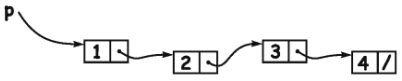
Element seznama navadno simbolično prikažemo z dvodelno škatlo, kjer je v enem delu vpisana vrednost elementa (v našem primeru vrednost spremenljivke `x`), iz drugega dela pa vodi puščica, ki kaže na naslednji element seznama. Naslednje slike prikazujejo nekaj načinov simboličnega prikaza elementa seznama:



Slika 20.1: Simbolični prikaz elementov 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).

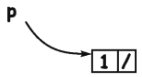
In kako bi simbolično prikazali cel seznam? Škatle z elementi enostavno povežemo skupaj. Vzemimo za primer seznam, ki hrani štiri elemente z vrednostmi `1`, `2`, `3` in `4` po vrsti. Simbolično ga narišemo, kot prikazuje slika:



Slika 20.2: Simbolični prikaz seznama.

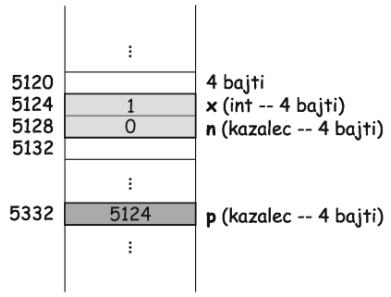
Tu je spremenljivka `p` kazalec na začetek seznama, element z vrednostjo `4` pa je zadnji element seznama.

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:



Slika 20.3: Seznam z enim elementom.

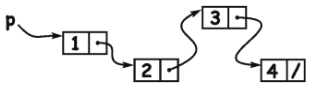
In kako je ta kratek seznam predstavljen v pomnilniku? Spremenljivka `p`, ki je kazalec na strukturo `el`, zavzame v pomnilniku štiri bajte, torej eno lokacijo na sliki 20.4. Na levi strani so napisani naslovi lokacij, ki so seveda za naš primer izmišljeni. Tako se naša spremenljivka `p` nahaja na naslovih 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.



Slika 20.4: Predstavitev seznama z enim elementom v pomnilniku.

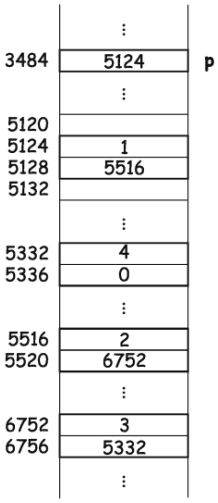
Vrednost spremenljivke `x` je `1`, kazalec `n` pa ne kaže nikamor, ker je to zadnji element v seznamu, zato ima vrednost `0`. Po dogovoru imajo kazalci, ki nikamor ne kažejo, vrednost `0` oziroma `NULL` (slednja je konstanta, definirana v zaglavni datoteki `stdio.h`). Kam pa kaže kazalec `p`? Na prvi 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:



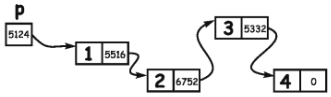
Slika 20.5: Seznam s štirimi elementi.

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 (glej sliko 20.6). 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 sliki 20.6 poudarjeni z močnejšo obrobo, podobno kot tudi začetek seznama.



Slika 20.6: Predstavitev seznama s štirimi elementi v pomnilniku.

Simbolično risbo seznama bi lahko predstavili tudi takole:



Slika 20.7: Simbolični zapis seznama s štirimi elementi (nekoliko drugače).

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, iskanje elementa, brisanje elementa in 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 prvega elementa seznama, temveč tudi do vseh ostalih elementov (torej do celega našega seznama), poleg tega pa bi ti elementi 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 (`p`). 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 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.

Če je spremenljivka `zacetek` kazalec na začetek našega seznama, potem bi ta seznam lahko izpisali z naslednjim klicem funkcije `izpisi`:

```
izpisi(zacetek);
```

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 zgodi prej. Tako bi zanko lahko zapisali:

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

Vrednost vsakega elementa 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 v splošnem 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 v tem primeru dostop do vrednosti `p->vrednost` povzročil napako).

Vrnimo se k funkciji iskanja elementa v seznamu, v kateri moramo na koncu še preveriti, ali smo element našli v seznamu, in vrniti 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` lahko torej zapišemo takole:

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

Pri seznamu, na katerega kaže `zacetek`, bi funkcijo za iskanje elementa lahko na primer uporabili v stavku `if`, ki preverja, ali je element z vrednostjo `5` v seznamu:

```
if(poisci(zacetek,5) == 1)
  printf("Element 5 je v seznamu.\n");
else
  printf("Elementa 5 ni v seznamu.\n");
```

V veliko primerov bi bila naša funkcija za iskanje elementa v seznamu nekoliko bolj uporabna, če bi namesto vrednosti 0 ali 1 vračala kar kazalec na najden element v seznamu (oziroma `NULL`, če iskanega elementa ni v seznamu).

Koda take funkcije je še nekoliko bolj preprosta, saj po zaključeni zanki `while` spremenljivka `p` že kaže na najden element (če elementa nismo našli, pa ima `p` vrednost `NULL`). Torej moramo po zaključku zanke le vrniti vrednost kazalca `p`:

```
struct element *poisci1(struct element *p, int vred) {
  while((p != NULL) && (p->vrednost != vred))
    p = p->naslednji;
  return(p);
}
```

Bodite pozorni, da smo morali spremeniti tudi glavo funkcije, saj vrednost, ki jo sedaj funkcija vrača, ni več `int` temveč kazalec na element seznama `struct element *`.

Primer uporabe zadnje funkcije je naslednji:

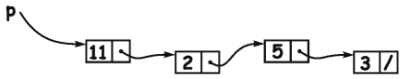
```
struct element *petka = poisci(zacetek, 5);
if(petka == NULL)
  printf("Elementa 5 ni v seznamu.\n");
```

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`).



Slika 20.8: Dodajanje elementa v seznam – začetni seznam.

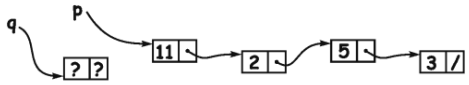
Temu seznamu želimo na začetek dodati nov element z vrednostjo `8`. Preden ta element dodamo, ga moramo seveda ustvariti, torej zanj rezervirati prostor v pomnilniku in mu določiti vrednosti obeh komponent.

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 slika 20.9:

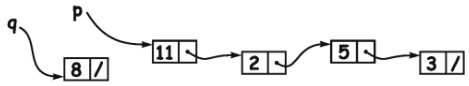


Slika 20.9: Dodajanje elementa v seznam – rezervacija prostora za nov element.

Zaenkrat sta obe komponenti strukture še nedefinirani (na sliki je namesto vrednosti zapisan vprašaj), ker jima še nismo priredili nobene vrednosti. Slednje naredimo z dvema prireditvenima stavkoma:

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

Začasno 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 spodnja slika (20.10):

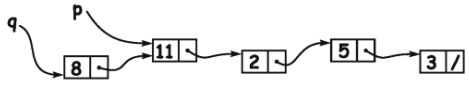


Slika 20.10: Dodajanje elementa v seznam – vpis vrednosti v nov element.

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:

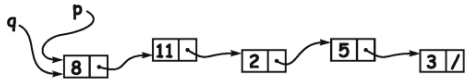


Slika 20.11: Dodajanje elementa v seznam – povezava novega elementa v seznam.

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

```
p = q;
```

Slednje lahko spet prikažemo simbolično (slika 20.12):



Slika 20.12: Dodajanje elementa v seznam – ponastavitev kazalca na začetek seznama.

Sedaj ko poznamo postopek dodajanja 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)
```

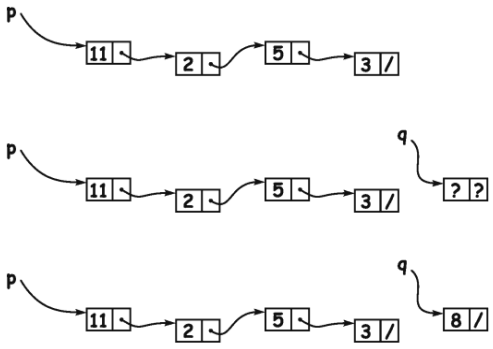
Telo funkcije zajema celoten postopek, ki smo ga že opisali. Pri tem lahko kodo tudi malo skrajšamo tako, da opustimo nepotrebne prireditve (`q->naslednji=NULL` in `p=q`). Funkcija vrne kar kazalec `q`, ki na koncu kaže na začetek celega seznama. Celotna funkcija je naslednja:

```
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 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 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 torej že poznamo in jih ne bomo ponovno opisovali; simbolično so predstavljeni na sliki 20.13.



Slika 20.13: Dodajanje elementa na konec seznama – ustvarjanje novega elementa.

Tu je stavek `q->naslednji=NULL` nujno potreben, saj bo element `q` po dodajanju zadnji element seznama, torej kazalec na naslednji element ne sme kazati nikamor naprej (mora imeti vrednost `NULL`).

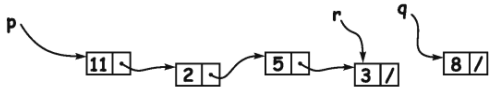
Za povezavo novo ustvarjenega elementa v obstoječi seznam potrebujemo še kazalec na zadnji element v seznamu (imenujmo ga `r`), saj novi element pride za zadnji element v seznam. In kako poiščemo zadnji element seznama? Ker je naša edina referenca le kazalec na začetek seznama `p`, moramo začeti s tem 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 v seznamu, saj takrat velja, da je `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 tako do komponente `r->naslednji` ne smemo dostopati, saj ne obstaja. 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:

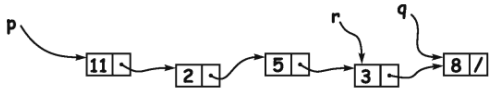


Slika 20.14: Dodajanje elementa na konec seznama – iskanje zadnjega elementa seznama.

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 slika 20.15:



Slika 20.15: Dodajanje elementa na konec seznama – povezava novega elementa v seznam.

Kot vidimo, pri tem kazalec `p` še vedno ostaja kazalec na začetek seznama, saj smo nov element dodali na konec neprazenega seznama (zato se začetek seznama ne spremeni).

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);
}
```

Če želimo v seznam, na katerega kaže `zacetek`, dodati element `8`, uporabimo naslednja klica funkcij:

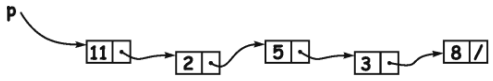
```
zacetek = dodajZacetek(zacetek,8);
zacetek = dodajKonec(zacetek,8);
```

Prva funkcija doda nov element na začetek seznama, druga pa na konec.

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:



Slika 20.16: Brisanje elementa – začetni seznam.

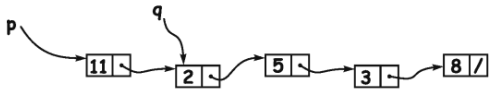
Iz tega seznama bi želeli zbrisati element, ki ima vrednost `5` (v našem primeru tretji element seznama).

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.

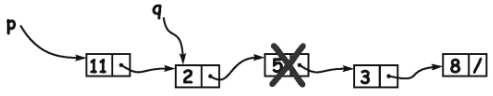
```
q = p;
while((q->naslednji!=NULL)&&(q->naslednji->vrednost!=vred))
  q = q->naslednji;
```

To lahko ponazorimo tudi s sliko:



Slika 20.17: Brisanje elementa – iskanje elementa pred elementom za brisanje.

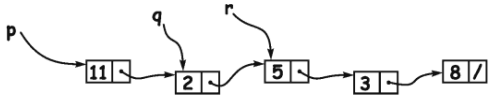
Ko se zanka zaključi, kazalec `q->naslednji` kaže na element, ki ga želimo brisati (če je ta element v seznamu, sicer pa ima vrednost `NULL`):



Slika 20.18: Brisanje elementa – zbrisati želimo element z vrednostjo `5`.

Priredimo to vrednost pomožnemu kazalcu `r`:

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



Slika 20.19: Brisanje elementa – nastavitev kazalca na element za brisanje.

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

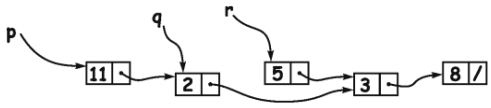
Sedaj, ko smo element našli (nanj kaže `r`), sledi drugi del, to je brisanje samega elementa iz seznama. To enostavno naredimo tako, da seznam prevežemo mimo tega elementa:

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

kar lahko enakovredno zapišemo tudi (glej sliko 20.19):

```
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:

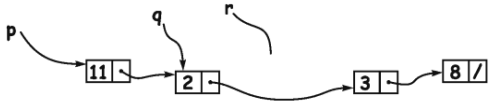


Slika 20.20: Brisanje elementa – prevezava seznama.

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 `r`, ki kaže na element za brisanje):

```
free(r);
```

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



Slika 20.21: Brisanje elementa – sprostitev pomnilnika zbrisanega elementa.

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

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

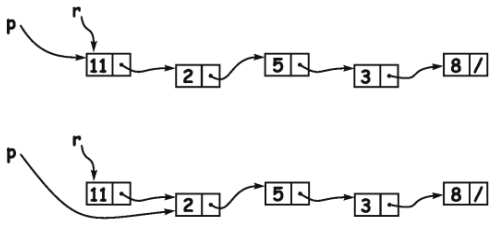
Tudi primer, ko je element, ki ga želimo zbrisati, prvi element seznama, zahteva posebno obravnavo. Recimo, da želimo v našem prvotnem seznamu zbrisati element z vrednostjo `11`. Najprej torej preverimo, ali je vrednost prvega elementa enaka vrednosti, ki jo brišemo, in v primeru izpolnjenega pogoja 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 prikazuje slika 20.22.



Slika 20.22: Brisanje prvega elementa v seznamu.

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);
}
```

Pa še primer uporabe te funkcije. Element z vrednostjo `5` brišemo iz seznama z naslednjim klicem (če elementa v seznamu ni, ostane seznam nespremenjen):

```
zacetek = brisi(zacetek,5);
```

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 za brisanje vseh elementov s podano vrednostjo ima torej 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);
}
```

Uporabimo jo podobno kot druge funkcije za dodajanje in brisanje elementov. Vse pojavitve števila `3` iz seznama, na katerega kaže `zacetek`, zbrišemo s klicem:

```
zacetek = brisiVse(zacetek,3);
```

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 = izprazni(zacetek);
```

Primer uporabe vseh navedenih funkcij za delo s kazalčnim seznamom skupaj s funkcijo `main` je prikazan v programu `seznam.c`, ki demonstrira delo z enosmernim neurejenim linearnim kazalčnim seznamom.

Vračanje vrednosti funkcije preko argumenta

Pri programiranju imamo vedno več načinov, na katere lahko rešimo nek 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 13. 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 dodan 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 tipa `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 v strukturi elementa, saj imata operatorja `.` in `->` višjo prioriteto kot operator `*`. Tako moramo pisati `(*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`.

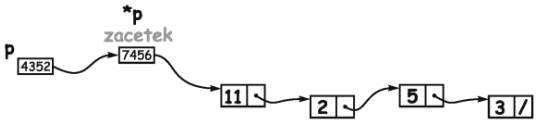
```
struct element *zacetek;
...
dodajZacetek1(&zacetek, 8);
```

Poglejmo si tak način dodajanja elementa še na sliki na primeru našega prejšnjega seznama s štirimi elementi (z vrednostmi `11`, `2`, `5` in `3`, po vrsti), kateremu dodamo na začetek element z vrednostjo `8`.

Na sliki 20.23 smo začetek seznama označili kot spremenljivko `zacetek`, v kateri je zapisan naslov prvega elementa seznama, to je (izmišljena vrednost) 7456.

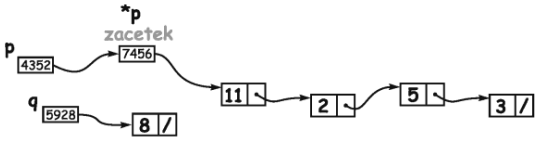
Slika 20.23: Kazalec na začetek seznama pred klicem funkcije dodajanja.

Ob klicu funkcije `dodajZacetek1` se naslov spremenljivke `zacetek` (recimo, da je to 4352) prenese kot argument v spremenljivko `p` (prenos po vrednosti). Vrednost spremenljivke `p` je torej 4352. Tako `p` kaže na `zacetek` oziroma, povedano z drugimi besedami, kaže na kazalec na začetek seznama. V sami funkciji `dodajZacetek1` spremenljivka `zacetek` ne obstaja, zato je na sliki 20.24 prikazana sivo. Na isto lokacijo kot `zacetek` pa se nanaša tudi `*p`.



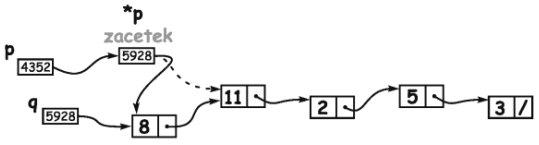
Slika 20.24: Kazalec na začetek seznama pri klicu funkcije (prenos parametrov po referenci).

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`.



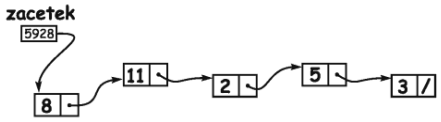
Slika 20.25: Dodajanje novega elementa na začetek seznama.

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 (glej sliko 20.26).



Slika 20.26: Sprememba kazalca na začetek seznama pri dodajanju elementa na začetek.

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 slike 20.27). 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.



Slika 20.27: Seznam po vrnitvi iz funkcije dodajanja.

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:

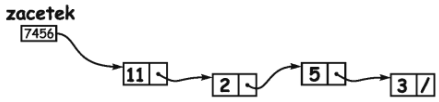
```
struct element *dodajZacetek(struct element *p, int vred) {
  struct element *q;

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

in primer klica funkcije:

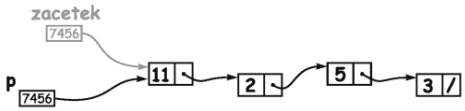
```
struct element *zacetek;
zacetek = dodajZacetek(zacetek, 8);
```

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



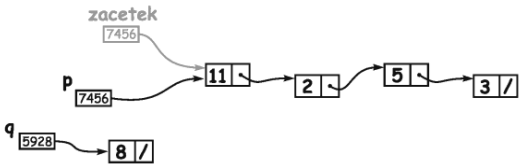
Slika 20.28: Kazalec na začetek seznama pred klicem funkcije dodajanja.

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 20.29 smo spremenljivko `zacetek` posivili, saj v funkciji ni dostopna.



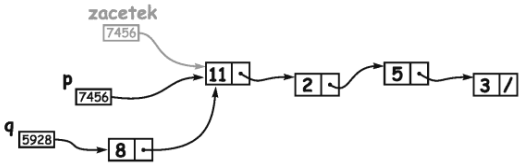
Slika 20.29: Kazalec na začetek seznama pri klicu funkcije (prenos parametrov po vrednosti).

Sledi ustvarjanje novega elementa. V našem primeru je ta element na naslovu 5928 in nanj kaže kazalec `q`, kot prikazuje slika 20.30.



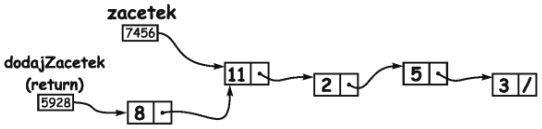
Slika 20.30: Dodajanje novega elementa na začetek seznama.

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`.



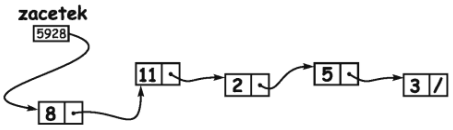
Slika 20.31: Sprememba kazalca na začetek seznama pri dodajanju elementa na začetek.

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



Slika 20.32: Seznam po vrnitvi iz funkcije dodajanja (funkcija vrača kazalec na začetek 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.

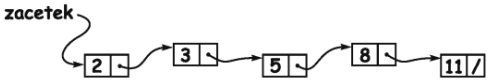


Slika 20.33: Kazalcu na začetek seznama priredimo vrednost, ki jo funkcija vrne.

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:



Slika 20.34: Urejen seznam.

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 ga nismo našli, je `p` enak `NULL` ali pa kaže na element z večjo vrednostjo od iskane. Zato popravimo 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 sicer podobna dodajanju v neurejen seznam, le da moramo najprej ugotoviti, na katero mesto v seznamu dodamo nov element. Če je vrednost, ki jo dodajamo, manjša od vrednosti prvega elementa seznama, dodamo nov element na začetek seznama:

```
if(p->vrednost >= vred) {
  q->naslednji = p;
  return(q);
}
```

Sicer poiščemo ustrezno mesto za nov element. Po seznamu se sprehajamo toliko časa, dokler ne najdemo elementa, ki je večji ali enak novemu elementu (razen če takega elementa ni v seznamu in v tem primeru pridemo do konca seznama):

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

V obeh primerih dodamo nov element za element, na katerega kaže `r`.

Celotna funkcija za urejeno dodajanje je torej naslednja (upoštevali smo urejenost števil od najmanjšega do največjega):

```
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 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);
}
```

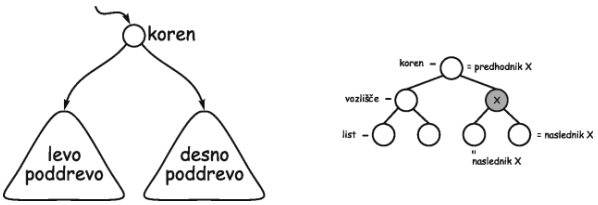
Program, ki demonstrira delo z enosmernim urejenim linearnim kazalčnim seznamom, je v datoteki `urejen.c`. V njem smo že opisanim funkcijam dodali še funkcijo `main`, ki prikazuje uporabo navedenih funkcij.

Binarna drevesa

Drevo kot abstraktni podatkovni tip smo spoznali že v 3. poglavju, kjer smo opisali tudi binarno iskalno drevo. V tem razdelku si bomo ogledali implementacijo binarnega drevesa v jeziku C s povezano strukturo, pri kateri ima vsako vozlišče neposredni referenci na svoja otroka (tako drevo prikazuje slika 3.11).

Za začetek ponovimo nekaj osnovnih pojmov, ki se navezujejo na drevesa. Drevo je hierarhična struktura 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: binarno drevo sestavljajo koren ter levo in desno poddrevo, ki sta tudi binarni drevesi. Seveda je drevo lahko tudi prazno (v tem primeru je koren enak `NULL`). Tako rekurzivno definicijo binarnega drevesa prikazuje slika 20.35:

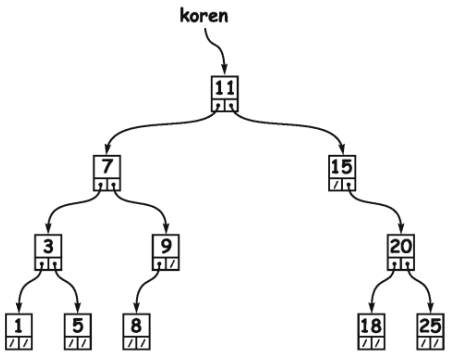


Slika 20.35: Binarno drevo je definirano rekurzivno.

Če so vozlišča binarnega drevesa urejena, strukturo imenujemo urejeno binarno drevo ali binarno iskalno 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 od vrednosti korena. Iz definicije sledi, da urejeno binarno drevo ne vsebuje podvojenih elementov (vsi elementi drevesa so različni).

Primer urejenega binarnega drevesa prikazuje slika 20.36. Drevo smo sestavili z zaporednim dodajanjem elementov `11`, `7`, `3`, `15`, `9`, `8`, `20`, `18`, `1`, `25` in `5`.



Slika 20.36: Primer binarnega drevesa.

V primeru drugačnega zaporedja vstavljanja elementov bi seveda lahko dobili drugačno drevo, saj je mesto vstavitve novega elementa odvisno tako od vrednosti tega elementa kot tudi od ostalih vrednosti, ki so že v drevesu.

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 drevesa in ga deklariramo z naslednjim stavkom:

```
struct vozlisce *koren;
```

Pri praznem drevesu je koren enak `NULL`, zato ga tudi tako inicializiramo:

```
koren = NULL;
```

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
  if(vred > koren->vrednost)
    return(poisci(koren->desno, vred));  // išči v desnem
}
```

Opisano funkcijo bi lahko na primer uporabili za iskanje elementa `5` v drevesu:

```
int obstaja = poisci(koren, 5);
```

Ali pa znotraj odločitvenega stavka:

```
if(poisci(koren, 5) == 1)
  printf("Element 5 je v drevesu.\n");
else
  printf("Elementa 5 ni v drevesu.\n");
```

Izpis drevesa

Binarno drevo lahko izpišemo na tri načine, odvisno od načina prehoda preko drevesa oziroma vrstnega reda izpisa vrednosti korena glede na obe poddrevesi. Če drevo izpisujemo v premem prehodu (*preorder*), potem najprej izpišemo koren, zatem celo levo poddrevo in na koncu celo desno poddrevo. Taki obliki izpisa rečemo tudi prefiksna (*prefix*) oblika.

Drug način obhoda drevesa postavi izpis korena drevesa na konec. Zaporedje levo poddrevo, desno poddrevo in koren tako imenujemo obratni prehod (*postorder*), ustrezno obliko izpisa pa imenujemo tudi postfiksna (*postfix*) oblika.

Tretji način pa predstavlja vmesni prehod (*inorder*), ki izpiše najprej celo levo poddrevo, nato koren in na koncu še desno poddrevo. Pripadajočo obliko izpisa imenujemo tudi infiksna (*infix*) oblika.

Izpis drevesa s slike 20.36 je v posameznih oblikah izpisa 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`

Vidimo, da nam infiksna oblika izpiše elemente drevesa po velikosti od najmanjšega do največjega (tako kot je drevo urejeno).

Funkcije za vse tri načine izpisa so podobne, razlikujejo se le v vrstnem redu stavkov za izpis korena in obeh poddreves. Vse tri funkcije so rekurzivne in imajo isti izstopni pogoj iz rekurzije: če je drevo prazno (`koren` je enak `NULL`), ga ne izpišemo in funkcija se takoj zaključi.

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);
}
```

Infiksna oblika izpisa:

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

Pa še primer klicev vseh treh funkcij za izpis drevesa, na katerega koren kaže `koren`:

```
izpisiIn(koren);
izpisiPre(koren);
izpisiPost(koren);
```

Katero obliko izpisa bomo izbrali, je navadno odvisno od problema in želenega načina prikaza drevesa. Tako se na primer pri urejenih drevesih navadno uporablja kar infiksna oblika, medtem ko sta pri obravnavi aritmetičnih izrazov pogosto uporabni tudi prefiksna in postfiksna oblika.

Dodajanje vozlišča v drevo

Novo vozlišče v urejeno binarno drevo dodamo vedno med liste drevesa. Mesto za novo vozlišče je zaradi urejenosti drevesa točno določeno z vrednostmi elementov drevesa.

Funkcijo za vstavljanje elementa v urejeno drevo bi lahko opisali 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);
}
```

Tudi tu smo uporabili rekurzivni postopek, saj je tak opis najenostavnejši glede na rekurzivno naravo dreves.

Uporaba funkcije za dodajanje je enostavna:

```
koren = dodaj(koren,11);
```

S tem stavkom smo v drevo, na katerega začetek kaže `koren`, dodali število `11`. Seveda se je lahko pri dodajanju elementa v drevo spremenil kazalec na koren drevesa, zato funkcija dodaj vrne ta kazalec, ki ga moramo prirediti spremenljivki `koren`.

Tudi pri dodajanju vrednosti, ki v drevesu že obstaja, se napisana funkcija pravilno obnaša in vrne kar nespremenjeno drevo (vrednost se ne doda, saj urejeno binarno drevo nima podvojenih elementov).

Poglejmo, zakaj. V postopku dodajanja, ko iščemo ustrezno mesto za nov element med listi drevesa, moramo zaradi urejenosti drevesa prej ali slej naleteti na koren, katerega vrednost je enaka vrednosti, ki jo želimo dodati. Ker v tem primeru koren ni prazen (pogoj prvega stavka `if`), niti ni nova vrednost manjša (oziroma večja) od vrednosti korena (drugi oziroma tretji stavek `if`), se izvede le stavek `return(koren)`, ki vrne nespremenjeno drevo in zaključi z rekurzivnimi klici. Če torej dodajamo v drevo že obstoječo vrednost, se drevo ne spremeni.

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, ki smo ga sicer že uporabili pri funkciji iskanja elementa v drevesu, 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 je vrednost tega elementa večja od vrednosti korena, element brišemo iz desnega poddrevesa.

Če pa je vrednost korena enaka vrednosti, ki jo želimo zbrisati, to pomeni, da moramo zbrisati koren drevesa. 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 izbrisali to vozlišče z najmanjšo vrednostjo desnega poddrevesa. Slednje lahko naredimo kar s klicem te iste funkcije `brisi` nad desnim poddrevesom. 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) //elementa ni v drevesu
    return(koren);

  if(vred < koren->vrednost) { //brišemo v levem poddrev.
    koren->levo = brisi(koren->levo, vred);
    return(koren);
  }

  if(vred > koren->vrednost) { //brišemo v desnem delu
    koren->desno = brisi(koren->desno, vred);
    return(koren);
  }

  if(vred == koren->vrednost) { //brišemo koren
    if(koren->levo == NULL) { //koren ima le desno poddrev.
      q = koren;
      koren = koren->desno;
      free(q);
    }
    else if(koren->desno == NULL) { //koren ima le levega
      q = koren;
      koren = koren->levo;
      free(q);
    }
    else {
    //poiščemo najmanjši element v desnem poddrevesu,
    //vrednost korena postane enaka vrednosti tega elementa
    //ter brišemo element, ki smo ga prestavili v koren
      q = koren->desno;
      while(q->levo != NULL)
        q = q->levo;
      koren->vrednost = q->vrednost;
      koren->desno = brisi(koren->desno, q->vrednost);
    }
    return(koren);
  }

}
```

Čeprav je funkcija brisanja elementa iz drevesa nekoliko bolj kompleksna, pa je njena uporaba enostavna. Za brisanje elementa `5` iz drevesa uporabimo naslednji stavek:

```
koren = brisi(koren,5);
```

Seveda se tudi ob brisanju elementa lahko spremeni kazalec na koren drevesa, zato ga funkcija `brisi` vrača; priredimo ga spremenljivki `koren`.

Brisanje celega drevesa

Brisanje celega drevesa je veliko bolj enostavno od brisanja podane vrednosti, saj sistematično brišemo le vse liste (ob tem pa notranja vozlišča postajajo listi). Pri tem seveda uporabimo rekurzijo. Izstopni pogoj je, da je koren enak `NULL`. Če pa koren obstaja (to pomeni, da je različen od `NULL`), moramo najprej izbrisati celo levo in celo desno poddrevo (če obstajata), potem pa lahko brišemo še koren. Brisanje korena pomeni sprostiti pomnilnik, ki ga zaseda koren. Ko koren drevesa zbrišemo, funkcija vrne `NULL` (drevo je prazno).

```
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);
}
```

Če želimo torej uničiti drevo (zbrisati vse njegove elemente), na katerega kaže `koren`, pokličemo funkcijo na naslednji način:

```
koren = izprazni(koren);
```

Po končani operaciji ima `koren` vrednost `NULL`, ki jo je vrnila funkcija `izprazni`.

Vse opisane funkcije za delo z binarnimi drevesi smo skupaj s funkcijo `main` zapisali v programu `drevo.c`, ki demonstrira delo z urejenim binarnim drevesom.

Literatura

Alfred V. Aho, John E. Hopcroft in Jeffrey D. Ullman: *Data Structures and Algorithms*, Addison-Wesley, 1983.

Donald E. Knuth: *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd Edition), Addison-Wesley, 1997.

U. Mesojedec in B. Fabjan: *Java 2: temelji programiranja*, Ljubljana: Pasadena, 2004.

J. Farrell: *Java Programming*, Third Edition, Thomson Course Technology, 2006.

R. Morelli: *Java, Java, Java: Object-oriented problem solving*, Prentice hall, 2000.

B. McLaughlin in D. Flanagan: *Java 1.5 Tiger*, O'Reilly Media, Inc., 2004.

B. Eckel: *Thinking in Java*, Prentice Hall PTR, 1998.

Spletni vir: *The Java Tutorial*, http://java.sun.com/docs/books/tutorial/index.html

Spletni vir: *Painting in AWT and Swing*,
http://java.sun.com/products/jfc/tsc/articles/painting/index.html

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.

Spletni vir: www.cs.waikato.ac.nz/Teaching/COMP134B/lectures/3_recursion.pdf

A. D. Marshall: *Programming in C, UNIX System Calls and Subroutines using C*, 1999.
Spletni vir: http://www.cs.cf.ac.uk/Dave/C/CE.html